Completeness of string analysis for dynamic languages

Completeness of String Analysis for Dynamic Languages

Vincenzo Arceri[b,*], Martina Olliaro[b,c], Agostino Cortesi[b], Isabella Mastroeni[a]

[a] *University of Verona. Department of Computer Science, Strada le Grazie 15, 37134 Verona, Italy*
[b] *Ca' Foscari University of Venice. Scientific Campus, Via Torino 155, 30172 Mestre, Venice, Italy*
[c] *Masaryk University. Faculty of Informatics, Botanickà 68A, 60200 Brno, Czech Republic*

Abstract

In Abstract Interpretation, completeness ensures that the analysis does not lose information with respect to the property of interest. In particular, for dynamic languages like JavaScript, completeness of string analysis is a key security issue, as poorly managed string manipulation code may easily lead to significant security flaws. In this paper, we provide a systematic and constructive approach for generating the completion of string domains for dynamic languages, and we apply it to the refinement of existing string abstractions. We also provide an effective procedure to measure the precision improvement obtained when lifting the analysis to complete domains.

Keywords: Abstract Interpretation, String Analysis, Completeness

1. Introduction

Despite the growth of support for string manipulation in programming languages, string manipulation errors still lead to code vulnerabilities that can be exploited by malicious agents, causing potential catastrophic damages. This is even more true in the context of web applications, where common programming languages used for web-based software development (e.g.,

*Corresponding author
Email addresses: vincenzo.arceri@unive.it (Vincenzo Arceri), martina.olliaro@unive.it (Martina Olliaro), cortesi@unive.it (Agostino Cortesi), isabella.mastroeni@univr.it (Isabella Mastroeni)

JavaScript) offer a wide range of dynamic features that make string manipulation dangerous.

String analysis is a static program analysis technique that computes, for each execution trace of the program given as input, the set of the possible string values that may reach a certain program point. String analysis, like any other non-trivial program analysis, is an undecidable task. Thus, a certain degree of approximation is necessary to find evidence of bugs and vulnerabilities in string manipulating code. In the recent literature, different approximation techniques for string analysis have been developed, such as [8]: automata-based [6, 11, 49, 48], abstraction-based [16, 2, 4, 15, 3, 50], constraint-based [1, 42, 45, 32], and grammar-based [36, 47], and have been used, in particular, to detect web application vulnerabilities [47, 48, 49].

In this paper[1] we focus on string analysis by means of the Abstract Interpretation theory [17, 18]. Cousot and Cousot have proposed Abstract Interpretation in the 70s as a theory of sound abstraction (or approximation) of the semantics of computer programs, and nowadays, it is widely integrated in software verification tools, and it is used to prove approximations correctness by means of rigorous mathematical methods. Since the introduction of the Abstract Interpretation theory, many abstract domains representing properties of interest about numerical domains values have been designed [17, 19, 25, 37, 10, 12, 38, 26, 43]. On the other hand, just in the last few years, the scientific community has taken an interest in the development of abstract domains for string analysis [16, 15, 2, 4, 34, 29, 39], some of them language-specific, such as those defined as part of the JavaScript static analysers: TAJS [27], SAFE [31], and JSAI [28].

Important features of Abstract Interpretation are *soundness* and *completeness* [18]. If soundness (or correctness), as a basic requirement, should always be guaranteed by static analysis tools to avoid the presence of false negatives, completeness is frequently not met. If completeness is satisfied, it means that the abstract computations do not lose information, during the abstraction process, with respect to a property of interest, and so the anal-

[1]This paper is a revised and extended version of [7]. Precisely, we have added an auxiliary procedure to facilitate the reader in understanding the fundamentals theorems of completeness. Moreover, we have introduced the analysis relative precision, which quantifies the increment of the analysis precision gained by analysing a program with a complete abstract domain with respect to when it is analysed with its original version. Finally, we have presented an experimental evaluation.

ysis can be considered optimal. In [24], Giacobazzi et al. highlighted the fact that completeness is an abstract domain property, and they presented a methodology to obtain complete abstract domains with respect to operations by minimally extending or restricting the underlying domains.

1.1. Paper contribution

The goal of this paper is to provide a way-to-proceed in the context of imprecise string abstractions. In particular, we exploit the complete shells' theoretical framework, constructively showing how to improve the precision of incomplete abstractions, without designing new string abstract domains.

We consider two JavaScript-specific string abstract domains defined as part of TAJS [27] and SAFE [31] static analysers, focusing on their completeness with respect to the main string-manipulating operations. We compute their complete versions, and we discuss the benefits of guaranteeing completeness in the context of Abstract Interpretation-based string analysis of dynamic languages. Finally, we present an effective procedure to measure the precision increment when analysing a program with a complete abstract domain.

1.2. Paper structure

Section 2 gives basics in mathematics and Abstract Interpretation. Section 3 recalls relevant concepts related to the completeness property in Abstract Interpretation that we will use throughout the whole paper. Moreover, a motivating example is given to show the importance of guaranteeing completeness in an Abstract Interpretation-based analysis with respect to strings. Section 4 defines our core language. Section 5 presents the completion of the string abstract domains integrated into TAJS and SAFE static analysers with respect to two operations of interest. Section 6 highlights the strengths and usefulness of the completeness approach to static analysis of JavaScript string manipulating programs. Section 7 defines the measurement procedure of the analysis precision increment. Section 8 concludes.

2. Background: Foundations of Abstract Interpretation

Mathematical notation. Given a set S, we denote by S^* the set of all the finite sequences of elements of S. If $s = s_0 \dots s_n \in S^*$, we denote by s_i the i-th element of s, and by $|s| = n + 1$ its length. We denote by $s[x/i]$ the sequence obtained replacing s_i in s with x. Given two sets S and T, we denote with $\mathcal{P}(S)$

the powerset of S, with $S \setminus T$ the set difference, with $S \subset T$ the strict inclusion relation, with $S \subseteq T$ the inclusion relation and with $S \times T$ the Cartesian product between S and T. We denote by S^n the n-ary Cartesian product of a set S, with $n \geq 2$. We denote by $\mathsf{f} : S \to T$ a function from elements of S (domain) to elements of T (image). Given $\mathsf{f} : S \to T$ and $\mathsf{g} : T \to Q$ we denote with $\mathsf{g} \circ \mathsf{f} : S \to Q$ their composition, i.e., $\mathsf{g} \circ \mathsf{f} = \lambda x.\mathsf{g}(\mathsf{f}(x))$. When f is a function of arity n, i.e., $\mathsf{f} : S^n \to T$, with $s \in S^n$ and $i \in [0, n)$, $\mathsf{f}^i_s = \lambda z.\mathsf{f}(s[z/i]) : S \to T$ is the same function where all the parameters but the i-th are fixed by s, namely $\mathsf{f}^i_s = \mathsf{f}(s_0, \ldots, s_{i-1}, x, s_{i+1}, \ldots, s_n)$.

Ordered structures. A set L with a partial ordering relation $\sqsubseteq$ is a poset and it is denoted by $\langle L, \sqsubseteq \rangle$. Given a poset $\langle L, \sqsubseteq \rangle$ and a $X \subseteq L$, an upper bound of X is an element $y \in L$ such that $x \sqsubseteq y$, $\forall x \in X$. An upper bound y for X is the least upper bound (lub) of X if and only if for every other upper bound y' of X it holds that $y \sqsubseteq y'$. The least upper bound, when it exists, it is unique. Dually, a lower bound of X is an element $y \in L$ such that $y \sqsubseteq x$, $\forall x \in X$. A lower bound y of X is the greatest lower bound (glb) of X if and only if for every other lower bound y' of X it holds that $y' \sqsubseteq y$. A poset $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$, where $\sqcup$ and $\sqcap$ are respectively the lub and glb operators of L, is a lattice if $\forall x, y \in L$ we have that $x \sqcap y$ and $x \sqcup y$ belong to L, and we say that it is also complete when for each $X \subseteq L$ we have that $\bigsqcap X, \bigsqcup X \in L$. Given a poset $\langle L, \sqsubseteq \rangle$ and $S \subseteq L$, we denote by $\max^{\sqsubseteq}(S) = \{x \in S \mid \forall y \in S.\, x \sqsubseteq y \Rightarrow x = y\}$ the set of the maximal elements of S in L w.r.t. $\sqsubseteq$. As usual, a complete lattice L, with ordering $\sqsubseteq$, lub $\sqcap$, glb $\sqcup$, greatest element (top) $\top$, and least element (bottom) $\bot$ is denoted by $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$. An *upper closure operator* on a poset $\langle L, \sqsubseteq \rangle$ is an operator $\rho : L \to L$ which is monotone (i.e., $x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y)$), idempotent (i.e., $\rho(\rho(x)) = \rho(x)$), and extensive (i.e., $x \sqsubseteq \rho(x)$). The set of all closure operators on a poset L is denoted by $uco(L)$.

Abstract Interpretation. Abstract Interpretation [17, 18] is a theoretical framework for sound reasoning about program semantic properties of interest, and can be equivalently formalized either as Galois connections or closure operators on a given concrete domain [18]. In this paper, we shall assume that the concrete domain D is a complete lattice w.r.t. to a certain partial order $\sqsubseteq_{\mathsf{D}}$. Observe that this condition is not restrictive, as if it is not matched it is sufficient to lift the concrete domain to its powerset, yielding a complete lattice w.r.t. the set inclusion partial order. This lifting operation in liter-

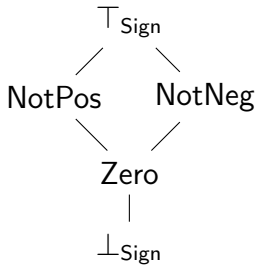


Figure 1: Sign abstract domain

ature is applied in the construction of a *collecting semantics* starting from any concrete semantics [17].

Let D and $\overline{\mathsf{D}}$ be complete lattices, which respectively represent a concrete and an abstract domain, a pair of monotone functions $\alpha : \mathsf{D} \to \overline{\mathsf{D}}$ and $\gamma : \overline{\mathsf{D}} \to \mathsf{D}$ forms a *Galois Connection* (GC) between D and $\overline{\mathsf{D}}$ if for every $\mathsf{d} \in \mathsf{D}$ and for every $\overline{\mathsf{d}} \in \overline{\mathsf{D}}$ we have $\alpha(\mathsf{d}) \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}} \Leftrightarrow \mathsf{d} \sqsubseteq_{\mathsf{D}} \gamma(\overline{\mathsf{d}})$. The function α (resp. γ) is the *left-adjoint* (resp. *right-adjoint*) to γ (resp. α), and it is additive (resp. co-additive). We denote a GC as $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$. Given $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$, then $\gamma \circ \alpha \in \mathit{uco}(\mathsf{D})$. In particular, an upper closure operator (over D) uniquely identifies a GC, and viceversa. For example, let us consider the abstract domain of Sign, abstracting integer sets (i.e., $\mathcal{P}(\mathbb{Z})$ is the concrete domain) reported in Figure 1. We can also see the GC between $\mathcal{P}(\mathbb{Z})$ and Sign by means of the corresponding upper closure operator $\rho_{\mathsf{Sign}} : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$, where its image can be either $\varnothing$, $\{0\}$, $\{n \leq 0 \mid n \in \mathbb{Z}\}$, $\{n \geq 0 \mid n \in \mathbb{Z}\}$ or $\mathbb{Z}$, corresponding to the concretization of the abstract elements Zero, NotPos, NotNeg, $\bot_{\mathsf{Sign}}$ and $\top_{\mathsf{Sign}}$, respectively, of the abstract domain Sign.

If D is a complete lattice, then $\langle \mathit{uco}(\mathsf{D}), \sqsubseteq, \sqcup, \sqcap, \lambda\mathsf{d}.\mathsf{D}, \mathtt{id}\rangle$ forms a complete lattice [46], which is the set of all possible abstractions of D, where the bottom element is $\mathtt{id} = \lambda\mathsf{d}.\mathsf{d}$, and for every $\rho, \eta \in \mathit{uco}(\mathsf{D})$, ρ is *more concrete than* η iff $\rho \sqsubseteq \eta$ iff $\forall \mathsf{d}' \in \mathsf{D}.\ \rho(\mathsf{d}') \sqsubseteq \eta(\mathsf{d}')$, and $(\rho \sqcap \eta)(\mathsf{d}) = \rho(\mathsf{d}) \sqcap \eta(\mathsf{d})$ and $(\rho \sqcup \eta)(\mathsf{d}) = \mathsf{d}$ iff $\rho(\mathsf{d}) = \eta(\mathsf{d}) = \mathsf{d}$. The operator $\rho \in \mathit{uco}(\mathsf{D})$ is disjunctive when $\rho(\mathsf{D})$ is a join-sublattice of D which holds iff ρ is additive [18]. Let L be a complete lattice, then $X \subseteq L$ is a Moore family of L if $X = \mathfrak{M}(X) = \{\sqcap S \mid S \subseteq X\}$, where $\sqcap\varnothing = \top$. The condition that any concrete element of D has the best abstraction in the abstract domain $\overline{\mathsf{D}}$, implies that $\overline{\mathsf{D}}$ is a Moore family of D. We denote by $\mathfrak{M}(X)$ the Moore closure of $X \subseteq \mathsf{D}$, that is the least subset of D, which is a Moore family of D, and

contains X. Given $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$, a concrete function $\mathsf{f} : \mathsf{D} \to \mathsf{D}$ may be not computable. We use a function $\mathsf{f}^\sharp : \overline{\mathsf{D}} \to \overline{\mathsf{D}}$ to *correctly* approximate f, namely $\mathsf{f}^\sharp$ must be *sound*.

Definition 1 (Soundness)**.** Given $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$ and a concrete function $\mathsf{f} : \mathsf{D} \to \mathsf{D}$, an abstract function $\mathsf{f}^\sharp : \overline{\mathsf{D}} \to \overline{\mathsf{D}}$ is sound w.r.t. f if

$$\forall \mathsf{d} \in \mathsf{D}.\ \alpha(\mathsf{f}(\mathsf{d})) \sqsubseteq_{\overline{\mathsf{D}}} \mathsf{f}^\sharp(\alpha(\mathsf{d}))$$

Among all the sound abstract functions $\mathsf{f}^\sharp$, we aim at the best one, namely the one that loses less information when approximating the function f. This property is captured by the notion of best correct approximation.

