An abstract domain for objects in dynamic programming languages

An abstract domain for objects in dynamic programming languages

Vincenzo Arceri, Michele Pasqua, and Isabella Mastroeni

University of Verona, Department of Computer Science, Italy
{vincenzo.arceri | michele.pasqua | isabella.mastroeni}@univr.it

Abstract. Dynamic languages, such as JavaScript, PHP, Python or Ruby, provide a memory model for objects data structures allowing programmers to dynamically create, manipulate, and delete objects' properties. Moreover, in dynamic languages it is possible to access and update properties by using strings: this represents a hard challenge for static analysis. In this paper, we exploit the finite state automata abstract domain, approximating strings, in order to define a novel abstract domain for objects. We design an abstract interpreter useful to analyze objects in a toy language, inspired by real-word dynamic programming languages. We then show, by means of minimal yet expressive examples, the precision of the proposed abstract domain.

1 Introduction

In the last years, dynamic languages such as JavaScript or PHP have gained a huge success in a very wide range of applications. This mainly happened due to the several features that such languages provide to developers, making the writing of programs easier and faster. One of this features is the way strings can be used to interact with programs' objects. Indeed, it is popular, especially in dynamic languages, to create, manipulate, and delete objects' properties at run-time, interacting with them using strings. If, on the one hand, this may help developers to simplify coding and to build applications faster, on the other hand, this may lead to misunderstandings and bugs in the produced code. Furthermore, because of these dynamic features, reasoning about dynamic programs by means of static analysis is quite hard, producing very often imprecise results.

For instance, let us consider the simple yet expressive example reported in Fig. 1, supposing that the value of the `if` guard is statically unknown. The value of `idx` is indeterminate after line 2 and it is updated at each iteration of the while loop (line 6). The `while` guard is also statically unknown and at each iteration we access `obj` with `idx`, incrementally saving the results in `n`. The goal is to statically retrieve the value of `idx` and `n` at the end of the program. It is worth noting that a crucial role here is played by the string abstraction used to approximate the value of `idx`, that is used to access `obj`. Indeed, adopting finite abstract domains, such as [13–15], will lead to infer that `idx` could be any possible string. Consequently, when `idx` is used to access `obj`, in order to

```
1 if (?) { idx = "a"; }
2 else { idx = "b"; };
3 n = 0; obj = new {a:1, aa:2, ab:3, ac:"world"};
4 while (?) {
5     n = n + obj[idx];
6     idx = concat(idx, "a");
7 }
8 obj[idx] = n; // value of idx and n ?
```

Fig. 1: Motivating example.

guarantee soundness, we need to access all properties of `obj`. For instance, we also have to consider the property `ac`, which is never used to access `obj` during the execution of the program. This ends up in an imprecise approximation of `idx` and, in turn, of `n`.

In this paper, we employ a more precise abstraction for string values. In particular, we abstract strings with the finite state automata abstract domain [2], able to derive precise results also when strings are modified in iterative constructs. Then, we define a novel abstract domain for objects, exploiting finite state automata. The idea is to abstract the objects' properties in the same domain used to abstract string values, namely the finite state automata abstract domain. We show that exploiting finite automata to abstract string values and objects properties produces precise results in abstract computations, in particular in objects' properties lookup and in objects' manipulation inside iterative constructs. We will formally present the objects abstract domain in Sec. 3.1.

Moreover, we use strings and objects abstract domains together with integers and booleans abstractions, presenting an abstract interpreter built upon the combination of these domains for a toy language, expressive enough to handle string operations, object declarations, objects' properties lookup and assignments.

2 Background

Notation Given a finite set of symbols Σ, we denote by Σ^* the Kleene-closure of Σ, i.e., the set of all finite sequences of symbols in Σ. We denote an element of Σ^*, called *string*, by $s \in \Sigma^*$. If $s = s_0 s_1 \dots s_n$, then the length of s is $|s| = n+1$ and the element in the i-th position is s_i. Given two strings s and s', ss' is their concatenation. We use the following notations: $\Sigma^i \triangleq \{s \in \Sigma^* \mid |s| = i\}$ and $\Sigma^{<i} \triangleq \bigcup_{0 \leq j < i} \Sigma^j$, for $i \in \mathbb{N}$. We follow [12] for automata notation. A finite state automaton is a tuple $\mathtt{A} = \langle Q, q_0, \Sigma, \delta, F \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is the (finite) alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is a set of final states. In particular, if $\delta \in Q \times \Sigma \to Q$ is a function, then $\mathtt{A}$ is called deterministic finite state automata (DFA). The class of languages recognized by finite state automata is the class of regular languages. Given an automaton $\mathtt{A}$, we denote the language accepted by $\mathtt{A}$ as $\mathscr{L}(\mathtt{A})$. A language $\mathfrak{L}$ is regular iff there exists a finite state automaton $\mathtt{A}$ such that $\mathfrak{L} = \mathscr{L}(\mathtt{A})$. From the Myhill-Nerode theorem [9], we have that for each regular language there exists a unique minimum automaton, i.e., with the minimum number of states, recognizing the language. Given a regular language

$\mathfrak{L}$, we denote by $\mathsf{Min}(\mathfrak{L})$ the minimum DFA $\mathtt{A}$ such that $\mathfrak{L} = \mathscr{L}(\mathtt{A})$. For space limitations, in the following we will refer to finite state automata by using the corresponding regular expressions, which are isomorphic to regular languages and, in turn, to finite state automata. Given two regular expressions $\mathtt{r_1}$ and $\mathtt{r_2}$, we denote by $\mathtt{r_1} \mid\mid \mathtt{r_2}$ the disjunction between $\mathtt{r_1}$ and $\mathtt{r_2}$, by $(\mathtt{r_1})^*$ the Kleene-closure of $\mathtt{r_1}$, and by $(\mathtt{r_1})^+$ the Kleene-closure of $\mathtt{r_1}$ with at least one repetition.

Given a partial function $f \in X \rightharpoonup Y$, we can define an equivalent total function $g \in X \to Y_\uparrow$, where $Y_\uparrow \triangleq Y \cup \{\uparrow\}$ and $\uparrow \notin Y$ denotes indefiniteness. The function g is defined as: $g(x) \triangleq f(x)$ when $f(x)$ is defined, and $g(x) \triangleq \uparrow$ otherwise. When we describe extensionally a function we omit the elements mapped to $\uparrow$, namely $g \in X \to Y_\uparrow$, described as $[x_1 \mapsto y_1 \ x_1 \mapsto y_1 \ \ldots \ x_n \mapsto y_n]$, is such that $g(x_i) = y_i$ for every $i \in \{1, 2, \ldots n\}$ and $g(x_i) = \uparrow$ otherwise.

Abstract interpretation The (concrete) semantic of a program is a representation of all its possible executions by means of a set of mathematical objects. This set is, in general, not computable. It is well known, due to Rice's theorem, that all non trivial properties of the concrete semantics of a program are undecidable. Abstract interpretation is born as a theory for soundly approximating the semantics of discrete dynamic systems. The approximation consists in the observation of the semantics at a specified level of abstraction, focusing only on some important aspects of computations. In this setting, abstract interpretation allows us to compute an abstract semantics of the program, depending on the properties of interest. The approximation is correct by design, in the sense that what holds in the abstract holds also in the concrete (no false negatives).

A theory of domains for abstract interpretation was defined in [7], based on the notion of *Galois insertion*. A Galois insertion (C, α, γ, A) consists of two partially ordered sets $\langle C, \leq_C \rangle$, $\langle A, \leq_A \rangle$ and two monotone functions $\alpha \in C \to A$, $\gamma \in A \to C$ such that for all c in C and a in A it holds: $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ and $\alpha \circ \gamma = \mathsf{id}$ (the identity function $\lambda x \,.\, x$). C is the concrete domain, A is the abstract domain, α is the abstraction function and γ is the concretization function. Sometimes, abstract interpretations are given by means of Galois connections (instead of Galois insertions), relaxing the constraints $\alpha \circ \gamma = \mathsf{id}$. Let $f \in C \to C$ be a function on the concrete domain and $f^\sharp \in A \to A$ be a function on the abstract domain. $f^\sharp$ is a sound (or correct) approximation of f if $f \circ \gamma \leq_C \gamma \circ f^\sharp$ or, equivalently, if $\alpha \circ f \leq_A f^\sharp \circ \alpha$ [7].

Nevertheless, Galois insertions/connections represent the optimal case: sometimes we have to settle for weaker forms of abstract interpretation, as in the case of the Polyhedra abstract domain [8], where we have only the concretization function γ. In this setting, the soundness is expressed just as: $f \circ \gamma \leq_C \gamma \circ f^\sharp$.

