Completeness of Abstract Domains for String Analysis of JavaScript Programs

Vincenzo Arceri[1], Martina Olliaro[2,3], Agostino Cortesi[2], and Isabella Mastroeni[1]

[1] University of Verona, Italy
{ vincenzo.arceri | isabella.mastroeni }@univr.it
[2] Ca' Foscari University of Venice, Italy
{ martina.olliaro | cortesi }@unive.it
[3] Masaryk University of Brno, Czech Republic

Abstract. Completeness in abstract interpretation is a well-known property, which ensures that the abstract framework does not lose information during the abstraction process, with respect to the property of interest. Completeness has been never taken into account for existing string abstract domains, due to the fact that it is difficult to prove it formally. However, the effort is fully justified when dealing with string analysis, which is a key issue to guarantee security properties in many software systems, in particular for JavaScript programs where poorly managed string manipulating code often leads to significant security flaws. In this paper, we address completeness for the main JavaScript-specific string abstract domains, we provide suitable refinements of them, and we discuss the benefits of guaranteeing completeness in the context of abstract-interpretation based string analysis of dynamic languages.

Keywords: String Abstract Domains, Abstract Interpretation Completeness, String Analysis

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 the web-based software development (e.g., JavaScript), offer a wide range of dynamic features that make string manipulation challenging.

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, as others non-trivial analyses in the programming languages field, is an undecidable task. Thus, a certain degree of approximation is necessary in order 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 [7]: automata-based [6, 9, 37, 38], abstraction-based [2–4, 11, 12, 39], constraint-based

[1,25,32,34], and grammar-based [28,36], and used, inter alia, with the purpose of detecting web application vulnerabilities [36–38].

In this paper we focus on string analysis by means of the abstract interpretation theory [13,14]. Abstract interpretation has been proposed by P. Cousot and R. Cousot 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 used to rigorous mathematical proofs of approximations correctness. Since the introduction of the abstract interpretation theory, many abstract domains that represent properties of interest about numerical domains values have been designed [8,10,13,15,18,19,29,30,33]. On the other hand, just in the last few years, scientific community has taken an interest in the development of abstract domains for string analysis [2,4,11,12,22,26,31], some of them language specific, such as those defined as part of the JavaScript static analysers: TAJS [20], SAFE [24], and JSAI [21].

Desirable features of abstract interpretation are *soundness* and *completeness* [14]. If soundness (or correctness), as a basic requirement, actually is often guaranteed by static analysis tools, 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 abstract interpretation can be considered optimal. In [17], authors 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

Due to the important role played by JavaScript in the current landscape, its extensive use of strings, and the difficulties in statically analyse it, we believe that an improvement in the accuracy of JavaScript-specific string abstract domains can lead to a preciser reasoning about strings.

Thus, in this paper, we study the completeness property, with respect to some string operations of interest, of two JavaScript-specific string abstract domains, i.e., those defined as part of SAFE [24] and TAJS [20] static analysers. Finally, we define their complete versions, and we discuss the benefits of guaranteeing completeness in the context of abstract interpretation based string analysis of dynamic languages.

1.2 Paper structure

Section 2 gives basics in mathematics and abstract interpretation. Section 3 presents important concepts related to the completeness property in abstract interpretation [17], that we will use through the whole paper. Moreover, a motivating example is given to show the importance to guarantee 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 domain integrated into SAFE [24] and TAJS [20] static analysers with respect to two operations of interest. Section 6 highlights the strengths and usefulness of the completeness approach to abstract-based static analysis of JavaScript string manipulating programs. Section 7 concludes and points out interesting aspects for future works.

2 Background

Mathematical notation. Given a set S, we denote by S^* the set of all the finite sequences of elements of S and by S^n the set of all finite sequences of S of length n. 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 $\wp(S)$ the powerset of S, with $S \setminus T$ the set difference, with $S \subset T$ the strict inclusion relation, and with $S \subseteq T$ the inclusion relation between S and T. A set L with ordering relation $\leq$ is a poset and it is denoted by $\langle L, \leq \rangle$. A poset $\langle L, \leq \rangle$ is a lattice if $\forall x, y \in L$ we have that $x \vee y$ and $x \wedge y$ belong to L, and we say that it is also complete when for each $X \subseteq L$ we have that $\bigvee X, \bigwedge X \in L$. Given a poset $\langle L, \leq \rangle$ and $S \subseteq L$, we denote by $\max(S) = \{x \in S \mid \forall y \in S.\, x \leq y \Rightarrow x = y\}$ the set of the maximal elements of S in L. As usual, a complete lattice L, with ordering $\leq$, least upper bound (lub) $\vee$, greatest lower bound (glb) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$ is denoted by $\langle L, \leq, \vee, \wedge, \top, \bot \rangle$. An *upper closure operator* on a poset $\langle L, \leq \rangle$ is an operator $\rho : L \to L$ which is monotone, idempotent, and extensive (i.e., $x \leq \rho(x)$) and it can be uniquely identified by the set of its fix-points. The set of all closure operators on a poset L is denoted by $uco(L)$. Given $f : S \to T$ and $g : T \to Q$ we denote with $g \circ f : S \to Q$ their composition, i.e., $g \circ f = \lambda x.g(f(x))$. Given $f : S^n \to T$, $s \in S^n$ and $i \in [0, n)$, we denote by $f^i_s = \lambda z.f(s[z/i]) : S \to T$ a generic i-th unary restriction of f.

