String Abstract Domains and Their Combination

Harald Søndergaard(✉)

School of Computing and Information Systems, The University of Melbourne,
Melbourne, VIC 3010, Australia
harald@unimelb.edu.au

Abstract. We survey recent developments in string static analysis, with an emphasis on how string abstract domains can be combined. The paper has formed the basis for an invited presentation given to LOPSTR 2021 and PPDP 2021.

1 Introduction

String manipulating programs are challenging for static analysis. The primary problem in static analysis, of finding a good balance between precision and efficiency, is particularly unwieldy for string analysis. For static reasoning about strings, many facets are of potential relevance, such as string length, shape, and the set of characters found in a string. Hence abstract interpretations of string manipulating programs tend to either employ an expressive but overly expensive abstract domain, or else combine a number of cheaper domains, each designed to capture a specific aspect of strings.

Much of the interest in the analysis of string manipulation is due to the fact that web programming and the scripting languages that are supported by web browsers make heavy use of strings. In the absence of static typing, strings, because of their flexibility, often end up being used as a kind of universal data structure. Moreover, scripting languages usually determine object attributes dynamically, treating an object as a lookup table that can associate any sort of information with an attribute, that is, with a string. And attributes may themselves be constructed dynamically. All this contributes to making the construction of robust, secure programs difficult. Hence much research on string analysis comes from communities that work with languages such as JavaScript, PHP and Python. The dynamic nature of these languages calls for combinations of non-trivial static and dynamic analysis, a continuing challenge.

This survey focuses on abstract interpretation. Much of it is based on Amadini *et al.* [2,3], from whom we take the concept of a "reference" abstract domain. We provide a Cook's Tour of string abstract domains, discuss how to combine domains, and show how reference domains can help domain combination, in theory and in practice.

© Springer Nature Switzerland AG 2022
E. De Angelis and W. Vanhoof (Eds.): LOPSTR 2021, LNCS 13290, pp. 1–15, 2022.
https://doi.org/10.1007/978-3-030-98869-2_1

2 Preliminaries

We start by summarising the main mathematical concepts underpinning abstract interpretation. Readers familiar with abstract interpretation and finite-state automata can skip this section.

A *poset* is a set, equipped with a partial order. A binary relation $\sqsubseteq$, defined on a set D, is a *partial order* iff it is reflexive, transitive and antisymmetric. Two elements $x, y \in D$ are *comparable* iff $x \sqsubseteq y$ or $y \sqsubseteq x$. A poset D is a *chain* iff, for each pair $x, y \in D$, x and y are comparable.

An element $x \in D$ is an *upper bound* for set $D' \subseteq D$ iff $x' \sqsubseteq x$ for all $x' \in D'$. Dually we may define a *lower bound* for D'. An upper bound x for D' is the *least upper bound* for D' iff, for every upper bound x' for D', $x \sqsubseteq x'$. We denote it (when it exists) by $\bigsqcup D'$. Dually we may define the *greatest lower bound* and denote it by $\bigsqcap D'$. We write $x \sqcup y$ for $\bigsqcup\{x, y\}$ and $x \sqcap y$ for $\bigsqcap\{x, y\}$.

Let $\langle D, \sqsubseteq \rangle$ and $\langle D', \leq \rangle$ be posets. A function $f : D \to D'$ is *isotone* iff $\forall x, y \in D : x \sqsubseteq y \Rightarrow f(x) \leq f(y)$. A function $f : D \to D$ is *idempotent* iff $\forall x \in D : f(f(x)) = f(x)$. Function f is *reductive* iff $\forall x \in D : f(x) \sqsubseteq x$; it is *extensive* iff $\forall x \in D : x \sqsubseteq f(x)$. A function which is isotone and idempotent is a *closure operator*. If it is also reductive, it is called a *lower closure operator* (lco). If it is extensive, it is called an *upper closure operator* (uco).

A poset $\langle D, \sqsubseteq \rangle$ is a *lattice* iff every finite subset $X \subseteq D$ has a least upper bound and a greatest lower bound—written $\bigsqcup X$ and $\bigsqcap X$, respectively. The lattice is *complete* iff the condition applies to every subset, finite or not. It is *bounded* iff it has a unique least element (often written $\bot$) and a unique greatest element (often written $\top$). We write the bounded lattice D as $\langle D, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$. A complete lattice is necessarily bounded.

Abstract interpretation is a *declarative* approach to static program analysis. An analysis is almost completely described by its associated *abstract domain*: the set $\mathcal{A}$ of abstractions of computation states used by the analysis. The abstract domain is usually a *bounded lattice* $\langle \mathcal{A}, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$. In our case, an element of $\mathcal{A}$ will be a *string property*. Strings are constructed from some unspecified alphabet Σ. Hence each element (or *abstract string*) $\hat{s} \in \mathcal{A}$ denotes a set of concrete strings $\gamma(\hat{s}) \in \mathcal{P}(\Sigma^*)$ via a *concretization function* γ such that $\hat{s} \sqsubseteq \hat{s}'$ iff $\gamma(\hat{s}) \subseteq \gamma(\hat{s}')$. Often γ has an adjoined function $\alpha : \mathcal{P}(\Sigma^*) \to \mathcal{S}$, the *abstraction function*, that is, we have a *Galois connection*: $\alpha(S) \sqsubseteq \hat{s}$ iff $S \subseteq \gamma(\hat{s})$. In this case both α and γ are necessarily isotone, $\alpha \circ \gamma$ is an lco, and $\gamma \circ \alpha$ is a uco. Moreover, every concrete operation $f : \mathcal{P}(\Sigma^*)^k \to \mathcal{P}(\Sigma^*)$ has a unique best counterpart on $\mathcal{S}$, namely $\lambda(\hat{s_1}, \ldots, \hat{s_k}) \,.\, (\alpha \circ f)(\gamma(\hat{s_1}), \ldots, \gamma(\hat{s_k}))$. Hence we can essentially identify a program analysis with the *abstract domain* it uses.