Finite state automata abstract domain We report here the finite state automata abstract domain presented in [2], that over-approximates strings as regular languages, represented by the minimum deterministic finite state automata recognizing them [9]. The domain is $\langle \textsc{Dfa}_{/\equiv}, \sqsubseteq_{\textsc{Dfa}}, \sqcup_{\textsc{Dfa}}, \sqcap_{\textsc{Dfa}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\Sigma^*) \rangle$, where $\textsc{Dfa}_{/\equiv}$ is the quotient set of $\textsc{Dfa}$ w.r.t. the equivalence relation induced

$$
\begin{array}{l}
\texttt{a} \in \text{AE} ::= x \mid n \mid \texttt{a + a} \mid \texttt{a - a} \mid \texttt{a * a} \mid \texttt{a / a} \\
\qquad \mid \texttt{length(s)} \mid \texttt{indexOf(}\texttt{s}_1\texttt{,}\texttt{s}_2\texttt{)} \\
\texttt{b} \in \text{BE} ::= x \mid \texttt{true} \mid \texttt{false} \mid \texttt{b \&\& b} \mid \texttt{b || b} \mid \texttt{! b} \mid \texttt{a} < \texttt{a} \\
\qquad \mid \texttt{a == a} \mid \texttt{s == s} \\
\texttt{s} \in \text{SE} ::= x \mid \texttt{"}s\texttt{"} \mid \texttt{substr(s,}\texttt{a}_1\texttt{,}\texttt{a}_2\texttt{)} \mid \texttt{charAt(s,a)} \mid \texttt{concat(}\texttt{s}_1\texttt{,}\texttt{s}_2\texttt{)} \\
\texttt{o} \in \text{OE} ::= \{\ \} \mid \{\ s_0 \texttt{ : } \texttt{e}_0\texttt{, } s_1 \texttt{ : } \texttt{e}_1\texttt{, } \ldots\texttt{, } s_n \texttt{ : } \texttt{e}_n\ \} \\
\texttt{e} \in \text{E} ::= \texttt{a} \mid \texttt{b} \mid \texttt{s} \mid x[\texttt{s}] \\
\texttt{st} \in \text{STMT} ::= \texttt{st ; st} \mid \texttt{skip} \mid x \texttt{ = e} \mid x \texttt{ = new o} \mid x[\texttt{s}] \texttt{ = e} \\
\qquad \mid \texttt{if b \{st\} else \{st\}} \mid \texttt{while b \{st\}}
\end{array}
$$

where $x \in \text{ID}$ (identifiers), $n \in \mathbb{Z}$ and $s, s_0, s_1, \ldots, s_n \in \Sigma^*$

Fig. 2: μJS syntax.

by language equality, $\sqsubseteq_{\text{DFA}}$ is the partial order induced by language inclusion, $\sqcup_{\text{DFA}}$ and $\sqcap_{\text{DFA}}$ are the least upper bound and the greatest lower bound, respectively. The minimum is $\mathsf{Min}(\varnothing)$, corresponding to the automaton recognizing the empty language and the maximum is $\mathsf{Min}(\Sigma^*)$, corresponding to the automaton recognizing any possible string over Σ. We abuse notation by representing equivalence classes in the domain $\text{DFA}_{/\equiv}$ by one of its automaton (usually the minimum), i.e., when we write $\texttt{A} \in \text{DFA}_{/\equiv}$ we mean $[\texttt{A}]_{\equiv}$. Since the domain $\text{DFA}_{/\equiv}$ is infinite, and it is not ACC, i.e., it contains infinite ascending chains, it is equipped with the parametric widening ∇^n_{DFA}. The latter is defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length $n \in \mathbb{N}$, a parameter used for tuning the widening precision [4, 10]. For instance, let us consider the automata $\texttt{A}, \texttt{A}' \in \text{DFA}_{/\equiv}$ recognizing the languages $\mathfrak{L} = \{\epsilon, a\}$ and $\mathfrak{L}' = \{\epsilon, a, aa\}$, respectively. The result of the application of the widening ∇^n_{DFA}, with $n = 1$, is $\texttt{A} \nabla^n_{\text{DFA}} \texttt{A}' = \texttt{A}''$ such that $\mathscr{L}(\texttt{A}'') = \{a^n \mid n \in \mathbb{N}\}$.

The μJS language In this paper, we adopt as core language μJS [2], whose syntax is reported in Fig. 2. This simple toy language is able to express arithmetic (AE), boolean (BE) and string expressions (SE). There is not implicit type conversion, since the problem of analyzing programs with implicit conversions had been already addressed in [1, 2]. Anyway, it is straightforward to merge our analysis with the ones proposed in [1, 2]. In addition, we augment μJS with objects (OE), where an object can be empty, denoted $\{\}$, or a finite set of comma-separated property-expression associations, denoted $\{\ s_0 \texttt{ : } \texttt{e}_0, s_1 \texttt{ : } \texttt{e}_1, \ldots, s_n \texttt{ : } \texttt{e}_n\ \}$.

Concerning the language's semantics, the execution of a μJS program relies on the notion of state, which is composed by environments and heaps, namely states $\sigma \in \text{STATE}$ are pairs $\langle \xi, \rho \rangle \in \text{ENV} \times \text{HEAP}$. An environment is a map from identifiers to values, namely $\text{ENV} \triangleq \text{ID} \to \text{VAL}$, while a heap is a map from addresses to objects, namely $\text{HEAP} \triangleq \text{ADDR} \to \text{OBJ}$. Values v have domain $\text{VAL} \triangleq \text{INT} \cup \text{BOOL} \cup \text{STR} \cup \text{ADDR} \cup \{\uparrow\}$, where $\text{INT} \triangleq \mathbb{Z}$, $\text{BOOL} \triangleq \{\texttt{true}, \texttt{false}\}$, $\text{STR} \triangleq \Sigma^*$, $\text{ADDR} \triangleq \{\underline{n} \mid n \in \mathbb{N}\}$ and $\uparrow$ denotes indefiniteness. An object $o \in \text{OBJ}$ is represented as a map that associates strings to values, namely $\text{OBJ} \triangleq$

$\textsc{Str} \to \textsc{Val}$. It is worth noting that there is no order relation between objects' properties, as it happens in standard programming languages. Environments update is defined as usual: $\xi[x \leftarrow v](y) \triangleq v$ when $x = y$, and $\xi[x \leftarrow v](y) \triangleq \xi(y)$ otherwise. The update for heaps and objects is analogous. The big-step semantics of a $\mu\mathsf{JS}$ program (i.e., a statement) is standard, following [1, 2], and it is captured by the function $[\![\mathsf{st}]\!] \in \textsc{State} \to \textsc{State}$. After showing the concrete semantics of object-related expressions, we will focus on the semantics of assignments, that slightly changes w.r.t. the standard one. As far as expression semantics is concerned, it is also standard [2]. We abuse notation denoting the semantics of an expression as $[\![\mathsf{e}]\!] \in \textsc{State} \to \textsc{Val}$. The evaluation of an object takes each association string-expression and it recursively evaluates the expressions. The result is a map containing the string-value associations.

$$[\![\{s_0 : \mathsf{e}_0, s_1 : \mathsf{e}_1, \ldots s_n : \mathsf{e}_n\}]\!]\sigma \triangleq [s_n \mapsto [\![\mathsf{e}_n]\!]\sigma] \bullet \ldots [s_1 \mapsto [\![\mathsf{e}_1]\!]\sigma] \bullet [s_0 \mapsto [\![\mathsf{e}_0]\!]\sigma]$$

$$\text{where } f \bullet g(s) \triangleq g(s) \text{ if } g(s) \neq \uparrow \wedge f(s) = \uparrow \text{ and } f \bullet g(s) \triangleq f(s) \text{ otherwise}$$

For example, the expression `{a:1, b:length("foo"), c:5+3}` evaluates to the object $[\mathtt{a} \mapsto \mathtt{1}\ \mathtt{b} \mapsto \mathtt{3}\ \mathtt{c} \mapsto \mathtt{8}]$. Following the JavaScript semantics, it is worth noting that, for instance, `{a:1, a:2}` evaluates to $[\mathtt{a} \mapsto \mathtt{2}]$, saving only the last association with the same property `a`. The semantics of objects' properties lookup checks whether the object contains a string-value association, where the string corresponds to the property. Hence, its definition is the following, supposing that $[\![\mathsf{s}]\!]\langle\xi, \rho\rangle = s \in \textsc{Str}$:

$$[\![x[\mathsf{s}]]\!]\langle\xi, \rho\rangle \triangleq \rho([\![x]\!]\langle\xi, \rho\rangle)(s) \text{ if } [\![x]\!]\langle\xi, \rho\rangle \in \textsc{Addr} \text{ and } [\![x[\mathsf{s}]]\!]\langle\xi, \rho\rangle \triangleq \uparrow \text{ otherwise}$$

In our core language, we allow only to access already stored objects (condition $[\![x]\!]\langle\xi, \rho\rangle \in \textsc{Addr}$). Moreover, it is worth noting that when we try to access a property s not present in the object pointed by x, then $\rho([\![x]\!]\langle\xi, \rho\rangle)(s)$ returns $\uparrow$.