Abstract interpretation. Abstract interpretation [13, 14] 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, which is a complete lattice C [14]. Let C and A be complete lattices, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a *Galois Connection* (GC) between C and A if for every $x \in C$ and for every $y \in A$ we have $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. The function α (resp. γ) is the *left-adjoint* (resp. *right-adjoint*) to γ (resp. α), and it is additive (resp. co-additive). If $\langle \alpha, \gamma \rangle$ is a GC between C and A then $\gamma \circ \alpha \in uco(C)$. If C is a complete lattice, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.C, \mathtt{id} \rangle$ forms a complete lattice [35], which is the set of all possible abstractions of C, where the bottom element is $\mathtt{id} = \lambda x.x$, and for every $\rho, \eta \in uco(C)$, ρ is *more concrete than* η iff $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$, $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$; $(\sqcup_{i \in I} \rho_i)(x) = x$ iff $\forall i \in I.\ \rho_i(x) = x$. The operator $\rho \in uco(C)$ is disjunctive when $\rho(C)$ is a join-sublattice of C which holds iff ρ is additive [14]. Let L be a complete lattice, then $X \subseteq L$ is a Moore family of L if $X = \mathcal{M}(X) = \{\wedge S \mid S \subseteq X\}$, where $\wedge \varnothing = \top$. The condition that any concrete element of C has the best abstraction in the abstract domain A, implies that A is a Moore family of C. We denote by $\mathcal{M}(X)$ the Moore closure of $X \subseteq C$, that is the least subset of C, which is a Moore family of C, and contains X. If $\langle \alpha, \gamma \rangle$ is a GC between C and A and $f : C \to C$ a concrete function, then $f^\sharp = \alpha \circ f \circ \gamma : A \to A$ is the best correct approximation of f in A. Let $\langle \alpha, \gamma \rangle$ be a GC between C and A, $f : C \to C$ be a concrete function and $f^\sharp : A \to A$ be an abstract function. The function $f^\sharp$ is a *sound approximation* of f if $\forall c \in C.\ \alpha(f(c)) \leq_A f^\sharp(\alpha(c))$. In abstract interpretation, there exist two notions of completeness. *Backward completeness* property focuses on complete abstractions of the inputs, while *forward completeness* focuses on complete abstractions of the outputs, both w.r.t. an operation of interest. In this paper, we focus on the more typical notion of completeness, i.e., backward completeness. Hence, when we will talk about completeness, we mean backward completeness. Given a GC $\langle \alpha, \gamma \rangle$ between C and A, a concrete function $f : C \to C$, and an abstract function $f^\sharp : A \to A$, then $f^\sharp$ is *backward complete* for f (for short complete) if $\forall c \in C.\ \alpha(f(c)) = f^\sharp(\alpha(c))$. If the backward completeness property is guaranteed, no loss of information arises during the input abstraction process, w.r.t. an operation of interest.

3 Making abstract interpretations complete

In this section, we give the notions and methodologies that we will use through the whole paper (and proposed in [17]), in order to constructively build, from an initial abstract domain, a novel 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 [17], it is worth noting that completeness is a property related to the underlying abstract domain. Starting from this fact, in [17], 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 A and B and an operator $f : A \to B$, 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 A or the other that *removes* the minimal number of abstract points from the output abstract domain B. The first approach captures the notion of *complete shell of* A, while the latter defines the *complete core of* B, 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, and so, even if it is complete for f, the complete core could worsen the precision of other operations.

On the other hand, complete shells augment the starting abstract domains (adding abstract points), and consequently it can not compromise the precision of other operations.

Below, we provide two important theorems proved in [17] that give 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.

Theorem 1 (*Complete shell of ρ relative to η). Let C and D be two posets and $f : C^n \to D$ be a continuous function. Given $\rho \in uco(C)$, then $\mathcal{S}^\rho_f : uco(D) \to uco(C)$ is the following domain transformer:*

$$\mathcal{S}^\rho_f(\eta) = \mathcal{M}(\rho \cup (\bigcup_{\substack{i\in[0,n)\\ x\in C^n, y\in\eta}} \max(\{z \in C \mid (f^i_x)(z) \leq_D y\})))$$

and it computes the complete shell of ρ relative to η.

As already mentioned above, the idea under 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 operator f. From Theorem 1, this is obtained adding to ρ the maximal elements in C, whose f image is dominated by elements in η, at least in one dimension i. Clearly, the so-obtained abstraction may be not an upper closure operator for C. Hence, 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., $f : C^n \to C$. In this case, given $\rho \in uco(C)$, absolute complete shells of ρ can be obtained as the greatest fix-point (*gfp*) of the domain transformer $\mathcal{S}^\rho_f$ (see Theorem 1), as stated by the following theorem.

Theorem 2 (*Absolute complete shell of ρ). Let C be a poset and $f : C^n \to C$ be a continuous function. Given $\rho \in uco(C)$, then $\overline{\mathcal{S}}^\rho_f : uco(C) \to uco(C)$ is the following domain transformer:*

$$\overline{\mathcal{S}}^\rho_f = gfp(\lambda\eta.\mathcal{S}^\rho_f(\eta))$$

and it computes the absolute complete shell of ρ.

The completeness property for the sign abstract domain, which approximates numerical values, has been discussed in [17]. The sign abstract domain is complete for the product operation, but it is not complete w.r.t. the sum. Indeed, the sign of $\mathsf{e}_1 + \mathsf{e}_2$ cannot be defined by simply knowing the sign of e_1 and e_2. In [17], authors computed the absolute complete shell of the sign domain w.r.t. the sum operation, and they showed it corresponds to the interval abstract domain [13].