We use standard notation and terminology for automata [24]. A deterministic finite automaton (DFA) R with alphabet Σ is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is the (non-empty) set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accept states, and $\delta : (Q \times \Sigma) \to Q$ is the transition function. The language recognised by R is written $\mathcal{L}(R)$. We use $\mathcal{L}_R(q)$ to denote the language recognised by $\langle Q, \Sigma, \delta, q, F \rangle$, that is, by a DFA identical to R, except q is considered the start state. We let $\delta^* : (Q \times \Sigma^*) \to Q$ be the generalised transition function defined by

$$\begin{aligned}\delta^*(q,\epsilon) &= q\\ \delta^*(q,x\ w) &= \delta^*(\delta(q,x),w)\end{aligned}$$

Let $q \rightarrow q'$ stand for $\exists x \in \Sigma(\delta(q,x)=q')$. The DFA $\langle Q,\Sigma,\delta,q_0,F\rangle$ is *trim* iff δ is a *partial* deterministic function on $Q \times \Sigma$, and for all $q \in Q\backslash\{q_0\}$, there is a $q' \in F$ such that $q_0 \rightarrow^+ q \wedge q \rightarrow^* q'$.

3 String Abstract Domains

A *string abstract domain* approximates concrete domain $\langle \mathcal{P}(\Sigma^*), \subseteq, \varnothing, \Sigma^*, \cap, \cup\rangle$. Figure 1 shows Hasse diagrams for some example string domains. We discuss them in Sect. 3.2, but first we look at some simple but general string domains.

3.1 Programming Language Agnostic String Abstract Domains

Exactly which strings a variable may be bound to at a given program point is an undecidable problem. For a simple program such as

```
x = "foo"
if (*)
    x = "zoo"
```

it is easy to tell that x, upon exit, will take its value from the set $\{\texttt{foo},\texttt{zoo}\}$. But in general we have to resort to finite descriptions of (possibly infinite) string sets, and reason with those. For example, we may approximate a set of strings by the characters they use. Then

$$\begin{aligned}\alpha_{\text{chars}}(\{\texttt{foo},\texttt{zoo}\}) &= \{\texttt{f},\texttt{o},\texttt{z}\}\\ \gamma_{\text{chars}}(\{\texttt{f},\texttt{o},\texttt{z}\}) &= \{w \in \Sigma^* \mid \mathit{chars}(w) \subseteq \{\texttt{f},\texttt{o},\texttt{z}\}\}\end{aligned}$$

Abstraction usually *over-approximates*; our description $\{\texttt{f},\texttt{o},\texttt{z}\}$ may have been intended to describe $\{\texttt{foo},\texttt{zoo}\}$, but is applies to an infinity of other strings, such as zoffo.

Domain	$\gamma(\{\texttt{foo},\texttt{zoo}\})$
$\mathcal{CI}$	$[\{\texttt{o}\},\{\texttt{f},\texttt{o},\texttt{z}\}]$
$\mathcal{SS}_2$	$\{\texttt{foo},\texttt{zoo}\}$
$\mathcal{SL}$	$[3,3]$
$\mathcal{PS}$	$\langle\epsilon,\texttt{oo}\rangle$

Let us list some examples of string domains. $\mathcal{CI} = \{\bot_{\mathcal{CI}}\} \cup \{[L,U] \mid L,U \in \mathcal{P}(\Sigma), L \subseteq U\}$ is the Character Inclusion domain. It provides *pairs* of character sets, the first of which hold characters that *must* be in any string described, the second characters that *may* be there. An example is given in the table on the right.

The String Set domain $\mathcal{SS}_k$ provides string sets up to cardinality k, representing all larger sets as $\top$. The String Length domain $\mathcal{SL}$ provides pairs $[\mathit{lo},\mathit{hi}]$ of natural numbers that give lower and upper bounds on the length of strings described, again exemplified in the table. Let us describe the Prefix/Suffix domain $\mathcal{PS}$ in some formal detail (for more detailed definitions of the other domains, see for example [3]).

$\mathcal{PS} = \{\bot_{\mathcal{PS}}\} \cup (\Sigma^* \times \Sigma^*)$. The meaning of an element of form $\langle p, s\rangle$ is the set of strings that have p as prefix and s as (possibly overlapping) suffix. Formally,

$$\begin{aligned} \gamma(\bot_{\mathcal{PS}}) &= \emptyset \\ \gamma(\langle p, s\rangle) &= \{p \cdot w \mid w \in \Sigma^*\} \cap \{w \cdot s \mid w \in \Sigma^*\} \end{aligned}$$

The largest element $\top_{\mathcal{PS}} = \langle \epsilon, \epsilon\rangle$ where ϵ is the empty string. $\sqsubseteq$, $\sqcup$ and $\sqcap$ are defined in terms of longest common prefix/suffix operations. Let $lcp(S)$ and $lcs(S)$ be the longest common prefix (respectively, suffix) of a string set S. Then $\langle p, s\rangle \sqsubseteq_{\mathcal{PS}} \langle p', s'\rangle$ iff $lcp(\{p, p'\}) = p' \wedge lcs(\{s, s'\}) = s'$, and $\langle p, s\rangle \sqcup_{\mathcal{PS}} \langle p', s'\rangle = \langle lcp\{p, p'\}, lcs\{s, s'\}\rangle$. The meet operation $\sqcap_{\mathcal{PS}}$ is induced—for details see Costantini *et al.* [9]. The abstract catenation operation is particularly simple in the case of $\mathcal{PS}$: $\langle p, s\rangle \odot_{\mathcal{PS}} \langle p', s'\rangle = \langle p, s'\rangle$, with $\bot_{\mathcal{PS}}$ as an annihilator. The abstract version of substring selection (from index i to j) is defined

$$\langle p, s\rangle[i..j]_{\mathcal{PS}} = \begin{cases} \langle p[i..j], p[i..j]\rangle & \text{if } j \le |p| \\ \langle p[i..|p|], \epsilon\rangle & \text{if } i \le |p| \le j \\ \top_{\mathcal{PS}} & \text{otherwise} \end{cases}$$

Most of these abstract operations have $O(|p| + |s|)$ cost.