Definition 2 (Best correct approximation)**.** Given $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$ and a concrete function $\mathsf{f} : \mathsf{D} \to \mathsf{D}$, the function $\alpha \circ \mathsf{f} \circ \gamma : \overline{\mathsf{D}} \to \overline{\mathsf{D}}$ is the best correct approximation of f.

In Abstract Interpretation, there exist two notions of completeness. *Backward completeness* property focuses on complete abstractions of the inputs, while *forward completeness* [23, 22, 21] focuses on complete abstractions of the outputs, both w.r.t. an operation of interest. In this paper, we focus on the more typical and best known notion of completeness, i.e., the backward completeness. In particular, the notion is obtained by enforcing the equality in the soundness condition reported in Definition 1, as reported in Definition 3.

Definition 3 (Backward completeness)**.** Given $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathsf{D}}$, a concrete function $\mathsf{f} : \mathsf{D} \to \mathsf{D}$ and an abstract function $\mathsf{f}^\sharp : \overline{\mathsf{D}} \to \overline{\mathsf{D}}$, $\mathsf{f}^\sharp$ is backward complete w.r.t. f if

$$\forall \mathsf{d} \in \mathsf{D}.\ \alpha(\mathsf{f}(\mathsf{d})) = \mathsf{f}^\sharp(\alpha(\mathsf{d}))$$

Having backward complete abstract functions is a desirable property since, when backward completeness is met, we have the guarantee that no loss of information arises during the input abstraction process, w.r.t. an operation of interest. For instance, the Sign numerical abstract domain, depicted in Figure 1, is backward complete w.r.t. the multiplication numerical operation. Indeed, in order to compute the sign of the expression $\mathsf{e}_1 * \mathsf{e}_2$ it is enough to know the sign of e_1 and e_2, without any loss of information during the abstraction of the operands.

In the rest of the paper, when we will talk about completeness, we mean backward completeness. While the best correct approximation always induces a sound abstraction, this is not the case of completeness. Indeed, given a concrete function $\mathsf{f} : \mathsf{D} \to \mathsf{D}'$, a pair of abstract domains $\langle \overline{\mathsf{D}}, \overline{\mathsf{D}}' \rangle \in uco(\mathsf{D}) \times uco(\mathsf{D}')$, with $\gamma_{\overline{\mathsf{D}},\mathsf{D}} : \overline{\mathsf{D}} \to \mathsf{D}$ the concretization function from $\overline{\mathsf{D}}$ to D and $\alpha_{\mathsf{D}',\overline{\mathsf{D}}'} : \mathsf{D}' \to \overline{\mathsf{D}}'$ the abstraction function from D' to $\overline{\mathsf{D}}'$, then there exists a complete function $\mathsf{f}^\sharp : \overline{\mathsf{D}} \to \overline{\mathsf{D}}'$ iff the best correct approximation $\alpha_{\mathsf{D}',\overline{\mathsf{D}}'} \circ \mathsf{f} \circ \gamma_{\overline{\mathsf{D}},\mathsf{D}} : \overline{\mathsf{D}} \to \overline{\mathsf{D}}'$ is complete (and it is equal to $\mathsf{f}^\sharp$) [24]. Given two posets D and D' and a function $\mathsf{f} : \mathsf{D}^n \to \mathsf{D}'$, we denote by $\Gamma(\mathsf{D}, \mathsf{D}', \mathsf{f})$ the set of pairs of abstract domains $\langle \rho, \eta \rangle \in uco(\mathsf{D}) \times uco(\mathsf{D}')$ which are complete for f. Hence, given a function $\mathsf{f} : \mathsf{D}^n \to \mathsf{D}'$, we can have more than one pair of complete domains $\langle \rho, \eta \rangle \in uco(\mathsf{D}) \times uco(\mathsf{D}')$ for f.

3. Background: Making Abstract Interpretations complete

In this Section, we recall notions of methodologies introduced in [24] that we will use through the whole paper, to constructively build, from an initial abstract domain, a new abstract domain that is complete w.r.t. an operation of interest. Finally, a motivating example showing the usefulness of completion of abstract domains for string analysis is given.

As reported in [24], it is worth noting that completeness is a property related to the underlying abstract domain. Starting from this fact, in [24], authors proposed a constructive method to manipulate the underlying incomplete abstract domain in order to get a complete abstract domain w.r.t. a certain operation. In particular, given two abstract domains $\overline{\mathsf{D}}$ and $\overline{\mathsf{D}}'$ and an operator $\mathsf{f} : \overline{\mathsf{D}}^n \to \overline{\mathsf{D}}'$ with $n \in \mathbb{N}$, the authors gave two different notions of completion of abstract domains w.r.t. f: the one that *adds* the minimal number of abstract points to the input abstract domain $\overline{\mathsf{D}}$ and the other that *removes* the minimal number of abstract points from the output abstract domain $\overline{\mathsf{D}}'$. The first approach captures the notion of *complete shell of* $\overline{\mathsf{D}}$, while the latter defines the *complete core of* $\overline{\mathsf{D}}'$, both w.r.t. an operator f.

Complete shell vs complete core. We will focus on the construction of complete shells of string abstract domains, rather than complete cores. This choice is guided by the fact that a complete core for an operation f removes abstract points from a starting abstract domain. So, even if it is complete for f, the complete core could worsen the precision of other operations.

Conversely, complete shells augment the starting abstract domains (adding abstract points), and consequently, they cannot compromise the precision of other operations.

Below, we recall two important theorems proved in [24] that provide a constructive method to compute abstract domain complete shells, defined in terms of an upper closure operator ρ. Precisely, the latter theorems present two notions of complete shells: *i. complete shells of* ρ *relative to* η (where η is an upper closure operator), meaning that they are complete shells of operations defined on ρ that return results in η, and *ii. absolute complete shells of* ρ, meaning that they are complete shells of operations that are defined on ρ and return results in ρ.

Definition 4 (Complete shell of ρ relative to η [24])**.** Let $\langle A, \sqsubseteq_A, \sqcup_A \rangle$ and $\langle B, \sqsubseteq_B, \sqcup_B \rangle$ be two posets and $f : A^n \to B$ be a continuous function. Given $\rho \in \mathit{uco}(A)$ and $\eta \in \mathit{uco}(B)$, then let $\mathcal{S}^{\eta}_{f} : \mathit{uco}(A) \to \mathit{uco}(A)$ be the domain transformer:

$$\mathcal{S}^{\eta}_{f}(\rho) \stackrel{\text{def}}{=} \bigsqcup_{\mathit{uco}(A)} \{\delta \in \mathit{uco}(A) \mid \delta \sqsubseteq \rho, \langle \delta, \eta \rangle \in \Gamma(A, B, f)\}.$$

If $\langle \mathcal{S}^{\eta}_{f}(\rho), \eta \rangle \in \Gamma(A, B, f)$, then $\mathcal{S}^{\eta}_{f}(\rho)$ is called *complete shell of* ρ *relative to* η *with respect to an operation* f.

As discussed in [24], Definition 4 does not offer a constructive methodology to compute $\mathcal{S}^{\eta}_{f}(\rho)$. Theorem 1 reports a constructive characterization of the complete shell of ρ relative to η w.r.t. f, making use of the Moore closure operator defined in Section 2.

Theorem 1 ([24])**.** Let $\langle A, \sqsubseteq_A, \sqcup_A \rangle$ and $\langle B, \sqsubseteq_B, \sqcup_B \rangle$ be two posets and $f : A^n \to B$ be a continuous function. Given $\rho \in \mathit{uco}(A)$, $\eta \in \mathit{uco}(B)$ and $\mathcal{S}^{\eta}_{f}(\rho)$ as in Definition 4, the following equality holds:

$$\mathcal{S}^{\eta}_{f}(\rho) = \mathfrak{M}(\rho \cup (\bigsqcup_{\substack{A \\ i \in [0,n) \\ x \in A^n, y \in \eta}} \max^{\sqsubseteq_A}(\{z \in A \mid (f^{i}_{x})(z) \sqsubseteq_B y\}))).$$

As already mentioned above, the idea behind the complete shell of ρ (input abstraction) relative to η (output abstraction) is to refine ρ adding the minimum number of abstract points to make ρ complete w.r.t. an operation f. By Theorem 1, this is obtained by adding to ρ the maximal elements in

A, whose f image is dominated by elements in η, at least in one dimension i. Clearly, the so-obtained abstraction may not be an upper closure operator for A. Hence, the Moore closure operator is applied. On the other hand, absolute complete shells are involved in the case in which the operator f of interest has same input and output abstract domain, i.e., $\mathsf{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$. In this case, given $\rho \in \mathit{uco}(\mathrm{A})$, absolute complete shells of ρ can be obtained as the greatest fix-point (*gfp*) of the domain transformer presented in Definition 4, as stated below.

Definition 5 (Absolute complete shell of ρ [24])**.** Let $\langle \mathrm{A}, \sqsubseteq_{\mathrm{A}}, \sqcup_{\mathrm{A}} \rangle$ be a poset and $\mathsf{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$ be a continuous function. Given $\rho \in \mathit{uco}(\mathrm{A})$, then let $\overline{\mathcal{S}}_{\mathsf{f}} : \mathit{uco}(\mathrm{A}) \to \mathit{uco}(\mathrm{A})$ be the domain transformer:

$$\overline{\mathcal{S}}_{\mathsf{f}}(\rho) \stackrel{\text{def}}{=} \bigsqcup_{\mathit{uco}(\mathrm{A})} \{\delta \in \mathit{uco}(\mathrm{A}) \mid \delta \sqsubseteq \rho, \langle \delta, \delta \rangle \in \Gamma(\mathrm{A}, \mathrm{A}, \mathsf{f})\}.$$

If $\langle \overline{\mathcal{S}}_{\mathsf{f}}(\rho), \rho \rangle \in \Gamma(\mathrm{A}, \mathrm{A}, \mathsf{f})$, then $\overline{\mathcal{S}}_{\mathsf{f}}(\rho)$ is called *absolute complete shell of* ρ *with respect to an operation* f.

Theorem 2 ([24])**.** Let A be a poset and $\mathsf{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$ be a continuous function. For $\rho \in \mathit{uco}(\mathrm{A})$, if $\mathcal{S}^{\rho}_{\mathsf{f}}(\rho)$ is the complete shell of ρ relative to ρ w.r.t. f, and if $\overline{\mathcal{S}}_{\mathsf{f}}(\rho)$ is the absolute complete shell of ρ w.r.t. f, then the following equality holds:

$$\overline{\mathcal{S}}_{\mathsf{f}}(\rho) = \mathit{gfp}(\lambda\rho.\mathcal{S}^{\rho}_{\mathsf{f}}(\rho)).$$

In [24], Giacobazzi et al. have discussed the completeness and incompleteness of the Sign abstract domain, approximating numerical values, depicted in Figure 1. Sign is complete for the product operation. Let $* : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ be the concrete product operation and $*_{\mathsf{Sign}} : \mathsf{Sign} \times \mathsf{Sign} \to \mathsf{Sign}$ be the corresponding abstract product operation, following the well known sign rules (e.g., $\mathsf{NotPos} *_{\mathsf{Sign}} \mathsf{NotPos} = \mathsf{NotNeg}$). Given the expression $\mathsf{e}_1 * \mathsf{e}_2$, the equality $\alpha_{\mathsf{Sign}}(\mathsf{e}_1 * \mathsf{e}_2) = \alpha_{\mathsf{Sign}}(\mathsf{e}_1) *_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\mathsf{e}_2)$ holds, with α_{Sign} being the abstraction function of Sign. As an example, consider the concrete expression $\{2,5\}*\{-1,-3\}$, then $\alpha_{\mathsf{Sign}}(\{2,5\}*\{-1,-3\}) = \alpha_{\mathsf{Sign}}(\{-2,-5,-6,-15\}) =$ NotPos is equal to $\alpha_{\mathsf{Sign}}(\{2,5\}) *_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\{-1,-3\}) = \mathsf{NotNeg} *_{\mathsf{Sign}} \mathsf{NotPos} =$ NotPos.

On the other hand, Sign is not complete for the sum operation. Let $+ : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ be the concrete sum operation and $+_{\mathsf{Sign}} : \mathsf{Sign} \times \mathsf{Sign} \to$

Algorithm 1: Relative complete shell procedure pseudo-code

Input: $\langle \mathrm{A}, \sqsubseteq_{\mathrm{A}} \rangle$, $\langle \mathrm{B}, \sqsubseteq_{\mathrm{B}} \rangle$ (posets),
$\rho \in \mathit{uco}(\mathrm{A})$, $\eta \in \mathit{uco}(\mathrm{B})$ (abstractions),
$\mathsf{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{B}$
Output: $\mathcal{S}^{\eta}_{\mathsf{f}}(\rho)$

1 $\mathrm{X} \leftarrow \varnothing$;
2 **foreach** $\mathrm{i} \in [0, \mathrm{n})$ ***and*** $\mathrm{x} \in \mathrm{A}$ ***and*** $\mathrm{y} \in \eta$ **do**
3 **let** $\mathrm{z} \in \mathrm{A}$ be the maximum element such that $\mathsf{f}^{\mathrm{i}}_{\mathrm{x}}(\mathrm{z}) \sqsubseteq_{\mathrm{B}} \mathrm{y}$;
4 **add** z to X;
5 **let** $\mathcal{S}^{\eta}_{\mathsf{f}}(\rho)$ be the Moore closure of $\rho \cup \mathrm{X}$;
6 **return** $\mathcal{S}^{\eta}_{\mathsf{f}}(\rho)$;

Sign be the corresponding abstract sum operation. Consider the concrete expression $\{2\} + \{-1, -2\}$, then $\alpha_{\mathsf{Sign}}(\{2\} + \{-1, -2\}) = \alpha_{\mathsf{Sign}}(\{0, 1\}) = \mathsf{NotNeg}$ is not equal to $\alpha_{\mathsf{Sign}}(\{2\}) +_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\{-1, -2\}) = \mathsf{NotNeg} +_{\mathsf{Sign}} \mathsf{NotPos} = \mathbb{Z}$.