3.1 Motivating example

A common feature of dynamic languages, such as PHP or JavaScript, is to be not typed. Hence, in those languages, it is allowed to change the variable type

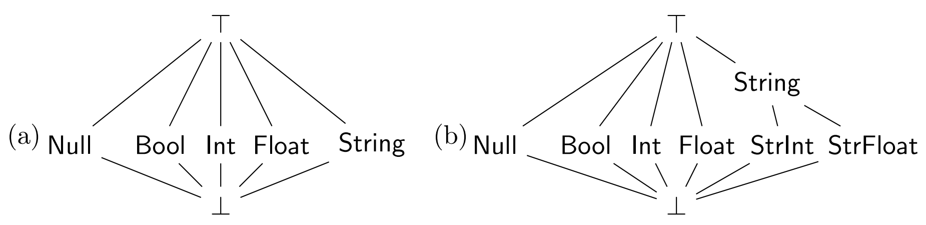


Fig. 1: (a) *Coalesced sum* abstract domain for PHP. (b) Complete shell of *coalesced sum* abstract domain w.r.t. the sum operation.

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 to static reasoning about variable types was to track the latter adopting the so-called *coalesced sum* abstract domain [5, 23], in order to detect whether a certain variable has constant type through the whole program execution. In Fig. 1a, we report the coalesced sum abstract domain for an intra-procedural version of PHP [5], that tracks null, boolean, integer, float and string types[4]. Consider the formal semantics of the sum operation in PHP [16]. When one of the operand 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 `4.4`. Let $+^\sharp$ be the abstract sum operation on the coalesced sum abstract domain. The type of the expression e is given by:

$$\alpha(\{\texttt{"2.4hello"}\}) +^\sharp \alpha(\{\texttt{"4"}\}) = \mathsf{String} +^\sharp \mathsf{String} = \top$$

The static type analysis based on the coalesced sum abstract domain returns $\top$ (i.e., any possible value), since the sum between two strings may 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: $\alpha(\sigma + \sigma') = \alpha(\sigma) +^\sharp \alpha(\sigma')$, e.g., $\alpha(\sigma + \sigma') = \mathsf{Float} \neq \alpha(\sigma) +^\sharp \alpha(\sigma') = \top$. 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, since the domain is not precise enough to distinguish between strings that may be implicitly converted to integers or floats.

Fig. 1b shows the complete shell of the coalesced sum abstract domain w.r.t. the sum. The latter adds two abstract values to the original domain, namely StrFloat and StrInt, that correspond to the abstractions of the strings that may be

[4] Closing the coalesced sum abstract domain by the powerset operation, a more precise abstract domain is obtained, called *union type* abstract domain [23], that tracks the set of types of a certain variable during program execution.

$$\begin{array}{l}
\mathsf{a} ::= n \in \textsc{Int} \cup \textsc{Float} \mid \texttt{a + a} \mid \\
\mid \texttt{a - a} \mid \texttt{a * a} \mid \texttt{a / a} \\
\mid \texttt{toNum(s)} \\
\mathsf{b} ::= \texttt{true} \mid \texttt{false} \mid \texttt{b \&\& b} \\
\mid \texttt{b || b} \mid \texttt{! b} \\
\mathsf{s} ::= \texttt{"s"} \\
\mid \texttt{concat(}\mathsf{s}_1\texttt{,}\mathsf{s}_2\texttt{)} \\
\mathsf{e} ::= x \mid \mathsf{a} \mid \mathsf{b} \mid \mathsf{s} \\
\mathsf{bl} ::= \texttt{\{ \}} \mid \texttt{\{ S \}} \\
\mathsf{S} ::= x\ \texttt{= e;} \mid \texttt{;} \mid \mathsf{bl} \\
\mid \texttt{if (b) } \mathsf{bl}_1 \texttt{ else } \mathsf{bl}_2 \\
\mid \texttt{while (b) } \mathsf{bl} \\
\mid \mathsf{S}_1\ \mathsf{S}_2
\end{array}$$

where $x \in \textsc{Id}$, $\mathsf{c} \in \Sigma$, $\mathsf{s} \in \Sigma^*$

Fig. 2: μDyn syntax.

$$\begin{array}{rl}
[\![x = \mathsf{e};]\!]\xi &= \xi[x \leftarrow [\![\mathsf{e}]\!]\xi] \\
[\![\texttt{if (b) } \mathsf{bl}_1 \texttt{ else } \mathsf{bl}_2]\!]\xi &= \begin{cases} [\![\mathsf{bl}_1]\!]\xi & [\![\mathsf{b}]\!]\xi = \texttt{true} \\ [\![\mathsf{bl}_2]\!]\xi & [\![\mathsf{b}]\!]\xi = \texttt{false} \end{cases} \\
[\![\texttt{while (b)bl}]\!]\xi &= [\![\texttt{if (b) \{bl while (b)bl\} else \{\}}]\!]\xi \\
[\![\{\}]\!]\xi = [\![;]\!]\xi &= \xi \qquad [\![\{\mathsf{S}\}]\!]\xi = [\![\mathsf{S}]\!]\xi \\
[\![\mathsf{S}_1\mathsf{S}_2]\!]\xi &= [\![\mathsf{S}_2]\!]([\![\mathsf{S}_1]\!]\xi) \\
\\
[\![\texttt{concat}(\mathsf{s},\mathsf{s}')]\!]\xi &= [\![\mathsf{s}]\!]\xi \cdot [\![\mathsf{s}']\!]\xi \\
[\![\texttt{toNum(s)}]\!]\xi &= \begin{cases} \mathcal{N}([\![\mathsf{s}]\!]\xi) & [\![\mathsf{s}]\!]\xi \in \Sigma^*_{\mathsf{Num}} \\ 0 & \text{otherwise} \end{cases}
\end{array}$$