Many other, often more sophisticated, string abstract domains have been proposed. For example, the *string hash* domain [20] ($\mathcal{SH}$) is a flat domain of hash values (integers) obtained through application of some string hash function.

3.2 Language Specific String Domains

Aspects of some scripting languages necessitate greater care in string analysis. For example, it may be important to distinguish strings that are valid representations of "numerical entities" such as `"-42.7"` or `"NaN"`. Figures 1(a–c) shows string abstract domains used by three different analyzers for JavaScript. A string set such as $\{\texttt{foo}, \texttt{NaN}\}$ is represented as $\top$, *NotSpecial*, or *NotUnsigned*, by the respective tools, all with different meanings.

Note that it is very difficult for a program analyzer to maintain precision in the presence of dynamic typing and implicit type conversion. Arceri *et al.* [4] retain some agility in their JavaScript analyzer through the use of a *tuple of abstractions* for each variable x: (x *qua* number, x *qua* string, ...), so that the relevant abstraction can be retrieved across conversions.

Amadini *et al.* [3] performed a comparison of a dozen common string abstract domains, shown in Fig. 1(d). Figure 1 also identifies the (non-extreme) elements of some domains not discussed so far, namely $\mathcal{NO}$, $\mathcal{NS}$, $\mathcal{UO}$, and the Constant String domain $\mathcal{CS}$. The details of these are not essential for this presentation, except we note that elements of $\mathcal{CS}$ are single strings, $\bot_{\mathcal{CS}}$, or $\top_{\mathcal{CS}}$, with the obvious meanings. The main message to take away is that potentially useful string domains are *legion*.

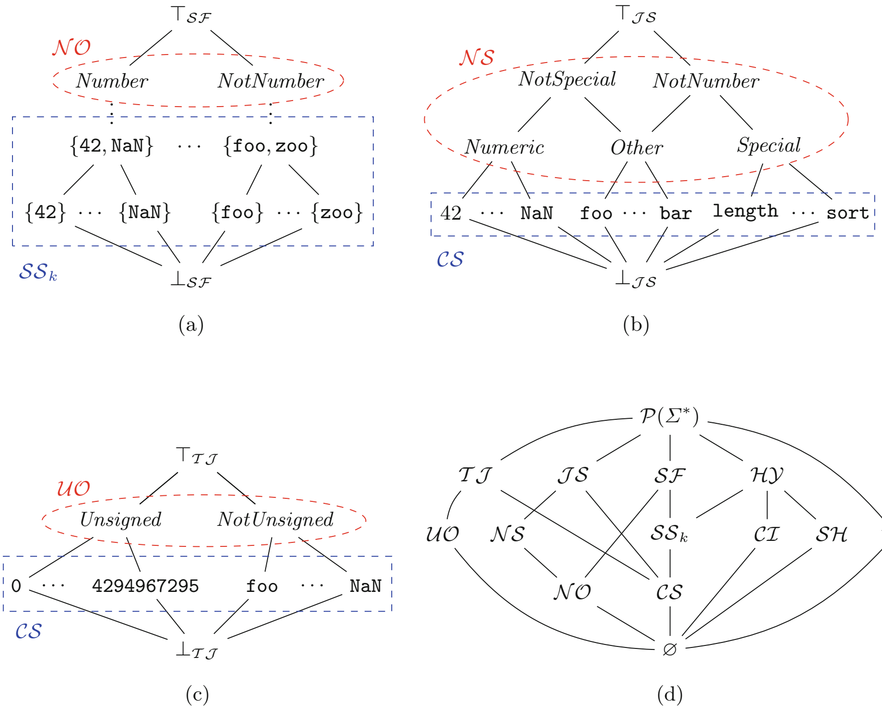


Fig. 1. String abstract domains used by different tools for JavaScript analysis: (a) SAFE [19], (b) JSAI [18], (c) TAJS [16]; (d) domains compared experimentally in Amadini *et al.* [3] ($\mathcal{HY}$ is the "hybrid" domain of Madsen and Andreasen [20])

3.3 Regular Expression-Like Domains

The "Bricks" domain of Costantini *et al.* [8] captures (sequences of) string sets with multiplicity. A *brick* is of form $[S]^{i,j}$ where S is a finite string set. $[S]^{i,j}$ represents the set $\bigcup_{i \leq k \leq j} \{w_1 \cdot w_2 \cdots w_k \mid (w_1, w_2, \ldots, w_k) \in S^k\}$. For example, $\{\mathtt{ab}, \mathtt{c}, \mathtt{abab}, \mathtt{cab}, \mathtt{cc}\}$ can be approximated by the brick $\{\mathtt{ab}, \mathtt{c}\}^{1,2}$. Elements of the *Bricks* domain are sequences of bricks, with the sequencing representing language catenation. The "Dashed Strings" of Amadini *et al.* [1] offer a variant of this using sequences of blocks $[S]^{i,j}$ which are like bricks, except S is a finite set of characters.

More expressive fragments of regular expressions are sometimes used [6, 21]. For example, Park *et al.* [21] approximate string sets by "atomic" regular expressions. These are regular expressions generated by the grammar

$$S \rightarrow \text{`}\epsilon\text{'} \mid \text{`}\Sigma^*\text{'} \mid A\ S \mid \text{`}\Sigma^*\text{'}A\ S \qquad A \rightarrow \mathtt{a}_1 \mid \ldots \mid \mathtt{a}_n$$

where $\Sigma = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ (we use quotes to stress that in this grammar, ϵ and Σ^* are *terminals*, not meta-symbols). The abstract domain used by Choi *et al.* [6]

is less constrained, and one could even contemplate the use of the whole class of regular languages, as done in the next section.

An example of a string abstract domain designed specifically for the analysis of C programs is the "M-Strings" domain, proposed by Cortesi *et al.* [7]. Recall that C uses the null character `\0` to indicate termination of a string. Hence the sequence of characters `zooo\0ba` represents a sequence of two strings. A set of C string sequences such as $\{$`zooo\0ba`, `dooo\0da`$\}$ is captured by the M-string $\langle\{0\}\top\{1\}\mathtt{o}\{4\}, \{5\}\top\{6\}\mathtt{a}\{7\}\rangle$. The numbers in braces are string index positions. The $\{1\}\mathtt{o}\{4\}$ component, for example, covers all C strings that have '`ooo`' from index 1 to 3, inclusive.