In [24], the absolute complete shell of Sign w.r.t. the sum operation has been computed, which corresponds to the interval abstract domain [17].

Domain completion procedure. To improve the understanding about how to obtain a complete domain w.r.t. an operation of interest we provide a step-by-step reading of the formula in Theorem 1 (i.e., *relative complete shell*) by means of the procedure reported in Algorithm 1. The algorithm takes as input two posets $\langle \mathrm{A}, \sqsubseteq_{\mathrm{A}} \rangle$, $\langle \mathrm{B}, \sqsubseteq_{\mathrm{B}} \rangle$, two closures $\rho \in \mathit{uco}(\mathrm{A})$, $\eta \in \mathit{uco}(\mathrm{B})$, which respectively correspond to the input and output abstraction, and a continuous function $\mathsf{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{B}$. The procedure returns the complete shell of ρ relative to η w.r.t. f. Algorithm 1 follows Theorem 1 and collects in X, for each dimension i and element of ρ, any element $\mathrm{z} \in \mathrm{A}$ whose f image is dominated by elements in η (lines 2-4). Then, the starting input abstraction ρ is joined with the new elements collected in X and since the so obtained result may not be a closure, the Moore closure operator is applied (line 5). Finally, the complete shell of ρ relative to η is returned at line 6.

We reported the procedure only to improve the understanding of the complete shell since, unfortunately, this is not an effective (decidable) procedure. Indeed, Algorithm 1 might diverge (at lines 2-4) when A is an infinite set or η is a not finite closure.

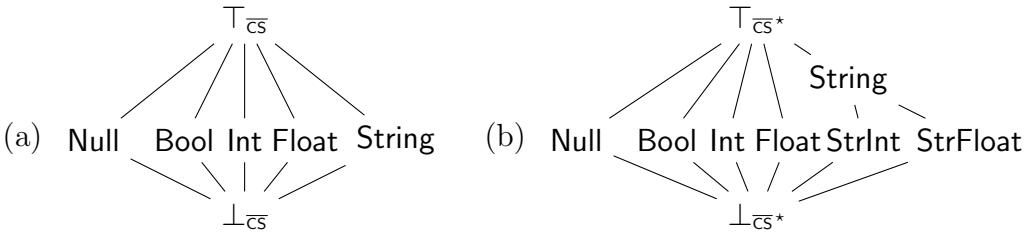


Figure 2: (a) *Coalesced Sum* abstract domain for PHP. (b) Complete shell of *Coalesced Sum* abstract domain w.r.t. the sum operation

3.1. Motivating example

A common feature of dynamic languages, such as PHP or JavaScript, is to be weakly typed. Hence, in those languages, it is allowed to change the variable type through the program execution. For example, in PHP, it is completely legal to write fragments such as `$x=1;$x=true;`, where the type of the variable `x` changes from integer to boolean. The first attempt for statically reasoning about variable types was adopting the so-called *Coalesced Sum* abstract domain ($\overline{\mathsf{CS}}$) [5, 30], to detect whether a certain variable has a constant type through the whole program execution. In Figure 2a, we report the *Coalesced Sum* abstract domain for an intra-procedural version of PHP [5], that tracks null, boolean, integer, float and string types[2]. Consider the formal semantics of the sum operation in PHP [20]. When one of the operands is a string, since the sum operation is feasible only between numbers, *implicit type conversion* occurs and converts the operand string to a number. In particular, if the prefix of the string is a number, it is converted to the maximum prefix of the string corresponding to a number, otherwise it is converted to 0. For example, the expression $e =$ `"2.4hello" + "4"` returns `6.4`. Let $\alpha_{\overline{\mathsf{CS}}}$ and $+_{\overline{\mathsf{CS}}}$ be the abstraction function and the abstract sum operation on the *Coalesced Sum* abstract domain respectively. The type of the expression e is given by: $\alpha_{\overline{\mathsf{CS}}}(\{\texttt{"2.4hello"}\}) +_{\overline{\mathsf{CS}}} \alpha_{\overline{\mathsf{CS}}}(\{\texttt{"4"}\}) = \mathsf{String} +_{\overline{\mathsf{CS}}} \mathsf{String} = \top_{\overline{\mathsf{CS}}}$. The static type analysis based on the *Coalesced Sum* abstract domain returns $\top_{\overline{\mathsf{CS}}}$ (i.e., any possible value), since the sum between two strings may

[2]By closing the *Coalesced Sum* abstract domain with the powerset operation we get a more precise domain called *union type* abstract domain [30], that tracks the set of types of a certain variable during program execution.

return either an integer or a float value. Precisely, the *Coalesced Sum* abstract domain is not complete w.r.t. the PHP sum operation, since for any string σ and σ', it does not meet the completeness condition, i.e., $\alpha_{\overline{\mathsf{CS}}}(\sigma \texttt{ + } \sigma') = \alpha_{\overline{\mathsf{CS}}}(\sigma) \; +_{\overline{\mathsf{CS}}} \; \alpha_{\overline{\mathsf{CS}}}(\sigma')$. Indeed, $\alpha_{\overline{\mathsf{CS}}}(\{\texttt{"2.4hello" + "4"}\}) = \mathsf{Float}$ is different from $\alpha_{\overline{\mathsf{CS}}}(\{\texttt{"2.4hello"}\}) \; +_{\overline{\mathsf{CS}}} \; \alpha_{\overline{\mathsf{CS}}}(\{\texttt{"4"}\}) = \top_{\overline{\mathsf{CS}}}$.

Intuitively, the *Coalesced Sum* abstract domain is not complete w.r.t. the sum operation due to the loss of precision that occurs during the abstraction process of the inputs. Indeed, the domain is not precise enough to distinguish between strings that may be implicitly converted to integers or floats.

Figure 2b shows the complete shell of the *Coalesced Sum* abstract domain w.r.t. the sum ($\overline{\mathsf{CS}}^{\star}$). The latter adds two abstract values to the original domain, namely $\mathsf{StrFloat}$ and StrInt, that correspond to the abstractions of the strings that may be implicitly converted to floats and to integers, respectively. Notice that the type analysis on the new abstract domain is now complete w.r.t. the sum operation, e.g., $\alpha_{\overline{\mathsf{CS}}^{\star}}(\{\texttt{"2.4hello" + "4"}\}) = \mathsf{Float}$ that is equal to $\alpha_{\overline{\mathsf{CS}}^{\star}}(\{\texttt{"2.4hello"}\}) \; +_{\overline{\mathsf{CS}}^{\star}} \; \alpha_{\overline{\mathsf{CS}}^{\star}}(\{\texttt{"4"}\}) = \mathsf{StrFloat} \; +_{\overline{\mathsf{CS}}^{\star}} \; \mathsf{StrInt} = \mathsf{Float}$.

As pointed out above, guaranteeing completeness in Abstract Interpretation is a desirable property that an abstract domain should aim to, since it ensures that no loss of precision occurs during the input abstraction process of the operation of interest. It is worth noting that *guessing* a complete abstract domain for a certain operation becomes particularly hard when the operation has tricky semantics, such as in our example or, more in general, in dynamic languages operations. For this reason, complete shells become important since they can mathematically guarantee completeness for a certain operation, starting from an abstract domain of interest.

4. The core dymamic language μDyn

We define μDyn, an imperative toy language expressive enough to handle some interesting behaviors related to strings in dynamic languages, e.g., implicit type conversion, inspired by the JavaScript programming language [35]. μDyn syntax is reported in Figure 3. The μDyn basic values are represented by the set

$$\textsc{Val} = \textsc{Int} \cup \textsc{Float} \cup \textsc{Bool} \cup \textsc{Str}$$

$$
\begin{aligned}
&\mathsf{a} ::= \mathrm{n} \mid \mathrm{x} \mid \mathsf{a} \texttt{ + } \mathsf{a} \mid \mathsf{a} \texttt{ - } \mathsf{a} \mid \mathsf{a} \texttt{ * } \mathsf{a} \mid \mathsf{a} \setminus \mathsf{a} \\
&\mid \texttt{toNum(s)} \mid \texttt{length(s)} \\
&\mathsf{b} ::= \mathrm{x} \mid \texttt{true} \mid \texttt{false} \mid \mathsf{b} \texttt{ \&\& } \mathsf{b} \mid \mathsf{b} \texttt{ || } \mathsf{b} \mid \texttt{ ! } \mathsf{b} \\
&\mathsf{s} ::= \mathrm{x} \mid \texttt{"s"} \mid \texttt{concat(}\mathsf{s}_1\texttt{,}\mathsf{s}_2\texttt{)} \\
&\mathsf{e} ::= \mathrm{x} \mid \mathsf{a} \mid \mathsf{b} \mid \mathsf{s} \\
&\mathsf{bl} ::= \{\ \} \mid \{\ \mathsf{S}\ \} \\
&\mathsf{S} ::= \mathrm{x} \texttt{ = e;} \mid \texttt{if (b) } \mathsf{bl}_1 \texttt{ else } \mathsf{bl}_2 \mid \texttt{while (b) } \mathsf{bl} \mid \mathsf{S}_1 \ \mathsf{S}_2
\end{aligned}
$$

where $\mathrm{x} \in \textsc{Id}$, $\mathtt{s} \in \Sigma^*$, $\mathrm{n} \in \textsc{Int} \cup \textsc{Float}$

Figure 3: μDyn syntax

such that $\textsc{Int} = \mathbb{Z}$ denotes the set of signed integers, $\textsc{Float}$ denotes the set of signed decimal numbers[3], $\textsc{Bool} = \{\texttt{true}, \texttt{false}\}$ denotes the set of booleans, and $\textsc{Str} = \Sigma^*$ denotes the set of strings over an alphabet Σ. Then, we consider Σ^* composed of two sets, $\Sigma^* = \Sigma^*_{\mathsf{Num}} \cup \Sigma^*_{\mathsf{NotNum}}$, where Σ^*_{Num} is the set of numeric strings (e.g., `"42"`, `"-7.2"`) $\Sigma^*_{\mathsf{NotNum}}$ is the set of non numeric strings (e.g., `"foo"`, `"-2a"`). Moreover, we consider Σ^*_{Num} additionally composed of four sets:

$$\Sigma^*_{\mathsf{Num}} = \Sigma^*_{\mathsf{UInt}} \cup \Sigma^*_{\mathsf{UFloat}} \cup \Sigma^*_{\mathsf{SInt}} \cup \Sigma^*_{\mathsf{SFloat}}$$

From left to right, they correspond to the set of unsigned integer strings, unsigned float strings, signed integer strings, and signed float strings, respectively.

μDyn programs are elements generated by S syntax rules. Program states $\textsc{State} = \{\xi \mid \xi : \textsc{Id} \to \textsc{Val}\}$ are maps from identifiers to values. The concrete semantics of μDyn statements follows[4] [5], and it is given by the function $[\![\cdot]\!]\cdot : \textsc{Stmt} \times \textsc{State} \to \textsc{State}$, inductively defined on the structure of the statements, as reported in Figure 4. We abuse notation in defining the

[3]In general, floats are represented in programming languages in the IEEE 754 double precision format. For the sake of simplicity, we use instead decimal numbers.

[4]Note that in this paper the semantics of the operations is detailed only for the ones that are relevant for string analysis. The complete concrete semantics of μDyn can be found in [5].

$$[\![x = \mathsf{e};]\!]\xi = \xi[x \leftarrow [\![\mathsf{e}]\!]\xi]$$

$$[\![\texttt{if (b) bl}_1 \texttt{ else bl}_2]\!]\xi = \begin{cases} [\![\mathsf{bl}_1]\!]\xi & [\![\mathsf{b}]\!]\xi = \texttt{true} \\ [\![\mathsf{bl}_2]\!]\xi & [\![\mathsf{b}]\!]\xi = \texttt{false} \end{cases}$$

$$[\![\texttt{while (b)bl}]\!]\xi = [\![\texttt{if (b) \{bl while (b)bl\} else \{\}}]\!]\xi$$

$$[\![\{\}]\!]\xi = \xi \qquad [\![\{\mathsf{S}\}]\!]\xi = [\![\mathsf{S}]\!]\xi$$

$$[\![\mathsf{S}_1\mathsf{S}_2]\!]\xi = [\![\mathsf{S}_2]\!]([\![\mathsf{S}_1]\!]\xi)$$

$$[\![\texttt{concat(s, s')}]\!]\xi = [\![\mathsf{s}]\!]\xi \cdot [\![\mathsf{s'}]\!]\xi \qquad [\![\texttt{length(s)}]\!]\xi = |[\![\mathsf{s}]\!]\xi|$$

$$[\![\texttt{toNum(s)}]\!]\xi = \begin{cases} \mathcal{N}([\![\mathsf{s}]\!]\xi) & [\![\mathsf{s}]\!]\xi \in \Sigma^*_{\mathsf{Num}} \\ 0 & \text{otherwise} \end{cases}$$

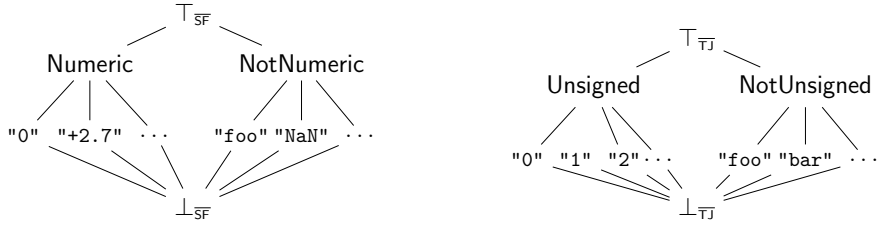
Figure 4: μDyn semantics

concrete semantics of expressions: $[\![\cdot]\!]\cdot : \textsc{Exp} \times \textsc{State} \to \textsc{Val}$. Figure 4 shows the formal semantics of two relevant expressions involving strings we focus on: `concat`, that concatenates two strings, and string-to-number operation, namely `toNum`, that takes a string as input and returns the number that it represents if the input string corresponds to a numerical strings, 0 otherwise. Given $\sigma \in \textsc{Str}$, we denote by $\mathcal{N}(\sigma) \in \textsc{Int} \cup \textsc{Float}$ the numeric value of a given string. For example, `toNum("4.2")` $= 4.2$ and `toNum("asd")` $= 0$.