Fig. 3: μDyn semantics.

implicitly converted to floats and to integers, respectively. Notice that, the type analysis on the novel abstract domain is now complete w.r.t. the sum operation. Indeed, the completeness condition also holds for the expression e, as shown below.

$$\begin{aligned}
\alpha(\{\texttt{"2.4hello" + "4"}\}) &= \mathsf{Float} \\
&= \alpha(\{\texttt{"2.4hello"}\}) +^\sharp \alpha(\{\texttt{"4"}\}) \\
&= \mathsf{StrFloat} +^\sharp \mathsf{StrInt} \\
&= \mathsf{Float}
\end{aligned}$$

As pointed out above, guaranteeing completeness in abstract interpretation is a precious and 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 a tricky semantics, such as in our example or, more in general, in dynamic languages operations. For this reason, complete shells become important since they are able to mathematically guarantee completeness for a certain operation, starting from an abstract domain of interest.

4 Core language

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, and inspired by the JavaScript programming language [27]. μDyn syntax is reported in Fig. 2. The μDyn basic values are represented by the set $\textsc{Val} = \textsc{Int} \cup \textsc{Float} \cup \textsc{Bool} \cup \textsc{Str}$, such that:

- $\textsc{Int} = \mathbb{Z}$ denotes the set of signed integers

- $\textsc{Float}$ denotes the set of signed decimal numbers[5]
- $\textsc{Bool} = \{\texttt{true}, \texttt{false}\}$ denotes the set of booleans
- $\textsc{Str} = \Sigma^*$ denotes the set of strings over an alphabet Σ

  We consider Σ^* composed of two sets, namely $\Sigma^* = \Sigma^*_{\mathsf{Num}} \cup \Sigma^*_{\mathsf{NotNum}}$, 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:

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

  which correspond to the set of unsigned integer strings, unsigned float strings, signed integer strings and signed float strings, respectively.

$\mu\mathsf{Dyn}$ programs are elements generated by S syntax rules. Program states $\textsc{State} : \textsc{Id} \rightarrow \textsc{Val}$, ranging over ξ, are partial functions from identifiers to values. The concrete semantics of $\mu\mathsf{Dyn}$ statements follows [5], and it is given by the function $[\![\cdot]\!]\cdot : \textsc{Stmt} \times \textsc{State} \rightarrow \textsc{State}$, inductively defined on the structure of the statements, as reported in Fig. 3. We abuse notation in defining the concrete semantics of expressions: $[\![\cdot]\!]\cdot : \textsc{Exp} \times \textsc{State} \rightarrow \textsc{Val}$. Fig. 3 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. We denote by $\mathcal{N}(\sigma) \in \textsc{Int} \cup \textsc{Float}$ the numeric value of a given string. For example, $\texttt{toNum("4.2")} = 4.2$ and $\texttt{toNum("asd")} = 0$.

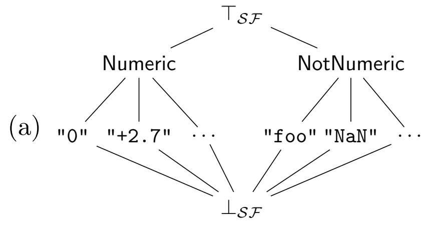
5 Making JavaScript string abstract domains complete

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, that are SAFE [24] and TAJS [20]. Both the abstract domains track important information on JavaScript strings, e.g., SAFE tracks numeric strings, such as `"2.5"` or `"+5"`, and TAJS is able to infer when a string corresponds to an unsigned integer, that may be used as array index.

For the sake of readability, we recast the original string abstract domains for $\mu\mathsf{Dyn}$, following the notation adopted in [4]. Fig. 4 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 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 D, we denote by $\alpha_D : \wp(\Sigma^*) \rightarrow D$ its abstraction function, by $\gamma_D : D \rightarrow \wp(\Sigma^*)$ its concretization function, and by $\rho_D : \wp(\Sigma^*) \rightarrow \wp(\Sigma^*) \in uco(D)$ the associated upper closure operator.

[5] Floats normally are represented in programming languages in the IEEE 754 double precision format. For the sake of simplicity, we use instead decimal numbers.



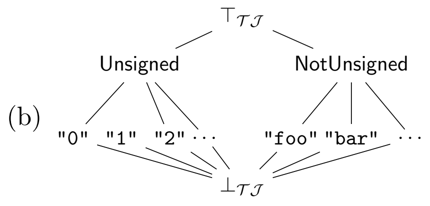


Fig. 4: (a) SAFE, (b) TAJS string abstract domains recasted for μDyn.