The semantics of generic statements is standard, here we explain only the semantics for assignments, which is also used for objects allocation and update. We have three cases: x `= e`, where `e` evaluates to a value; x `= new o`, where `o` evaluates to an object; x[s] `= e`, where `s` evaluates to a string and `e` evaluates to a value. In the first case, we only update the environment, following the typical concrete semantics of assignments. In the second case, we need to allocate the object into a new address which x will point to. Then, both environment and heap are properly updated. In the third case, we update the object pointed by x in the heap. Formally, let $\underline{n} \in \textsc{Addr}$ be a fresh, i.e., not-used, address:

$$[\![x = \mathsf{e}]\!]\langle\xi, \rho\rangle \triangleq \langle\xi[x \leftarrow [\![\mathsf{e}]\!]\langle\xi, \rho\rangle], \rho\rangle$$

$$[\![x = \texttt{new o}]\!]\langle\xi, \rho\rangle \triangleq \langle\xi[x \leftarrow \underline{n}], \rho[\underline{n} \leftarrow [\![\mathsf{o}]\!]\langle\xi, \rho\rangle]\rangle$$

$$[\![x[\mathsf{s}] = \mathsf{e}]\!]\langle\xi, \rho\rangle \triangleq \big\langle\xi, \rho\big[\xi(x) \mapsto \rho(\xi(x))[[\![\mathsf{s}]\!]\langle\xi, \rho\rangle \leftarrow [\![\mathsf{e}]\!]\langle\xi, \rho\rangle]\big]\big\rangle$$

As a final remark, we point out that in our extension of $\mu\mathsf{JS}$ we do not model features such as pointer arithmetic, objects comparisons and implicit type conversion (e.g., `x = 1 ; y = true ; z = x == y` leads to an error).

3 Static Analysis of μJS

In order to reason about a μJS program we need to take into account all its possible executions, by means of the so called collecting semantics. Our concrete collecting semantics is a classic post-conditions semantics, computing state invariants at every statement. It is defined as the direct-image lift of the big-step semantics of μJS, hence it is a function from sets of states to sets of states. We denote by $(\!|\mathsf{st}|\!) \in \wp(\textsc{State}) \to \wp(\textsc{State})$ the concrete collecting semantics. For instance, the collecting semantics for assignments involving expressions, is defined as $(\!|x = \mathsf{e}|\!)X \triangleq \{[\![x = \mathsf{e}]\!]\sigma \mid \sigma \in X\}$. The semantics is similarly defined for the other constructs and for assignments involving objects. In particular, the collecting semantics for conditionals and loops is defined, as usual, as:

$$(\!|\texttt{if b } \{\, \mathsf{st}_1 \,\} \texttt{ else } \{\, \mathsf{st}_2 \,\}|\!)X \triangleq (\!|\mathsf{st}_1|\!)\mathsf{filter}_{\mathsf{b}}(X) \cup (\!|\mathsf{st}_2|\!)\mathsf{filter}_{!\mathsf{b}}(X)$$

$$(\!|\texttt{while b } \{\, \mathsf{st} \,\}|\!)X \triangleq \mathsf{filter}_{!\mathsf{b}}\big(\mathsf{lfp}\ \lambda T\,.\, X \cup (\!|\mathsf{st}|\!)\mathsf{filter}_{\mathsf{b}}(T)\big)$$

Here $\mathsf{filter}_{\mathsf{b}} \in \wp(\textsc{State}) \to \wp(\textsc{State})$ is a filtering function, namely it filters out the states that do not fulfill the boolean condition b. Unfortunately, we are not able to compute the concrete collecting semantics, since it is an infinite mathematical object. Hence, in order to perform static analysis, we approximate the collecting semantics, following the abstract interpretation framework. In order to make the computation, and in turn the analysis, feasible we need an abstract semantics $(\!|\mathsf{st}|\!)^\sharp$ computer-representable and ensuring termination of the analysis. Ideally, the abstract semantics computes on abstract states in $\textsc{State}^\sharp$, approximations of the concrete ones. Precisely, $\textsc{State}^\sharp$ is an approximation of $\wp(\textsc{State})$, with a concretization $\gamma \in \textsc{State}^\sharp \to \wp(\textsc{State})$. The abstract semantics must be sound, meaning that what we prove in the abstract also holds for the concrete semantics. Put it in abstract interpretation terms, this means that for every $\sigma^\sharp \in \textsc{State}^\sharp$ we have that $(\!|\mathsf{st}|\!)\gamma(\sigma^\sharp) \subseteq \gamma((\!|\mathsf{st}|\!)^\sharp\sigma^\sharp)$. Before defining the abstract semantics, we focus on the *objects abstract domain*, which is the core of our paper and it is used to represent, possibly infinite, sets of concrete objects.

3.1 Abstract Objects

As previously introduced, in order to make the analysis feasible, we need to finitely represent an infinite set of states. We start here with our representation of infinite sets of objects, namely we define an abstract domain approximating $\wp(\textsc{Obj})$. First, we have a non-relational abstraction between objects-properties and values, i.e., we abstract $\wp(\textsc{Obj})$ in $\wp(\textsc{Str}) \to \wp(\textsc{Val})$.

Then we abstract $\wp(\textsc{Str})$ with the automata domain, while for $\wp(\textsc{Val})$ we abstract separately each type of values in its abstract domain, obtaining the product domain $\textsc{Val}^\sharp \triangleq \textsc{Int}^\sharp \times \textsc{Bool}^\sharp \times \textsc{Str}^\sharp \times \wp(\textsc{Addr}^\sharp) \times \{\texttt{def}, \texttt{?}\}$. For numeric values we can use any non-relational domain, such as integer intervals. $\textsc{Bool}^\sharp \triangleq \{\bot, \texttt{tt}, \texttt{ff}, \top\}$ is isomorphic to $\wp(\textsc{Bool})$ and for sets of strings we use the automata domain, namely $\textsc{Str}^\sharp \triangleq \textsc{Dfa}_{/\equiv}$. As we will see in the next subsection, we approximate heaps with an allocation-site abstraction of $\textsc{Addr}$.

So, possibly infinite sets of addresses are abstracted into finite sets of allocation sites, namely $\textsc{Addr}^\sharp \triangleq \textsc{Lines}$, where Lines is the finite set of lines of code of a given program. Here we abstract $\wp(\textsc{Addr})$ in $\wp(\textsc{Addr}^\sharp)$, since an abstract object could have more than one allocation site. The domain $\{\texttt{def}, \texttt{?}\}$ is isomorphic to $\wp(\{\uparrow\})$ and `def` represents the absence of indefiniteness while `?` represents potential indefiniteness. An abstract value $v^\sharp = \langle i^\sharp, b^\sharp, s^\sharp, A, u^\sharp\rangle \in \textsc{Val}^\sharp$ represents the union of the elements taken from every single-type abstraction:

$$\gamma_{\mathrm{V}}(v^\sharp) = \gamma_{\mathrm{I}}(i^\sharp) \cup \gamma_{\mathrm{B}}(b^\sharp) \cup \gamma_{\mathrm{S}}(s^\sharp) \cup \bigcup_{l \in A} \gamma_{\mathrm{A}}(l) \cup \gamma_{\mathrm{U}}(u^\sharp)$$

where γ_{I} is the concretization defined in the numerical non-relational domain, $\gamma_{\mathrm{B}}(\bot) \triangleq \varnothing$, $\gamma_{\mathrm{B}}(\texttt{tt}) \triangleq \{\texttt{true}\}$, $\gamma_{\mathrm{B}}(\texttt{ff}) \triangleq \{\texttt{false}\}$, $\gamma_{\mathrm{B}}(\top) \triangleq \{\texttt{true}, \texttt{false}\}$, γ_{S} is the concretization for the automata domain (i.e., the language recognized by the given automaton) and $\gamma_{\mathrm{U}}(\texttt{def}) = \varnothing$, $\gamma_{\mathrm{U}}(\texttt{?}) = \{\uparrow\}$. The concretization for addresses γ_{A} will be introduced in Sec. 3.2, when we deal with abstract heaps. Briefly, the concretization of a given allocation site is the set of all possible addresses that can be allocated at that line of code. The abstract join $\sqcup^\sharp_{\mathrm{V}}$ and the partial order $\sqsubseteq^\sharp_{\mathrm{V}}$ for $\textsc{Val}^\sharp$ are defined pointwise.