5. Making JavaScript string abstract domains complete

In this section, we study the completeness of two string abstract domains integrated into two state-of-the-art JavaScript static analysers based on Abstract Interpretation, namely TAJS [27] and SAFE [31]. Both the abstract domains track important information on JavaScript strings, e.g., TAJS can infer when a string corresponds to an unsigned integer, that may be used as array index, and SAFE tracks numeric strings, such as `"2.5"` or `"+5"`.

For the sake of readability, we recast the original string abstract domains for μDyn, following the notation adopted in [4]. Figure 5 depicts them. Notice that the original abstract domain part of SAFE analyser treats the string `"NaN"` as a numeric string. Since our core language does not provide



(a) (b)

Figure 5: (a) SAFE, (b) TAJS string abstract domains recasted for μDyn

the primitive value `NaN`, the corresponding string, i.e., `"NaN"`, has no particular meaning here, and it is treated as a non-numerical string.

For each string abstract domain $\overline{\mathsf{D}}$, we denote by $\alpha_{\overline{\mathsf{D}}} : \mathcal{P}(\Sigma^*) \to \overline{\mathsf{D}}$ its abstraction function, by $\gamma_{\overline{\mathsf{D}}} : \overline{\mathsf{D}} \to \mathcal{P}(\Sigma^*)$ its concretization function, and by $\rho_{\overline{\mathsf{D}}} : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*) \in \mathit{uco}(\overline{\mathsf{D}})$ the associated upper closure operator.

5.1. Completing TAJS string abstract domain

Figure 5b depicts the string abstract domain $\overline{\mathsf{TJ}}$, the recasted version of the domain integrated into the TAJS static analyser [27]. $\overline{\mathsf{TJ}}$ splits the strings into $\mathsf{Unsigned}$, that denotes the strings corresponding to unsigned numbers, and $\mathsf{NotUnsigned}$, any other string. Hence, for example, $\alpha_{\overline{\mathsf{TJ}}}(\{\texttt{"9"}, \texttt{"+9"}\}) = \top_{\overline{\mathsf{TJ}}}$ and $\alpha_{\overline{\mathsf{TJ}}}(\{\texttt{"9.2"}, \texttt{"foo"}\}) = \mathsf{NotUnsigned}$. Before reaching these abstract values, $\overline{\mathsf{TJ}}$ precisely tracks each string.

Here we focus on the `toNum` (i.e., string-to-number) operation. Since this operation clearly involves numbers, in Figure 6 we report the TAJS numerical abstract domain, denoted by $\overline{\mathsf{TJ}}_{\mathsf{N}}$. The latter domain behaves similarly to $\overline{\mathsf{TJ}}$, distinguishing between unsigned and not unsigned integers.

Below we define the abstract semantics of the string-to-number operation for $\overline{\mathsf{TJ}}$. In particular, we define the function:

$$[\![\texttt{toNum}(\bullet)]\!]^{\overline{\mathsf{TJ}}} : \overline{\mathsf{TJ}} \to \overline{\mathsf{TJ}}_{\mathsf{N}}$$

that takes as input a string abstract value in $\overline{\mathsf{TJ}}$, and returns an integer abstract value in $\overline{\mathsf{TJ}}_{\mathsf{N}}$.

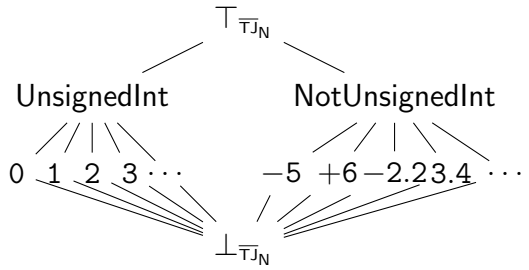


Figure 6: TAJS numerical abstract domain

$$[\![\texttt{toNum(s)}]\!]^{\overline{\mathsf{TJ}}} = \begin{cases} \bot_{\overline{\mathsf{TJ}}_\mathsf{N}} & [\![\mathsf{s}]\!]^{\overline{\mathsf{TJ}}} = \bot_{\overline{\mathsf{TJ}}} \\ [\![\texttt{toNum}(\sigma)]\!] & [\![\mathsf{s}]\!]^{\overline{\mathsf{TJ}}} = \sigma \in \Sigma^* \\ \mathsf{UnsignedInt} & [\![\mathsf{s}]\!]^{\overline{\mathsf{TJ}}} = \mathsf{Unsigned} \\ \top_{\overline{\mathsf{TJ}}_\mathsf{N}} & [\![\mathsf{s}]\!]^{\overline{\mathsf{TJ}}} = \mathsf{NotUnsigned} \vee [\![\mathsf{s}]\!]^{\overline{\mathsf{TJ}}} = \top_{\overline{\mathsf{TJ}}} \end{cases}$$

When the input evaluates to $\bot_{\overline{\mathsf{TJ}}}$, the bottom is propagated, and $\bot_{\overline{\mathsf{TJ}}_\mathsf{N}}$ is returned (first row). If the input evaluates to a single string value, i.e., a concrete string, the abstract semantics relies on its concrete one (second row), as single strings are precisely captured by $\overline{\mathsf{TJ}}$. When the input evaluates to the string abstract value Unsigned, the integer abstract value UnsignedInt is returned (third row). Indeed, in the concrete scenario, an unsigned and not float numeric string, which exactly represents the strings approximated by Unsigned in $\overline{\mathsf{TJ}}$, is converted into the correspondent numeric value by the string-to-number operation. Therefore, the abstraction in $\overline{\mathsf{TJ}}_\mathsf{N}$ of the numeric value of all the strings approximated by Unsigned is UnsignedInt. Finally, when the input evaluates to NotUnsigned or $\top_{\overline{\mathsf{TJ}}}$, the top abstract value $\top_{\overline{\mathsf{TJ}}_\mathsf{N}}$ is returned (fourth row). In the second case, i.e., when the input is evaluated to $\top_{\overline{\mathsf{TJ}}}$, it is trivial to notice that the result of the abstract string-to-number operation represents the best correct approximation. However, in the first case, it is not straightforward. Note that NotUnsigned approximates not numeric strings and signed and/or float numeric strings. Thus the only safe abstraction of the numeric value of strings approximated by NotUnsigned is $\top_{\overline{\mathsf{TJ}}_\mathsf{N}}$.

Lemma 1. The $\overline{\mathsf{TJ}}$ abstract domain is not complete with respect to the

toNum operation. In particular[5], $\exists \mathrm{S} \in \mathcal{P}(\Sigma^*)$ such that:

$$\alpha_{\overline{\mathsf{TJ}}_{\mathsf{N}}}([\![\texttt{toNum}(\mathrm{S})]\!]) \subsetneq [\![\texttt{toNum}(\alpha_{\overline{\mathsf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathsf{TJ}}}$$

Proof of Lemma 1. As a counterexample to completeness, consider the set $\mathrm{S} = \{\texttt{"2.3"}, \texttt{"3.4"}\}$. We can show that $\alpha_{\overline{\mathsf{TJ}}_{\mathsf{N}}}([\![\texttt{toNum}(\mathrm{S})]\!]) \neq [\![\texttt{toNum}(\alpha_{\overline{\mathsf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathsf{TJ}}}$. Indeed,

- $\alpha_{\overline{\mathsf{TJ}}_{\mathsf{N}}}([\![\texttt{toNum}(\mathrm{S})]\!]) = \mathsf{NotUnsignedInt}$
- $[\![\texttt{toNum}(\alpha_{\overline{\mathsf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathsf{TJ}}} = [\![\texttt{toNum}(\mathsf{NotUnsigned})]\!]^{\overline{\mathsf{TJ}}} = \top_{\overline{\mathsf{TJ}}_{\mathsf{N}}}$.

□

The completeness condition does not hold because the $\overline{\mathsf{TJ}}$ string abstract domain loses too much information during the abstraction process, and the latter information cannot be retrieved during the abstract toNum operation. In particular, when non-numeric strings and unsigned integer strings are converted to numbers by toNum, they are mapped to the same value, namely 0. Indeed, $\overline{\mathsf{TJ}}$ does not differentiate between non-numeric and unsigned integer string values, and this is the principal cause of the $\overline{\mathsf{TJ}}$ incompleteness w.r.t. toNum. Additionally, more precision can be obtained if we could differentiate numeric strings holding float numbers from those holding integer numbers. Thus, to make $\overline{\mathsf{TJ}}$ complete w.r.t. toNum, we have to derive the complete shell of the $\overline{\mathsf{TJ}}$ string abstract domain relative to the $\overline{\mathsf{TJ}}_{\mathsf{N}}$ numerical abstract domain, by applying Theorem 1.

Definition 6 (Complete shell of $\overline{\mathsf{TJ}}$)**.** Let $\rho_{\overline{\mathsf{TJ}}} \in \mathit{uco}(\mathcal{P}(\Sigma^*))$ and $\rho_{\overline{\mathsf{TJ}}_{\mathsf{N}}} \in \mathit{uco}(\mathcal{P}(\textsc{Int} \cup \textsc{Float}))$ be the upper closure operators related to $\overline{\mathsf{TJ}}$ and $\overline{\mathsf{TJ}}_{\mathsf{N}}$ abstract domains respectively. Given $[\![\texttt{toNum}(\bullet)]\!]^{\overline{\mathsf{TJ}}} : \overline{\mathsf{TJ}} \to \overline{\mathsf{TJ}}_{\mathsf{N}}$, we define by $\overline{\mathsf{TJ}}^{\star}$ the transformer $\mathcal{S}^{\rho_{\overline{\mathsf{TJ}}_{\mathsf{N}}}}_{\texttt{toNum}}(\rho_{\overline{\mathsf{TJ}}})$.

By Definition 4, $\overline{\mathsf{TJ}}^{\star}$ is the complete shell of $\rho_{\overline{\mathsf{TJ}}}$ relative to $\rho_{\overline{\mathsf{TJ}}_{\mathsf{N}}}$ with respect to the toNum operation. As already argued in Section 5, to compute $\overline{\mathsf{TJ}}^{\star}$ we use the constructive characterization given by Theorem 1.

Figure 7 depicts $\overline{\mathsf{TJ}}^{\star}$. In particular, the abstract points inside dashed boxes are the abstract values added during the computation of $\overline{\mathsf{TJ}}^{\star}$, the

[5]We abuse notation denoting with $[\![\cdot]\!]$ the additive lift to set of basic values of the concrete semantics, i.e., the collecting semantics.

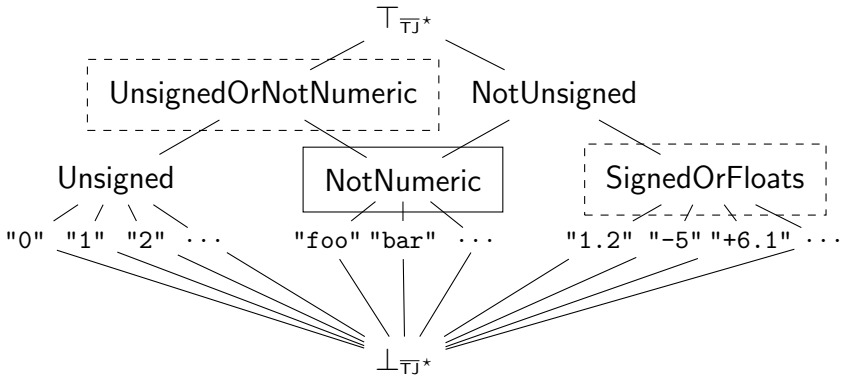


Figure 7: Complete shell of $\rho_{\overline{\mathsf{TJ}}}$ relative to $\rho_{\overline{\mathsf{TJ}}_\mathsf{N}}$ w.r.t. `toNum`

point inside the solid box is instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in $\overline{\mathsf{TJ}}$. A non-intuitive point added in $\overline{\mathsf{TJ}}^\star$ is SignedOrFloats, namely the abstract value s.t. its concretization contains any float string and the signed integers. This abstract point is added during the computation of $\overline{\mathsf{TJ}}^\star$. In particular, following Algorithm 1, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathsf{TJ}}_\mathsf{N}}$, this abstract point is computed at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathsf{TJ}}_\mathsf{N}}(\mathsf{NotUnsignedInt})$, namely

$$\mathsf{SignedOrFloats} \in \max{}^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{toNum}(\mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathsf{TJ}}_\mathsf{N}}(\mathsf{NotUnsignedInt})\})$$

Informally: which is the maximal set of strings Z s.t. `toNum`(Z) is dominated by NotUnsignedInt? To obtain from `toNum`(Z) only values dominated by NotUnsignedInt, the maximal set doing so is exactly the set of the float strings and the signed strings. Other strings, such that: unsigned integer strings or not numerical strings are excluded, since they are both converted to unsigned integers, and they would violate the dominance relation. Similarly, the abstract point UnsignedOrNotNumeric is added to the complete shell $\overline{\mathsf{TJ}}^\star$. Following again Algorithm 1, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathsf{TJ}}_\mathsf{N}}$, the above abstract element is computed at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathsf{TJ}}_\mathsf{N}}(\mathsf{UnsignedInt})$, namely

$$\mathsf{UnsignedOrNotNumeric} \in \max{}^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid \texttt{toNum}(\mathrm{Z}) \subseteq \gamma_{\overline{\mathsf{TJ}}_\mathsf{N}}(\mathsf{UnsignedInt})\})$$

To obtain from `toNum`(Z) only values dominated by UnsignedInt, the maximal set doing so is exactly the set of the unsigned integer strings and the non-

numerical strings, since the latter are converted to 0. Any other string, such as a signed or float string, would violate the dominance relation.