5.1 Completing SAFE string abstract domain

Fig. 4a depicts the string abstract domain $\mathcal{SF}$, i.e., the recasted version of the domain involved into SAFE [24] static analyser. It splits strings into the abstract values: Numeric (i.e., numerical strings) and NotNumeric (i.e., all the other strings). Before reaching these abstract values, $\mathcal{SF}$ precisely tracks each string. For instance, $\alpha_{\mathcal{SF}}(\{$"+9.6", "7"$\})=$ Numeric, and $\alpha_{\mathcal{SF}}(\{$"+9.6", "bar"$\}) = \top_{\mathcal{SF}}$.

We study the completeness of $\mathcal{SF}$ w.r.t. `concat` operation. Fig. 5 presents the abstract semantics of the concatenation operation for $\mathcal{SF}$, that is:

$$[\![\texttt{concat}(\bullet, \bullet)]\!]^{\mathcal{SF}} : \mathcal{SF} \times \mathcal{SF} \to \mathcal{SF}$$

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

Lemma 1. *$\mathcal{SF}$ is not complete w.r.t.* `concat`. *In particular*[6], $\forall S_1, S_2 \in \wp(\Sigma^*)$ *we have that:*

$$\alpha_{\mathcal{SF}}([\![\texttt{concat}(S_1, S_2)]\!]) \subsetneq [\![\texttt{concat}(\alpha_{\mathcal{SF}}(S_1), \alpha_{\mathcal{SF}}(S_2))]\!]^{\mathcal{SF}}$$

Consider $S_1 = \{$"2.2", "2.3"$\}$ and $S_2 = \{$"2", "3"$\}$. The completeness property does not hold:

$$\alpha_{\mathcal{SF}}([\![\texttt{concat}(S_1, S_2)]\!]) = \textsf{Numeric} \neq \top_{\mathcal{SF}} = [\![\texttt{concat}(\alpha_{\mathcal{SF}}(S_1), \alpha_{\mathcal{SF}}(S_2))]\!]^{\mathcal{SF}}$$

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

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

Fig. 5: SAFE `concat` abstract semantics.

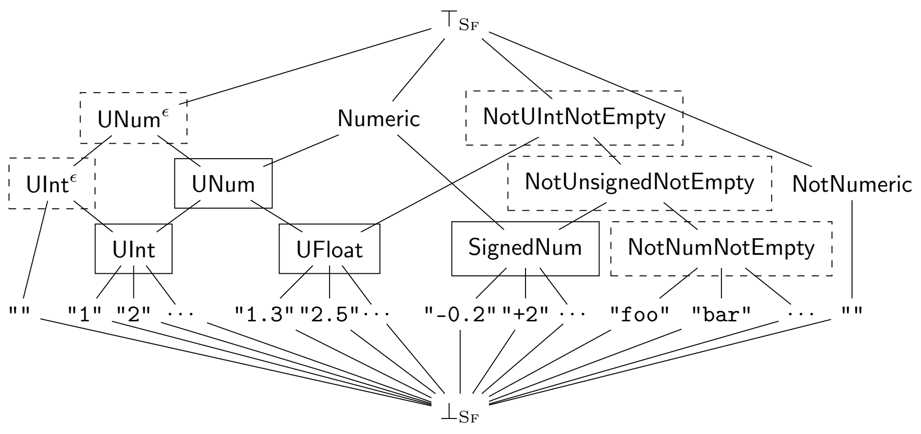


Fig. 6: Absolute complete shell of $\rho_{\mathcal{SF}}$ w.r.t. `concat`.

The $\mathcal{SF}$ abstract domain loses too much information during the abstraction process; information that can not be retrieved during the abstract concatenation. Intuitively, to gain completeness w.r.t. `concat` operation, $\mathcal{SF}$ should improve the precision of the numerical strings abstraction, e.g., discriminating between float and integer strings. Following Theorem 2, we can formally construct the absolute complete shell of $\rho_{\mathcal{SF}}$ w.r.t. `concat` operation $\overline{\mathcal{S}}^{\rho_{\mathcal{SF}}}_{\texttt{concat}}$, and we denote it by Sf. This leads to a novel abstract domain, given in Fig. 6, that is complete for `concat`.

In particular, the points inside dashed boxes are the abstract values added during the iterative computations of Sf, the points inside standard boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in $\mathcal{SF}$. The meaning of abstract values in Sf is intuitive. In order to satisfy the completeness property, Sf splits the Numeric abstract value, already taken into account in $\mathcal{SF}$, into all the strings corresponding to unsigned integer (UInt), unsigned floats (UFloat), and signed numbers ($\mathsf{SignedNum}$). Moreover, particular importance is given to the empty string, since the novel abstract domain specifies whether each abstract value contains `""`. Indeed, the UInt^{ϵ} abstract value represents the strings corresponding to unsigned integer or to the empty string, and the UNum^{ϵ} abstract value represents the strings corresponding to unsigned numbers or to the empty string. An unexpected abstract value considered in $\mathrm{S}\textsc{f}$ is $\mathsf{NotUnsignedNotEmpty}$, such that:

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

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 $\mathrm{S}\textsc{f}$ following the computation of the formula below:

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

Informally speaking, we are wondering the following question: *which is the maximal set of strings s.t. concatenated to any possible numerical string will produce any possible non-numerical string?* Indeed, in order 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 $\mathsf{NotUnsignedNotEmpty}$.

