A Theory of Regular Expressions over Uninterpreted String Variables

Ticing Kao, Anish Ahuja, and Caleb Stanford

University of California, Davis

Abstract. In the traditional satisfiability modulo theories (SMT) theory of strings, strings and regular expressions live in separate worlds: strings are symbolic, while regular expressions are required to be concrete literals. We introduce a theory of regexes which unifies these worlds, supporting regexes defined over uninterpreted string variables (Uninterpreted String Regexes, or USRs) integrated in the same grammar. We show that this theory is highly expressive: it generalizes studied extensions of regexes with pattern variables, capture groups, and back-references, and can directly encode a variety of existing SMT constructs from the SMT-LIB 2.0 theory of strings. To achieve matching and analysis of USRs, since satisfiability is undecidable, we propose a theory of Brzozowski and Antimirov derivatives of USRs. Our Antimirov derivative incorporates a novel approach using *string substitutions* which are performed on the uninterpreted variables on-the-fly; this eliminates symbolic checks on the first character of a potential string match and instead reduces satisfiability of USRs via a lightweight procedure to unify string substitutions, pruning those that are not compatible. We implement both derivative-based algorithms and evaluate them on standard SMT benchmarks, as well as handwritten USR cases. We find that our Antimirov implementation solves a significant fraction of benchmarks in our set (about 70% of those solved by CVC5 and Z3), through pure translation to USRs, and would lead to benefits over existing solvers if run in a portfolio setting. We also find that some existing solvers (in particular, CVC5) can naturally handle USRs.

1 Introduction

Regular expressions (regexes) are supported by most mainstream programming languages, and the theory of string constraints incorporating regexes is widely used for static analysis of string-manipulating programs [7,3] and other applications [9,52]. This has led to interest over the last decade in algorithms and practical solvers for string and regular expression constraints [17,15,43,34,1,32,33,66,41,4,52]. However, there remains a gap between regexes that arise in practice, and the theoretical and formal basis for regex constraints (interpreted as languages of strings); practical regexes may incorporate, for example, (1) unicode and other symbolic alphabets, not just a finite alphabet [72,37]; (2) Boolean operations (e.g., intersection of regular expressions $R_1 \cap R_2$) in many settings, not just union, concatenation and star [66,52]; (3) advanced features like capture groups and

```
(declare-fun a () String)
(declare-fun b () String)
(assert (str.in_re a
     (re.diff (re.union (re.* (str.to_re "b")) (str.to_re "z"))
     (re.++ (re.* (str.to_re "z")) (str.to_re b)))))
(assert (= 0 (str.len b)))
(check-sat)
```

Fig. 1: Although not allowed by the SMT theory of strings, regexes containing string variables have exposed bugs in prior versions of state-of-the-art solvers (GitHub 2021 [44]). Here, `str.to_re b` (denoting a regex matching the string sort `b`) was erroneously applied to a string variable `b` rather than a string literal.

back-references [12,70,27,54], which can be captured as string transducers [41,29]. In this paper, we study a different, simple extension of regexes which unifies many of the above: regexes extended with string variables. For example, the regex

$$\overline{w}\mathtt{a}\overline{w}$$

includes one uninterpreted string variable, $\overline{w}$; it matches strings which are formed by a string $\overline{w}$, followed by character `a`, followed by the same string $\overline{w}$ again. The language of the regex is then a non-regular language of strings which begin and end with the same text, separated by an `a`.

Such regexes over uninterpreted variables (also known as "non-ground regexes" [44]) are technically disallowed by the SMT-LIB 2.0 syntax for the theory of strings [68]. However, there is some evidence that such constraints arise in practice; for example, they appeared in a series of GitHub bug reports in 2021 for existing SMT solvers (Figure 1). The crux of the issue is that the SMT syntax `str.to_re s` can be used to create a regular expression that matches a string `s`, where `s` is of a *string sort*; however, there is nothing in the grammar that explicitly distinguishes string variables from other values of sort String, so users can inadvertently write such expressions. Similarly, a user might write `(str.in_re s1 (re.option (str.to_re s2)))` to indicate that `s1` is supposed to be a string that is either empty or equal to `s2`. Motivated by these and similar examples, we set out to identify a theory that could solve such constraints, and this is the focus of the present paper. We shall see that in fact, regexes with string variables – while arising in such obscure cases – turn out to be useful far beyond these cases; they encompass much of the existing SMT string theory syntax, as well as existing classes of regexes that are extended with patterns and capture groups, as we will show in Section 4.

Uninterpreted String Regular Expressions. We propose Uninterpreted String Regexes (USRs), which extend classical regex with symbolic uninterpreted string variables, symbolic predicates (over characters), all Boolean operations (intersection, union, and complement), and if-then-else expressions. We introduce the theory with motivating examples in Section 2 and the syntax and semantics in Section 3. USRs have two different semantics of note: first, the main semantics which is given with respect to a valuation over *all* uninterpreted string variables;

and second, the "projection" semantics which is given as a language of strings, implicitly an existential quantifier over all uninterpreted string variables. Although nonemptiness for the resulting theory is naturally undecidable (Theorem 1), the theory can express many SMT string constructs within a unified framework (Propositions 1 and 2).

The incorporation of string variables also allows the theory to support capture groups and patterns: for example, the regex with one capture group $(\mathtt{a}^*)_1\mathtt{b}\backslash 1$ (representing a string of as followed by a b followed by the same string of as) can be expressed as the USR $(\mathtt{a}^* \cap \overline{w})\mathtt{b}\overline{w}$. Generalizing this result, we show that in expressiveness the most general class of languages recognized by USRs contains the existing classes PATEXP (introduced and studied by Câmpeanu and Yu [24]), regexes with back-references (Freydenberger [39]), and HREG (introduced by Bordihn, Dassow, and Holzer [18]) (Theorem 2). The full class lies between these classes and the context-sensitive languages (see Theorem 4 and Figure 3).

Since conversion to automata may result in an infinite state space — and would be prohibitive even in the finite case due to Boolean operations [67] — we next develop a theory of derivatives over USRs, following the classical theory of Brzozowski [21] and Antimirov [8] in Section 5. Our Brzozowski derivatives= mirror the classical case, but also incorporate if-then-else expressions [71,66]. However, Brzozowski derivatives can suffer from blowup due to the combination of Boolean operations, string indexing, and if-then-else terms. To address this, we propose Antimirov derivatives with *substitution terms* (Section 5.2), which avoid if-then-else terms in the construction (Theorem 9). Both derivative constructions (for which we prove correctness) immediately yield sound and partially complete algorithms for satisfiability of USRs.

Our key insight for the Antimirov derivatives is to perform a substitution in string variables *on-the-fly*, meaning that we do not need to separately query an SMT solver for the character theory. To understand this distinction, consider the derivative of the USR $\overline{w}\overline{w}$ with respect to a character x. Under the Brzozowski derivative, it becomes the term $\mathtt{IF}(\overline{w}[0] = x, \overline{w}[1:]\overline{w}, \varnothing)$. In order to solve for nullability (i.e., matching the empty string) on such a derivative term, we need to then check whether the expression $w[0] = x$ is satisfiable over the character theory. For Antimirov derivatives, we instead employ the following trick: we replace $\overline{w}$ with $x\overline{w}$, in all instances, when evaluating the derivative: we then return the derivative as a pair $(\overline{w}x\overline{w}, f)$, where $\overline{w}$ now represents the tail of the previous instance of $\overline{w}$ and f is a *substitution* which maps $\overline{w} \mapsto x\overline{w}$. Notice that the first instance of $\overline{w}$ has had an x peeled off (implicitly), while the second instance has correspondingly been replaced by $x\overline{w}$. When these derivatives are combined (for example, from different parts of a regex), differing substitutions can then be combined via lightweight calls to a union-find data structure. We also establish a linear complexity bound on the size of Antimirov derivatives (in the size of the regex and number of derivatives) for the subclass of USRs involving no complement or if-then-else terms (Theorem 10).

We implement the theory and our algorithms in an experimental SMT solver (Section 6). The implementation works via pure reduction to USR derivatives, with some minor optimizations for predicate simplifications and character range constraints. Our evaluation asks three questions: Do (Brzozowski or Antimirov) derivatives of USRs solve existing standard SMT benchmark classes? Would they lead to concrete improvements when evaluated in a portfolio setting? Finally, do existing SMT solvers successfully handle USR constraints in some cases? We find that our experimental solver solves close to 70% of our benchmarks via encoding constraints into USRs and can handle SMT constraints with explicitly declared string variables; one-to-one comparison with Z3 and CVC5 shows that some benchmarks are solved significantly faster (at least half an order of magnitude) by USR derivatives, i.e., would be improved in a portfolio setting ($N = 100$ benchmarks over Z3 and $N = 84$ benchmarks over CVC5). Interestingly, we find that to some extent, existing solvers *can* currently handle USR cases, when evaluated with their latest versions: among 41 benchmarks which contain regexes with string variables, the latest version of CVC5 solves 38 of them. Our Antimirov implementation is a close second with 36, ahead of both Z3 and Brzozowski.

In summary, our contributions are as follows:

- We introduce Uninterpreted String Regexes (USRs), which extends the theory of regular expressions to support arbitrary uninterpreted string variables. (Section 3)
- We show that USRs are highly expressive: many existing SMT constraints can be encoded directly as USRs, and they generalize studied classes of regexes involving patterns, capture groups, and back-references. (Section 4)
- We define a theory of Brzozowski derivatives using if-then-else terms and Antimirov derivatives using substitutions evaluated on the fly for incrementally solving constraints over USRs; and we prove both theories correct with respect to the intended semantics. (Section 5)
- We evaluate our theory experimentally on standard SMT benchmarks benchmarks, and compare it to existing state-of-the-art solvers, including the first evaluation of these solvers for explicit USR cases. (Section 6)

2 Motivation

In this section, we motivate the definition of USRs and Brzozowski and Antimirov derivatives. Our first example is taken from an existing case written by an SMT solver user on GitHub, similar to the one described in the introduction, and reduces to nonemptiness on USRs. The Brzozowski derivative of R works by returning symbolic *if-then-else* terms, while the Antimirov derivative avoids this by returning a pair (R', f) where f is a string substitution (calculated on-the-fly) for the string variables present in R. Both of these derivative constructions lead to a sound and partially complete decision procedure (i.e., terminating for the nonempty case) for constraints involving USRs.