3.4 The Class of Regular Languages as an Abstract Domain $\mathcal{RL}$

Perhaps the most natural choice for a string abstract domain is the bounded lattice $\langle \mathit{Reg}, \subseteq, \varnothing, \Sigma^*, \cap, \cup\rangle$ of regular languages. A obvious representation for the elements are (possibly trim) minimal DFAs [4]. The advantages of this choice are significant: This abstract domain is very expressive and regular languages and DFA operations are well understood. Unusually, the domain $\mathcal{RL}$ is closed not only under intersection, but also under complement—a rare occurrence. At least one implementation is publicly available [4].

There are, however, also significant drawbacks: First, to enable containment tests, automata need to be maintained in deterministic, ideally minimal, form; this normalisation is very expensive. Second, there is a clear risk of *size explosion*, as DFA size is unbounded; for $\mathcal{L}(R) \cap \mathcal{L}(R')$ and $\mathcal{L}(R) \cup \mathcal{L}(R')$ it can be as bad as $|R| \cdot |R'|$. Finally there is the termination problem. Unlike other abstract domains discussed so far, $\mathcal{RL}$ has infinite ascending chains, so that Kleene iteration may not terminate without "widening".

4 Widening

Where an abstract domain has infinite ascending chains, termination is usually ensured by defining a *widening* operator ∇. This is an upper bound operator ($a \sqsubseteq a \nabla b$ and $b \sqsubseteq a \nabla b$) with the property that, for any sequence $\{a_1, a_2, \ldots\}$, the sequence $b_1 = a_1, b_{i+1} = b_i \nabla a_{i+1}$ stabilises in finitely many steps [10].

The design of widening operators is a difficult art. Done well, widening can be very powerful, but there is usually no single natural design that presents itself. For highly expressive domains it often becomes hard to balance convergence rate with preservation of precision.

Bartzis and Bultan [5] pioneered the design of widening for automata. We will explain their widening with an example—for exact definitions, the reader is referred to the original papers discussed in this section.

Example 1. Consider this pseudo-code (from [2]) involving a while loop:

```
x = "aaa"
while (*)
```

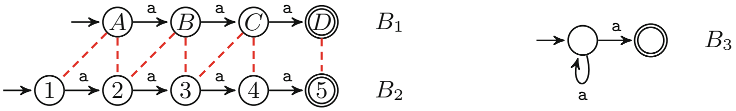


Fig. 2. DFA widening a la Bartzis and Bultan [5]

```
if (length(x) < 4) x = "a" + x
```

When the loop is first entered, the variable has the value aaa. After one iteration, the value is aaaa. Figure 2 shows the (trim, deterministic) automata for these values, $B_1 = \langle Q_1, \Sigma, \delta_1, q_{01}, F_1 \rangle$ and $B_2 = \langle Q_2, \Sigma, \delta_2, q_{02}, F_2 \rangle$. The idea now is to weaken B_2 by merging some of its states. Which states to merge is determined by similarity with B_1 as follows. Consider the relation

$$\rho = \left\{ (q_1, q_2) \in Q_1 \times Q_2 \;\middle|\; \begin{array}{l} (a) \mathcal{L}_{B_1}(q_1) = \mathcal{L}_{B_2}(q_2), \textbf{or} \\ (b) q_1 \in (Q_1 \setminus F_1), q_2 \in (Q_2 \backslash F_2), \text{ and for some} \\ \quad w \in \Sigma^*, q_1 = \delta_1^*(q_{01}, w) \text{ and } q_2 = \delta_2^*(q_{02}, w) \end{array} \right\}$$

For our example, we have $\rho = \{(A, 2), (B, 3), (C, 4), (D, 5), (A, 1), (B, 2), (C, 3)\}$. Now the idea is to form the reflexive transitive closure of $\rho^{-1} \circ \rho$, to create an equivalence relation on Q_2. The result of widening will be the corresponding quotient automaton—states of B_2 that belong to the same equivalence class are merged. For our example there are two classes, $\{1, 2, 3, 4\}$ and $\{5\}$, and the resulting automaton is B_3 shown in Fig. 2. □

Two comments are relevant. First, the automata that result from this type of widening are usually non-deterministic, and in an implementation, widening needs to be followed by determinisation. Moreover, as shown by D'Silva [14], widening is generally sensitive to the shape of the resulting DFA, and for best results, minimisation is also required. Second, as pointed out by Bartzis and Bultan [5], the method as described is not strictly a widening, as it does not guarantee stabilisation in finite time. Bartzis and Bultan mention that a guarantee can be secured by dropping the "q_1 and q_2 are reject states" part of the condition (b) in the set comprehension above, but the cost is an intolerable loss of precision; their discussion ([5] page 326) underlines the tension between precision and convergence. For our example, the automaton that results when the condition is weakened recognises $\mathtt{a}^*$, rather than the more precise $\mathcal{L}(B_3) = \mathtt{a}^+$.

D'Silva [14] conducted a deeper study of a variety of families of widening for automata. These generalise the Bartzis-Bultan approach in a number of ways, including the way relevant state equivalence classes are identified. For example, in the "k-tails" approach, the relation ρ is determined by considering only strings of length k or less. Let $\mathcal{L}_R^k(q) = \{w \in \mathcal{L}_R(q) \mid |w| \leq k\}$. Then for automata S_1 and S_2, $(s_1, s_2) \in \rho$ iff $\mathcal{L}_{S_1}^k(s_1) = \mathcal{L}_{S_2}^k(s_2)$.