The last point added in the complete shell is NotNumeric. In particular, this abstract point is added by the Moore closure between Unsigned and UnsignedOrNotNumeric. The remaining abstract points, namely Unsigned and NotUnsigned, were already present in the original TAJS string abstract domain, and they are leaved unaltered in its complete shell w.r.t. `toNum`.

Example 1. Consider again the string set $\mathrm{S} = \{\texttt{"2.3"}, \texttt{"3.4"}\}$ of Example 5.1. We can show that in $\overline{\mathsf{TJ}}^{\star}$, $\alpha_{\overline{\mathsf{TJ}}_{\mathsf{N}}}(\llbracket \texttt{toNum}(\mathrm{S}) \rrbracket) = \llbracket \texttt{toNum}(\alpha_{\overline{\mathsf{TJ}}^{\star}}(\mathrm{S})) \rrbracket^{\overline{\mathsf{TJ}}^{\star}}$. Indeed, $\alpha_{\overline{\mathsf{TJ}}_{\mathsf{N}}}(\llbracket \texttt{toNum}(\mathrm{S}) \rrbracket) = \mathsf{NotUnsignedInt}$ and,

$$\begin{aligned} \llbracket \texttt{toNum}(\alpha_{\overline{\mathsf{TJ}}^{\star}}(\mathrm{S})) \rrbracket^{\overline{\mathsf{TJ}}^{\star}} &= \llbracket \texttt{toNum}(\mathsf{SignedOrFloats}) \rrbracket^{\overline{\mathsf{TJ}}^{\star}} \\ &= \mathsf{NotUnsignedInt} \end{aligned}$$

Note that, the new abstract semantics $\llbracket \texttt{toNum}(\bullet) \rrbracket^{\overline{\mathsf{TJ}}^{\star}}$, handling the abstract points added by the complete shell, corresponds to the best correction approximation, namely $\alpha_{\overline{\mathsf{TJ}}^{\star}} \circ \llbracket \texttt{toNum}(\bullet) \rrbracket \circ \gamma_{\overline{\mathsf{TJ}}^{\star}} : \overline{\mathsf{TJ}}^{\star} \to \overline{\mathsf{TJ}}_{\mathsf{N}}$ (see Definition 2).

5.2. Completing SAFE string abstract domain

Figure 5a depicts the string abstract domain $\overline{\mathsf{SF}}$, i.e., the recasted version of the domain involved into SAFE [31] static analyser. It splits strings into the abstract values: Numeric (i.e., numerical strings) and NotNumeric (i.e., all the other strings). As for $\overline{\mathsf{TJ}}$, before reaching these abstract values, $\overline{\mathsf{SF}}$ precisely tracks single string values. For instance, $\alpha_{\overline{\mathsf{SF}}}(\{\texttt{"+9.6"}, \texttt{"7"}\}) = \mathsf{Numeric}$, and $\alpha_{\overline{\mathsf{SF}}}(\{\texttt{"+9.6"}, \texttt{"bar"}\}) = \top_{\overline{\mathsf{SF}}}$.

We study the completeness of $\overline{\mathsf{SF}}$ w.r.t. `concat` operation. Figure 8 presents the abstract semantics of the concatenation operation for $\overline{\mathsf{SF}}$, that is:

$$\llbracket \texttt{concat}(\bullet, \bullet) \rrbracket^{\overline{\mathsf{SF}}} : \overline{\mathsf{SF}} \times \overline{\mathsf{SF}} \to \overline{\mathsf{SF}}$$

In particular, when both abstract values correspond to single strings, the standard string concatenation is applied (second row, second column). In the case in which one abstract value, involved in the concatenation, is a string and the other is Numeric (third row, second column and second row, third column) we distinguish two cases: if the string is empty or corresponds to an unsigned integer we can safely return Numeric, otherwise NotNumeric is returned. This

	$[\![\texttt{concat}(\mathsf{s}_1,\mathsf{s}_2)]\!]^{\overline{\mathsf{SF}}}$				
$[\![\mathsf{s}_1]\!]^{\overline{\mathsf{SF}}}$ \ $[\![\mathsf{s}_2]\!]^{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\sigma_2 \in \Sigma^*$	Numeric	NotNumeric	$\top_{\overline{\mathsf{SF}}}$
$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$
$\sigma_1 \in \Sigma^*$	$\bot_{\overline{\mathsf{SF}}}$	$\sigma_1 \cdot \sigma_2$	$\begin{cases} \mathsf{Numeric} & \sigma_1 = \texttt{""} \text{ or } \\ & \sigma_1 \in \Sigma^*_{\mathsf{UInt}} \\ \mathsf{NotNumeric} & \text{otherwise} \end{cases}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$
Numeric	$\bot_{\overline{\mathsf{SF}}}$	$\begin{cases} \mathsf{Numeric} & \sigma_2 = \texttt{""} \text{ or } \\ & \sigma_2 \in \Sigma^*_{\mathsf{UInt}} \\ \mathsf{NotNumeric} & \text{otherwise} \end{cases}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$
NotNumeric	$\bot_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	NotNumeric	$\top_{\overline{\mathsf{SF}}}$
$\top_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$	$\top_{\overline{\mathsf{SF}}}$

Figure 8: SAFE `concat` abstract semantics

happens because, when two float strings (hence numerical strings) are concatenated, a non-numerical string is returned (e.g., $\texttt{concat}(\texttt{"1.1"},\texttt{"2.2"}) = \texttt{"1.12.2"}$). For the same reason, when both input abstract values are Numeric, the result is not guaranteed to be numerical. Indeed, $[\![\texttt{concat(Numeric,Numeric)}]\!]^{\overline{\mathsf{SF}}} = \top_{\overline{\mathsf{SF}}}$.

Lemma 2. The $\overline{\mathsf{SF}}$ abstract domain is not complete with respect to the `concat` operation. In particular, $\exists \mathrm{S}_1, \mathrm{S}_2 \in \mathcal{P}(\Sigma^*)$ such that:

$$\alpha_{\overline{\mathsf{SF}}}([\![\texttt{concat}(\mathrm{S}_1,\mathrm{S}_2)]\!]) \sqsubsetneq [\![\texttt{concat}(\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_1),\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathsf{SF}}}$$

Proof of Lemma 2. As a counterexample to completeness, consider the sets $\mathrm{S}_1 = \{\texttt{"2.2"},\texttt{"2.3"}\}$ and $\mathrm{S}_2 = \{\texttt{"2"},\texttt{"3"}\}$. Then, in $\overline{\mathsf{SF}}$, $\alpha_{\overline{\mathsf{SF}}}([\![\texttt{concat}(\mathrm{S}_1,\mathrm{S}_2)]\!]) \neq [\![\texttt{concat}(\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_1),\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathsf{SF}}}$. Indeed, $\alpha_{\overline{\mathsf{SF}}}([\![\texttt{concat}(\mathrm{S}_1,\mathrm{S}_2)]\!]) = \mathsf{Numeric}$ and,

$$[\![\texttt{concat}(\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_1),\alpha_{\overline{\mathsf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathsf{SF}}} = [\![\texttt{concat}(\mathsf{Numeric},\mathsf{Numeric})]\!]^{\overline{\mathsf{SF}}} = \top_{\overline{\mathsf{SF}}}$$

□

The $\overline{\mathsf{SF}}$ abstract domain loses too much information during the abstraction process, which can not be retrieved during the abstract concatenation. Intuitively, to gain completeness w.r.t. `concat` operation, $\overline{\mathsf{SF}}$ should improve the precision of the numerical strings abstraction, e.g., discriminating between float and integer strings.

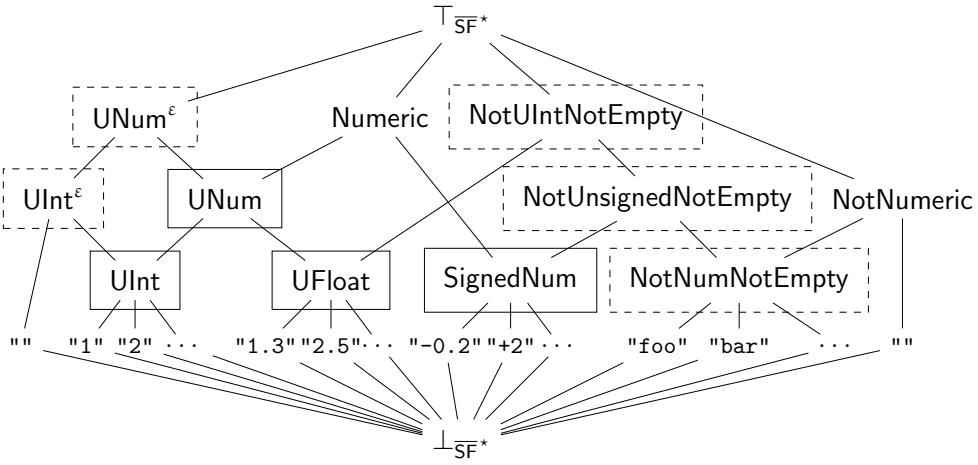


Figure 9: Absolute complete shell of $\rho_{\overline{\mathsf{SF}}}$ w.r.t. `concat`

Definition 7 (Complete shell of $\overline{\mathsf{SF}}$)**.** Let $\rho_{\overline{\mathsf{SF}}} \in uco(\mathcal{P}(\Sigma^*))$ be the upper closure operator related to $\overline{\mathsf{SF}}$ abstract domain. Given $[\![\texttt{concat}(\bullet,\bullet)]\!]^{\overline{\mathsf{SF}}} : \overline{\mathsf{SF}} \times \overline{\mathsf{SF}} \to \overline{\mathsf{SF}}$, we define by $\overline{\mathsf{SF}}^\star$ the transformer $\overline{\mathcal{S}}_{\texttt{concat}}(\rho_{\overline{\mathsf{SF}}})$.

By Definition 5, $\overline{\mathsf{SF}}^\star$ is the absolute complete shell of $\rho_{\overline{\mathsf{SF}}}$ with respect to the `concat` operation.

By Theorem 2, the transformer $\overline{\mathcal{S}}_{\texttt{concat}}(\rho_{\overline{\mathsf{SF}}})$ is equal to the Moore closure of the union between $\overline{\mathsf{SF}}$ and the binary operator defined in Table 1. Table 1 depicts the first iteration of the fix-point computation of Theorem 2, where $\bot_{\overline{\mathsf{SF}}}$ and $\top_{\overline{\mathsf{SF}}}$ rows and columns are omitted for space limitations; note that Table 1 also corresponds to the whole fix-point result of Theorem 2, since the fix-point is reached at the next step. In particular, $\forall \mathrm{X} \in \mathcal{P}(\Sigma^*).\ \max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat}(\mathrm{Z},\mathrm{X})]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\bot_{\overline{\mathsf{SF}}})\}) = \bot_{\overline{\mathsf{SF}}}$, $\max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat}(\mathrm{X},\mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\bot_{\overline{\mathsf{SF}}})\}) = \bot_{\overline{\mathsf{SF}}}$ and $\max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat}(\mathrm{X},\mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\bot_{\overline{\mathsf{SF}}})\}) = \top_{\overline{\mathsf{SF}}}$, when $\mathrm{X} \neq \bot_{\overline{\mathsf{SF}}}$, $\max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat}(\mathrm{Z},\mathrm{X})]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\bot_{\overline{\mathsf{SF}}})\}) = \bot_{\overline{\mathsf{SF}}}$, when $\mathrm{X} \neq \top_{\overline{\mathsf{SF}}}$. The complete shell $\overline{\mathsf{SF}}^\star$ is depicted in Figure 9.[6] In particular, the points inside dashed boxes are the abstract values added during the iterative computations of $\overline{\mathsf{SF}}^\star$, the points inside solid boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were

[6]The empty strings (`""`) at the most-left and most-right sides of Figure 9 are not distinct elements but they are only duplicated in order to declutter the figure.

X \ Y	Numeric	NotNumeric	"n" $\in \mathbb{Z}$	"f" $\in \mathbb{F}$	"s" $\in$ NotNum
Numeric	{""} $\cup$ UInt	[NotNum $\setminus$ {""}] $\cup$ NotUInt $\cup$ NotUFloat	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$
NotNumeric	$\bot_{\overline{\mathsf{SF}}}$	NotNumeric	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$
"n" $\in \mathbb{Z}$	{""} $\cup$ UInt $\cup$ UFloat	[NotNum $\setminus$ {""}] $\cup$ NotUInt $\cup$ NotUFloat	$\bot_{\overline{\mathsf{SF}}} \vee str$	$\bot_{\overline{\mathsf{SF}}} \vee str$	$\bot_{\overline{\mathsf{SF}}} \vee str$
"f" $\in \mathbb{F}$	{""} $\cup$ UInt	[NotNum $\setminus$ {""}] $\cup$ Float $\cup$ NotUInt	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}} \vee str$	$\bot_{\overline{\mathsf{SF}}} \vee str$
"s" $\in$ NotNum	$\bot_{\overline{\mathsf{SF}}}$	NotNumeric	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}}$	$\bot_{\overline{\mathsf{SF}}} \vee str$

Table 1: First iteration of $\max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat(X, Z)}]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\mathrm{Y})\})$

already in $\overline{\mathsf{SF}}$. The meaning of abstract values in $\overline{\mathsf{SF}}^\star$ is intuitive. In order to satisfy the completeness property, $\overline{\mathsf{SF}}^\star$ splits the Numeric abstract value, already taken into account in $\overline{\mathsf{SF}}$, into all the strings corresponding to unsigned integer (UInt), unsigned floats (UFloat), and signed numbers (SignedNum). Moreover, particular importance is given to the empty string since the new abstract domain specifies whether each abstract value contains "". Indeed, the $\mathsf{UInt}^\varepsilon$ abstract value represents the strings corresponding to unsigned integer or the empty string. The $\mathsf{UNum}^\varepsilon$ abstract value represents the strings corresponding to unsigned numbers or the empty string. An unexpected abstract value considered in $\overline{\mathsf{SF}}^\star$ is NotUnsignedNotEmpty, such that:

$$\gamma_{\overline{\mathsf{SF}}^\star}(\mathsf{NotUnsignedNotEmpty}) = \{\sigma \mid \sigma \in \Sigma^*_{\mathsf{SInt}} \cup \Sigma^*_{\mathsf{SFloat}} \cup (\Sigma^*_{\mathsf{NotNum}} \setminus \{\texttt{""}\})\}$$

Namely, the abstract point whose concretization corresponds to the set of any non-numerical string, except the empty string, and any string corresponding to a signed number. This abstract point has been added to $\overline{\mathsf{SF}}^\star$, following Theorem 2. Since the absolute complete shell is the greatest fix-point of the relative one (that in our case is reached after one iteration), the corresponding procedure is the greatest fix-point of Algorithm 1, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathsf{SF}}}$. In particular, the abstract element is computed

at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathsf{SF}}}(\mathsf{NotNumeric})$, x is $\gamma_{\overline{\mathsf{SF}}}(\mathsf{Numeric})$ and $\mathrm{i} = 1$, namely

$$\mathsf{NotUnsignedNotEmpty} \in \max^{\subseteq}(\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(\gamma_{\overline{\mathsf{SF}}}(\mathsf{Numeric}), \mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathsf{SF}}}(\mathsf{NotNumeric}))$$

Informally: which is the maximal set of strings s.t. concatenated to any possible numerical string will produce any possible non-numerical string? Indeed, to be sure to obtain non-numerical strings, the maximal set doing so is exactly the set of any non-numerical non-empty string, and any string corresponding to a signed number, that is $\mathsf{NotUnsignedNotEmpty}$.