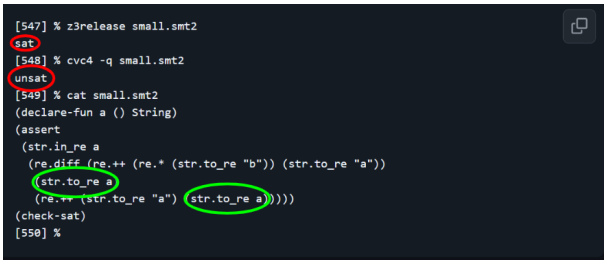

Fig. 2: Output of a prior version of Z3 [74] and CVC4 [35] on a case similar to Figure 1. String variables are shown in green with conflicting output in red.

Example 1. Consider the user-written SMT constraint from Figure 2; this example exposed a bug in a prior version of Z3. Here a is declared a string variable with `(declare-fun a () String)`; to distinguish this from the character literal `a`, we replace the string variable a with $\overline{w_1}$. When presented as an SMT constraint, the goal is to find some satisfying assignment to $\overline{w_1}$; we encode this using USRs using intersection and complement to express the regex difference `re.diff`: $\mathtt{b}^*\mathtt{a} \cap (\overline{w_1})^c \cap (\mathtt{a}\overline{w_1})^c$. Since we want $\overline{w_1}$ to match this USR, the entire assertion then reduces to nonemptiness of the following USR:

$$\overline{w_1} \cap (\mathtt{b}^*\mathtt{a} \cap (\overline{w_1})^c \cap (\mathtt{a}\overline{w_1})^c).$$

Brzozowski derivative. So satisfiability reduces to nonemptiness of USRs; to solve the latter problem, we first consider Brzozowski derivative. Using a slightly simpler example of $R = \overline{w}0 \cap \overline{w}^*$, similar to other derivative-based solvers [66,50], we first utilize a *nullable* function (a regex is *nullable* if it matches the empty string); returning an *if-then-else* term [71]: $N(R) = \mathbf{IF}\,(|\overline{w}| = 0, \epsilon, \varnothing)\,\varnothing \cap \epsilon$. The resulting expression is not satisfiable, so we take the derivative with respect to a fresh character variable x_1 denoted $d(R, x_1)$ and simplify:

$$R' = d(R, x_1) = (\mathbf{IF}\,(\overline{w}[0] = x_1, \overline{w}[1:]0, \varnothing) \cup \mathbf{IF}\,(|\overline{w}| = 0 \wedge x_1 = 0, \epsilon, \varnothing)) \\ \cap \mathbf{IF}\,(\overline{w}[0] = x_1, \overline{w}[1:]\overline{w}^*, \varnothing)$$

We can check that R' is again not nullable, so we take the derivative a second time to get $R'' = d(R', x_2)$. $N(R'')$ gives a satisfiable branch with $\overline{w}[0] = x_1$, $|\overline{w}| = 1$, $x_2 = 0$, and $\overline{w}[0] = x_2$, so R is nonempty.

Antimirov derivative. Unfortunately, the above construction via the Brzozowski derivative leads to increasingly large nested sequences of if-then-else terms on each successive call. The Antimirov derivative $\Delta(R, x)$ avoids this by returning a pair (R', f) of a regex and a *string substitution* f, which tracks requirements on the first character of the string variable $\overline{w}$. The substitution is calculated as we take the derivative of each component of the intersection, i.e. in $\overline{w}0$ and $\overline{w}^*$. For the first component $\overline{w}$ we get two cases, one when $\overline{w}$ is nonempty, and one

when it is empty, leading to the pairs $(\overline{w}0, \langle \overline{w} \mapsto x\overline{w} \rangle)$ and $(\epsilon, \langle \overline{w} \mapsto \epsilon, x \mapsto 0 \rangle)$, respectively. The second component requires $\overline{w}$ to be nonempty, so only has one derivative $(\overline{w}(x\overline{w})^*, \langle \overline{w} \mapsto x\overline{w} \rangle)$. When these are intersected, the empty case is then pruned as it is incompatible with the second component, and we end with:

$$\Delta(R, x) = \{(\overline{w}0 \cap \overline{w}(x\overline{w})^*, \langle \overline{w} \mapsto x\overline{w} \rangle)\}$$

Notice that there are no if-then-else terms in the above, as the joint conditions on the first character of each string variable are automatically combined during the intersection calculation described above. After nullable of the above USR returns $\varnothing$, we repeat this process with a second derivative and get the nullable regex $R_2'' = \epsilon \cap \epsilon(x_1\epsilon)^*$, so R is satisfiable.

3 Uninterpreted String Regular Expressions

3.1 Preliminaries

Let Σ be a finite alphabet. We use σ to denote a generic element of Σ, teletype $\mathtt{a}, \mathtt{b} \in \Sigma$ to denote concrete character literals, and $s, t, \ldots \in \Sigma^*$ to denote strings over Σ. We distinguish strings from *string variables*, denoted $\overline{w_i}$, which are uninterpreted variables over the SMT theory of strings We sometimes use $\overline{w}$ to denote a generic variable $\overline{w_i}$ when the context is clear. We also distinguish character literals from *character variables*, denoted $\overline{c_i}$, and character expressions, denoted $x, y, \ldots$. Assigning string and character variables to literals is done through a standard *valuation function* $\nu : \{\overline{w_i}, \overline{c_i}\}_i \to \Sigma^*$, such that $\nu(\overline{c_i}) \in \Sigma$ for all i. We let Val denote the set of all valuations. String indexing $s[j]$ and slicing $s[j:]$ are operations defined on Σ^*; we use the convention that these return either a string in Σ^* or $\varnothing$ if j is out of bounds.

3.2 Syntax and Semantics

We define the set of Uninterpreted String Regexes (USRs) which allows for predicates (in if-then-else terms), complement and intersection, and string slicing applied to string variables. The grammar for USRs R, predicates ϕ, and character expressions C is as follows:

$$R ::= \varepsilon \mid \varnothing \mid C \mid \overline{w_i} \mid \overline{w_i}[j] \mid \overline{w_i}[j:] \mid R \cup R \mid R \cap R \mid R \cdot R \mid (R)^* \mid (R)^c \mid \mathbf{IF}\,(\phi, R, R)$$
$$\phi ::= C{=}C \mid |\overline{w_i}| = j \mid \overline{w_i}[j] = C \mid \overline{w_i}[j] = \overline{w_k}[l] \mid T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$
$$C ::= \sigma \mid \overline{c_i}.$$

We sometimes refer to USRs as simply *regexes*.

We define the semantics functions C, P, and R for characters, predicates, and USRs recursively (each takes an expression of the given type and a valuation ν). For character expressions: $C(\sigma, \nu) = \{\sigma\}$ and $C(\overline{c_i}, \nu) = \{\nu(\overline{c_i})\}$. For predicates: $P(x = y, \nu) = \mathtt{True}$ if $C(x, \nu) = C(y, \nu)$, $P(|\overline{w_i}| = j, \nu) = \mathtt{True}$ if $|\nu(\overline{w_i})| = j$, $P(\overline{w_i}[j] =$

$x, \nu) = \texttt{True}$ if $\nu(\overline{w_i})[j] = C(x, \nu)$, $P(\overline{w_i}[j] = \overline{w_k}[l], \nu) = \texttt{True}$ if $\nu(\overline{w_i}[j]) = \nu(\overline{w_k}[l])$, $P(T, \nu) = \texttt{True}$, $P(F, \nu) = \texttt{False}$, $P(\phi_1 \wedge \phi_2, \nu) = P(\phi_1, \nu) \wedge P(\phi_2, \nu)$, $P(\phi_1 \vee \phi_2, \nu) = P(\phi_1, \nu) \vee P(\phi_2, \nu)$, and $P(\neg\phi_1, \nu) = \neg P(\phi_1, \nu)$. Finally, for USRs: $L(\varepsilon, \nu) = \{\varepsilon\}$, $L(\varnothing, \nu) = \varnothing$, $L(\overline{w_i}, \nu) = \{\nu(\overline{w_i})\}$, $L(\overline{w_i}[j], \nu) = C(\overline{w_i})[j]$ element-wise, $L(\overline{w_i}[j:], \nu) = C(\overline{w_i})[j:]$ element-wise, $L(R_1 \cup R_2, \nu) = L(R_1, \nu) \cup L(R_2, \nu)$, $L(R_1 \cdot R_2, \nu) = L(R_1, \nu) \cdot L(R_2, \nu)$, $L((R_1)^*, \nu) = (L(R_1), \nu)^*$, $L(R_1 \cap R_2, \nu) = L(R_1, \nu) \cap L(R_2, \nu)$, $L((R_1)^c, \nu) = \Sigma^* \setminus L(R_1, \nu))$, and for the if-then-else case:

$$L(\textbf{IF}\,(\phi, R_1, R_2)\,, \nu) = L(R_1, \nu) \text{ if } P(\phi, \nu),\ L(R_2, \nu) \text{ otherwise.}$$

Example 2. Let us find the language of $\textbf{IF}\,(\overline{c_1} = \texttt{a}, \overline{w_1}\texttt{a}\overline{w_2 c_1}, \texttt{b})$ with valuation ν with $\nu(\overline{w_1}) = \texttt{cat}, \nu(\overline{w_2}) = \texttt{dog}, \nu(\overline{c_1}) = \texttt{a}$:

$$L(\textbf{IF}\,(\overline{c_1} = \texttt{a}, \overline{w_1}\texttt{a}\overline{w_2 c_1}, \texttt{b})\,, \nu) = \left\{ \begin{array}{cc} \texttt{catadoga} & \text{if } \texttt{a} = \texttt{a} \\ \texttt{b} & \text{otherwise} \end{array} \right\} = \{\texttt{catadoga}\}.$$

Definition 1. *(Language Projection) Let R be a USR. We define the* language projection $L\downarrow(R)$ *as the set of all strings accepted by R over all valuations:*

$$L\downarrow(R) = \bigcup_{\nu \in Val} L(R, \nu).$$

3.3 Decidability

The following theorem shows that the nonemptiness problem (and hence also, language inclusion) is undecidable in general for the language projection of USRs. This motivates more focused algorithms which aim to decide membership and nonemptiness efficiently on practical benchmarks.

Theorem 1. *For any alphabet Σ, the language of USRs that match at least one string:*

$$\left\{\langle R\rangle \ :\ L\downarrow(R) \text{ is nonempty}\right\}$$

is undecidable.