Theorem 3. *ρ_{SF} is the absolute complete shell of $\rho_{\mathcal{SF}}$ w.r.t. `concat` operation and it is complete for it.*

For example, consider again $S_1 = \{\texttt{"2.2"}, \texttt{"2.3"}\}$ and $S_2 = \{\texttt{"2"}, \texttt{"3"}\}$. Given $\mathrm{S}\textsc{f}$, the completeness condition holds:

$$\alpha_{\mathrm{SF}}([\![\texttt{concat}(S_1, S_2)]\!]) = \mathsf{UFloat} = [\![\texttt{concat}(\alpha_{\mathrm{SF}}(S_1), \alpha_{\mathrm{SF}}(S_2))]\!]^{\mathrm{SF}}$$
$$= [\![\texttt{concat}(\mathsf{UFloat}, \mathsf{UInt})]\!]^{\mathrm{SF}}$$

5.2 Completing TAJS string abstract domain

Fig. 4b depicts the string abstract domain $\mathcal{TJ}$, the recasted version of the domain integrated into TAJS static analyser [20]. Differently from $\mathcal{SF}$, it splits the strings into $\mathsf{Unsigned}$, that denotes the strings corresponding to unsigned numbers, and $\mathsf{NotUnsigned}$, any other string. Hence, for example, $\alpha_{\mathcal{TJ}}(\{\texttt{"9"}, \texttt{"+9"}\}) = \top_{\mathcal{TJ}}$ and $\alpha_{\mathcal{TJ}}(\{\texttt{"9.2"}, \texttt{"foo"}\}) = \mathsf{NotUnsigned}$. As for $\mathcal{SF}$, before reaching these abstract values, $\mathcal{TJ}$ precisely tracks single string values.

In this section, we focus on the `toNum` (i.e., string-to-number) operation. Since this operation clearly involves numbers, in Fig. 7 we report the TAJS numerical abstract domain, denoted by $\mathcal{TJ}_{\mathsf{N}}$. The latter domain behaves similarly to $\mathcal{TJ}$, distinguishing between unsigned and not unsigned integers. Below we define the abstract semantics of the string-to-number operation for $\mathcal{TJ}$. In particular, we define the function:

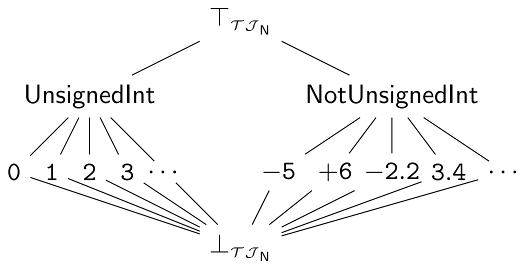

$\top_{\mathcal{TJ}_\mathsf{N}}$
UnsignedInt
NotUnsignedInt
0
1
2
3
···
−5
+6
−2.2
3.4
···
$\bot_{\mathcal{TJ}_\mathsf{N}}$

Fig. 7: TAJS numerical abstract domain.

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

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

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

When the input evaluates to $\bot_{\mathcal{TJ}}$, bottom is propagated and $\bot_{\mathcal{TJ}_\mathsf{N}}$ is returned (first row). While, if the input evaluates to a single string value, the abstract semantics relies on its concrete one (second row). When the input evaluates to the string abstract value $\mathsf{Unsigned}$ (third row), the integer abstract value $\mathsf{UnsignedInt}$ is returned. Finally, when the input evaluates to $\mathsf{NotUnsigned}$ or $\top_{\mathcal{JS}}$, the top integer abstract value is returned (fourth row).

Lemma 2. *$\mathcal{TJ}$ is not complete w.r.t.* `toNum`*. In particular, $\forall S \in \wp(\Sigma^*)$ we have that:*

$$\alpha_{\mathcal{TJ}}([\![\texttt{toNum}(S)]\!]) \subsetneq [\![\texttt{toNum}(\alpha_{\mathcal{TJ}}(S))]\!]^{\mathcal{TJ}}$$

Consider $S = \{\texttt{"2.3"}, \texttt{"3.4"}\}$. The completeness property does not hold:

$$\alpha_{\mathcal{TJ}}([\![\texttt{toNum}(S)]\!]) = \mathsf{NotUnsignedInt} \neq \top_{\mathcal{TJ}_\mathsf{N}} = [\![\texttt{toNum}(\alpha_{\mathcal{TJ}}(S))]\!]^{\mathcal{TJ}}$$

Again, the completeness condition does not hold because the $\mathcal{TJ}$ 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, $\mathcal{TJ}$ does not differentiate between non-numeric and unsigned integer string values, and this is the principal cause of the $\mathcal{TJ}$ 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, in order to

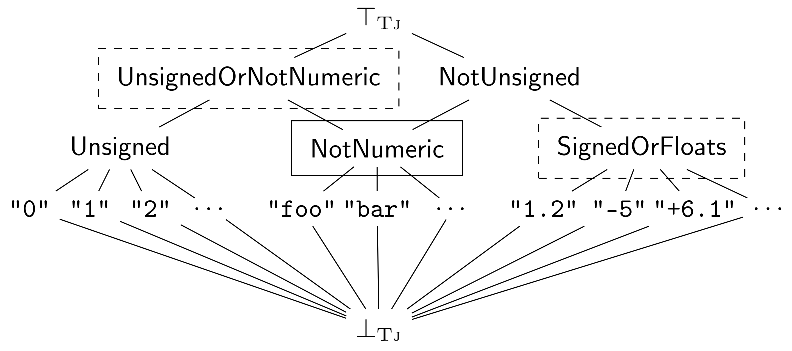


Fig. 8: Complete shell of $\rho_{\mathcal{TJ}}$ relative to $\rho_{\mathcal{TJ}_\mathsf{N}}$ w.r.t. `toNum`.