The partial order $\sqsubseteq^\sharp_{\mathrm{O}}$ for $\textsc{Obj}^\sharp$ is the pointwise ordering between functions, i.e., $o^\sharp_1 \sqsubseteq^\sharp_{\mathrm{O}} o^\sharp_2 \triangleq (\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv} \,.\, o^\sharp_1(\mathtt{A}) \sqsubseteq^\sharp_{\mathrm{V}} o^\sharp_2(\mathtt{A}))$. This order is not optimal but it does not harm the analysis since, as we can see in Sec. 3.1, the order can be strengthen. Analogously, the join for $\textsc{Obj}^\sharp$ is defined as $\bigsqcup^\sharp_{\mathrm{O}} X \triangleq \lambda \mathtt{A} \,.\, \bigsqcup^\sharp_{\mathrm{V}} \{o^\sharp(\mathtt{A}) \mid o^\sharp \in X\}$. It is straightforward to see that $\langle \textsc{Obj}^\sharp, \sqsubseteq^\sharp_{\mathrm{O}}\rangle$ is a lattice, with minimum mapping every automaton to the tuple composed by the minimum of each value-type domain, and maximum mapping every automaton to the tuple composed by the maximum of each value-type domain. The concretization $\gamma_{\mathrm{O}} \in \textsc{Obj}^\sharp \to \wp(\textsc{Obj})$ is defined as:

$$\gamma_{\mathrm{O}}(o^\sharp) \triangleq$$
$$\left\{ o \in \textsc{Obj} \;\middle|\; \forall s \in \textsc{Str}\, \exists \mathtt{A} \in \textsc{Dfa}_{/\equiv} \,.\, (s \in \gamma_{\mathrm{S}}(\mathtt{A}) \wedge (o(s) \in \gamma_{\mathrm{V}}(o^\sharp(\mathtt{A})) \vee \varnothing(s) = \uparrow)) \right\}$$

In order to optimize the implementation of the abstract domain, we represent singleton sets of strings as they are, instead of converting them into automata. Indeed, it is worth noting that we can partition the finite state automata abstract domain as $\textsc{Dfa}_{/\equiv} = \textsc{Dfa}^{\mathbf{1}}_{/\equiv} \cup \textsc{Dfa}^{\omega}_{/\equiv}$, where $\textsc{Dfa}^{\mathbf{1}}_{/\equiv} \triangleq \{\mathtt{A} \in \textsc{Dfa}_{/\equiv} \mid |\mathscr{L}(\mathtt{A})| = 1\}$, namely the set of finite state automata that recognize singleton languages, and $\textsc{Dfa}^{\omega}_{/\equiv} \triangleq \textsc{Dfa}_{/\equiv} \setminus \textsc{Dfa}^{\mathbf{1}}_{/\equiv}$, namely the set of finite state automata that recognizes languages of size 0 or size greater than 1 (possibly infinite). Clearly $\textsc{Dfa}^{\mathbf{1}}_{/\equiv}$ is isomorphic to Str, hence we can equivalently define abstract objects as maps in $\textsc{Obj}^\sharp \triangleq (\textsc{Str} \cup \textsc{Dfa}^{\omega}_{/\equiv}) \to \textsc{Val}^\sharp$.

In order to show how our objects abstract domain works, we consider a simple yet expressive μJS example (Fig. 3, where we suppose that the boolean guards of `while` and `if` statements are statically unknown). The fragment declares the object `o` at line 1, and its abstract value at lines 1-7 is reported in Fig. 4a. Then, it indefinitely iterates over the string variable `idx` at lines 3-6 appending either the strings `"x"` or `"y"`. Finally, `idx` is used to access the object `o` at line 7. Let us suppose to statically analyze the above program with the abstract

```
1  o = new {x:1, y:2, z:3};
2  idx = "x";
3  while (?) {
4      if (?) { idx = concat(idx, "x") }
5      else { idx = concat(idx, "y") }
6  };
7  o[idx] = 7;
```

Fig. 3: μJS program example.

(a) $\left[\begin{array}{c} \texttt{x} \mapsto [1,1] \\ \texttt{y} \mapsto [2,2] \\ \hline \texttt{z} \mapsto [3,3] \\ \hline - \end{array}\right]$ (b) $\left[\begin{array}{c} \texttt{x} \mapsto [1,1] \\ \texttt{y} \mapsto [2,2] \\ \hline \texttt{z} \mapsto [3,3] \\ \hline \texttt{x}(\texttt{x} \mid\mid \texttt{y})^* \mapsto [7,7] \end{array}\right]$ (c) $\left[\begin{array}{c} \texttt{x} \mapsto [1,7] \\ \texttt{y} \mapsto [2,2] \\ \hline \texttt{z} \mapsto [3,3] \\ \hline \texttt{x}(\texttt{x} \mid\mid \texttt{y})^+ \mapsto [7,7] \end{array}\right]$

Fig. 4: (a) Abstract value of `o` after line 1 of the fragment reported in Fig. 3 (b) Abstract value of `o` after line 6. (c) Normal form of `o` after line 6.

domain previously presented. Since the number of iterations of the `while`-loop is statically unknown, the computation of the value of `idx`, abstracted as a finite state automaton, may diverge. In order to enforce termination, the automata widening ∇^n_{DFA} is applied. Tuning ∇^n_{DFA} with $n = 3$, the abstract value of `idx` at line 7, after the `while` computation, corresponds to the automaton expressed by the regular expression $\texttt{x}(\texttt{x} \mid\mid \texttt{y})^*$. Since `idx` does not represent just a single string, when we analyze `o[idx]` we may have to overwrite an object property (e.g., `x`) and add new properties to `o` (e.g., `xyy`). Since the abstract value of `idx` expresses an infinite number of object properties, we call this property *summary property*. The abstract value after line 6 is depicted in Fig. 4b, where the summary property $\texttt{x}(\texttt{x} \mid\mid \texttt{y})^*$ is added to the object reported in Fig. 4a. Note that in the abstract object updated at line 7, the abstract properties `x` and $\texttt{x}(\texttt{x} \mid\mid \texttt{y})^*$ share the common concrete property `x`. In particular, the value of `o["x"]` may be either 1 or 7. We aim at an objects' representation where every property does not share any property with the others, namely when objects are in normal form.

Normalization We now formally define the notion of abstract object normal form. Given an abstract object $o^\sharp \in \textsc{Obj}^\sharp$, we denote by $\mathsf{props}(o^\sharp) \subseteq \textsc{Str}^\sharp$ the set of its abstract properties, namely the properties which are not undefined. We remind that $\textsc{Str}^\sharp$ is the optimized version of the automata domain, i.e., $\textsc{Str}^\sharp = \textsc{Str} \cup \textsc{Dfa}^\omega_{/\equiv}$. Formally, $\mathsf{props}(o^\sharp) \triangleq \{p \in \textsc{Str}^\sharp \mid o^\sharp(p) = \langle i^\sharp, b^\sharp, s^\sharp, A, u^\sharp \rangle \wedge u^\sharp = \texttt{def}\}$. Abstract properties represent sets of concrete properties. Hence, given $p \in \mathsf{props}(o^\sharp)$, we abuse notation denoting by $\mathscr{L}(p)$ the language of the concrete properties captured by p. $\mathscr{L}(p)$ is the language recognized by the corresponding automaton, when $p \in \textsc{Dfa}^\omega_{/\equiv}$ and it is the language $\{p\}$ when $p \in \textsc{Str}$.

Definition 1 (Abstract object normal form). *An abstract object $o^\sharp \in \textsc{Obj}^\sharp$ is in normal form when:*

$$\forall p \in \mathsf{props}(o^\sharp)\,.\,|\mathscr{L}(p)| \in \{1, \omega\} \wedge \forall p_1, p_2 \in \mathsf{props}(o^\sharp)\,.\,\mathscr{L}(p_1) \cap \mathscr{L}(p_2) = \varnothing$$

Algorithm 1: $\mathsf{Norm} \in \textsc{Obj}^\sharp \to \textsc{Obj}^\sharp$ algorithm