Proof. We reduce from the quantifier-free theory of natural numbers over $\{+, \mathrm{lcm}\}$, where lcm is the least common multiple operation.[1] To that end, let φ be any formula over $\{+, \mathrm{lcm}\}$; we may assume without loss of generality that φ is purely conjunctive with clauses of the form $x_1 = x_2 + x_3$, $x_1 = \mathrm{lcm}(x_2, x_3)$, $x_1 \neq x_2$, or $x_1 = n$, where x_i are variables and n is a constant. We will construct a USR R by concatenating a series of USRs such that $L\downarrow(R)$ is nonempty iff φ is satisfiable.

For each natural number variable x in φ, we create a corresponding string variable w_x such that the length of w_x will correspond to x, and we concatenate the regex $w_x \cap 1^*$ to enforce it only uses a unary alphabet. For any addition

[1] This theory is undecidable by reduction from Hilbert's 10th problem [55]: we encode $x^2 := \mathrm{lcm}(x, x-1) + x$ and then multiplication via $2(x \cdot y) := (x+y)^2 - x^2 - y^2$.

constraint $x_1 = x_2 + x_3$, we concatenate $w_{x_1} \cap (w_{x_2} \cdot w_{x_3})$. For an LCM constraint $x_1 = \text{lcm}(x_2, x_3)$, we encode via Kleene star as two USRs: $w_{x_1} \cap S$ and $w_{x_1} \cap (S \cdot S)^c$, where $S = (w_{x_2})^* \cap (w_{x_3})^* \cap \epsilon^c$. Finally, for $x_1 \neq x_2$ and $x_1 = n$ we use $w_{x_1} \cap (w_{x_2})^c$ and $w_{x_1} \cap 1^n$, respectively. Each concatenated regex is nonempty for a particular assignment $w_{x_i} = 1^k$ iff the corresponding integer constraint is satisfied for the corresponding assignment $x_i = k$. □

4 Expressiveness

This section is about expressiveness of USRs. We first show that USRs can express a significant subset of the SMTLIB theory of strings [68], without any special syntax for this purpose. In particular, they can express string equality and concatenation; length constraints, when the length is compared to a constant; string indexing, when the index is a constant; string prefixes, suffixes, substrings; and lexicographic string comparison.

Second, we formally study three language classes: S-USR, I-USR, and B-USR. Define S-USR as the class of languages that can be recognized as the language projection $L \downarrow (R)$ for R containing no intersection and complement (S for *simple*); I-USR similarly, but allowing intersection; and B-USR allowing both intersection and complement. These classes are related to each other and to the class `REG` of regular languages by `REG` $\subseteq$ S-USR $\subseteq$ I-USR $\subseteq$ B-USR. We prove several relationships between these classes and studied extensions of `REG`, summarized in Figure 3.

4.1 Encoding SMT Constraints

Proposition 1. *For the following SMTLIB constraints ϕ, we can compute R_t, R_f such that $\phi[v]$ is true if and only if $L(R_t, v)$ is non-empty and false if and only if $L(R_f, v)$ is non-empty. Let `s1, s2, s3` be string variables, `c` be a character expression, and `n` be a constant integer:*

- `(str.= s1 s2)`
- `(str.= (str.++ s1 s2) s3)`
- `(= (str.len s1) n)`
- `(< (str.len s1) n)`
- `(> (str.len s1) n)`
- `(= (str.at s1 n) c)`
- `(= (str.substr s1 i j) s2)`
- `(str.contains s1 s2)`
- `(str.prefixof s1 s2)`
- `(str.suffixof s1 s2)`

Proof. Here, an example for the first case is provided. The remaining cases use similar techniques and are provided in the appendix.

Let $R_t = \texttt{s1} \cap \texttt{s2}$ and $R_f = \texttt{s1} \cap (\texttt{s2})^c$. If `s1` is equal to `s2`, $L(R_t, v) = \{v(\texttt{s1})\} \cap \{v(\texttt{s1})\} = \{v(\texttt{s1})\}$, which is nonempty. Conversely, if `s1` is not equal to `s2`, then $L(R_t, v) = \varnothing$. Similarly for R_f; for instance, if $L(R_f, v)$ is nonempty, then that means $v(\texttt{s1})$ is in the language $\{v(\texttt{s2})^c\}$, so `s1` is not equal to `s2`. □

Proposition 2. *The SMTLIB constraint $\phi =$ `(str.< s1 s2)` can be expressed via USRs. Formally, we can compute USRs R_t, R_f such that for any valuation v of free variables in ϕ, ϕ is true if and only if R_t is nonempty, and false if and only if R_f is non-empty. That is, $\phi[v]$ is true if and only if there is some valuation v' extending the valuation of v such that $L(R_t, v')$ is non-empty.*

Proof. We only need to encode R_t, as the R_f case is equivalent to the disjunction of `str.>` and `str.=`. The idea is that we should encode the first expression where `s1` and `s2` might differ. There are two cases: `s1` is a prefix of `s2`, or `s1` and `s2` differ at a certain index. For the first case, we can use $(\texttt{s1} \cdot \Sigma^*) \cap \texttt{s2}$, and for the second, we introduce a fresh variable `p` for the common prefix, and use $\texttt{s1} \cap (\texttt{p} \cdot c \cdot \texttt{s1}')$ concatenated with $\texttt{s1} \cap (\texttt{p} \cdot d \cdot \texttt{s2}')$, where we enumerate over all c and d characters such that $c < d$ and take the union. This encoding works because the prefix `p` is uniquely determined by any two strings in lexicographic ordering (in fact, any two strings share a unique longest common prefix). □

4.2 Comparison to existing pattern regex classes

For the purposes of this section, it is sufficient to consider USRs only with plain characters, string variables, epsilon, union, concat, star, intersection, and complement over a finite alphabet; it is possible to reduce nonemptiness of any other USR to one of this form, as all other constraints can be expanded by enumerating the first n characters of each string variable (c.f. SFromP in Section 5.2). We show relationships of S-USR, I-USR, and B-USR with the following classes: HREG consisting of regexes under homomorphic replacement considered by [19], PATEXP considered by [19,24], and $\texttt{RegEx}(k)$ defined by [39].

We divide PATEXP into PATEXP-CSY [24] and PATEXP-BDH [19]; the class is defined subtly differently in the two papers, and the definitions are not obviously equivalent. In the case of CSY, multiple substitutions of the same string variable across pattern languages need not be consistent, which is not true of BDH pattern expressions. However, by introducing separate string variables per pattern, a BDH pattern expression can encode CSY semantics, so PATEXP-CSY is a subset of PATEXP-BDH. The class S-USR is a subset of PATEXP-CSY because any simple USR can be written as a pattern expression with homomorphic replacement by Σ^*. For the class $\mathcal{C} = \bigcup_{k \geq 0} \texttt{RegEx}(k)$, note that S-USR $\subseteq \mathcal{C} \subseteq$ I-USR, as any simple USR can be written trivially with backreferences, and any binding $(R)\%w$ in $\mathcal{C}$ can be expressed as $\overline{w} \cap R$ in I-USR. Finally, we also show that, similar to some of these other classes, USRs are incomparable with the context-free languages and contained in the context-sensitive languages. The proof of Theorem 2 can be found in the Appendix; it works by identifying a sub-class of I-USR*s* which is closed under homomorphic replacement: this allows intersections only of the form $\overline{w_i} \cap R_i$, where R_i is the same for all $\overline{w_i}$ instances.

Theorem 2. HREG $\subseteq$ I-USR.

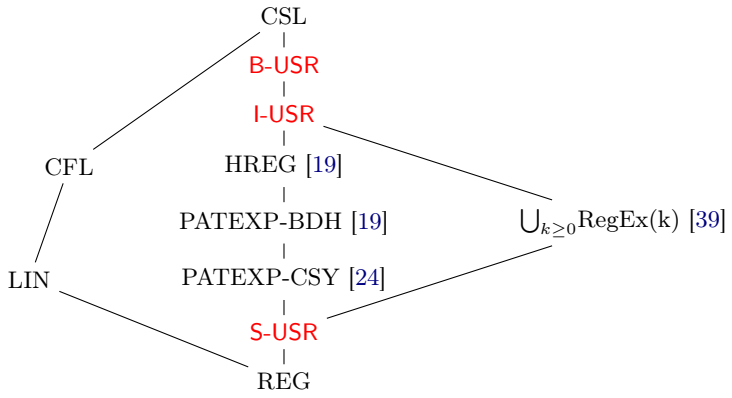

Fig. 3: Relation between expressiveness of classes of USRs considered in our paper (shown in red: S-USR, I-USR, and B-USR) and known classes of extensions of regular languages.

Theorem 3. *The class* S-USR *is incomparable with the class* CFL *of context-free languages.*

Proof. For CFL $\not\subseteq$ S-USR, it is known that CFL and HREG are incomparable, and S-USR $\subseteq$ HREG. For the other direction S-USR $\not\subseteq$ CFL, it suffices to show that there are languages $L \in$ S-USR such that $L \notin$ CFL. The language $L = \{wcw | w \in \{a, b\}^*\}$ satisfies this requirement. □

Theorem 4. B-USR $\subseteq$ CSL.

Proof. We show that for any fixed USR R, on input s, there is a nondeterministic linear-space algorithm to decide if $s \in L\downarrow(R)$. If $s \in L\downarrow(R)$, then there exists a valuation v such that $s \in L(R, v)$. We can show via structural induction that given this valuation, there exists another valuation v' bound by restricting the length of all mappings of $\overline{w_i}$ to strings of max length $|s| + 1$ such that $L(R, v)$ and $L(R, v')$ contain the same strings up to length $|s|$. Thus, we can non-deterministically pick this valuation instead. Using structural induction and this fixed valuation with mapping of max length $|s| + 1$, we can use structural induction to find a deterministic algorithm that will match s to $L(R, v)$ in linear space. □

5 Derivatives

In this section, we define derivatives of USRs, starting with Brzozowski, and then Antimirov. We need some auxiliary functions, in particular the *nullable* function which takes different forms and is used to define the derivative of concatenation. We also state and prove correctness theorems, culminating in

the correctness of the Brzozowski and Antimirov derivatives in capturing the intended semantics (the latter theorem is considerably more difficult to state and prove, due to the presence of substitutions); these results establish the basis for a sound and partially complete decision procedure (Theorem 8). We also establish a subclass closure result and linear complexity bound on the Antimirov derivative in Theorems 9 and 10 for regexes without if-then-else and complement.

5.1 Brzozowski Derivatives with Transition Regexes

We will first define the predicate nullable function, which will evaluate to ϵ if and only if the USR accepts the empty string, and will evaluate to $\varnothing$ otherwise, matching the classical nullable definition for derivatives.