Example 2. Consider again the string sets $\mathrm{S}_1 = \{\texttt{"2.2"}, \texttt{"2.3"}\}$ and $\mathrm{S}_2 = \{\texttt{"2"}, \texttt{"3"}\}$ of Example 5.2. We can show that in $\overline{\mathsf{SF}}^{\star}$, $\alpha_{\overline{\mathsf{SF}}^{\star}}([\![\mathtt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) = [\![\mathtt{concat}(\alpha_{\overline{\mathsf{SF}}^{\star}}(\mathrm{S}_1), \alpha_{\overline{\mathsf{SF}}^{\star}}(\mathrm{S}_2))]\!]^{\overline{\mathsf{SF}}^{\star}}$. Indeed, $\alpha_{\overline{\mathsf{SF}}^{\star}}([\![\mathtt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) = \mathsf{UFloat}$ and $[\![\mathtt{concat}(\alpha_{\overline{\mathsf{SF}}^{\star}}(\mathrm{S}_1), \alpha_{\overline{\mathsf{SF}}^{\star}}(\mathrm{S}_2))]\!]^{\overline{\mathsf{SF}}^{\star}} = [\![\mathtt{concat}(\mathsf{UFloat}, \mathsf{UInt})]\!]^{\overline{\mathsf{SF}}^{\star}} = \mathsf{UFloat}$.

As in the TAJS case, the new abstract semantics $[\![\mathtt{concat}(\bullet, \bullet)]\!]^{\overline{\mathsf{SF}}^{\star}}$, handling the abstract points added by the complete shell, corresponds the best correct approximation, namely $\alpha_{\overline{\mathsf{SF}}^{\star}} \circ [\![\mathtt{concat}(\bullet, \bullet)]\!] \circ \gamma_{\overline{\mathsf{SF}}^{\star}} : \overline{\mathsf{SF}}^{\star} \times \overline{\mathsf{SF}}^{\star} \to \overline{\mathsf{SF}}^{\star}$ (see Definition 2).

6. What do we gain from using a complete abstract domain?

Now, we discuss and evaluate the benefits of adopting the complete shells reported in Section 5 and, more in general, complete domains, w.r.t. a certain operation. In particular, we compare the μDyn versions of the string abstract domains adopted by SAFE and TAJS with their corresponding complete shells. We discuss the complexity of the complete shells, and finally we argue how adopting complete abstract domains can be useful into static analysers.

Precision. In the previous section, we focused on the completeness of the string abstract domains integrated into SAFE and TAJS, for μDyn, w.r.t. two string operations, namely `concat` and `toNum`, respectively. While string concatenation is common in any programming language, `toNum` assumes critical importance for dynamic languages, where implicit type conversion is provided. It is worth noting that, in this paper we have only considered `toNum` operation but the completion construction of other implicit type conversion operations (e.g., `toBool` or `toString`) is similar to the one reported in Section 5.1. Since type conversion is often hidden from the developer, aim

```
1 var obj = {
2     "foo" : 1,
3     "bar" : 2,
4     "1.2" : 3,
5     "2.2" : "hello"
6 }
7
8 y = obj[idx];
```

Figure 10: JavaScript object access example

to completeness of the analysis increases the precision of such operations. For instance, let `x` be a variable, at a certain program execution point. `x` may have concrete value in the set $S = \{\texttt{"foo"}, \texttt{"bar"}\}$. If S is abstracted into the original TAJS string abstract domain, its abstraction will corresponds to NotUnsigned, losing the information about the fact that the concrete value of `x` surely does not contain numerical values. Hence, when the abstract value of S is used as input of `toNum`, the result will return $\top_{\overline{\mathsf{TJ}}_{\mathsf{N}}}$, i.e., any possible concrete integer value. Conversely, abstracting S in $\overline{\mathsf{TJ}}^{\star}$ (the absolute complete shell of $\overline{\mathsf{TJ}}$ relative to `toNum` discussed in Section 5.1) leads to a more precise abstraction, since $\overline{\mathsf{TJ}}^{\star}$ is able to differentiate between non-numerical and numerical strings. In particular, the abstract value of S in $\overline{\mathsf{TJ}}^{\star}$ is NotNumeric, and $[\![\texttt{toNum}(\mathsf{NotNumeric})]\!]^{\overline{\mathsf{TJ}}^{\star}}$ will precisely return 0.

Adopting a complete shell w.r.t. a certain operation does not compromise the precision of the others. For example, the JavaScript fragment reported in Figure 10 and let us analyze it with the domain of TAJS. Suppose that the value of `idx` is the abstraction, in the starting TAJS string abstract domain, of the string set $S = \{\texttt{"foo"}, \texttt{"bar"}\}$, namely the abstract value NotUnsigned. The variable `idx` is used to access the property of the object `obj` at line 8 and, to guarantee soundness, it accesses *all* the properties of `obj`, including the fields `"1.2"` and `"2.2"`, introducing noise in the abstract computation, since `"1.2"` and `"2.2"` are false positives values introduced by the abstraction of the values of `idx`. If we analyse the same JavaScript fragment with the absolute complete shell (w.r.t. `toNum` operation) of the TAJS string abstract domain defined in Section 5.1, we obtain more precise results. Indeed, in this case, the value of `idx` corresponds to the the abstract value NotNumeric, and when it is used to access the object `obj` at line 8, only `"foo"` and `"bar"` are accessed, since they are the only non-numerical string properties of `obj`.

Qualitative evaluation of complete shells. We evaluate the complete shells we have provided in the previous section from a qualitative point of view. As usual in static analysis by Abstract Interpretation, there is a trade-off between precision and efficiency: choosing a more precise abstract domain may compromise the efficiency of the abstract computations. A representative example is reported in [24]: the complete shell of the sign abstract domain w.r.t. addition is the interval abstract domain. Hence, starting from a finite height abstract domain (signs) we obtain an infinite height abstract domain (intervals). In particular, fix-point computations on signs converge, while on intervals may diverge. Indeed, after the completion, the interval abstract domain should also be equipped with a widening operator [17] to still guarantee termination. A worst-case scenario is when the complete shells w.r.t. a certain operation exactly corresponds to the collecting abstract domain, i.e., the concrete domain. Clearly, we cannot use the concrete domain due to undecidability reasons, but this suggests us to change the starting abstract domain, since it is not able to track any information related to the operation of interest. An example is the suffix abstract domain [16] with `substring` operation: since this abstract domain tracks only the common suffix of a strings set, it can not track the information about the indexes of the common suffix, and the complete shell of the suffix abstract domain w.r.t. `substring` would lead to the concrete domain. Hence, if the focus of the abstract interpreter is to improve the precision of the `substring` operation, we should change the abstract domain with a more precise one for `substring`, such as the finite state automata [6] abstract domain.

Consider now the complete shells reported in Section 5. The obtained complete shells still have finite height, hence termination is still guaranteed without the need to equip the complete shells with widening operators. Moreover, the complexity of the string operations of interest is preserved after completion. Indeed, in both TAJS and SAFE starting abstract domains, `concat` and `toNum` operations have constant complexity, respectively, and the same complexity is preserved in the corresponding complete shells. It is worth noting that also the complexity of the abstract domain-related operations, such as least upper bound, greatest lower bound and the ordering operator, is preserved in the complete shells. Hence, to conclude, as far as the complete shells we have reported for TAJS and SAFE are concerned, there is no worsening when we substitute the original string abstract domains with the corresponding complete shells, and this leads, as we have already mentioned before, to completeness during the input abstraction process w.r.t. the

relative operations, namely `concat` for SAFE and `toNum` for TAJS.

False positives reduction. In static analysis, a certain degree of abstraction must be added in order to obtain decidable procedures to infer invariants on a generic program. Clearly, using less precise abstract domains lead to an increase of *false positive* values of the computed invariants. In particular, after a program is analysed, this burdens the phase of false positive detection: when a program is analysed, the following phase consists in detecting which values of the invariants derived by the static analyser are spurious, namely those values that certainly are not computed by the concrete execution of the program of interest.

In particular, using imprecise (i.e., not complete) abstract domains clearly increases the number of false positives in the abstract computation of the static analyser, burdening the next phase of detection of the spurious values. Conversely, adopting (backward) complete abstract domains w.r.t. a certain operation reduce the numbers of false positives introduced during the abstract computations, at least in the input abstraction process. Clearly, in this way, the next phase of detecting false positives will be lighter since less noise has been introduced during the abstract computation of the invariants. For example, let us consider the following simplified example, inspired by [41].

```
if (valid(in))
   safeOnlyIfValid(in);
else
   safeOnlyIfInvalid(in);
```

It is common, in web developing, to check and sanitize user inputs before performing some action in a secure way, e.g., for preventing XSS or injections attacks. For instance, we can suppose that the function `valid` in the above example checks whether the input string `in` does not contain the string "$<$`script`": if so, it returns `true`, `false` otherwise. Hence, if `in` is valid, some safe action with valid inputs is performed (`true` branch), otherwise another safe action with non-valid input is performed (`false` branch). Let us suppose that the value of `in` may be one of the strings set S = {`"+1"`, `"+2"`}. Note that, in any case, the program execution always lead to the execution of the `true` branch, since all the possible strings of `in` are valid, i.e., do not contain the string "$<$`script`". The abstract value of `in`, using the starting TAJS abstract domain, is NotUnsigned. Hence, the `if`-guard cannot be statically determined, since a NotUnsigned string may be contain or not the

string "$<$ `script`". Consequently, both `if` branches must be taken into account, leading to false alarms both in the `true` and `false` branches. Using the complete shell of TAJS w.r.t. `toNum` instead leads to a more precise abstract value for `in`, namely SignedOrFloat. In this case, the `if` boolean guard is satisfied (since any string approximated by SignedOrFloat does not contain the string "$<$ `script`") and no false alarms are raised during the execution of the program.

7. Relative Precision

In this section, we recall the notion of pseudo-distance on abstract domains, firstly defined in [33], which takes into account the order relation between abstract elements (together with the fact that different abstract elements might approximate the same set of concrete values) and their possible incomparability. Then we formalise the increment of the precision obtained when analysing a program with a complete abstract domain with respect to the original version of the domain.

7.1. Abstract domains precision: an overview

It is well known that abstract domains precision can be qualitatively compared by exploiting the information they are able to capture [13, 14]. However, quantitatively evaluating the precision of abstract domains has been proven to be quite challenging and frequently resulted in ad hoc measures.

Di Pierro and Wiklicky [40], introduced the notion of probabilistic Abstract Interpretation, used to numerically estimate the incompleteness of numerical abstract domains. Sotin [44] presented the notion of precision of a numerical abstract value, measuring the volume it describes, with the purpose of quantitatively comparing the precision of numerical abstract domains. Finally, Casso et al. [9] wanted to compare the precision of different analyses on logic programs. Thus, they proposed distances in two abstract domains used in constraint logic programming and they extended them to distances between the results of different analyses of a given program.

Logozzo et al. [33] gave a more general definition of pseudo-distance between abstract domain elements that allows quantifying the error of approximating a concrete element in an abstract domain.

Definition 8 (Pseudo-distance [33])**.** Let $\langle \overline{\mathsf{D}}, \sqsubseteq_{\overline{\mathsf{D}}}, \bot_{\overline{\mathsf{D}}}, \top_{\overline{\mathsf{D}}}, \sqcap_{\overline{\mathsf{D}}}, \sqcup_{\overline{\mathsf{D}}} \rangle$ be an abstract domain. A function $\delta : \overline{\mathsf{D}} \times \overline{\mathsf{D}} \to \mathbb{R} \cup \{+\infty\}$ is a *pseudo-distance* over

the abstract domain $\overline{\mathsf{D}}$ if and only if, for any $\overline{\mathsf{d}}, \overline{\mathsf{d}}', \overline{\mathsf{d}}'' \in \overline{\mathsf{D}}$, it satisfies the following axioms:

- *non-negativity*: $\delta(\overline{\mathsf{d}}, \overline{\mathsf{d}}') \geq 0$
- *if-identity*: $\overline{\mathsf{d}} =_{\overline{\mathsf{D}}} \overline{\mathsf{d}}' \Rightarrow \delta(\overline{\mathsf{d}}, \overline{\mathsf{d}}') = 0$
- *symmetry*: $\delta(\overline{\mathsf{d}}, \overline{\mathsf{d}}') = \delta(\overline{\mathsf{d}}', \overline{\mathsf{d}})$
- *weak triangle inequality*: $\overline{\mathsf{d}} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}'' \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}' \Rightarrow \delta(\overline{\mathsf{d}}, \overline{\mathsf{d}}'') \leq \delta(\overline{\mathsf{d}}, \overline{\mathsf{d}}') + \delta(\overline{\mathsf{d}}', \overline{\mathsf{d}}'')$

We recall the path length distance definition.