make $\mathcal{TJ}$ complete w.r.t. `toNum`, we have to derive the complete shell of the $\mathcal{TJ}$ string abstract domain relative to the $\mathcal{TJ}_\mathsf{N}$ numerical abstract domain, applying Theorem 1. In particular, let $\rho_{\mathcal{TJ}}$ and $\rho_{\mathcal{TJ}_\mathsf{N}}$ be the upper closure operators related to $\mathcal{TJ}$ and $\mathcal{TJ}_\mathsf{N}$ abstract domains, respectively. By applying Theorem 1, we obtain $\mathcal{S}^{\rho_{\mathcal{TJ}}}_{\mathtt{toNum}}(\mathcal{TJ}_\mathsf{N})$ (depicted in Fig. 8), i.e., the complete shell of $\rho_{\mathcal{TJ}}$ relative to $\rho_{\mathcal{TJ}_\mathsf{N}}$ w.r.t. `toNum`, and we denote it by $\textsc{Tj}$.

In particular, the abstract points inside dashed boxes are the abstract values added during the iterative computations of $\textsc{Tj}$, the points inside the standard boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in $\mathcal{TJ}$. A non-intuitive point added by $\textsc{Tj}$ 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 iterative computation of $\textsc{Tj}$, following the formula below:

$$\mathsf{SignedOrFloats} \in \max(\{Z \in \wp(\Sigma^*) \mid [\![\mathtt{toNum}(Z)]\!] \subseteq \gamma_{\mathcal{TJ}}(\mathsf{NotUnsigned})\})$$

Informally speaking, we are wondering the following question: *which is the maximal set of strings Z s.t.* `toNum`(Z) *is dominated by* NotUnsigned*?* In order to obtain from `toNum`(Z) only values dominated by NotUnsigned, 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 absolute complete shell $\textsc{Tj}$, when the following formula is computed:

$$\mathsf{UnsignedOrNotNumeric} = \max(\{Z \in \wp(\Sigma^*) \mid \mathtt{toNum}(Z) \subseteq \gamma_{\mathcal{TJ}}(\mathsf{Unsigned})\})$$

In order to obtain from `toNum`(Z) only abstract values dominated by Unsigned, 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.

Theorem 4. *ρ_{TJ} is the complete shell of $\rho_{\mathcal{TJ}}$ relative to $\rho_{\mathcal{TJ}_{\mathsf{N}}}$ w.r.t.* `toNum` *operation and hence it is complete for it.*

For example, consider again the string set $S = \{\texttt{"2.3"}, \texttt{"3.4"}\}$. Given TJ, the completeness condition holds:

$$\begin{aligned}\alpha_{\text{TJ}}([\![\texttt{toNum}(S)]\!]) &= \mathsf{NotUnsignedInt}\\ &= [\![\texttt{toNum}(\alpha_{\text{TJ}}(S))]\!]^{\mathcal{TJ}}\\ &= [\![\texttt{toNum}(\mathsf{SignedOrFloats})]\!]^{\mathcal{TJ}}\end{aligned}$$

6 What 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 $\mu\mathsf{Dyn}$ 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 $\mu\mathsf{Dyn}$, w.r.t. two string operations, namely `concat` and `toNum`, respectively. While string concatenation is common in any programming language, `toNum` assumes critical importance in the dynamic language context, mostly where implicit type conversion is provided. Since type conversion is often hidden from the developer, aim 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 starting TAJS string abstract domain, its abstraction will corresponds to $\mathsf{Unsigned}$, 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 $\top_{\mathcal{TJ}_{\mathsf{N}}}$, i.e., any possible concrete integer value. Conversely, abstracting S in TJ (the absolute complete shell of $\mathcal{TJ}$ relative to `toNum` discussed in Section 5.2) leads to a more precise abstraction, since TJ is able to differentiate between non-numerical and numerical strings. In particular, the abstract value of S in TJ is $\mathsf{NotNumeric}$, and $[\![\texttt{toNum}(\mathsf{NotNumeric})]\!]^{\mathcal{TJ}}$ will precisely return 0.

Adopting a complete shell w.r.t. a certain operation does not compromise the precision of the others. For example, consider again the original string abstract domain into TAJS static analyser and the following JavaScript fragment.

```
var obj = {
    "foo" : 1,
    "bar" : 2,
    "1.2" : 3,
    "2.2" : "hello"
}

y = obj[idx];
```

Suppose that the value of `idx` is the abstraction, in the starting TAJS string abstract domain, of the string set $S = \{$`"foo"`, `"bar"`$\}$, namely the abstract value NotUnsinged. 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`, included 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 5.2, 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`.

Complexity of the complete shells. We evaluate the complexity of the complete shells we have provided in the previous section. As usual in static analysis by abstract interpretation, there exists a trade-off between precision and efficiency: choose a preciser abstract domain may compromise the efficiency of the abstract computations. A representative example is reported in [17]: 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 be equipped also with a widening [13] in order 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 suggest 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 [12] 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 phase after consists to detect which values of the invariants derived by the static analyser are spurious values, namely values that are not certainly computed by the concrete execution of the program of interest. In particular, using imprecise (i.e., not complete) abstract domains clearly augment the number of false positives in the abstract computation of the static analyser, burdening the next phase of detection of the spurious values. On the other hand, 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 detection of false positives will be lighten since less noise has been introduced during the abstract computation of the invariants. Consider againt the JavaScript fragment reported in the previous paragraph. As we already discussed before, using the starting TAJS abstract domain to abstract the variable `idx` leads to a loss of precision, since the spurious value `"foo"` and `"bar"` are taken into account in its abstract value, namely `Unsigned`. Using the complete shell of TAJS w.r.t. `toNum` instead does not add noise when `idx` is used to access `obj`.