Definition 2. *(Predicate Nullable). The* predicate nullable *function $N : USR \rightarrow USR$ is defined by the following rules: $N(\varnothing) = \varnothing, N(\epsilon) = \epsilon, N(C) = \varnothing, N(R_1 \cup R_2) = N(R_1) \cup N(R_2), N(R_1 \cdot R_2) = N(R_1) \cdot N(R_2), N(R^*) = \epsilon, N(R_1 \cap R_2) = N(R_1) \cap N(R_2), N(R^c) = \epsilon \cap N(R)^c$.*

Cases unique to USRs such as string variables and if-then-else have the following rules:

$$N(\overline{w_i}) = \boldsymbol{IF}(|\overline{w_i}| = 0, \epsilon, \varnothing) \qquad N(\overline{w_i}[j:]) = \boldsymbol{IF}(|\overline{w_i}| = j, \varepsilon, \varnothing)$$
$$N(\overline{w_i}[j]) = \varnothing \qquad N(\boldsymbol{IF}(\phi, R_1, R_2)) = \boldsymbol{IF}(\phi, N(R_1), N(R_2))$$

Theorem 5. *(Correctness of Predicate Nullable). For any USR R and valuation v, if $\epsilon \in L(R, v)$, then $L(N(R), v) = \{\epsilon\}$ and if $\epsilon \notin L(R, v)$, then $L(N(R), v) = \varnothing$.*

Definition 3. *(Derivatives of USRs). The derivative of a USR R with respect to a character expression x is a function $D : USR \times C \rightarrow USR$. Let $\overline{c_i}$ be character variable. When x is a character literal and R does not contain variables or if-then-else, D is identical to Brzozowski derivative. Recursive cases that are not concatenation or if-then-else are also identical to Brzozowski derivative. The following are the cases unique to USRs:*

$$\begin{aligned}
D(\sigma, x) &= \boldsymbol{IF}(\sigma = x, \varepsilon, \varnothing) \\
D(\overline{c_i}, x) &= \boldsymbol{IF}(\overline{c_i} = x, \varepsilon, \varnothing) \\
D(\overline{w_j}, x) &= \boldsymbol{IF}(\overline{w_j}[0] = x, \overline{w_j}[1:], \varnothing) \\
D(\overline{w_j}[k], x) &= \boldsymbol{IF}(\overline{w_j}[k] = x, \varepsilon, \varnothing) \\
D(\overline{w_j}[k:], x) &= \boldsymbol{IF}(\overline{w_j}[k] = x, \overline{w_j}[k+1:], \varnothing) \\
D(R_1 \cdot R_2, x) &= D(R_1, x) \cdot R_2 \cup N(R_1) \cdot D(R_2, x) \\
D(\boldsymbol{IF}(\phi, R_1, R_2), x) &= \boldsymbol{IF}(\phi, D(R_1, x), D(R_2, x))
\end{aligned}$$

Theorem 6. *(Correctness of Derivative). For any USR R, character expression x, and valuation v, Let $R' = D(R, x)$, then $\forall z \in L(R', v)$, $L'(x, v) \cdot z \in L(R, v)$.*

5.2 Antimirov Derivatives with Substitution Terms

With the Antimirov derivative, we encode constraints on strings through substitutions. Substitutions are mappings from string and character variables to a small subset of USRs needed to restrict valid variable assignments when taking the derivative. The syntax of a substitution is a set of mappings delimited by $\langle\rangle$. Semantically, each substitution f corresponds to a restricted set of valuations $Val_f \subseteq Val$. This restriction can also be described as an onto function $F_f : Val \to Val_f$ and a one-to-one right-inverse map $F'_f : Val_f \to Val$. For example, $f = \langle \overline{c_c} \mapsto \sigma \rangle$ enforces $F_f(v)(\overline{c_c}) = \sigma$.

There is a subset of substitutions named *simple substitutions* that syntactically and semantically behave well with USRs. A simple substitution f can be applied on a USR R with $R[f]$ which replaces all variables in R with its output from f. The identity substitution, $\mathbb{1}$ does nothing when applied with $R[\mathbb{1}] = R$. For substitutions f, g, other operations include merging two simple substitutions to combine their semantics denoted $f\#g$, substitution difference to remove the semantics of one substitution from another denoted $f - g$, and complementing a set of substitutions to get another set with opposite semantics denoted $\{f_1, f_2, \ldots, f_n\}^T$. $f\#g$ corresponds to intersecting the two sets of valuations $\mathrm{Val}_f \cap \mathrm{Val}_g$. SFromP converts predicates to substitutions and is used to check nullability of if-then-else expressions. The technical details of the semantics of substitutions and operation definitions are in the Appendix.

We first define substitution nullable which takes a USR R and returns a set of substitutions, that when applied to R yields a nullable USR.

Definition 4 (Substitution Nullable). *A USR R is nullable when $\Pi(R)$ is non-empty where Π is defined as*

$$
\begin{aligned}
\Pi(\sigma) = \Pi(\overline{c_i}) = \Pi(\varnothing) = \Pi(\overline{w_i}[j]) &= \varnothing \\
\Pi(\epsilon) &= \{\mathbb{1}\} \\
\Pi(\overline{w_i}) &= \{\langle \overline{w_i} \mapsto \epsilon \rangle\} \\
\Pi(\overline{w_i}[j:]) &= \{\langle \overline{w_i} \mapsto \sigma_1 \cdots \sigma_{j-1} \rangle | \sigma_k \in \Sigma\} \\
\Pi(P \cup Q) &= \Pi(P) \cup \Pi(Q) \\
\Pi(PQ) = \Pi(P \cap Q) &= \{f\#g | f \in \Pi(P), g \in \Pi(Q), f\#g \neq \bot\} \\
\Pi(\textbf{\textit{IF}}(\phi, P, Q)) &= \{f\#g | f \in \Pi(P), g \in \mathit{SFromP}(\phi)\} \\
&\quad \cup \{f\#g | f \in \Pi(Q), g \in \mathit{SFromP}(\neg\phi)\} \\
\Pi(R^c) &= \Pi(R)^c
\end{aligned}
$$

Lemma 1 (Correctness of Nullability). *(1) If $f \in \Pi(R)$ for some USR R, then for every valuation v, $\epsilon \in L(R[f], v)$. (2) If $\epsilon \in L(R, v)$ for some valuation v, then there is some $f \in \Pi(R)$ such that $v \in Val_f$.*

Using substitution nullability, we define the Antimirov derivative for USRs. The derivative returns a set of pairs $(P, f) \in USR \times Subs$. The USR component

behaves similarly to classical Antimirov partial derivatives. The substitution component gives the constraints required to be applied on the input USR for the corresponding USR component to capture the derivative. In other words, we can find which partial derivative the suffix of a string corresponds to based on the valuation used to determine the language containing the string.

Definition 5. *The Antimirov derivative is a function* $\Delta : USR \times CharExpr \to P(USR \times Subs)$ *defined by:*

$$\Delta(\varnothing, x) = \varnothing \qquad \Delta(P^c, x) = \{(D(P^c, x), \mathbb{1})\}$$

$$\Delta(\sigma, x) = \begin{cases} \{(\epsilon, \langle x \mapsto \sigma \rangle)\}, x = \overline{c_i} \\ \{\epsilon, \mathbb{1}\}, x = \sigma \\ \varnothing, x \neq \sigma \end{cases} \qquad \begin{aligned} \Delta(\overline{w_i}[j], x) &= \{(D(\overline{w_i}[j], x), \mathbb{1})\} \\ \Delta(\overline{w_i}[j:], x) &= \{(D(\overline{w_i}[j:], x), \mathbb{1})\} \\ \Delta(\overline{w_i}, x) &= \{(\overline{w_i}, \langle \overline{w_i} \mapsto x\overline{w_i} \rangle)\} \end{aligned}$$

$$\Delta(\overline{c_i}, x) = \{\epsilon, \langle \overline{c_i} \mapsto x \rangle\} \qquad \Delta(P \cup Q, x) = \Delta(P, x) \cup \Delta(Q, x)$$

$$\begin{aligned} \Delta(PQ, x) &= \{(P'Q[f], f) | (P', f) \in \Delta(P, x)\} \\ &\quad \cup \bigcup_{g \in \Pi(P)} \{(Q'[g - f], f \# g) | f \# g \neq \bot, (Q', f) \in \Delta(Q, x)\} \\ \Delta(P^*, x) &= \{(P'P[f]^*, f) | (P', f) \in \Delta(P, x)\} \\ \Delta(P \cap Q, x) &= \{(P'[g - f] \cap Q'[f - g], f \# g) | \\ &\quad (P', f) \in \Delta(P, x), (Q', g) \in \Delta(Q, x), h \neq \bot\} \\ \Delta(\boldsymbol{IF}(\phi, P, Q), x) &= \{(\boldsymbol{IF}(\phi, P', \varnothing), f) | \\ &\quad (P', f) \in \Delta(P, x)\} \cup \{(\boldsymbol{IF}(\phi, \varnothing, Q'), g) | (Q', g) \in \Delta(Q, x)\} \end{aligned}$$

Definition 6. *The null projection of* R *outputs either* ϵ *or* $\varnothing$ *depending on whether there exists a valuation* ν *such that* $\epsilon \in L(R, \nu)$*. We denote it* $N \downarrow (R) := \epsilon$ *if* $\Pi(R)$ *is nonempty and* $\varnothing$ *otherwise.*

Corollary 1. $\sigma^{-1}(L \downarrow (R)) = \bigcup_{(R', f) \in \Delta(R, \sigma)} L \downarrow (R')$ *where* σ^{-1} *refers to the language derivative with respect to* σ*.*

By concatenating σ, taking the union over the alphabet and adding in the empty string case using null projection, we obtain the following identity from the corollary: $L \downarrow (R) = N \downarrow (R) \cup \bigcup_{\sigma \in \Sigma} \bigcup_{(R', f) \in \Delta(R, \sigma)} \sigma L \downarrow (R')$.

Theorem 7 (Correctness of Derivative).