Data: $o^\sharp \in \textsc{Obj}^\sharp$
Result: $\mathsf{Norm}(o^\sharp)$
1 **foreach** $p \in \mathsf{props}(o^\sharp)$ **do**
2 $v^\sharp \leftarrow o^\sharp(p)$;
3 **if** $|\mathscr{L}(p)| \notin \{1, \omega\}$ **then**
4 remove p from $o^\sharp$;
5 **foreach** $s \in \mathscr{L}(p)$ **do**
6 $o^\sharp \leftarrow o^\sharp \bullet^\sharp [s \mapsto v^\sharp]$;
7 **foreach** $p_1 \in \mathsf{props}(o^\sharp)$ **do**
8 $v_1^\sharp \leftarrow o^\sharp(p_1)$; remove p_1 from $o^\sharp$; $\mathsf{normalized} \leftarrow \texttt{false}$;
9 **foreach** $p_2 \in \mathsf{props}(o^\sharp)$ **do**
10 $v_2^\sharp \leftarrow o^\sharp(p_2)$;
11 **if** $p_1 \sqcap_{\mathrm{S}}^\sharp p_2 \neq \mathsf{Min}(\varnothing) \wedge p_1 \neq p_2$ **then**
12 $\mathsf{normalized} \leftarrow \texttt{true}$;
13 $o^\sharp \leftarrow o^\sharp \bullet^\sharp [p_1 \sqcap_{\mathrm{S}}^\sharp p_2 \mapsto o^\sharp(p_1 \sqcap_{\mathrm{S}}^\sharp p_2) \sqcup_{\mathrm{V}}^\sharp v_1^\sharp \sqcup_{\mathrm{V}}^\sharp v_2^\sharp]$;
14 $o^\sharp \leftarrow o^\sharp \bullet^\sharp [p_1 \smallsetminus_{\mathrm{S}}^\sharp p_2 \mapsto o^\sharp(p_1 \smallsetminus_{\mathrm{S}}^\sharp p_2) \sqcup_{\mathrm{V}}^\sharp v_1^\sharp]$;
15 $o^\sharp \leftarrow o^\sharp \bullet^\sharp [p_2 \smallsetminus_{\mathrm{S}}^\sharp p_1 \mapsto o^\sharp(p_2 \smallsetminus_{\mathrm{S}}^\sharp p_1) \sqcup_{\mathrm{V}}^\sharp v_2^\sharp]$;
16 remove p_2 from $o^\sharp$;
17 **if** $!\mathsf{normalized}$ **then** $o^\sharp \leftarrow o^\sharp \bullet^\sharp [p_1 \mapsto v_1^\sharp]$;
18 **return** $o^\sharp$;

Informally, we say that an abstract object is in normal form when each property p represents only a single string (i.e., $|\mathscr{L}(p)| = 1$) or an infinite language (i.e., $|\mathscr{L}(p)| = \omega$) and it does not share any concrete property with other abstract properties. Hence, a normal form abstract object has two kind of properties: p is a *non-summary property*, if $|\mathscr{L}(p)| = 1$, and p is a *summary property*, if $|\mathscr{L}(p)| = \omega$. For instance, the abstract object in Fig. 4a is in normal form, since any abstract property expresses concrete properties that are not expressed by other abstract properties and it only contains non-summary properties. Instead, the abstract object in Fig. 4b is not in formal form, despite it has only summary and non-summary properties, since the string `x` is expressed by the non-summary property `x` and by the summary property $\texttt{x}(\texttt{x} \mid\mid \texttt{y})^*$.

During abstract computations, it may happen that abstract objects are not in normal form, so we need to normalize them. We rely on the function $\mathsf{Norm} \in \textsc{Obj}^\sharp \to \textsc{Obj}^\sharp$ that normalizes an abstract object and its behaviour is captured by the algorithm reported by Alg. 1, where $o_1^\sharp \bullet^\sharp o_2^\sharp$ is defined as:

$$\text{let } \langle i_1^\sharp, b_1^\sharp, s_1^\sharp, A_1, u_1^\sharp \rangle = o_1^\sharp(p), \langle i_2^\sharp, b_2^\sharp, s_2^\sharp, A_2, u_2^\sharp \rangle = o_2^\sharp(p) \text{ in}$$
$$o_1^\sharp \bullet^\sharp o_2^\sharp(p) \triangleq o_2^\sharp(p) \text{ if } u_2^\sharp \neq \texttt{?} \wedge u_1^\sharp = \texttt{?} \text{ and } o_1^\sharp \bullet^\sharp o_2^\sharp(p) \triangleq o_1^\sharp(p) \text{ otherwise}$$

In the algorithm, the operators $\sqcap_{\mathrm{S}}^\sharp$ and $\smallsetminus_{\mathrm{S}}^\sharp$ are the operators $\sqcap_{\textsc{Dfa}}$ and $\smallsetminus_{\textsc{Dfa}}$, respectively, of the automata domain adapted to its optimized versions $\textsc{Str} \cup \textsc{Dfa}^\omega_{/\equiv}$. The first part of Alg. 1, namely lines 1-6, checks if any property of $o^\sharp$ is summary or non-summary. If it finds a property p such that $|\mathscr{L}(p)| \notin \{1, \omega\}$ then the algorithm first remove that property from the object, and then looks at its language (that is finite) and adds any single property captured

by p with its old corresponding value. All the automata operations reported above and the check $|\mathscr{L}(p)| \notin \{1, \omega\}$ can be performed with linear complexity w.r.t. the number of state of the automata. For example, let consider the object $[\mathtt{x} \mid\mid \mathtt{y} \mapsto [5,5]]$, the algorithm returns as result the normal form abstract object $[\mathtt{x} \mapsto [5,5], \mathtt{y} \mapsto [5,5]]$. The idea of the second part of Alg. 1 (lines 7-17) is to check, for any $p_1 \in \mathsf{props}(o^\sharp)$, if it shares at least a concrete property with any other $p_2 \in \mathsf{props}(o^\sharp)$ (lines 11-16). This boils down to check whether the intersection between p_1 and p_2 is not empty. If so, three new abstract properties are created in $o^\sharp$ (note that p_1 is removed at line 8 and p_2 will be removed at line 16). In particular:

- the property $p_1 \sqcap^\sharp_{\mathrm{S}} p_2$ points to the join of the previous values of p_1 and p_2 and the previous value (if present) of $p_1 \sqcap^\sharp_{\mathrm{S}} p_2$ in $o^\sharp$ (line 13);
- the property $p_1 \smallsetminus^\sharp_{\mathrm{S}} p_2$ points to the join of the previous value of p_1 and the previous value (if present) of $p_1 \smallsetminus^\sharp_{\mathrm{S}} p_2$ in $o^\sharp$ (line 14);
- the property $p_2 \smallsetminus^\sharp_{\mathrm{S}} p_1$ points to the join of the previous value of p_2 and the previous value (if present) of $p_2 \smallsetminus^\sharp_{\mathrm{S}} p_1$ in $o^\sharp$ (line 15);

Otherwise, if p_1 does not share any property with other abstract properties of $o^\sharp$, the association $\langle p_1, o^\sharp(p_1)\rangle$ is simply added to $o^\sharp$ (line 17). For example, let us consider again the abstract object reported in Fig. 4b. The result obtained by applying Alg. 1 is the abstract object reported in Fig. 4c.

Proposition 1. *Given $o^\sharp \in \textsc{Obj}^\sharp$, the abstract object $\mathsf{Norm}(o^\sharp)$, computed by Alg. 1, is in normal form (Def. 1). Moreover, we have that $\gamma_{\mathrm{O}}(o^\sharp) = \gamma_{\mathrm{O}}(\mathsf{Norm}(o^\sharp))$.*

As we have mentioned in Sect. 3, normalization strengthens the abstract order between objects. For example, the objects $[\mathtt{a} \mapsto [1,1], \mathtt{b} \mapsto [1,1]]$ and $[\mathtt{a} \mid\mid \mathtt{b} \mapsto [1,2]]$ are not comparable, but, if we normalize the second object (i.e., in $[\mathtt{a} \mapsto [1,2], \mathtt{b} \mapsto [1,2]]$), then we have $[\mathtt{a} \mapsto [1,1], \mathtt{b} \mapsto [1,1]] \sqsubseteq^\sharp_{\mathrm{O}} \mathsf{Norm}([\mathtt{a} \mid\mid \mathtt{b} \mapsto [1,2]])$.