Definition 9 (Path length distance [33])**.** Let $\langle \overline{\mathsf{D}}, \sqsubseteq_{\overline{\mathsf{D}}}, \bot_{\overline{\mathsf{D}}}, \top_{\overline{\mathsf{D}}}, \sqcap_{\overline{\mathsf{D}}}, \sqcup_{\overline{\mathsf{D}}} \rangle$ be an abstract domain. The path length distance $\delta_{plen} : \overline{\mathsf{D}} \times \overline{\mathsf{D}} \to \mathbb{R} \cup \{+\infty\}$ is defined as

$$\delta_{plen}(\overline{\mathsf{d}}, \overline{\mathsf{d}}') = \begin{cases} plen(\overline{\mathsf{d}}, \overline{\mathsf{d}}') & \overline{\mathsf{d}} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}' \\ plen(\overline{\mathsf{d}}', \overline{\mathsf{d}}) & \overline{\mathsf{d}}' \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}} \\ +\infty & \text{otherwise} \end{cases}$$

where $plen(\overline{\mathsf{d}}, \overline{\mathsf{d}}')$ computes the distance between two abstract values $\overline{\mathsf{d}}, \overline{\mathsf{d}}' \in \overline{\mathsf{D}}$, when they are comparable, and it is defined as follows.[7]

$$plen(\overline{\mathsf{d}}, \overline{\mathsf{d}}') = \min\{\mathrm{n} \in \mathbb{N} \mid \{\overline{\mathsf{d}}_0, \overline{\mathsf{d}}_1, \ldots, \overline{\mathsf{d}}_\mathrm{n}\} \in \mathcal{P}(\overline{\mathsf{D}}), \overline{\mathsf{d}}_0 = \overline{\mathsf{d}}, \overline{\mathsf{d}}_\mathrm{n} = \overline{\mathsf{d}}', \\ \forall \mathrm{i} \in [0, \mathrm{n}). \overline{\mathsf{d}}_\mathrm{i} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}_{\mathrm{i}+1} \wedge \nexists \overline{\mathsf{d}}'' \in \overline{\mathsf{D}} \setminus \{\overline{\mathsf{d}}_\mathrm{i}, \overline{\mathsf{d}}_{\mathrm{i}+1}\}. \overline{\mathsf{d}}_\mathrm{i} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}'' \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}_{\mathrm{i}+1}\}$$

Observe that, if $\overline{\mathsf{d}} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}'$, $plen(\overline{\mathsf{d}}, \overline{\mathsf{d}}')$ corresponds to the length of the minimal path from $\overline{\mathsf{d}}$ to $\overline{\mathsf{d}}'$ in the Hasse diagram representation of $\overline{\mathsf{D}}$ w.r.t. $\sqsubseteq_{\overline{\mathsf{D}}}$. For instance, let us consider the Sign abstract domain reported in Figure 1 and suppose to compute the path length distance between $\bot_{\mathsf{Sign}}$ and $\top_{\mathsf{Sign}}$. In the abstract domain, two paths exist between $\bot_{\mathsf{Sign}}$ and $\top_{\mathsf{Sign}}$, namely $\{\bot_{\mathsf{Sign}}, \mathsf{NotPos}, \top_{\mathsf{Sign}}\}$ and $\{\bot_{\mathsf{Sign}}, \mathsf{NotNeg}, \top_{\mathsf{Sign}}\}$. Following the definition of *plen* reported above, we have to take the minimum path length of the paths set, hence $plen(\bot_{\mathsf{Sign}}, \top_{\mathsf{Sign}}) = 2$ (path length distance starts counting from 0).

[7] Notice that the original definition of *plen* in [33] contains a typo, as it refers to the partial order in a chain by disregarding its transitive closure.

```
[1]nums = "";
[2]notnums = "";
[3]i = 0;
[4]while (i < length(s)) {
   [5]if (toNum(charAt(str, i)) == 0) {
      [6]notnums = concat(notnums, charAt(s, i));[7]
   } else {
      [8]nums = concat(nums, charAt(s, i));[9]
   }
   [10]i = i + 1;[11]
}[12]
```

Figure 11: Example of μDyn annotated program

7.2. Measuring precision gained by complete shells

We aim at computing the distance and, in turn, the increment of precision, between abstract points of an abstract domain $\overline{\mathsf{D}}$ and its complete shell, w.r.t. a certain operation of interest f, noting that the abstract values of $\overline{\mathsf{D}}$ are all contained in its complete shell. In order not to clutter the notation, given an abstract domain $\overline{\mathsf{D}}$, we denote by $\mathsf{Shell}_{\mathsf{f}}(\overline{\mathsf{D}})$ its complete shell w.r.t the function f.

Let P be a μDyn program and let denote by $\mathsf{Lab}(\mathsf{P})$ the program points of P. We denote by $\mathsf{l}_\mathsf{i} \in \mathsf{Lab}(\mathsf{P})$ its i-th program point and by $\mathsf{Vars}(\mathsf{P})$ the program variables. An example of program annotated with its program points is reported in Figure 11. Given such a program, as usual in static program analysis, the goal is to compute the abstract values associated to each variable $\mathtt{x} \in \mathsf{Vars}(\mathsf{P})$ at each program point $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$. We denote by $\xi^{\overline{\mathsf{D}}} : \mathsf{Vars}(\mathsf{P}) \to \overline{\mathsf{D}}$ the abstract state associating each variable to an abstract value of $\overline{\mathsf{D}}$. When it is clear from the context, we denote the abstract state by ξ. Hence, given an abstract domain $\overline{\mathsf{D}}$ and a program P, the result of the analysis of P using the abstract domain $\overline{\mathsf{D}}$ (and the corresponding abstract semantics $[\![\cdot]\!]^{\overline{\mathsf{D}}}$) is defined by

$$\mathit{Analysis}(\mathsf{P}, \overline{\mathsf{D}}, \xi_\varnothing) = \{(\mathsf{l}_\mathsf{i}, \xi_\mathsf{i}) \mid \mathsf{l}_\mathsf{i} \in \mathsf{Lab}(\mathsf{P}), \xi_\mathsf{i} = [\![\mathsf{P}]\!]^{\overline{\mathsf{D}}}\xi_\varnothing \text{ at program point } \mathsf{l}_\mathsf{i}\}$$

where $\xi_\varnothing$ is the (initial) empty abstract state. For example, let us analyse the μDyn program with the original TAJS abstract domain reported in Figure 5b. The abstract state holding at program point 12 (namely the exit program point) is $\xi_{12} = \{\mathtt{i} \mapsto \mathsf{UnsignedInt}, \mathtt{nums} \mapsto \top_{\overline{\mathsf{TJ}}}, \mathtt{notnums} \mapsto \top_{\overline{\mathsf{TJ}}}\}$, indeed the pair $(12, \xi_{12}) \in \mathit{Analysis}(\mathsf{P}, \overline{\mathsf{TJ}}, \xi_\varnothing)$.

We can exploit the definition of path length distance reported in Definition 9 to define, for each program point $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$, the distance from $\bot_{\overline{\mathsf{D}}}$ and each abstract value associated with each $\mathtt{x} \in \mathsf{Vars}(\mathsf{P})$.

Definition 10 (Set of $\bot$-distances)**.** Let P be a μDyn program, $\overline{\mathsf{D}}$ an abstract domain and δ_{plen} the path length distance defined in Definition 9. The set of $\bot_{\overline{\mathsf{D}}}$-distances for a given $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$ is defined as follows.

$$\Phi_{\mathsf{l}}(\mathsf{P}, \overline{\mathsf{D}}) = \{(\mathtt{x}, \delta_{plen}(\bot_{\overline{\mathsf{D}}}, \xi(\mathtt{x}))) \mid (\mathsf{l}, \xi) \in \mathit{Analysis}(\mathsf{P}, \overline{\mathsf{D}}, \xi_{\varnothing}), \mathtt{x} \in \mathsf{Vars}(\mathsf{P})\}$$

For example, let us consider again ξ_{12}, previously defined, that is the abstract state holding at the program point 12 of the program reported in Figure 11 analyzing it with $\overline{\mathsf{TJ}}$ abstract domain. Hence, the set of the $\bot_{\overline{\mathsf{TJ}}}$-distances at program point 12 is $\Phi_{12}(\mathsf{P}, \overline{\mathsf{TJ}}) = \{(\mathtt{i}, 2), (\mathtt{nums}, 3), (\mathtt{notnums}, 3)\}$.

Note that, by the weak triangle inequality, given two abstract elements $\overline{\mathsf{d}}, \overline{\mathsf{d}}' \in \overline{\mathsf{D}}$, if $\overline{\mathsf{d}} \sqsubseteq_{\overline{\mathsf{D}}} \overline{\mathsf{d}}' \Rightarrow \delta_{plen}(\bot_{\overline{\mathsf{D}}}, \overline{\mathsf{d}}) \leq \delta_{plen}(\bot_{\overline{\mathsf{D}}}, \overline{\mathsf{d}}')$. Moreover, since $\bot_{\overline{\mathsf{D}}}$ is comparable with any abstract value $\mathrm{a} \in \overline{\mathsf{D}}$, $\delta(\bot_{\overline{\mathsf{D}}}, \overline{\mathsf{d}}) \neq \infty$.

Definition 11. [Precision entropy at a program point] Let P be a μDyn program and $\overline{\mathsf{D}}$ an abstract domain. Given a program point $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$, the precision entropy of $\overline{\mathsf{D}}$ for the program P at l is defined as

$$\mathbb{P}_{\mathsf{l}}(\mathsf{P}, \overline{\mathsf{D}}) = \sum_{(\mathtt{x}_{\mathrm{i}}, \mathrm{d}_{\mathrm{i}}) \in \Phi_{\mathsf{l}}(\mathsf{P}, \overline{\mathsf{D}})} \mathrm{d}_{\mathrm{i}}$$

where d_{i} is the path length distance from $\bot_{\overline{\mathsf{D}}}$ to the abstract value of $\mathtt{x}_{\mathrm{i}}$ at the program point l.

We can use Definition 11 to define the precision entropy of an analysis for a given program.

Definition 12. [Precision entropy] Let P be a μDyn program and $\overline{\mathsf{D}}$ be an abstract domain. The precision entropy $\mathbb{P}(\mathsf{P}, \overline{\mathsf{D}})$ of the abstract domain A for the program P is defined as

$$\mathbb{P}(\mathsf{P}, \overline{\mathsf{D}}) = \sum_{\mathsf{l} \in \mathsf{Lab}(\mathsf{P})} \mathbb{P}_{\mathsf{l}}(\mathsf{P}, \overline{\mathsf{D}})$$

The precision entropy $\mathbb{P}(\mathsf{P}, \overline{\mathsf{D}})$ says how much information the analysis based on the abstract domain $\overline{\mathsf{D}}$ is able to express: the more $\mathbb{P}(\mathsf{P}, \overline{\mathsf{D}})$ is low,

the more the analysis based on $\overline{\mathsf{D}}$ is precise for the program P. Using this metric, we are able to compare the analysis results based on a certain abstract domain and its corresponding complete shell. As we have already mentioned at the beginning of Section 7, any abstract point of $\overline{\mathsf{D}}$ is contained in its complete shell. Moreover, it is worth noting that, given a variable $\mathtt{x} \in \mathsf{Vars}(\mathsf{P})$ and a certain program point $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$, the abstract value computed by the analysis on $\overline{\mathsf{D}}$ associated to $\mathtt{x}$ is always comparable with the abstract value computed by the analysis on $\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})$ associated to $\mathtt{x}$. Formally, let consider $(\mathsf{l}, \xi) \in \mathit{Analysis}(\mathsf{P}, \overline{\mathsf{D}}, \xi_\varnothing)$ and $(\mathsf{l}, \xi') \in \mathit{Analysis}(\mathsf{P}, \mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}}), \xi_\varnothing)$, for some program point $\mathsf{l} \in \mathsf{Lab}(\mathsf{P})$, we have that $\forall \mathtt{x} \in \mathsf{Vars}(\mathsf{P}).\ \xi'(\mathtt{x}) \sqsubseteq_{\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})} \xi(\mathtt{x})$. This can be expressed by the following predicate

$$\mathbb{P}(\mathsf{P}, \mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})) \leq \mathbb{P}(\mathsf{P}, \overline{\mathsf{D}}).$$

Informally speaking, for a program point l, the analysis result of $\mathtt{x}$ on $\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})$ is always dominated (i.e., less than or equal to) by the analysis result of $\mathtt{x}$ on $\overline{\mathsf{D}}$ (contained in $\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})$). This fact directly comes from the dominance relation involved in the definition of complete shells given in Theorems 1 and 2. For this reason, we can always compare the analysis results produced by $\overline{\mathsf{D}}$ and $\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})$ for any variable and any program point.

This leads us to an automatic procedure to compute how much the analysis performed by $\mathsf{Shell}_\mathsf{f}(\overline{\mathsf{D}})$ is better than the one performed by $\overline{\mathsf{D}}$.

The Clam *static analyzer.* The Clam static analyzer for $\mu\mathsf{Dyn}$ programs[8] implements the TAJS and SAFE string abstract domains and their corresponding complete shells discussed in Section 5. The abstract interpreter is parametric, as it can be tuned by selecting the string abstract domain to analyze a given program. Other string abstract domains can be easily plugged into our static analyzer, without re-implementing the underlying abstract interpreter. Moreover, it is possible to check the precision entropy, discussed in Section 7, of an abstract domain and its complete shell. In this way, it is possible to check at which program point and for which variables the complete shell-based analysis gains precision w.r.t. the analysis on the original abstract domain. In the following, we will consider TAJS and its complete shell: note that, in our analyzer, both the domains specify the path length distances, as defined in Definition 9, from the bottom element to

[8]Available at `https://github.com/VincenzoArceri/clam`

Variable	$\overline{\mathsf{TJ}}$	$\overline{\mathsf{TJ}}^{\star}$	$\mathbb{P}_{12}(\mathsf{P}, \overline{\mathsf{TJ}}) - \mathbb{P}_{12}(\mathsf{P}, \overline{\mathsf{TJ}}^{\star})$
`s`	UnsignedStr	UnsignedStr	0
`i`	UnsignedInt	UnsignedInt	0
`nums`	$\top_{\overline{\mathsf{TJ}}}$	UnsignedOrNotNumeric	1
`notnums`	$\top_{\overline{\mathsf{TJ}}}$	UnsignedOrNotNumeric	1

Table 2: Analysis output results with $\overline{\mathsf{TJ}}$ and $\overline{\mathsf{TJ}}^{\star}$

each of their abstract points (that are computed off-line and *hard-coded*) by computing all the lengths of the complete chains linking the bottom element with that points and picking the minimal one.