- *If* $(P, f) \in \Delta(R, x)$ *and* $s \in L(P, \nu)$ *then there exists* $F_f(\nu) \in Val_f$ *such that* $F_f(\nu)(x)s \in L(R, F_f(\nu))$*.*
- *If* $s \in L(R, \nu)$ *and* $s \neq \epsilon$*, then there exists* $(P, f) \in \Delta(R, x)$ *such that enforcing* $\nu(x) = s[0]$ *makes it be in* Val_f *and* $s[1:] \in L(P, F'_f(\nu))$*. If* x *is a symbol in* Σ*, then no such enforcement is required.*

5.3 Decision procedures and results

In the Appendix, we show how derivatives can be used to solve satisfiability via a search algorithm on the derivative space (with Antimirov and Brzozowski variants); as with other derivative-based algorithms, our algorithm proceeds by repeatedly taking the derivative of any unvisited regexes until a nullable USR is discovered or the entire state space is explored. The algorithm uses a *union find* procedure to unify string substitutions from different derivative branches on-the-fly, so we do not need to check whether different branches are satisfiable. This also produces an algorithm for matching when applied to the regex $s \cap R$ for a concrete string s. We obtain the following result (proof in the Appendix).

Theorem 8. *The satisfiability algorithm is sound and partially complete: if $L \downarrow (R) \neq \varnothing$, it returns SAT; and if $L \downarrow (R) = \varnothing$, it either returns UNSAT or diverges.*

It is desirable that the derivative suffers minimal blowup in the size of the regex produced. We obtain the following results towards this end, which help justify the Antimirov approach. The first result is immediate upon inspection of the definition of Antimirov derivative and substitution-nullable; the proof of the second is in the Appendix.

Theorem 9. *If R is a USR which contains no if-then-else or complement sub-expressions, and $(R', f) \in \Delta(R, x)$, then R' contains no if-then-else or complement sub-expressions.*

Theorem 10. *Let R be any USR with k string variables containing no if-then-else or complement sub-expressions. Let R' be an nth Antimirov derivative of R. Then $|R'| \leq n \cdot k + c$, where c is a constant dependent on R but not n.*

6 Implementation and Evaluation

We implemented USRs and the Brzozowski and Antimirov satisfiability algorithms in a prototype SMT solver for regex constraints, comprising approximately 5250 lines of Rust code. The implementation runs in both Antimirov and Brzozowski modes; it uses a parser to convert SMT-LIB syntax files into Uninterpreted String Regexes, which are analyzed using derivatives to determine satisfiability. Our implementation incorporates two additional optimizations which differ from what has been presented in the theory in the rest of the paper. First, we include an optimization to store range constraints (e.g., `[a-z]`) in a more compact form, rather than enumerating the finite alphabet (this required modifying substitutions to allow for ranges in the Antimirov derivative). Second, we include some optimizations for predicate simplification involving $\textbf{IF}\,(\phi, R_1, R_2)$ terms. The Brzozowski solver additionally makes calls to an SMT solver to determine whether constraints over the character theory are satisfiable; for this purpose we use the Rust bindings for Z3 version 4.13.4. Our evaluation aims to address the following research questions:

RQ1 Of existing standard SMT benchmarks, how many can be solved directly via USR derivatives?

RQ2 Does our solver based on USRs improve the state of the art when combined with a portfolio of existing solvers?

RQ3 How do existing state-of-the-art SMT solvers compared with our solver specifically perform on benchmarks which contain USR constraints, either implicitly or explicitly?

To answer **RQ1**, we compiled a set of 5538 benchmarks. We collected standard benchmarks from SMTLib and prior work: 960 from Norn, 1976 from Slog, 343 from SyGuS, and 2172 from regex-smt-benchmarks and RegExLib [66,65]. To these we added 57 handwritten USR benchmarks described below.

To answer **RQ2**, we compared the times for our solvers and existing solvers on individual benchmarks. By plotting this data into a scatter plot, we can measure the number of benchmarks that are significantly faster with one or the other solver, which corresponds to time savings in a portfolio setting.

To answer **RQ3**, we wrote a series of handwritten and new USR-specific benchmarks. These include from the GitHub thread mentioned in the introduction [44], some benchmarks based on the SMT syntax translations from Section 4, and other difficult cases. For the SMT syntax benchmarks we generate both an *explicit* case – which directly contains one or more USRs – and an *implicit* cases – which does not contain string variables, but contain constructs such as `str.replace` which can be encoded via USRs; the latter are a useful point of comparison for solvers which do not support USRs.

Experimental setup and baselines. We use version 4.13.4 of Z3 and version 1.2.0 of CVC5, ran each solver with a 10s timeout, and compared the answer with the correct label (if provided with the benchmark); otherwise, we compared with the answer provided by CVC5, as it appears to be sound for the benchmark sets we used. If the baseline solver did not return a result, we marked the answer as "unchecked" and conservatively considered it correct. An answer of "unknown" is counted as an error (i.e. unsupported case). In summary, a correct result can be either sat, unsat, or unchecked, while an incorrect result can be either wrong, a timeout, or an error. We followed existing SMT community practices [20] in generating the plots. The experiments were run on an MSI Raider GE76 12UE with an Intel Core i7 CPU and 16GB of RAM.

Results. The results for **RQ1** are shown in Figure 4. Both CVC5 and Z3 achieve over 5000 benchmarks solved with a large fraction of benchmarks solved within the first 0.1 seconds. `usr-a`, which uses the Antimirov derivative, tapers off at close to 4000 of these, and `usr-b`, which uses Brzozowski derivatives, at less than 2000. The results for **RQ2** are shown in Figure 5. In both plots, CVC5 and Z3 completed many tests faster than `usr-a` as represented by the large line of dots close to the x-axis; however, in both plots two orthogonal clusters of points are

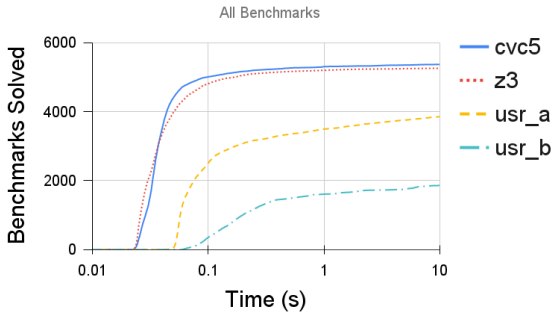

Fig. 4: Cactus plot of benchmarks solved in time t or less (t on the x-axis on a logarithmic scale) for CVC5, Z3, `usr-a` (ours), and `usr-b` (ours).

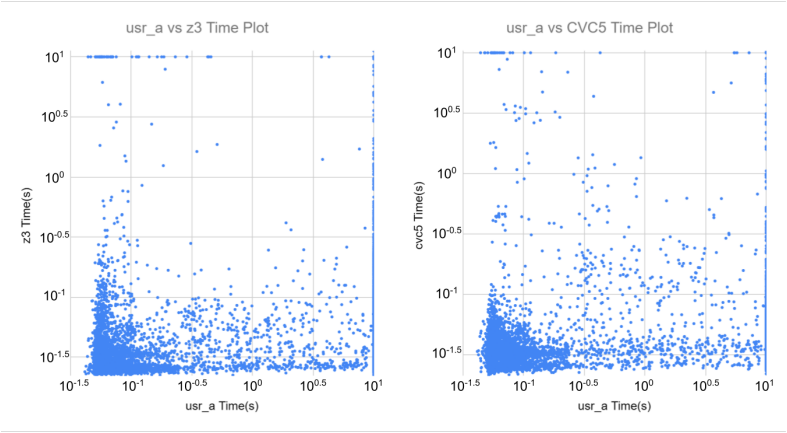

Fig. 5: Time scatter plots for `usr-a` vs. CVC5 (left) and Z3 (right). Each dot represents a single test where the x-axis is the time for `usr-a` and the y-axis is the time for either CVC5 or Z3.

visible, above and below the $y = x$ line. As seen in Figure 5 with dots above the $y = x$, of 5538 benchmarks, `usr-a` solved 370 faster than Z3, and 245 faster than CVC5. Finally, for **RQ3** in Figure 6, as mentioned earlier, we split our handwritten cases into explicit and implicit cases; CVC5 solves even explicit cases well, with 41 benchmarks against 38 for `usr-a`. Combined, `usr-a` solves 50/57 cases and `usr-b` solves 42/57.

Conclusions. Although neither the `usr-a` or `usr-b` solvers achieves state-of-the-art performance compared to CVC5 and Z3, they exhibit strong performance on a significant fraction of benchmarks. Through pure reduction to USRs, `usr-a` was able to solve 69% of the benchmarks and `usr-b` was able to solve 33% of

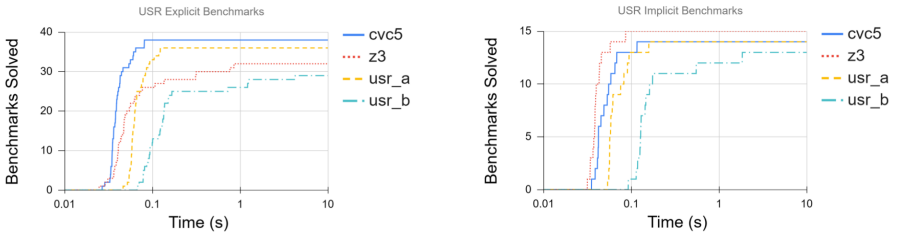

Fig. 6: Cumulative number of tests solved over "explicit" (left) and "implicit" (right) USR benchmarks for CVC5, Z3, `usr-a`, and `usr-b`.

the benchmarks taken. For RQ2, we find 100 benchmarks at least half an order of magnitude faster than Z3, and 84 at least half an order of magnitude faster than CVC5; these are cases that would see significant benefits if using `usr-a` in a portfolio setting. For RQ3, interestingly, although uninterpreted string variables are not supported officially by SMTLib, both state-of-the-art solvers correctly solve many explicit benchmarks, suggesting that their algorithms were adapted to incorporate regexes over uninterpreted strings at some point. However, neither Z3 nor CVC5 solves all USR benchmarks in our dataset with 47/57 solved by Z3 and 52/57 solved by CVC5.

Threats to validity. The `usr-a` and `usr-b` solvers crash or return wrong results on some existing SMT-Lib benchmarks that we tested due to unsupported cases: we currently do not support `Bool` and Int variables within SMT constraints as well as unicode escapes in string literals which gets treated as multiple concatenated character literals as opposed to individual unicode characters.