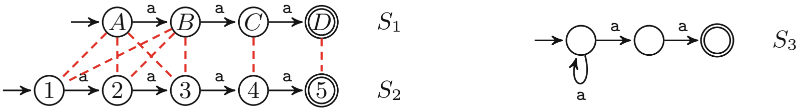


Fig. 3. 1-tails widening a la D'Silva [14]

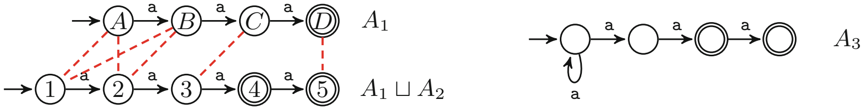


Fig. 4. Widening a la Arceri *et al.* [4]

Example 2. Consider again Example 1. The automata are shown (renamed) in Fig. 3. In this case, $\rho = \{(A, 1), (A, 2), (A, 3), (B, 1), (B, 2), (B, 3), (C, 4), (D, 5)\}$ and $(\rho^{-1} \circ \rho)^*$ induces three equivalence classes, $\{1, 2, 3\}$, $\{4\}$, and $\{5\}$. Hence the result of widening is S_3, shown in Fig. 3. □

Again, D'Silva warns that the methods discussed may not always be widenings in the classical sense, as the convergence guarantees that are on offer are conditional on a variety of parameters.

D'Silva's ideas have been adopted for practical string analysis by Arceri, Mastroeni and Xu [4]. Here the decision about which states to merge is based on the k-tails principle just exemplified, but Arceri *et al.* replaces the induced widening operator ∇_k by ∇ defined by $A_1 \nabla A_2 = A_1 \nabla_k\ (A_1 \sqcup A_2)$.

Example 3. Figure 4 shows the result for our running example. As comparison is now against $A_1 \sqcup A_2$, we have $\rho = \{(A, 1), (A, 2), (B, 1), (B, 2), (C, 3), (D, 5)\}$, and $(\rho^{-1} \circ \rho)^*$ induces four equivalence classes, $\{1, 2\}$, $\{3\}$, $\{4\}$ and $\{5\}$. The result of widening is A_3 in Fig. 4. Once S_3 and A_3 are determinised and minimised, they are identical (and more precise than B_3). □

5 Combining Domains

The study by Amadini *et al.* [3] included *combinations* of different string abstract domains, but it focused on *direct products* of the domains involved.

5.1 Direct Products

Suppose the n abstract domains $\langle \mathcal{A}_i, \sqsubseteq_i, \bot_i, \top_i, \sqcap_i, \sqcup_i \rangle$ $(i = 1, \ldots, n)$ all abstract a concrete domain $\mathcal{C}$. Their *direct product* is $\langle \mathcal{A}, \sqsubseteq, \bot, \top, \sqcap, \sqcup, \rangle$ with:

- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$
- $(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n)$ iff $a_i \sqsubseteq_i b_i$ for all $i \in [1..n]$
- $\bot = (\bot_1, \ldots, \bot_n)$ and $\top = (\top_1, \ldots, \top_n)$

- $(a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap_1 b_1, \ldots, a_n \sqcap_n b_n)$
- $(a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup_1 b_1, \ldots, a_n \sqcup_n b_n)$
- $\gamma(a_1, \ldots, a_n) = \bigcap_{i=1}^{n} \gamma_i(a_i)$ and $\alpha(c) = (\alpha_1(c), \ldots, \alpha_n(c))$

The direct product generally induces a concretisation function which is not injective.

Example 4. Consider string analysis using $\mathcal{SL} \times \mathcal{CI}$. Of the two descriptions $([3,3], [\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{\mathtt{a},\mathtt{b},\mathtt{c}\}])$ and $([0,3], [\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}])$, the former is strictly smaller than the latter, by the component-wise ordering of the direct product. But $\gamma([3,3], [\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{\mathtt{a},\mathtt{b},\mathtt{c}\}]) = \gamma([0,3], [\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}])$; each represents the string set $\{\mathtt{abc}, \mathtt{acb}, \mathtt{bac}, \mathtt{bca}, \mathtt{cab}, \mathtt{cba}\}$. The components of the second description are unnecessarily imprecise. □

In an analysis based on the direct product, no exchange of information happens between the component domains, often leading to an unwanted loss of precision.

5.2 Reduced Products

The *mathematical* solution is to force γ to be injective. Consider the equivalence relation $\equiv$ defined by

$$(a_1, \ldots, a_n) \equiv (b_1, \ldots, b_n) \text{ iff } \gamma(a_1, \ldots, a_n) = \gamma(b_1, \ldots, b_n)$$

The *reduced product* $\mathcal{A}' = \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$ is the quotient set of $\equiv$:

$$\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n = \{[(a_1, \ldots, a_n)]_\equiv \mid a_1 \in \mathcal{A}_1, \ldots, a_n \in \mathcal{A}_n\}$$

Define (the injective) $\gamma : \mathcal{A}' \to \mathcal{C}$ and $\alpha : \mathcal{C} \to \mathcal{A}'$ by

$$\begin{array}{lcl} \gamma([(a_1, \ldots, a_n)]_\equiv) & = & \bigcap_{i=1}^{n} \gamma_i(a_i) \\ \alpha(c) & = & [(\alpha_1(c), \ldots, \alpha_n(c))]_\equiv \end{array}$$

If a greatest lower bound exists (say $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are complete lattices) then $[(a_1, \ldots, a_n)]_\equiv$ is identified with its minimal representative: $\bigsqcap([(a_1, \ldots, a_n)]_\equiv)$. Moreover, if each (γ_i, α_i) is a Galois connection then so is (γ, α).

Reduced products are easy to define but generally hard to realise. Algorithms for the required operations are far from obvious. Moreover, an incremental approach to analysis where many abstract domains are involved does not appear possible. To quote Cousot, Cousot and Mauborgne [12], "The implementation of the most precise reduction (if it exists) can hardly be modular since in general adding a new abstract domain to increase precision implies that the reduced product must be completely redesigned". Section 6 suggests a remedy.

5.3 Paraphrasing: Translating Approximate Information

It is natural to propose some kind of information exchange to translate insight from one component of a domain product to other components, in order to calculate minimal representatives of equivalence classes. Let us call this improvement of one component using information from another *paraphrasing*.