3.2 Abstract Semantics

Abstract states in $\textsc{State}^\sharp$ are composed by abstract environments and abstract heaps, so we have an abstraction from $\wp(\textsc{Env} \times \textsc{Heap})$ to $\wp(\textsc{Env}) \times \wp(\textsc{Heap})$. As an abstract representation of the heap, we use a classic allocation-site abstraction of $\textsc{Addr}$ [16]. Possibly infinite sets of addresses are abstracted into finite sets of allocation sites, namely $\textsc{Addr}^\sharp \triangleq \textsc{Lines}$, where $\textsc{Lines}$ is the finite set of lines of code of a given program. Given a μJS program, we suppose to have a labeling assigning to each statement of the program a unique line of code (a natural number). Then, we define two functions, $\mathsf{line} \in \textsc{stmt} \to \textsc{Lines}$ and $\mathsf{code} \in \textsc{Lines} \to \textsc{stmt}$, returning the line of code of a given statement and the statement assigned to a given line of code, respectively. The concretization is

$$\gamma_{\mathrm{A}}(l) \triangleq \left\{ \underline{n} \in \textsc{Addr} \;\middle|\; \exists \langle \xi, \rho\rangle \in \textsc{State}\,.\, \begin{array}{l} [\![\mathsf{code}(l)]\!]\langle \xi, \rho\rangle = \langle \xi', \rho'\rangle \wedge \\ \rho(\underline{n}) = \lambda s\,.\uparrow \wedge\, \rho'(\underline{n}) \neq \lambda s\,.\uparrow \end{array} \right\}$$

meaning that the concretization of a given allocation site l is the set of all possible addresses that can be allocated at that line of code. An abstract heap is a map

$$(\!|n|\!)^\sharp_{\mathrm{A}} \triangleq \alpha_{\mathrm{I}}(\{n\}) \quad (\!|x|\!)^\sharp_{\mathrm{A}}\langle\xi^\sharp,\rho^\sharp\rangle \triangleq \xi^\sharp(x) \quad (\!|\mathsf{a}_1 + \mathsf{a}_2|\!)^\sharp_{\mathrm{A}}\sigma^\sharp \triangleq (\!|\mathsf{a}_1|\!)^\sharp_{\mathrm{A}}\sigma^\sharp +^{\mathrm{I}} (\!|\mathsf{a}_2|\!)^\sharp_{\mathrm{A}}\sigma^\sharp$$
$$(\!|\mathtt{true}|\!)^\sharp_{\mathrm{B}}\sigma^\sharp \triangleq \mathtt{tt} \quad (\!|x|\!)^\sharp_{\mathrm{B}}\langle\xi^\sharp,\rho^\sharp\rangle \triangleq \xi^\sharp(x) \quad (\!|\mathsf{b}_1 \,||\, \mathsf{b}_2|\!)^\sharp_{\mathrm{B}}\sigma^\sharp \triangleq (\!|\mathsf{b}_1|\!)^\sharp_{\mathrm{B}}\sigma^\sharp \sqcup^\sharp_{\mathrm{B}} (\!|\mathsf{b}_2|\!)^\sharp_{\mathrm{B}}\sigma^\sharp$$
$$(\!|\{\}|\!)^\sharp_{\mathrm{O}}\sigma^\sharp \triangleq \lambda p\,.\,\langle\bot,\bot,\mathsf{Min}(\varnothing),\varnothing,\mathtt{?}\rangle$$
$$(\!|\{s_0 : \mathsf{e}_0, s_1 : \mathsf{e}_1, \ldots s_n : \mathsf{e}_n\}|\!)^\sharp_{\mathrm{O}}\sigma^\sharp \triangleq$$
$$[s_n \mapsto (\!|\mathsf{e}_n|\!)^\sharp_{\mathrm{E}}\sigma^\sharp] \bullet^\sharp \ldots [s_1 \mapsto (\!|\mathsf{e}_1|\!)^\sharp_{\mathrm{E}}\sigma^\sharp] \bullet^\sharp [s_0 \mapsto (\!|\mathsf{e}_0|\!)^\sharp_{\mathrm{E}}\sigma^\sharp]$$
$$(\!|x[\mathsf{s}]|\!)^\sharp_{\mathrm{E}}\langle\xi^\sharp,\rho^\sharp\rangle \triangleq$$
$$\begin{cases} \bigsqcup^\sharp_{\mathrm{V}}\{\rho^\sharp(l)(p) \mid l \in \xi^\sharp(x) \wedge \mathscr{L}(p) \cap \mathscr{L}((\!|\mathsf{s}|\!)^\sharp_{\mathrm{S}}\langle\xi^\sharp,\rho^\sharp\rangle) \neq \varnothing\} & \text{if } \xi^\sharp(x) \in \textsc{Addr}^\sharp \\ \langle\bot,\bot,\mathsf{Min}(\varnothing),\varnothing,\mathtt{?}\rangle \end{cases}$$
$$\mathsf{filter}^\sharp_{\mathtt{true}}(\sigma^\sharp) \triangleq \sigma^\sharp \quad \mathsf{filter}^\sharp_x(\langle\xi^\sharp,\rho^\sharp\rangle) \triangleq \begin{cases} \langle\xi^\sharp,\rho^\sharp\rangle & \text{if } \xi^\sharp(x) \in \{\mathtt{tt},\top\} \\ \sigma^\sharp_\bot & \text{otherwise} \end{cases}$$
$$\mathsf{filter}^\sharp_{\mathsf{b}_1||\mathsf{b}_2}(\sigma^\sharp) \triangleq \mathsf{filter}^\sharp_{\mathsf{b}_1}(\sigma^\sharp) \sqcup^\sharp \mathsf{filter}^\sharp_{\mathsf{b}_2}(\sigma^\sharp) \quad \mathsf{filter}^\sharp_{\mathtt{false}}(\sigma^\sharp) \triangleq \sigma^\sharp_\bot$$

Fig. 5: Abstract semantics for expressions and objects and the abstract filter

associating abstract addresses, i.e., lines of code, to abstract objects, namely $\textsc{Heap}^\sharp \triangleq \textsc{Addr}^\sharp \to \textsc{Obj}^\sharp$. As we have already seen, an abstract object is a map associating an automaton with an abstract value.

For what concerns environments, we consider a non-relational abstraction, approximating every identifier separately. This means that we abstract from $\wp(\textsc{Id} \to \textsc{Val})$ to $\textsc{Id} \to \wp(\textsc{Val})$. Abstract environments are maps from identifiers to abstract values, namely $\textsc{Env}^\sharp \triangleq \textsc{Id} \to \textsc{Val}^\sharp$, exploiting the abstraction between $\wp(\textsc{Val})$ and $\textsc{Val}^\sharp$ we have introduced in the previous subsection. Finally, abstract states are, as in the concrete, pairs of abstract environments and abstract heaps, namely $\textsc{State}^\sharp \triangleq \textsc{Env}^\sharp \times \textsc{Heap}^\sharp$. The definition of the abstract join $\sqcup^\sharp$ and the partial order $\sqsubseteq^\sharp$ for $\textsc{State}^\sharp$ is straightforward.

The abstract semantics is then a function $(\!|\mathsf{st}|\!)^\sharp \in \textsc{State}^\sharp \to \textsc{State}^\sharp$, computing on abstract states. It relies on the abstract semantics for expressions $(\!|\mathsf{e}|\!)^\sharp_{\mathrm{E}} \in \textsc{State}^\sharp \to \textsc{Val}^\sharp$, on the abstract semantics for objects $(\!|\mathsf{o}|\!)^\sharp_{\mathrm{O}} \in \textsc{State}^\sharp \to \textsc{Obj}^\sharp$ and on the abstract filtering function $\mathsf{filter}^\sharp_{\mathsf{b}} \in \textsc{State}^\sharp \to \textsc{State}^\sharp$[1]. All of them must be sound w.r.t. their concrete counterparts, namely $(\!|\mathsf{e}|\!)\gamma(\sigma^\sharp) \subseteq \gamma_{\mathrm{V}}((\!|\mathsf{e}|\!)^\sharp_{\mathrm{E}}\sigma^\sharp)$, $(\!|\mathsf{o}|\!)\gamma(\sigma^\sharp) \subseteq \gamma_{\mathrm{O}}((\!|\mathsf{o}|\!)^\sharp_{\mathrm{O}}\sigma^\sharp)$ and $\mathsf{filter}_{\mathsf{b}}(\gamma(\sigma^\sharp)) \subseteq \gamma(\mathsf{filter}^\sharp_{\mathsf{b}}(\sigma^\sharp))$, for every $\sigma^\sharp \in \textsc{State}^\sharp$. In Fig. 5 we have a part of the definition of the abstract semantics for expressions and objects and the abstract filter, where $\sigma^\sharp_\bot$ is the minimum of the lattice $\langle\textsc{State}^\sharp,\sqsubseteq^\sharp\rangle$. The abstract semantics for statements is quite standard:

$$(\!|\mathsf{st}_1\ ;\ \mathsf{st}_2|\!)^\sharp\sigma^\sharp \triangleq (\!|\mathsf{st}_2|\!)^\sharp((\!|\mathsf{st}_1|\!)^\sharp\sigma^\sharp) \quad (\!|\mathtt{skip}|\!)^\sharp\sigma^\sharp \triangleq \sigma^\sharp$$
$$(\!|\mathtt{if}\ \mathsf{b}\ \{\,\mathsf{st}\,\}\ \mathtt{else}\ \{\,\mathsf{st}\,\}|\!)^\sharp\sigma^\sharp \triangleq (\!|\mathsf{st}_1|\!)^\sharp\mathsf{filter}^\sharp_{\mathsf{b}}(\sigma^\sharp) \sqcup^\sharp (\!|\mathsf{st}_2|\!)^\sharp\mathsf{filter}^\sharp_{!\mathsf{b}}(\sigma^\sharp)$$
$$(\!|\mathtt{while}\ \mathsf{b}\ \{\,\mathsf{st}\,\}|\!)^\sharp\sigma^\sharp \triangleq \mathsf{filter}^\sharp_{!\mathsf{b}}\big(\mathsf{lfp}\ \lambda\sigma^\sharp_w\,.\,\sigma^\sharp \sqcup^\sharp (\!|\mathsf{st}|\!)^\sharp\mathsf{filter}^\sharp_{\mathsf{b}}(\sigma^\sharp_w)\big)$$

Concerning generic assignments, the abstract semantics follows the definition of the concrete one, so we have three cases: $x = \mathsf{e}$, where e evaluates to a value;

[1] We assume that all negations ! have been removed using DeMorgan's laws and usual arithmetic laws: $!\,(\mathsf{b}_1 \,||\, \mathsf{b}_2) \equiv !\,\mathsf{b}_1\ \&\&\ !\,\mathsf{b}_2$, $!\,(\mathsf{a}_1 < \mathsf{a}_2) \equiv (\mathsf{a}_2 < \mathsf{a}_1 \,||\, \mathsf{a}_2 == \mathsf{a}_1)$, etc.