7 Related Work

Pattern regex, capture groups, back-references, and other classes. USRs are most closely related to other extensions of regex involving *patterns* (e.g., $p\mathtt{a}p$ where p is a regex pattern) and *capture groups with back-references* (e.g., $(\Sigma^*)_1\mathtt{a}\backslash 1$, where $\backslash 1$ is a back-reference to the string matched by the capture group subscripted by $()_1$). Both of these examples match the same language as the USR $\overline{w}\mathtt{a}\overline{w}$, but this does not generalize; in Section 4 we give several inclusions between our three classes – S-USR, I-USR, and B-USR – to the existing studied language classes, showing a close relationship, but no case of equality. We give a more detailed comparison below with these and other classes.

Regexes extended with backreferences were defined by Câmpeanu, Salomaa, and Yu [23] together with the CSY pumping lemma; this class has been called Ref-REG and been studied by Schmid [61,62], Freydenberger [39] and others [25,47].

Semantics is ill-defined in cases where the back-reference is nested inside either a union or a star [24], but this can be solved via ref-words [62]. We establish a concrete comparison of USRs to Freydenberger's `RegEx(k)` hierarchy, a subset of Ref-REG which contains S-USR and is contained in I-USR. *Pattern expressions* were defined by Câmpeanu and Yu [24], given as a sequence of patterns each matching a regex containing other pattern variables. The resulting class PATEXP has been defined differently by [18]; we relate our classes to both definitions by showing that S-USR $\subseteq$ `PATEXP-CSY` $\subseteq$ `PATEXP-BDH` $\subseteq$ I-USR. *Regexes under homomorphic replacement* gives rise to the class `HREG`, generalizing `PATEXP-BDH` and contained in I-USR. *Regexes with capture groups and lookarounds* have been recently shown to support matching in some cases with linear complexity [12,70,27,54]. Another widely considered extension of regexes is with intersection and complement (often also called *extended* regexes) [67,64,48]. These are equivalent in expressiveness to classical regular expressions; as we allow Boolean operations, USRs are more expressive but no less succinct. Symbolic regular expressions and their related automata models [37,36,38] are also relevant; these allow for uninterpreted characters over a Boolean algebra, as opposed to uninterpreted strings. To our knowledge, regexes over uninterpreted string variables as defined in this paper have not been considered.

String and Regex SMT Solvers. A wide variety of practical string and regex solvers have been developed by the SMT string solving [68] and Constraint Programming (CP) communities, and have various applications [69,9]. Actively maintained solvers include Z3 (Z3's sequence solver) [57,66], Z3str3/str4 [76,16,13], CVC5 [10,50], Ostrich [58,30,29], and Z3-Noodler [73,34] Many modern techniques descend from those developed by earlier solvers, such as Z3-Trau [75,2,1], CVC4 [35,11,49], Sloth [42], Norn [5], Z3str2 [77], and Hampi [46]. See Amadini [7] for a survey. The underlying theory is based on identifying practical decision procedures for various fragments (e.g., [14,17,51,41,31,49]), with associated decidability results [53,63,6,40,28] and open problems [22,40,28]. While nonemptiness of USRs is undecidable, our present goal is to achieve a satisfiability algorithm that works on-the-fly (see also [43]) and is appropriate for practical benchmarks.

Derivatives of Regular Expressions. Derivatives were first defined by Brzozowski in 1964 [21], and notably extended by Antimirov to the nondeterministic case in 1996 [8]. Derivatives have been lifted to regexes with intersection and complement in different ways [45,26,66], and can also be used for matching [56,60,59]. The idea of if-then-else terms for our Brzozowski derivatives comes from [66,71].

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Dolby, J., Janků, P., Lin, H.H., Holík, L., Wu, W.C.: Efficient handling of string-number conversion. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 943–957 (2020)

2. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: Smt solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5. IEEE (2018)
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 150–166. Springer (2014)
4. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An smt solver for string constraints. In: International conference on computer aided verification. pp. 462–469. Springer (2015)
5. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An smt solver for string constraints. In: International Conference on Computer Aided Verification. pp. 462–469. Springer (2015)
6. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17. pp. 277–293. Springer (2019)
7. Amadini, R.: A survey on string constraint solving. ACM Computing Surveys (CSUR) **55**(1), 1–38 (2021)
8. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science **155**(2), 291–319 (1996)
9. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for aws access policies using smt. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
10. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
11. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)
12. Barrière, A., Pit-Claudel, C.: Linear matching of javascript regular expressions. Proceedings of the ACM on Programming Languages **8**(PLDI), 1336–1360 (2024)
13. Berzish, M.: Z3str4: a solver for theories over strings (2021)
14. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: String theories involving regular membership predicates: From practice to theory and back. In: Combinatorics on Words: 13th International Conference, WORDS 2021, Rouen, France, September 13–17, 2021, Proceedings 13. pp. 50–64. Springer (2021)
15. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: Towards more efficient methods for solving regular-expression heavy string constraints. Theoretical Computer Science **943**, 50–72 (2023)
16. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59. IEEE (2017)
17. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An smt solver for regular expressions and linear arithmetic over string length. In: International Conference on Computer Aided Verification. pp. 289–312. Springer (2021)

18. Bordihn, H., Dassow, J., Holzer, M.: Extending regular expressions with homomorphic replacement. RAIRO-Theoretical Informatics and Applications **44**(2), 229–255 (2010)
19. Bordihn, H., Dassow, J., Holzer, M.: Extending regular expressions with homomorphic replacement. RAIRO - Theoretical Informatics and Applications **44**(2), 229–255 (5 2010), http://eudml.org/doc/250764
20. Brain, M., Davenport, J.H., Griggio, A.: Benchmarking solvers, sat-style. In: SC^2@ ISSAC (2017)
21. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM (JACM) **11**(4), 481–494 (1964)
22. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. In: The Collected Works of J. Richard Büchi, pp. 671–683. Springer (1990)
23. Câmpeanu, C., Salomaa, K., Yu, S.: A formal study of practical regular expressions. International Journal of Foundations of Computer Science **14**(06), 1007–1018 (2003)
24. Câmpeanu, C., Yu, S.: Pattern expressions and pattern automata. Information Processing Letters **92**(6), 267–274 (2004)
25. Carle, B., Narendran, P.: On extended regular expressions. In: International Conference on Language and Automata Theory and Applications. pp. 279–289. Springer (2009)
26. Caron, P., Champarnaud, J.M., Mignot, L.: Partial derivatives of an extended regular expression. In: Language and Automata Theory and Applications, LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer (2011)
27. Chattopadhyay, A., Li, A.W., Mamouras, K.: Verified and efficient matching of regular expressions with lookaround. In: Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 198–213 (2025)
28. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (2017)
29. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proceedings of the ACM on Programming Languages **6**(POPL), 1–31 (2022)
30. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
31. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
32. Chen, Y.F., Chocholatỳ, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Solving string constraints with lengths by stabilization. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 2112–2141 (2023)
33. Chen, Y.F., Chocholatỳ, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 24–33. Springer (2024)
34. Chen, Y.F., Havlena, V., Hečko, M., Holík, L., Lengál, O.: A uniform framework for handling position constraints in string solving. Proceedings of the ACM on Programming Languages **9**(PLDI), 550–575 (2025)
35. CVC4: (2020), https://github.com/CVC4/CVC4

36. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 541–553 (2014)
37. D'Antoni, L., Veanes, M.: Automata modulo theories. Communications of the ACM **64**(5), 86–95 (2021)
38. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: International Conference on Computer Aided Verification. pp. 47–67. Springer (2017)
39. Freydenberger, D.D.: Extended regular expressions: Succinctness and decidability. Theory of Computing Systems **53**(2), 159–193 (2013)
40. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: What is decidable about strings? (2011)
41. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proceedings of the ACM on Programming Languages **2**(POPL), 1–32 (2017)
42. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proceedings of the ACM on Programming Languages **2**(POPL), 1–32 (2017)
43. Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. pp. 377–386 (2010)
44. jiwonparc, U., Su, Z., et al.: Invalid model in string formula (GitHub Issue 5140 — Z3Prover/z3). https://github.com/Z3Prover/z3/issues/5140 (2021), [Accessed 09-07-2025]
45. Keil, M., Thiemann, P.: Symbolic solving of extended regular expression inequalities. In: FSTTCS'14. pp. 175–186. LIPIcs (2014)
46. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 105–116 (2009)
47. Komendantsky, V.: Matching problem for regular expressions with variables. In: International Symposium on Trends in Functional Programming. pp. 149–166. Springer (2012)
48. Kupferman, O., Zuhovitzky, S.: An improved algorithm for the membership problem for extended regular expressions. In: International Symposium on Mathematical Foundations of Computer Science. pp. 446–458. Springer (2002)
49. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A dpll (t) theory solver for a theory of strings and regular expressions. In: International Conference on Computer Aided Verification. pp. 646–662. Springer (2014)
50. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: International Symposium on Frontiers of Combining Systems. pp. 135–150. Springer (2015)
51. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation xss. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 123–136 (2016)
52. Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Majumdar, R., Nowotka, D.: Solving string constraints using sat. In: International Conference on Computer Aided Verification. pp. 187–208. Springer (2023)
53. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)

54. Mamouras, K., Chattopadhyay, A.: Efficient matching of regular expressions with lookaround assertions. Proceedings of the ACM on Programming Languages **8**(POPL), 2761–2791 (2024)
55. Matiyasevich, Y.: Diophantine sets. Russian Mathematical Surveys **27**(5), 124 (1972)
56. Moseley, D., Nishio, M., Perez Rodriguez, J., Saarikivi, O., Toub, S., Veanes, M., Wan, T., Xu, E.: Derivative based nonbacktracking real-world regex matching with backtracking semantics. Proceedings of the ACM on Programming Languages **7**(PLDI), 1026–1049 (2023)
57. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS'08. pp. 337–340. LNCS, Springer (2008)
58. Ostrich: (2020), https://github.com/uuverifiers/ostrich/
59. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. Journal of Functional Programming **19**(2), 173–190 (2009)
60. Saarikivi, O., Veanes, M., Wan, T., Xu, E.: Symbolic regex matcher. In: Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I 25. pp. 372–378. Springer (2019)
61. Schmid, M.L.: Inside the class of regex languages. In: Yen, H.C., Ibarra, O.H. (eds.) Developments in Language Theory. pp. 73–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
62. Schmid, M.L.: Characterising regex languages by regular languages equipped with factor-referencing. Information and Computation **249**, 1–17 (2016)
63. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: International Workshop on Word Equations and Related Topics. pp. 85–150. Springer (1990)
64. Sen, K., Roşu, G.: Generating optimal monitors for extended regular expressions. Electronic Notes in Theoretical Computer Science **89**(2), 226–245 (2003)
65. Stanford, C.: GitHub - cdstanford/regex-smt-benchmarks: A collection of regular expression benchmarks for SMT string solvers. https://github.com/cdstanford/regex-smt-benchmarks (2025), [Accessed 26-01-2026]
66. Stanford, C., Veanes, M., Bjørner, N.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 620–635. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454066, https://doi.org/10.1145/3453483.3454066
67. Stockmeyer, L., Meyer, A.: Word problems requiring exponential time. Preliminary report, in 5th Symp. on Theory of Computing (1973)
68. Tinelli, C., Barrett, C., Fontaine, P.: SMT-LIB The Satisfiability Modulo Theories Library — smt-lib.org. https://smt-lib.org/theories-UnicodeStrings.shtml (2024), [Accessed 09-07-2025]
69. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1232–1243 (2014)
70. Varatalu, I.E., Veanes, M., Ernits, J.: Re#: High performance derivative-based regex matching with intersection, complement, and restricted lookarounds. Proceedings of the ACM on Programming Languages **9**(POPL), 1–32 (2025)
71. Veanes, M.: On symbolic derivatives and transition regexes. Technical Report