As an example, $[\{\mathtt{a},\mathtt{b},\mathtt{c}\},\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}] \in \mathcal{CI}$ can be seen as an "$\mathcal{SL}$ paraphraser" $\lambda v.\ v \sqcap_{\mathcal{SL}} [3,\infty]$ which tightens an $\mathcal{SL}$ component appropriately. More generally, the $\mathcal{SL}$ paraphraser corresponding to $[L,U] \in \mathcal{CI}$ is $\lambda v.\ v \sqcap_{\mathcal{SL}} [|L|,\infty]$.

Here is another example of a paraphraser. We can view $\langle p,s\rangle \in \mathcal{PS}$ as a $\mathcal{CI}$ paraphraser $P_{\mathcal{PS}}^{\mathcal{CI}}\langle p,s\rangle : \mathcal{CI} \to \mathcal{CI}$:

$$P_{\mathcal{PS}}^{\mathcal{CI}}\langle p,s\rangle(v) = \begin{cases} [L \cup X, U] \text{ if } v = [L,U] \text{ and } L \cup X \subseteq U \\ \qquad\qquad\qquad \text{where } X = \mathit{chars}(p) \cup \mathit{chars}(s) \\ \bot_{\mathcal{CI}} \qquad \text{otherwise} \end{cases}$$

Granger [15] proposed an important "local increasing iterations" technique to improve the precision of abstract interpretation. Granger's technique can also be used to improve the direct product of domains—it effectively uses paraphrasing systematically and repeatedly. Let us call the results *Granger products*.

However, when many abstract domains are involved, we soon run into a combinatorial problem. If we have n abstract domains, we can have $n(n-1)$ paraphrasers $P_i^j : \mathcal{A}_i \to \mathcal{A}_j \to \mathcal{A}_j$, so even for small n, a large number of "translation tools" are needed. The strain of juggling many different kinds of information, delivered through different abstract domains, becomes prohibitive. As we have seen, this is typically the situation we are faced in string analysis. Note that if paraphrasers of type $(\mathcal{A}_1 \times \cdots \times A_k) \to \mathcal{A}_j \to \mathcal{A}_j$ are allowed (for $k > 1$), the number of possible paraphrasers is well beyond quadratic.

5.4 One-on-One Paraphrasing

Even if each one-on-one paraphraser $P_i^j(a_i)$ is an lco, it may have to be applied repeatedly. The combined effect of paraphrasing until no more tightening is possible comes down to computing the *greatest fixed point* of P defined by

$$P(a_1,\ldots,a_n) = \begin{pmatrix} a_1 \sqcap \bigsqcap_{i\in[1..n]} P_i^1(a_i)(a_1) \\ \vdots \\ a_n \sqcap \bigsqcap_{i\in[1..n]} P_i^n(a_i)(a_n) \end{pmatrix}$$

This is the approach suggested by Granger [15], and further developed by Thakur and Reps [25].

However, sometimes one-on-one paraphrasing falls short [2]:

Example 5. Let $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ and consider the combination of abstractions

$$x = [5,6] \in \mathcal{SL} \qquad y = [\Sigma,\Sigma] \in \mathcal{CI} \qquad z = \langle \mathtt{ab}, \mathtt{aba}\rangle \in \mathcal{PS}$$

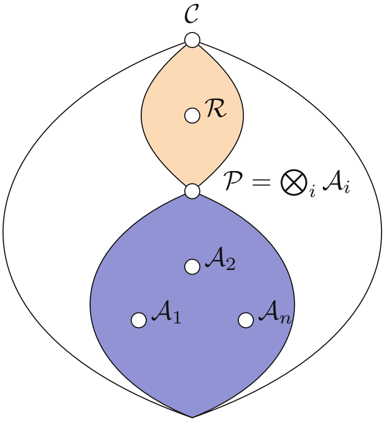
A system of optimal paraphrasers for this example leads to an equation system whose solution is simply (x,y,z).

That is, application of P provides no improvement (in this case P acts as the identity function). To see this, note that the knowledge (in y) that a string s uses the whole alphabet does not allow us to improve on x, nor on z. Conversely, neither x nor z can improve on y, since y is as precise as $\mathcal{CI}$ will allow. And, x and z clearly cannot improve on each other.

In contrast, in $P = \mathcal{SL} \otimes \mathcal{CI} \otimes \mathcal{PS}$, (x, y, z) denotes $\varnothing$, since no string satisfies all three constraints. To see this, note that the combination of y and z allows only strings of length 7 or more—strings must have form $\mathtt{ab}\Sigma^*\mathtt{c}\Sigma^*\mathtt{d}\Sigma^*\mathtt{aba}$ or $\mathtt{ab}\Sigma^*\mathtt{d}\Sigma^*\mathtt{c}\Sigma^*\mathtt{aba}$. □

The example shows that sometimes no amount of (repeated) one-on-one paraphrasing will lead to the optimal reduction. Nor is this kind of paraphrasing enough, in general, to implement optimal transfer functions. Amadini *et al.* [2] have suggested an alternative that involves the use of what they call a reference domain.

6 Reference Abstract Domains



When a large number of abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$ need to be combined, we may look for a way of obtaining the effect of a reduced product, while avoiding the combinatorial explosion of paraphrasing. Amadini *et al.* [2] propose the use of an additional domain that can act as a *mediator* among the n given domains, a "reference" domain. This domain should be as expressive as each $\mathcal{A}_i$. This way it is, if anything, "closer" to the concrete domain than the reduced product is. In the diagram on the right, $\mathcal{C}$ is the concrete semantic domain and $\mathcal{P}$ is the reduced product of the n domains $\mathcal{A}_1$ to $\mathcal{A}_n$. For an abstract domain $\mathcal{R}$ to fill the role of reference domain, it must be located as suggested in the diagram.

We can then achieve the effect of using $\mathcal{P}$ by recasting each of the n components in $\mathcal{RL}$, taking the greatest lower bound of the results in $\mathcal{RL}$, and translating that back into the n abstract domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The combined effect of this amounts to the application of a "strengthening" function (an lco) $\mathbb{S} : (\mathcal{A}_1 \times \ldots \times \mathcal{A}_n) \to (\mathcal{A}_1 \times \ldots \times \mathcal{A}_n)$.