(a) $\left[\begin{array}{c} \mathtt{b} \mapsto [2,2] \\ \hline \mathtt{c} \mapsto [3,3] \\ \hline \mathtt{a(a)}^* \mapsto [4,4] \end{array}\right]$ (b) $\left[\begin{array}{c} \mathtt{a} \mapsto [1,1] \\ \mathtt{b} \mapsto [2,2] \\ \hline \mathtt{c} \mapsto [3,3] \\ \hline \mathtt{a}^* \mapsto [4,4] \end{array}\right]$

Fig. 6: Example of materialization.

$x = \mathsf{o}$, where o evaluates to an object; $x[\mathsf{s}] = \mathsf{e}$, where s evaluates to a string and e evaluates to a value. In the first, we have to modify the abstract environment, setting x to the (abstract) evaluation of e. In the second, we need to update the abstract address pointed by the identifier x, with the line of code of the assignment. Then we have to update the abstract object pointed, in the abstract heap, by the new line of code with the (abstract) evaluation of o. Formally:

$$(\!|x = \mathsf{e}|\!)^\sharp \langle \xi^\sharp, \rho^\sharp \rangle \triangleq \langle \xi^\sharp [x \leftarrow (\!|\mathsf{e}|\!)^\sharp_{\mathrm{E}} \langle \xi^\sharp, \rho^\sharp \rangle], \rho^\sharp] \rangle$$
$$(\!|x = \mathtt{new}\,\mathsf{o}|\!)^\sharp \langle \xi^\sharp, \rho^\sharp \rangle \triangleq \langle \xi^\sharp [x \leftarrow \{\mathsf{line}(x = \mathtt{new}\,\mathsf{o})\}], \rho^\sharp [\mathsf{line}(x = \mathtt{new}\,\mathsf{o}) \leftarrow (\!|\mathsf{o}|\!)^\sharp_{\mathrm{O}} \langle \xi^\sharp, \rho^\sharp \rangle] \rangle$$

As a third case, we have the abstract semantics of object-property update, namely $x[\mathsf{s}] = \mathsf{e}$, where *materialization* occurs. As we have already mentioned before, we allow to update only the objects that have been already stored into the heap. Suppose that $v^\sharp = (\!|\mathsf{e}|\!)^\sharp_{\mathrm{E}} \langle \xi^\sharp, \rho^\sharp \rangle$, $p = (\!|\mathsf{s}|\!)^\sharp_{\mathrm{S}} \langle \xi^\sharp, \rho^\sharp \rangle$ and $\{l_1, \dots l_n\} = \xi^\sharp(x)$:

$$\text{let } o_i^\sharp = \mathsf{Norm}(\rho^\sharp(l_i)[p \leftarrow v^\sharp \sqcup_{\text{VAL}^\sharp} \rho^\sharp(l_i)(p)]), \text{ with } i \in \{1, \dots n\} \text{ in}$$
$$(\!|x[\mathsf{s}] = \mathsf{e}|\!)^\sharp \langle \xi^\sharp, \rho^\sharp \rangle \triangleq \langle \xi^\sharp, \rho^\sharp [l_1 \leftarrow o_1^\sharp, \dots\ l_n \leftarrow o_n^\sharp] \rangle$$

The abstract semantics of $x[\mathsf{s}] = \mathsf{e}$ does not update the environment, since it only needs to update properties of abstract objects stored into the heap. For each location $l \in \textsc{Addr}^\sharp$, associated to the identifier x (i.e., the ones contained in $\xi^\sharp(x)$), the abstract semantics updates $\rho^\sharp(l)$, at the property p, with the lub between $v^\sharp$ (i.e., the abstract evaluation of the expression e) and the previous value of $\rho^\sharp(l)(p)$. This corresponds to a *weak update* of the object contained in x [3]. Before storing the updated abstract object in $\rho^\sharp(l)$, the latter is normalized. In this paper, we only perform weak updates. We could improve the precision of the analysis performing a *must-may analysis* in order to differentiate between properties that certainly point to some value and properties that may point to others. This can be done improving the proposed analysis using standard techniques, such as the ones reported in [3, 16, 17].

For example, let us suppose that $\rho^\sharp(l)$ is the object reported in Fig. 6(a) and we want to update the property $\mathtt{a}$, with the interval $[1, 1]$. Applying these values to the previously defined abstract semantics, we obtain, at the allocation site l, the abstract object reported in Fig. 6(b). We say that the property $\mathtt{a}$ has been *materialized*, since, before the update, it was part of a summary property, and after the update it is a non-summary property. We say that a (concrete) property is materialized when a string of an abstract object passes, during the update, from a summary property to a non-summary property. It is worth noting that normalization take care of materialization. The abstract semantics is sound w.r.t. the concrete collecting semantics, i.e., it computes an over-approximation of state invariants at every statement.

(a)
```
1 o = new {a:1};
2 key = "a";
3 while (?) {
4     key = concat(key, "a");
5     o[key] = 1;
6 };
```

(b) $\left[\begin{array}{c} \mathtt{a} \mapsto [1,1] \\ \mathtt{aa} \mapsto [1,1] \\ \mathtt{aaa} \mapsto [1,1] \\ \hline \mathtt{aaaa(a)}^* \mapsto [1,1] \end{array}\right]$

Fig. 7: (a) μJS fragment, (b) Value of `o` after `while`-loop.

Theorem 1 (Soundness). *For every μJS program* $\mathsf{st} \in \textsc{stmt}$ *we have that:*

$$\forall \sigma^\sharp \in \textsc{State}^\sharp \,.\, (\!|\mathsf{st}|\!)\gamma(\sigma^\sharp) \sqsubseteq \gamma((\!|\mathsf{st}|\!)^\sharp \sigma^\sharp)$$

3.3 Widening

The domain $\langle \textsc{State}^\sharp, \sqsubseteq^\sharp \rangle$ is not ACC, i.e., it contains infinite ascending chains, because of the intervals abstract domain, the automata abstract domain and the novel objects abstract domain. Hence, fix-point computations in our abstract interpreter may diverge, if we do not introduce an extrapolation operator. In order to enforce termination, the abstract domain $\textsc{Val}^\sharp$ is equipped with the widening operator $\nabla_{\text{V}} \in \textsc{Val}^\sharp \times \textsc{Val}^\sharp \to \textsc{Val}^\sharp$ defined point-wisely. In particular, the intervals domain is equipped with its well-known widening defined in [7], the automata abstract domain is equipped with the widening ∇^n_{DFA}, reported in Sec. 2, while for addresses and booleans we can just use their least upper bound (they are finite). We can define the widening operator $\nabla_\xi \in \textsc{Env}^\sharp \times \textsc{Env}^\sharp \to \textsc{Env}^\sharp$ between environments upon ∇_{V}, applied point-wisely. For instance, suppose to use the widening ∇^n_{DFA}, with $n = 3$, for the finite state automata. We have that $[x \mapsto \langle [1,1], \bot, \mathsf{Min(aaa)}, \varnothing, \mathtt{def}\rangle] \; \nabla_\xi \; [x \mapsto \langle [2,2], \bot, \mathsf{Min(aaaa)}, \varnothing, \mathtt{def}\rangle]$ is equal to the abstract environment $[x \mapsto \langle [1,+\infty], \bot, \mathsf{Min(a^*)}, \varnothing, \mathtt{def}\rangle]$. Fix-point computations may also diverge on heaps, since also $\textsc{Heap}^\sharp$ is not ACC, due to the objects abstract domain. In particular, this happens because we model objects' properties with the finite state automata domain, which is not ACC. Anyway, a slight extension of the join $\sqcup^\sharp_{\text{O}}$ is enough to guarantee termination of heap computations, exploiting the widening of the finite state automata domain. Informally speaking, abstract string values, in `while`-loop computations, always converge since finite state automata domain is equipped with a widening.