72. Veanes, M., Ball, T., Ebner, G., Zhuchko, E.: Symbolic automata: Omega-regularity modulo theories. Proceedings of the ACM on Programming Languages **9**(POPL), 33–66 (2025)
73. VeriFIT: GitHub - VeriFIT/z3-noodler: The Z3-Noodler String Solver. https://github.com/VeriFIT/z3-noodler (2024), [Accessed 11-07-2025]
74. Z3: (2020), https://github.com/z3prover/z3
75. Z3-Trau: (2020), https://github.com/diepbp/z3-trau
76. Z3str3: (2020), https://sites.google.com/site/z3strsolver/
77. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design **50**(2-3), 249–288 (2017)

A Appendix

A.1 Example of String Constraints as USRs (Sec 2)

This example shows how constraints over certain functions from the SMTLIB string theory can be reduced to satisfiability of USRs. Suppose we have the constraint `s2 = (str.substr s1 i n)`. This constraint requires `s2` to be the substring of `s1` that starts from `s1`'s `ith` index with length `n` characters. When `i` is out of bounds, `s2` should be the empty string and if `n` reaches the end of `s1`, `s2` should reach the end of `s1`. More concretely, if `s1`="apple", `i`=1 and `n`=3, then `s2`="ppl". To encode as a USR, we want the USR to contain the string variables s_1 and s_2 where the language is nonempty when the assignment of these string variables satisfies the string constraint. This means the encoding is satisfiable if and only if the constraint is satisfiable. The encoding is

$$(s_1 \cap .^{i+n}.^*)(s_2 \cap .^n)(.^i s_2.^* \cap s_1) \cup (s_1 \cap (.^i.^*)^c)(s_2 \cap \epsilon) \cup (s_1 \cap (.^{i+n}.^*)^c)(.^i s_2 \cap s_1)$$

Intuitively, we split the string constraint into multiple length and position constraint on s_1 and s_2 then connect them by logical ANDs and ORs which correspond to concatenation and union in a USR. Each term corresponds to a length constraint to enforce s_2 to be in the correct part of s_1 based on the given indices.

To fully remove string constraints in an SMT setting, we also need to encode the constraint's negation. We can encode the negation, `s2 != (str.substr s1 i n)` using the same method. This yields

$$\begin{aligned}&(s_1 \cap .^{i+n}.^*)(s_2 \cap .^n)((.^i s_2.^*)^c \cap s_1) \cup (s_1 \cap .^{i+n}.^*)(s_2 \cap (.^n)^c)\\ &\cup (s_1 \cap (.^i.^*)^c)(s_2 \cap \epsilon^c) \cup (s_1 \cap (.^{i+n}.^*)^c)(s_1 \cap .^i.^*)((.^i s_2)^c \cap s_1)\end{aligned}$$

There are two standard cases in the first two terms of the union where s_1 is at least length $i+n$, then s_2 can either be of length n or not. The two edge cases are the same, but with complemented constraints on s_2 like $(s_2 \cap \epsilon^c)$ in the third term and $((.^i s_2)^c \cap s_1)$ in the fourth term of the union. We will see further examples of encoding SMT string constraints in Section 4.

A.2 Proof of Proposition 1 (Sec 4)

Proof.
- `(str.= s1 s2)`: Let $R_1 = \texttt{s1} \cap \texttt{s2}$ and $R_2 = \texttt{s1} \cap (\texttt{s2})^c$. If `s1` is equal to `s2`, $L(R_1, v) = \{v(\texttt{s1})\} \cap \{v(\texttt{s1})\} = \{v(\texttt{s1})\}$, which is nonempty. Conversely, if `s1` is not equal to `s2`, then $L(R_1, v) = \varnothing$. Similarly for R_2; for instance, if $L(R_2, v)$ is nonempty, then that means $v(\texttt{s1})$ is in the language $\{v(\texttt{s2})^c\}$, so `s1` is not equal to `s2`.
- `(str.= (str.++ s1 s2) s3)`: Let $R_1 = (\texttt{s1} \cdot \texttt{s2}) \cap \texttt{s3}$ and $R_2 = (\texttt{s1} \cdot \texttt{s2})^c \cap (\texttt{s3})$. For any valuation, each of R_1 and R_2 matches either the singleton language $\{\texttt{s3}\}$ or the empty set, depending on whether $v(\texttt{s3}) = v(\texttt{s1}) \cdot v(\texttt{s2})$.
- `(= (str.len s1) n)`: Let $R_1 = \texttt{s} \cap \Sigma^{\texttt{n}}$ and $R_2 = \texttt{s} \cap (\Sigma^{\texttt{n}})^c$.

- `(< (str.len s1) n)`: Let $R_3 = \mathtt{s} \cap (\Sigma^{\mathtt{n}} \cdot \Sigma^*)^c$ and $R_4 = \mathtt{s} \cap (\Sigma^{\mathtt{n}} \cdot \Sigma^*)$.
- `(> (str.len s1) n)`: Let $R_5 = \mathtt{s} \cap (\Sigma^{\mathtt{n+1}} \cdot \Sigma^*)$ and $R_6 = \mathtt{s} \cap (\Sigma^{\mathtt{n+1}} \cdot \Sigma^*)^c$.
- `(= (str.at s1 n) c)`: Let $R_1 = \mathtt{s1} \cap (\Sigma^n c \Sigma^*)$ and $R_2 = \mathtt{s1} \cap (\Sigma^n \cdot c \cdot \Sigma^*)^c$.
- `(= (str.substr s1 i j) s2)`: The proof idea is similar to the previous case, except we have to consider cases for when i is in bounds of the length of the string but j is out of bounds, and for when both i and j are out of bounds (in these cases the SMT standard defines the output to be ϵ). Each case amounts to a concatenation constraint as before, where we use Σ^i for the first prefix, and Σ^{j-i} to constrain the length of the substring. All together, we get the regex

  $$R_1 = (\mathtt{s1} \cap (\Sigma^i \mathtt{s2} \Sigma^*))(\mathtt{s2} \cap \Sigma^{j-i}) \cup (\mathtt{s1} \cap (\Sigma^i \mathtt{s2}))(\mathtt{s2} \cap (\Sigma^{j-i} \Sigma^*)^c \cup (\mathtt{s1} \cap (\Sigma^i \Sigma^*)^c.$$

  For R_2, we simply add a string variable $\mathtt{s2}'$, and add a constraint that $\mathtt{s2}' \neq \mathtt{s2}$ (using the negative encoding for equality), concatenating it to the above constraint for $\mathtt{s2}'$.
- `(str.contains s1 s2)`: Let $R_1 = \mathtt{s1} \cap (\Sigma^* \mathtt{s2} \Sigma^*)$ and $R_2 = \mathtt{s1} \cap (\Sigma^* \mathtt{s2} \Sigma^*)^c$.
- `(str.prefixof s1 s2)`: Let $R_1 = \mathtt{s2} \cap (\mathtt{s1} \Sigma^*)$ and $R_2 = \mathtt{s2} \cap (\mathtt{s1} \Sigma^*)^c$,
- `(str.suffixof s1 s2)`: Let $R_1 = \mathtt{s2} \cap (\Sigma^* \mathtt{s1})$ and $R_2 = \mathtt{s2} \cap (\Sigma^* \mathtt{s1})^c$. For any fixed valuation, R_1 and R_2 match either to singletons or the empty set, as in the previous proofs.

A.3 Proof of Theorem 2 (Sec 4)

We first prove the following two lemma then prove the theorem.

Lemma 2. *Let R be a USR that does not contain $\overline{w}$ and let v be any valuation. Then*

$$L(\overline{w} \cap R, v) = \begin{cases} v(\overline{w}) & \text{if } v(\overline{w}) \in L(R, v) \\ \varnothing & \text{else.} \end{cases}$$

Proof. By unfolding the definition of L.

Lemma 3. *Define the class $\mathcal{C}$ of I-USRs such that they only contain character literals, union, concatenation, Kleene , and intersections of the form $(\overline{w_i} \cap R_i)$ where R_i is the same for all $\overline{w_i}$ and R_i does not contain $\overline{w_i}$, $R_i \in \mathcal{C}$, and $L \downarrow (R_i)$ is nonempty. Then the class of languages recognized by $\mathcal{C}$ is closed under homomorphic replacement $L_1 \Leftarrow_\sigma L_2$.*

Proof. Let $P, Q \in \mathcal{C}$ with $L_1 = L \downarrow (P)$ and $L_2 = L \downarrow (Q)$, WLOG assume that all string variables occurring in P and Q are distinct. We also need to assume L_2 is nonempty before proceeding; if $L_2 = \varnothing$ then $L_1 \Leftarrow_\sigma L_2 = \varnothing$ so we are done.

Let P' be P with all instances of σ replaced by $\overline{w} \cap Q$, where $\overline{w}$ is fresh. We want to show that $L \downarrow (P') = L_1 \Leftarrow_\sigma L_2$.