Let us consider the μDyn program reported in Figure 11, where the value of `s` is statically unknown. The program takes the strings `s` and puts its non-zero numerical characters into `nums` and the others into `notnums`. If the variable `s` is initialized as "`24kobe8`", at the program point 12 the value of `nums` is "`248`" and the value of of `notnumbs` is "`kobe`". Let us consider TAJS and its complete shell and let analyze the program with both the abstract domains. When the variable `s` is initialized with the abstract value UnsignedStr, the analysis output, for the exit point is reported in Table 2, where the second and third columns correspond to the result analysis of $\overline{\mathsf{TJ}}$ and $\overline{\mathsf{TJ}}^{\star}$ for the corresponding variable, respectively, and the last column is the increment precision gained by performing the analysis on $\overline{\mathsf{TJ}}^{\star}$. Observe that, for the variables `s` and `i`, we have no precision improvement, as stated by the last column of Table 2. Concerning variables `nums` and `notnumbs`, the analysis on $\overline{\mathsf{TJ}}$ returns the top values, whereas the analysis on $\overline{\mathsf{TJ}}^{\star}$ leads to a precision increment since it returns the abstract value UnsignedOrNotNumeric, for both variables. The precision increment is stated by the last column of Table 2 for the variables `nums` and `notnumbs`, since their abstract values (UnsignedOrNotNumeric) on the analysis on $\overline{\mathsf{TJ}}^{\star}$ is distant 1 from the abstract values of the analysis on $\overline{\mathsf{TJ}}$ ($\top_{\overline{\mathsf{TJ}}}$). The total precision entropy of the analysis on $\overline{\mathsf{TJ}}$ and the analysis on $\overline{\mathsf{TJ}}^{\star}$ is 54 and 44, respectively, so the overall precision increment when considering the complete shell, in this case, is equal to 10.

8. Conclusion

This paper focused on backward completeness in JavaScript-specific string abstract domains, and provided in particular the complete shells of TAJS and SAFE string abstract domains w.r.t. `toNum` and `concat` operations, together with an effective procedure to measure the precision improvement of the analysis when moving to the complete shell. Our results can be easily applied also to JSAI string abstract domain [28], as it can be seen as an extension of the SAFE domain. The next challenge is to investigate forward completeness [24], the property that guarantees that no loss of precision occurs during the output abstraction process of a given operation. We aim to integrate the two completeness methodologies within an industrial JavaScript static analyzer to deeply evaluate their actual cost and overall impact.

Acknowledgment. Special thanks to the very professional reviewers, that contributed to improve the technical quality of the submitted manuscript.

References

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, J. Stenman, Norn: An SMT Solver for String Constraints, in: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, 462–469, URL `https://doi.org/10.1007/978-3-319-21690-4_29`, 2015.

[2] R. Amadini, G. Gange, F. Gauthier, A. Jordan, P. Schachte, H. Søndergaard, P. J. Stuckey, C. Zhang, Reference Abstract Domains and Applications to String Analysis, Fundam. Inform. 158 (4) (2018) 297–326, URL `https://doi.org/10.3233/FI-2018-1650`.

[3] R. Amadini, G. Gange, P. J. Stuckey, G. Tack, A Novel Approach to String Constraint Solving, in: Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings, 3–20, URL `https://doi.org/10.1007/978-3-319-66158-2_1`, 2017.

[4] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, C. Zhang, Combining String Abstract

Domains for JavaScript Analysis: An Evaluation, in: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, 41–57, URL https://doi.org/10.1007/978-3-662-54577-5_3, 2017.

[5] V. Arceri, S. Maffeis, Abstract Domains for Type Juggling, Electron. Notes Theor. Comput. Sci. 331 (2017) 41–55, URL https://doi.org/10.1016/j.entcs.2017.02.003.

[6] V. Arceri, I. Mastroeni, Static Program Analysis for String Manipulation Languages, in: Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019, 19–33, URL https://doi.org/10.4204/EPTCS.299.5, 2019.

[7] V. Arceri, M. Olliaro, A. Cortesi, I. Mastroeni, Completeness of Abstract Domains for String Analysis of JavaScript Programs, in: Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings, 255–272, URL https://doi.org/10.1007/978-3-030-32505-3_15, 2019.

[8] T. Bultan, F. Yu, M. Alkhalaf, A. Aydin, String Analysis for Software Verification and Security, Springer, ISBN 978-3-319-68668-4, URL https://doi.org/10.1007/978-3-319-68670-7, 2017.

[9] I. Casso, J. F. Morales, P. López-García, R. Giacobazzi, M. V. Hermenegildo, Computing Abstract Distances in Logic Programs, in: M. Gabbrielli (Ed.), Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers, vol. 12042 of *Lecture Notes in Computer Science*, Springer, 57–72, URL https://doi.org/10.1007/978-3-030-45260-5_4, 2019.

[10] L. Chen, A. Miné, P. Cousot, A Sound Floating-Point Polyhedra Abstract Domain, in: Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings, 3–18, URL https://doi.org/10.1007/978-3-540-89330-1_2, 2008.

[11] A. S. Christensen, A. Møller, M. I. Schwartzbach, Precise Analysis of String Expressions, in: Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings, 1–18, URL https://doi.org/10.1007/3-540-44898-5_1, 2003.

[12] R. Clarisó, J. Cortadella, The octahedron abstract domain, Sci. Comput. Program. 64 (1) (2007) 115–139, URL https://doi.org/10.1016/j.scico.2006.03.009.

[13] A. Cortesi, G. Filé, W. H. Winsborough, Comparison of Abstract Interpretations, in: Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings, 521–532, URL https://doi.org/10.1007/3-540-55719-9_101, 1992.

[14] A. Cortesi, G. Filé, W. H. Winsborough, The Quotient of an Abstract Interpretation, Theor. Comput. Sci. 202 (1-2) (1998) 163–192, URL https://doi.org/10.1016/S0304-3975(97)00137-0.

[15] A. Cortesi, M. Olliaro, M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs, in: 2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018, 1–8, URL https://doi.org/10.1109/TASE.2018.00009, 2018.

[16] G. Costantini, P. Ferrara, A. Cortesi, A suite of abstract domains for static analysis of string values, Softw., Pract. Exper. 45 (2) (2015) 245–287, URL https://doi.org/10.1002/spe.2218.

[17] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, 238–252, URL https://doi.org/10.1145/512950.512973, 1977.

[18] P. Cousot, R. Cousot, Systematic Design of Program Analysis Frameworks, in: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, 269–282, URL https://doi.org/10.1145/567752.567778, 1979.

[19] P. Cousot, N. Halbwachs, Automatic Discovery of Linear Restraints Among Variables of a Program, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, 84–96, URL https://doi.org/10.1145/512760.512770, 1978.

[20] D. Filaretti, S. Maffeis, An Executable Formal Semantics of PHP, in: ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings, 567–592, URL https://doi.org/10.1007/978-3-662-44202-9_23, 2014.

[21] R. Giacobazzi, I. Mastroeni, Transforming Abstract Interpretations by Abstract Interpretation, in: M. Alpuente, G. Vidal (Eds.), Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, vol. 5079 of *Lecture Notes in Computer Science*, Springer, 1–17, URL https://doi.org/10.1007/978-3-540-69166-2_1, 2008.

[22] R. Giacobazzi, I. Mastroeni, Making abstract models complete, Math. Struct. Comput. Sci. 26 (4) (2016) 658–701, URL https://doi.org/10.1017/S0960129514000358.

[23] R. Giacobazzi, E. Quintarelli, Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking, in: P. Cousot (Ed.), Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings, vol. 2126 of *Lecture Notes in Computer Science*, Springer, 356–373, URL https://doi.org/10.1007/3-540-47764-0_20, 2001.

[24] R. Giacobazzi, F. Ranzato, F. Scozzari, Making abstract interpretations complete, J. ACM 47 (2) (2000) 361–416, URL https://doi.org/10.1145/333979.333989.

[25] P. Granger, Static Analysis of Arithmetical Congruences, International Journal of Computer Mathematics - IJCM 30 (1989) 165–190.

[26] P. Granger, Static Analysis of Linear Congruence Equalities among Variables of a Program, in: TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium

on Trees in Algebra and Programming (CAAP'91), 169–192, URL https://doi.org/10.1007/3-540-53982-4_10, 1991.

[27] S. H. Jensen, A. Møller, P. Thiemann, Type Analysis for JavaScript, in: Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings, 238–255, URL https://doi.org/10.1007/978-3-642-03237-0_17, 2009.

[28] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, B. Hardekopf, JSAI: a static analysis platform for JavaScript, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, 121–132, URL https://doi.org/10.1145/2635868.2635904, 2014.

[29] S. Kim, W. Chin, J. Park, J. Kim, S. Ryu, Inferring Grammatical Summaries of String Values, in: Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings, 372–391, URL https://doi.org/10.1007/978-3-319-12736-1_20, 2014.

[30] E. Kneuss, P. Suter, V. Kuncak, Phantm: PHP analyzer for type mismatch, in: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, 373–374, URL https://doi.org/10.1145/1882291.1882355, 2010.

[31] H. Lee, S. Won, J. Jin, J. Cho, S. Ryu, SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript, in: FOOL'12, URL https://doi.org/10.1.1.387.1030, 2012.

[32] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. W. Barrett, M. Deters, An efficient SMT solver for string constraints, in: Formal Methods in System Design 48 (3) (2016) 206–234, URL https://doi.org/10.1007/s10703-016-0247-6.

[33] F. Logozzo, Towards a Quantitative Estimation of Abstract Interpretations, in: Workshop on Quantitative Analysis of Software, Microsoft, URL https://www.microsoft.com/en-us/research/publication/

towards-a-quantitative-estimation-of-abstract-interpretations/, 2009.

[34] M. Madsen, E. Andreasen, String Analysis for Dynamic Field Access, in: Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, 197–217, URL https://doi.org/10.1007/978-3-642-54807-9_12, 2014.

[35] S. Maffeis, J. C. Mitchell, A. Taly, An Operational Semantics for JavaScript, in: Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings, 307–325, URL https://doi.org/10.1007/978-3-540-89330-1_22, 2008.

[36] Y. Minamide, Static approximation of dynamically generated Web pages, in: Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005, 432–441, URL https://doi.org/10.1145/1060745.1060809, 2005.

[37] A. Miné, The octagon abstract domain, in: High. Order Symb. Comput. 19 (1) (2006) 31–100, URL https://doi.org/10.1007/s10990-006-8609-1.

[38] R. Oucheikh, I. Berrada, O. E. Hichami, The 4-Octahedron Abstract Domain, in: Networked Systems - 4th International Conference, NETYS 2016, Marrakech, Morocco, May 18-20, 2016, Revised Selected Papers, 311–317, URL https://doi.org/10.1007/978-3-319-46140-3_25, 2016.

[39] C. Park, H. Im, S. Ryu, Precise and scalable static analysis of jQuery using a regular expression domain, in: Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016, 25–36, URL https://doi.org/10.1145/2989225.2989228, 2016.

[40] A. D. Pierro, H. Wiklicky, Measuring the Precision of Abstract Interpretations, in: Logic Based Program Synthesis and Transformation,

10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers, 147–164, URL https://doi.org/10.1007/3-540-45142-0_9, 2000.

[41] M. D. Preda, R. Giacobazzi, I. Mastroeni, Completeness in Approximate Transduction, in: X. Rival (Ed.), Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, vol. 9837 of *Lecture Notes in Computer Science*, Springer, 126–146, URL https://doi.org/10.1007/978-3-662-53413-7_7, 2016.

[42] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A Symbolic Execution Framework for JavaScript, in: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA, 513–528, URL https://doi.org/10.1109/SP.2010.38, 2010.

[43] A. Simon, A. King, J. M. Howe, Two Variables per Linear Inequality as an Abstract Domain, in: Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers, 71–89, URL https://doi.org/10.1007/3-540-45013-0_7, 2002.

[44] P. Sotin, Quantifying the Precision of Numerical Abstract Domains, Research Report, URL https://hal.inria.fr/inria-00457324, 2010.

[45] M. Veanes, P. de Halleux, N. Tillmann, Rex: Symbolic Regular Expression Explorer, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, 498–507, URL https://doi.org/10.1109/ICST.2010.15, 2010.

[46] M. Ward, The Closure Operators of a Lattice, Annals of Mathematics 43 (2) (1942) 191–196.

[47] G. Wassermann, Z. Su, Sound and precise analysis of web applications for injection vulnerabilities, in: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, 32–41, URL https://doi.org/10.1145/1250734.1250739, 2007.

[48] F. Yu, M. Alkhalaf, T. Bultan, O. H. Ibarra, Automata-based symbolic string analysis for vulnerability detection, Formal Methods in

System Design 44 (1) (2014) 44–70, URL https://doi.org/10.1007/s10703-013-0189-1.

[49] F. Yu, T. Bultan, M. Cova, O. H. Ibarra, Symbolic String Verification: An Automata-Based Approach, in: Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings, 306–324, URL https://doi.org/10.1007/978-3-540-85114-1_21, 2008.

[50] F. Yu, T. Bultan, B. Hardekopf, String Abstractions for String Verification, in: Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings, 20–37, URL https://doi.org/10.1007/978-3-642-22306-8_3, 2011.