7 Conclusion

This paper addressed the problem of backward completeness in JavaScript-purpose string abstract domains, and provides, in particular, the complete shells of TAJS and SAFE string abstract domains w.r.t. `concat` and `toNum` operations. Our results can be easily applied also to JSAI string abstract domain [21], as it can be seen as an extension of the SAFE domain. The next issue we would like to investigate concerns forward completeness [17], meaning that no loss of precision occurs during the output abstraction process of a certain operation, and the integration of the completeness methodologies. As a final goal of our research, we aim to integrate the notion of complete shell into an industrial JavaScript static analyzer, so that, depending on the target program, an optimal string abstract domain is automatically selected from a set of domains and their complete shells, based on the specific string operations the program makes use of.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT Solver for String Constraints. In: CAV '15. pp. 462–469 (2015)

2. Amadini, R., Gange, G., Gauthier, F., Jordan, A., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Reference Abstract Domains and Applications to String Analysis. Fundam. Inform. **158**(4), 297–326 (2018)
3. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A Novel Approach to String Constraint Solving. In: CP '17. pp. 3–20 (2017)
4. Amadini, R., Jordan, A., Gange, G., Gauthier, F., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In: TACAS '17. pp. 41–57 (2017)
5. Arceri, V., Maffeis, S.: Abstract Domains for Type Juggling. Electr. Notes Theor. Comput. Sci. **331**, 41–55 (2017)
6. Arceri, V., Mastroeni, I.: Static Program Analysis for String Manipulation Languages. In: VPT'19. To appear. (2019)
7. Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.: String Analysis for Software Verification and Security. Springer (2017)
8. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: APLAS '08. pp. 3–18 (2008)
9. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. pp. 1–18 (2003)
10. Clarisó, R., Cortadella, J.: The Octahedron Abstract Domain. Sci. Comput. Program. **64**(1), 115–139 (2007)
11. Cortesi, A., Olliaro, M.: M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In: TASE'18. pp. 1–8 (2018)
12. Costantini, G., Ferrara, P., Cortesi, A.: A Suite of Abstract Domains for Static Analysis of String Values. Softw., Pract. Exper. **45**(2), 245–287 (2015)
13. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL'77. pp. 238–252 (1977)
14. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL'79. pp. 269–282 (1979)
15. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL'78. pp. 84–96 (1978)
16. Filaretti, D., Maffeis, S.: An Executable Formal Semantics of PHP. In: ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings. pp. 567–592 (2014)
17. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making Abstract Interpretations Complete. J. ACM **47**(2), 361–416 (2000)
18. Granger, P.: Static Analysis of Arithmetical Congruences. International Journal of Computer Mathematics - IJCM **30**, 165–190 (01 1989)
19. Granger, P.: Static Analysis of Linear Congruence Equalities among Variables of a Program. In: Abramsky, S., Maibaum, T.S.E. (eds.) TAPSOFT '91". pp. 169–192. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
20. Jensen, S.H., Møller, A., Thiemann, P.: Type Analysis for JavaScript. In: SAS '09. pp. 238–255 (2009)
21. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: a Static Analysis Platform for JavaScript. In: FSE '14. pp. 121–132 (2014)
22. Kim, S., Chin, W., Park, J., Kim, J., Ryu, S.: Inferring Grammatical Summaries of String Values. In: APLAS '14. pp. 372–391 (2014)
23. Kneuss, E., Suter, P., Kuncak, V.: Phantm: PHP Analyzer for Type Mismatch. In: FSE '10. pp. 373–374 (2010)

24. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In: FOOL'12 (2012)
25. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**(3), 206–234 (2016)
26. Madsen, M., Andreasen, E.: String Analysis for Dynamic Field Access. In: CC '14. pp. 197–217 (2014)
27. Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: APLAS '08. pp. 307–325 (2008)
28. Minamide, Y.: Static Approximation of Dynamically Generated Web Pages. In: WWW '05. pp. 432–441 (2005)
29. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006)
30. Oucheikh, R., Berrada, I., Hichami, O.E.: The 4-Octahedron Abstract Domain. In: NETYS '16. pp. 311–317 (2016)
31. Park, C., Im, H., Ryu, S.: Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain. In: DLS '16. pp. 25–36 (2016)
32. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A Symbolic Execution Framework for JavaScript. In: S&P '10. pp. 513–528 (2010)
33. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: LOPSTR '02. pp. 71–89 (2002)
34. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST '10. pp. 498–507 (2010)
35. Ward, M.: The Closure Operators of a Lattice. Annals of Mathematics **43**(2), 191–196 (1942)
36. Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In: PLDI'07. pp. 32–41 (2007)
37. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based Symbolic String Analysis for Vulnerability Detection. Formal Methods in System Design **44**(1), 44–70 (2014)
38. Yu, F., Bultan, T., Cova, M., Ibarra, O.H.: Symbolic String Verification: An Automata-Based Approach. In: SPIN '08. pp. 306–324 (2008)
39. Yu, F., Bultan, T., Hardekopf, B.: String Abstractions for String Verification. In: SPIN '11. pp. 20–37 (2011)