Let us consider the μJS fragment reported in Fig. 7a and suppose that the boolean guard value is statically unknown. At each iteration on the `while`-loop, the string `"a"` is concatenated to the string value of `key` and then it is used to add a new property to the object `o`. If the $\textsc{Dfa}_{/\equiv}$ were not equipped with a widening, the computation of the value of `key` would diverge. Since convergence of string computations is enforced by the widening ∇^n_{DFA} (with $n = 3$), also the computations of objects' properties of `o` converge. Indeed, the `while`-loop converges and the abstract interpreter produces, for the variable `o`, the (normalized) object reported in Fig. 7b. Clearly, the simple object join is enough for objects' properties convergence but it is not for the associated value. For example, let consider the μJS fragment reported in Fig. 8a. In this case, the number of properties of the object `o` does not increase in the `while`-loop but the value of the property `a` increases at each iteration. The idea behind the widening for objects

(a)
```
1  o = new {a:1};
2  while (?) {
3      o["a"] = o["a"] + 1;
4  };
```

(b) $\left[\begin{array}{c} \mathtt{a} \mapsto [1, +\infty] \\ \hline \text{-} \end{array}\right]$

Fig. 8: (a) μJS fragment, (b) Value of `o` after `while`-loop.

is to apply the widening of values point-wisely between the properties of the two objects. Hence, we define the widening on $\textsc{Obj}^\sharp$ as: $o_1^\sharp \nabla_{\text{O}} o_2^\sharp \triangleq \lambda p \,.\, o_1^\sharp(p) \nabla_{\text{V}} o_2^\sharp(p)$. Coming back to the example, applying the widening defined above, the abstract value of `o` after the `while`-loop is reported in Fig. 8b. We then use this widening in order to define the widening for abstract heaps and, in turn, for abstract states.

Motivating example We now illustrate the so far defined analysis on the example reported in the introduction (Fig. 1). It is worth noting that, in this example, objects' widening does not occur. We have already commented it with the fragments reported in Fig. 7 and Fig. 8. The goal of the analysis is to reason about the value of `idx` (and, in turn, of `n`) at the end of the execution. At the beginning of the first iteration of the while loop, the value of `n` is $\langle [0,0], \bot, \mathsf{Min}(\varnothing), \varnothing, \mathtt{def}\rangle$ and the value of `idx` is $\langle \bot, \bot, (\mathtt{a} \parallel \mathtt{b}), \varnothing, \mathtt{def}\rangle$. The latter is used during the first iteration to access `obj` and then the result is stored in `n` (line 5). Since the property `b` is not present in `obj`, only the property `a` is accessed by `idx`, and the value of `n` is $\langle [1,1], \bot, \mathsf{Min}(\varnothing), \varnothing, \mathtt{def}\rangle$. Before starting the next iteration, `idx` is updated at line 6 and its value becomes $\langle \bot, \bot, (\mathtt{aa} \parallel \mathtt{ba}), \varnothing, \mathtt{def}\rangle$.

Widening is applied before starting new iterations. Supposing to apply the widening ∇^n_{DFA}, $n = 1$, and the widening for intervals, the values of the variables before the second iteration are: $\mathtt{n} = \langle [0, +\infty], \bot, \mathsf{Min}(\varnothing), \varnothing, \mathtt{def}\rangle$, $\mathtt{idx} = \langle \bot, \bot, (\mathtt{a} \parallel \mathtt{b} \parallel \mathtt{aa} \parallel \mathtt{ab}), \varnothing, \mathtt{def}\rangle$ since, in this case, the widening for automata coincides with the automata join. In the second iteration `idx` accesses the properties `a` and `aa`, hence `n` gets the value $[1, +\infty] = [0, +\infty] + ([1,1] \sqcup [2,2])$. Similarly to the previous iteration, `idx` becomes $\langle \bot, \bot, (\mathtt{aa} \parallel \mathtt{ba} \parallel \mathtt{aaa} \parallel \mathtt{aba}), \varnothing, \mathtt{def}\rangle$. Before starting the new iteration we apply the widening, obtaining the values $\mathtt{n} = \langle [0, +\infty], \bot, \mathsf{Min}(\varnothing), \varnothing, \mathtt{def}\rangle$ and $\mathtt{idx} = \langle \bot, \bot, (\mathtt{a} \parallel \mathtt{b})\,\mathtt{a}^\star, \varnothing, \mathtt{def}\rangle$. The third iteration does not change the values of `n` and `idx`, hence the fixpoint is reached.

Finally, at line 8, the value of `n` is assigned to `obj[idx]`, updating the abstract object `obj` as follows (we omit bottom values): $[\mathtt{a} \mapsto [0, +\infty], \mathtt{aa} \mapsto [0, +\infty], \mathtt{ab} \mapsto [3,3], \mathtt{ac} \mapsto \mathsf{Min}(\{\texttt{"world"}\}), (\mathtt{a} \parallel \mathtt{b})\,\mathtt{a}^* \setminus \{\mathtt{a}, \mathtt{aa}\} \mapsto [0, +\infty]]$. The summary property $(\mathtt{a} \parallel \mathtt{b})\,\mathtt{a}^* \setminus \{\mathtt{a}, \mathtt{aa}\}$ is added and only the properties `a` and `aa` are modified. Properties already present in `obj` remain unaltered (e.g., `ab` and `ac`).

4 Discussion and conclusion

We have proposed an abstract domain suitable for the analysis of objects' properties in dynamic programming languages. The novelty consists in exploiting *finite state automata*, in order to approximate objects' properties. This leads to a better precision (less false positives), compared to state-of-the-art domains approximating strings (for instance, [5, 6]). A key aspect of our abstract domain

is the *normal form for objects* and, in the paper, we have presented a normalization algorithm: it transforms objects in their normal form. An object is in normal form if and only if it has only two kind of properties: *summary* and *non-summary*. The idea behind summarization, and hence materialization, is not new in static analysis, and comes from the well-known shape analysis [16]. For example, this idea has been adopted in [11], where the authors present a static analyzer for PHP that also involve heap analysis, where the heap, in their abstraction, is made of summary heap identifiers and non-summary heap identifiers. In particular, in [11], a summary heap identifier summarizes all the elements of the heap that could be updated by statically unknown assignments. We have adopted the same idea with the difference that we may have more summary properties, expressed by automata recognizing infinite languages, rather than a single summary property that merges together heap elements updated by statically unknown assignments. The idea of summarization has been also taken into account in [3], where the authors propose the recency abstraction, which consists in representing each abstract allocation site with two memory regions, namely the *most recently allocated block* and *the not most recently allocated blocks*. The latter is basically a summary memory region, since more than one block may be allocated. Recency abstraction has been implemented also in TAJS [13], an abstract interpretation-based static analyzer for JavaScript, showing that such abstraction outperforms other abstract allocation-based techniques. As future work, we aim to implement our objects' abstract domain upon TAJS. We believe that the combination of our abstract domain and the recency abstraction can produce good results, w.r.t. analysis precision, and it would be interesting to make a comparison with TAJS and other JavaScript static analyzers, such as SAFE [15] and JSAI [14].

References

1. Arceri, V., Maffeis, S.: Abstract domains for type juggling. Electr. Notes Theor. Comput. Sci. **331** (2017)
2. Arceri, V., Mastroeni, I.: Static program analysis for string manipulation languages. In: VPT'19 (2019). https://doi.org/10.4204/EPTCS.299.5
3. Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: SAS'16 (2006). https://doi.org/10.1007/11823230_15
4. Bartzis, C., Bultan, T.: Widening Arithmetic Automata. In: CAV'04 (2004)
5. Cortesi, A., Olliaro, M.: M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In: TASE'18 (2018)
6. Costantini, G., Ferrara, P., Cortesi, A.: A Suite of Abstract Domains for Static Analysis of String Values. Softw., Pract. Exper. **45**(2) (2015)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
9. Davis, M.D., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages: Fund. of Theor. CS. Academic Press Professional, Inc. (1994)
10. D'Silva, V.: Widening for Automata. MsC Thesis, Inst. Fur Inform. - UZH (2006)

11. Hauzar, D., Kofron, J.: Framework for static analysis of PHP applications. In: ECOOP'15 (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.689
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
13. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: SAS '09 (2009). https://doi.org/10.1007/978-3-642-03237-0_17
14. 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 (2014)
15. 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)
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
17. Wilhelm, R., Sagiv, S., Reps, T.W.: Shape analysis. In: CC'00 (2000). https://doi.org/10.1007/3-540-46423-9_1