The same idea was present already in work on symbolic abstraction by Reps and Thakur [22], albeit in a rather different form. The authors cast the problem of symbolic abstraction in logical terms, associating abstract domains with different logic fragments. In that setting, a reference domain is simply a sufficiently expressive fragment.

Example 6. Take the case of abstract domains for linear arithmetic constraints. The well-known polyhedral domain [13] is very expressive and also expensive. Instead we may want to combine two cheaper analyses, namely intervals [11] and Karr's domain of affine equations [17]. However, reduced products of numeric abstract domains, while mathematically straightforward, are difficult to implement. Noting that *translations* between each cheap domain and the polyhedral domain are inexpensive, we may choose to use the latter as reference domain. For example, assume we have

$$x \in [5, \infty], z \in [1, 10] \text{ and } 2x - y = 8, x + 2z = 7$$

We translate each to polyhedral form and calculate the meet there:

$$(5 \leq x, 1 \leq z, z \leq 10) \ \sqcap \ (2x - y = 8, x + 2z = 7)$$

Then we translate the result $(x = 5, y = 2, z = 1)$ back into the interval domain, as well as into Karr's domain. Note how each component is strengthened. □

In the presence of a large number of incomparable abstract domains, a suitable reference domain offers several advantages:

- Simplicity of implementation, as its requires $2n$ translation functions rather than a quadratic number.
- Modular implementation; incorporating yet another domain $\mathcal{A}_{n+1}$ is mostly straight-forward.
- Potentially lower computational cost at runtime.

Amadini *et al.* [2] find the idea particularly useful in the context of string abstract domains, since $\mathcal{RL}$ presents itself as a natural candidate for reference domain.

Algorithms for translations to and from $\mathcal{RL}$ are given by Amadini *et al.* [2], for the string abstract domains $\mathcal{CS}$, $\mathcal{SL}$, $\mathcal{CI}$, and $\mathcal{PS}$. Trim DFAs are used to represent regular languages, and most translations turn out to be straightforward. Here we give just one example, namely the translation of an element $\langle p, s \rangle \in \mathcal{PS}$ into an element of $\mathcal{RL}$, in the form of a trim DFA. Assume the DFA that recognises p is $\langle Q, \Sigma, \delta, q_0, \{q_f\} \rangle$ and let q^* be the unique state for which $\delta(q^*) = q_f$. Note that this DFA is very easily built. Let $\langle Q', \Sigma, \delta', q'_0, \{q'_f\} \rangle$ be the Knuth-Morris-Pratt DFA for s (a recogniser of $\Sigma^* s$). Again, this DFA is easily built directly, rather than going via an NFA (see for example [23] page 765). The DFA for $\langle p, s \rangle$ is $\langle (Q \cup Q') \backslash \{q_f\}, \Sigma, \delta[q^* \mapsto \delta^*(q'_0, p)] \cup \delta', q_0, \{q'_f\} \rangle$.

Example 7. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and consider $\langle \mathtt{ab}, \mathtt{ba} \rangle \in \mathcal{PS}$ (denoting $\mathtt{ab}\Sigma^* \cap \Sigma^*\mathtt{ba}$). Figure 5 shows the DFA for `ab` (left), the KMP DFA for `ab` (middle), and their assembly into a DFA for the prefix/suffix description $\langle \mathtt{ab}, \mathtt{ba} \rangle$. □

Translating back to prefix/suffix form is no harder. Given a trim DFA, one can extract the longest prefix in $O(|Q|)$ time, by following transitions from the start state, stopping when an accept state is reached, or when fan-out exceeds 1. Collecting the longest prefix is slightly more complicated [2] and requires $O(|\delta||Q|)$ time.

Example 8. Let us now combine $\mathcal{PS}$, $\mathcal{CI}$, $\mathcal{SL}$ and $\mathcal{CS}$ analysis, using $\mathcal{RL}$ to provide the precision of a reduced product. In the context of $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, consider the description

$$\langle \langle \mathtt{ab}, \mathtt{ba} \rangle, [\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}], [0, 3], \top_{\mathcal{CS}} \rangle \in (\mathcal{PS} \times \mathcal{CI} \times \mathcal{SL} \times \mathcal{CS})$$

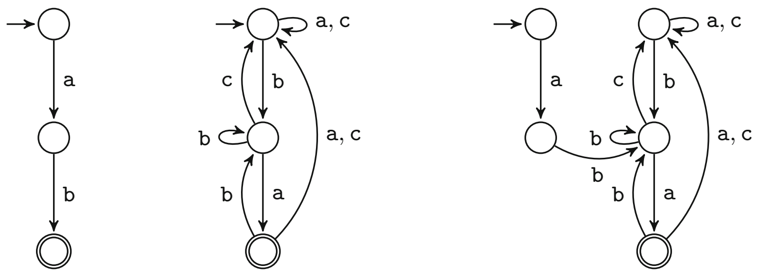
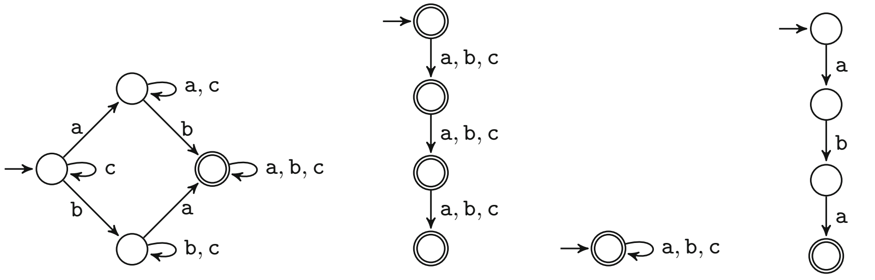


Fig. 5. Constructing a trim DFA for $\langle \mathtt{ab}, \mathtt{ba}\rangle \in \mathcal{PS}$



$\gamma_{\mathcal{CI}}([\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}])$ $\gamma_{\mathcal{SL}}([0, 3])$ $\gamma_{\mathcal{CS}}(\top_{\mathcal{CS}})$ Product

Fig. 6. Simpler descriptions as DFAs. The rightmost DFA is the product of the three on the left, together with the DFA from Fig. 5

The DFA for $\gamma_P S(\langle \mathtt{ab}, \mathtt{ba}\rangle)$ is the one we just calculated (Fig. 5 right). The other three are shown in Fig. 6, together with the product of all four. This product automaton recognises $\{\mathtt{aba}\}$. The refined information is then sent back to the elementary domains, to yield $\langle\langle \mathtt{aba}, \mathtt{aba}\rangle, [\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}], [3, 3], \mathtt{aba}\rangle$. Notice the improved precision, especially for the $\mathcal{CS}$ component which has been strengthened from Σ^* to the singleton $\{\mathtt{aba}\}$. □

The generated product automata are not, in general, minimal, and we do not avoid the cost of minimisation. Keeping automata trim pays off by simplifying some translation operations, and the cost of trimming is low—linear in the size of a DFA.

Let us finally revisit the example that we started from. Our last example shows how simple abstract domains, when synchronised through an expressive reference domain, can yield a precise result. In Sect. 4, the use of widening led to an imprecise result such as $\mathtt{aaa}^*$ to be produced. Here we avoid widening altogether.

Example 9. Consider again the while loop from Example 1. Assume analysis uses the direct product $\mathcal{PS} \times \mathcal{CI} \times \mathcal{SL}$, but utilises $\mathcal{RL}$ as reference domain. At the first entry of the loop, the description of the variable is $\langle\langle \mathtt{aaa}, \mathtt{aaa}\rangle, [\{\mathtt{a}\}, \{\mathtt{a}\}], [3, 3]\rangle$.

At the end of the loop body it is $\langle\langle \mathtt{aaaa}, \mathtt{aaa}\rangle, [\{\mathtt{a}\}, \{\mathtt{a}\}], [4, 4]\rangle$. Analysis finds that the join of the two, $\langle\langle \mathtt{aaa}, \mathtt{aaa}\rangle, [\{\mathtt{a}\}, \{\mathtt{a}\}], [3, 4]\rangle$, is a fixed point, that is, an invariant for the loop entry, and the possible values of the variable at exit are identified precisely as $\gamma(\langle\langle \mathtt{aaaa}, \mathtt{aaa}\rangle, [\{\mathtt{a}\}, \{\mathtt{a}\}], [4, 4]\rangle = \{\mathtt{aaaa}\}$. □

7 Conclusion

From the perspective of abstract interpretation, string analysis is interesting, as it gives rise to a plethora of natural but very different abstract domains, with very different degrees of expressiveness. A particular challenge is how to manage this multitude, that is, how to combine many string abstract domains. We have discussed some approaches to this, paying special attention to the use of the class of regular languages as a "reference" domain, used to mediate between other abstract domains.

There would appear to be considerable scope for improved string analysis. For highly dynamic programming languages, it is likely that combinations of static and dynamic analysis will be needed, to help solve the pressing problems in software reliability and security.

Acknowledgements. I wish to thank my string analysis collaborators: Roberto Amadini, Graeme Gange, François Gauthier, Alexander Jordan, Peter Schachte, Peter Stuckey and Chenyi Zhang. Part of our work was supported by the Australian Research Council through Linkage Project LP140100437.

References

1. Amadini, R., Gange, G., Stuckey, P.J.: Dashed strings for string constraint solving. Artif. Intell. **289**, 103368 (2020)
2. Amadini, R., et al.: Reference abstract domains and applications to string analysis. Fund. Inform. **158**, 297–326 (2018). https://doi.org/10.3233/FI-2018-1650
3. Amadini, R., et al.: Combining string abstract domains for JavaScript analysis: an evaluation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 41–57. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_3
4. Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ECMAScript string manipulation programs. Appl. Sci. **10**, 3525 (2020)
5. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 321–333. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_25
6. Choi, T.-H., Lee, O., Kim, H., Doh, K.-G.: A practical string analyzer by the widening approach. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 374–388. Springer, Heidelberg (2006). https://doi.org/10.1007/11924661_23
7. Cortesi, A., Lauko, H., Olliaro, M., Ročkai, P.: String abstraction for model checking of C programs. In: Biondi, F., Given-Wilson, T., Legay, A. (eds.) SPIN 2019. LNCS, vol. 11636, pp. 74–93. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30923-7_5
8. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 505–521. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_34

9. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw. Pract. Exp. **45**(2), 245–287 (2015)
10. Cousot, P.: Principles of Abstract Interpretation. MIT Press, Cambridge (2021)
11. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM Publications (1977). https://doi.org/10.1145/512950.512973
12. Cousot, P., Cousot, R., Mauborgne, L.: A framework for combining algebraic and logical abstract interpretations, September 2010. Working paper https://hal.inria.fr/inria-00543890
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, pp. 84–97. ACM Publications (1978). https://doi.org/10.1145/512760.512770
14. D'Silva, V.: Widening for automata. Diploma thesis, University of Zürich (2006)
15. Granger, P.: Improving the results of static analyses of programs by local decreasing iterations. In: Shyamasundar, R. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 68–79. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-56287-7_95
16. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_17
17. Karr, M.: Affine relationships among variables of a program. Acta Inform. **6**, 133–151 (1976). https://doi.org/10.1007/BF00268497
18. Kashyap, V., et al.: JSAI: a static analysis platform for JavaScript. In: FSE 2014, pp. 121–132. ACM Publications (2014). https://doi.org/10.1145/2635868.2635904
19. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In: FOOL 2012 (2012). https://doi.org/10.1145/2384616.2384661
20. Madsen, M., Andreasen, E.: String analysis for dynamic field access. In: Cohen, A. (ed.) CC 2014. LNCS, vol. 8409, pp. 197–217. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54807-9_12
21. Park, C., Im, H., Ryu, S.: Precise and scalable static analysis of jQuery using a regular expression domain. In: DSL 2016, pp. 25–36. ACM Publications (2016). https://doi.org/10.1145/2989225.2989228
22. Reps, T., Thakur, A.: Automating abstract interpretation. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 3–40. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_1
23. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Pearson Education, London (2011)
24. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Thomson Course Technology (2012)
25. Thakur, A., Reps, T.: A method for symbolic computation of abstract operations. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 174–192. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_17