For $R \in \mathcal{C}$, say that a valuation v is *consistent with* R and $\overline{w_i}$ if for the subexpression $\overline{w_i} \cap R_i$, $v(\overline{w_i}) \in L(R_i, v)$. We first claim that, for any arbitrary $R \in \mathcal{C}$

$$L \downarrow (R) = \bigcup_{v \text{ consistent}} L(R, v).$$

To prove the subclaim, we first show the language of any inconsistent valuation is a subset of the language for some consistent valuation, and this language is nonempty. That is, for any $R \in \mathcal{C}$, and valuation v, there exists v' consistent with every string variable such that $L(R, v) \subseteq L(R, v') \neq \varnothing$. Such v' is constructed by modifying inconsistent $v(\overline{w_i})$ in a specific order and leaving consistent $v(\overline{w_i})$ identical in v'. The order is determined by a topological sort of a graph of string variable dependencies, where string variables are nodes and there is an edge between $(wi, \overline{w_j})$ when R_i contains $\overline{w_j} \cap R_j$. By the previous lemma, making a valuation consistent with a string variable will always increase the language. Thus, the v' construction gives $L(R, v) \subseteq L(R, v') \neq \varnothing$.

We have that every inconsistent valuation induces a valuation whose language is a superset. We also know the language projection unions over all valuations which includes consistent valuations. Thus,

$$L \downarrow (R) = \bigcup_{v \text{ consistent}} L(R, v).$$

Denote the set of valuations consistent over all string variables with USR R as $\mathcal{V}$. We claim that the set of valuations consistent with P' can be partitioned into sets $\mathcal{V}_i$ such that given any $v \in \mathcal{V}_i \subseteq \mathcal{V}$ we have $L(P, v) \Leftarrow_\sigma L(Q, v) = \bigcup_{v' \in \mathcal{V}_i} L(P', v')$.

We define the partition via an equivalence relation. Two valuations are equivalent if they agree on everything excluding how they map the new string variable introduced by P'.

Next we unwrap the definition of $\Leftarrow_\sigma$ from [18].

$$L(P, v) \Leftarrow_\sigma L(Q, v) = \{u_1 s u_2 \dots s u_{k+1} | u_1 \sigma u_2 \dots \sigma u_{k+1} \in L(P, v) \\ \sigma \text{ does not occur in } u_1 u_2 \dots u_{k+1} \text{ and } s \in L(Q, v)\}$$

We proceed to show all elements of $\bigcup_{v' \in \mathcal{V}_i} L(P', v')$ have the same properties by structural induction on P which implies our equality.

1. If $P = \varnothing, \epsilon, \sigma'$ where $\sigma' \neq \sigma$, then no replacement happens, $P = P'$, and the property holds.
2. If $P = \sigma$, then $P' = \overline{w_i} \cap Q$. Then, $\bigcup_{v' \in \mathcal{V}_i} L(P', v')$ contains exactly every string in $L(Q, v)$. Furthermore, σ is the only element in $L(P, v)$ so the property holds.

3. If $P = P_1 P_2$, let $P' = P_1' P_2'$ and $s \in \bigcup_{v' \in \mathcal{V}_i} L(P_1' P_2', v')$. Then, $s = s_1 s_2$ such that $s_1 \in \bigcup_{v' \in \mathcal{V}_i} L(P_1', v')$ and $s_2 = \bigcup_{v' \in \mathcal{V}_i} L(P_2', v')$. By inductive hypothesis, s_1 and s_2 both satisfy the property for P_1' and P_2' so their concatenation must also satisfy the property for P'.
4. If $P = P_1 \cup P_2$, let $P' = P_1' \cup P_2'$ and $s \in \bigcup_{v' \in \mathcal{V}_i} L(P_1' \cup P_2', v')$. $s \in \bigcup_{v' \in \mathcal{V}_i} L(P_1', v') \cup \bigcup_{v' \in \mathcal{V}_i} L(P_2', v')$. By inductive hypothesis s satisfies the property for either P_1' or P_2' so s must also satisfy the property for $P_1' \cup P_2' = P'$
5. If $P = P_1^*$, then $P' = P_1'^*$ and $s \in \bigcup_{v' \in \mathcal{V}_i} L(P', v')$ has the form $s = s_1^n$ for some n with $s_1 \in \bigcup_{v' \in \mathcal{V}_i} L(P_1'^*, v')$. By inductive hypothesis s_1 satisfies the property for P_1', so their concatenation also satisfies the property for $P_1'^*$.
6. If $P = \overline{w_i} \cap P_1$, then $P' = \overline{w_i} \cap P_1'$ and $s \in \bigcup_{v' \in \mathcal{V}_i} L(P', v')$ which contains only $v\overline{w_i} \in L(P', v)$. Thus, by inductive hypothesis it satisfies the property for P_1' so it also satisfies the property for P'.

By the above, we now know that

$$
\begin{aligned}
L_1 \Leftarrow_\sigma L_2 & \\
&= L \downarrow (P) \Leftarrow_\sigma L \downarrow (Q) \\
&= \left(\bigcup_{v \text{ consistent}} L(P, v) \right) \Leftarrow_\sigma \left(\bigcup_{v \text{ consistent}} L(Q, v) \right) \\
&= \bigcup_{v \text{ consistent with } P} \; \bigcup_{v \text{ consistent with } Q} (L(P, v) \Leftarrow_\sigma L(Q, v)) \\
&= \bigcup_{v \text{ consistent}} L(P, v) \Leftarrow_\sigma L(Q, v) \\
&= \bigcup_{v \text{ consistent}} \; \bigcup_{v' \in [v]} L(P', v') \\
&= \bigcup_{v \text{ consistent}} L(P', v) \\
&= L \downarrow (P').
\end{aligned}
$$

Where $[v]$ refers to the equivalence class containing v.

A.4 Proof of Theorem 4 (Sec)

Lemma 4. *Given a fixed USR R, there exists a deterministic algorithm to decide $s \in L(R, v)$ in space linear in n, where n is the maximum length of s and $v(\overline{w_i})$ for all i.*

Proof. We proceed by induction on R. For the base cases, if $R = \varnothing, \epsilon, C$ or $\overline{w_i}$, then $L(R, v)$ will evaluate to a string literal, so matching can be done in linear space.

Now, suppose that R_1 and R_2 are regular expressions that can be matched under a fixed valuation in linear space. Then, if $R = R_1 \cup R_2$, first match s with $L(R_1, v)$. If the string is accepted, we are done. Else, match s with $L(R_2, v)$. Because this is a sum of linear space operations, the union can be done in linear space.

If $R = R_1 \cap R_2$, then match s with both R_1 and R_2. If s is in $L(R_1, v)$ and $L(R_2, v)$, then $s \in L(R, v)$.

If $R = R_1 \cdot R_2$, then traverse every possible split of s into strings s_1, s_2. Match s_1 to $L(R_1, v)$ and s_2 to $L(R_2, v)$. If all are accepted, $s \in L(R, v)$. Storing the split of s takes $\log n$ space, so our algorithm is still linear in space.

If $R = (R_1)^*$, then test every possible split of s into strings $s_1, s_2, \ldots s_k$. Match each substring to $L(R_1, v)$. If all are accepted, $s \in L(R, v)$. Storing the splits can be done in linear space, so performing all matchings will be linear in space over n.

If $R = (R_1)^c$, then match s with $L(R_1, v)$. Then $s \notin L(R_1, v) \iff s \in L(R, v)$.

In all instances, the space required is at most a sum of linear space matches, so determining $s \in L(R, v)$ can be done in linear space.

Lemma 5. *Define the equivalence relation $\equiv_n$ on 2 languages L_1, L_2 where $L_1 \equiv_n L_2$ if for all strings $|s| \leq n$, $s \in L_1 \iff s \in L_2$. Then, for any USR R, and any 2 valuations v_1, v_2, if $\forall w, v_1(w)[: n+1] = v_2(w)[: n+1]$, then $L(R, v_1) \equiv_n L(R, v_2)$.*

Proof. We shall prove via structural induction on R.

If $R = \varnothing, \epsilon, \sigma$, then the valuation function is not applied on any $\overline{w_i}$, so $L(R, v_1) = L(R, v_2)$ is trivially true.

In the case that $R = \overline{w_i}$, for any s such that $|s| \leq n$, $s \in L(R, v_1) \iff v_1(\overline{w_i}) = s$. Because $v_1(\overline{w_i})$ and $v_2(\overline{w_i})$ agree on the first $n+1$ characters of $\overline{w_i}$, $v_2(\overline{w_i}) = s$ as well, so $s \in L(R, v_2)$.

Now, suppose R_1 and R_2 are $USRs$ such that $\forall \overline{w_i}, v_1(w)[: n+1] = v_2(w)[: n+1] \implies L(R, v_1) \equiv_n L(R, v_2)$. Additionally, let $|s| \leq n$. In each case, by symmetry, we only must prove one direction of the equivalence relation.

Let $R = R_1 \cup R_2$. Then, $s \in L(R, v_1) \implies s \in L(R_1 \cup R_2, v_1) \implies s \in L(R_1, v_1) \cup L(R_2, v_1)$. Suppose $s \in L(R_1, v_1)$. Then by $\equiv_n$, $s \in L(R_1, v_2)$. Similarly, $s \in L(R_2, v_1) \implies s \in L(R_2, v_2)$, so $s \in L(R_1, v_2) \cup L(R_2, v_2) \implies s \in L(R_1 \cup R_2, v_2) \implies s \in L(R, v_2)$.

In the case that $R = R_1 \cap R_2$, $s \in L(R, v_1) \implies s \in L(R_1 \cap R_2, v_1) \implies s \in L(R_1, v_1) \wedge s \in L(R_2, v_1) \implies s \in L(R_1, v_2) \wedge s \in L(R_2, v_2) \implies s \in L(R_1 \cap R_2, v_2) \implies s \in L(R, v_2)$.

In the case that $R = R_1 \cdot R_2$, $s \in L(R, v_1) \implies s \in L(R_1 \cdot R_2, v_1)$. We can then split s into s_1 and s_2, where $s_1 \in L(R_1, v_1)$ and $s_2 \in L(R_2, v_1)$. By the induction

hypothesis, $s_1 \in L(R_1, v_2) \wedge s_2 \in L(R_2, v_2) \implies s_1 \cdot s_2 \in L(R_1, v_1) \cdot L(R_2, v_1) \implies s \in L(R_1 \cdot R_2, v_2) \implies s \in L(R, v_2)$.

In the case that $R = (R_1)^*$, $s \in L(R, v_1) \implies s \in L((R_1)^*, v_1)$. Then, we can split s into $s_1, s_2, \ldots, s_k$, where each $s_i \in L(R_1, v_1)$. Because each is a substring, each must be on length at most n, so $s \in L(R, v_1) \implies s_1, \ldots, s_k \in L(R_1, v_1) \implies s_1, \ldots, s_k \in L(R_1, v_2) \implies s \in L((R_1)^*, v_2) \implies s \in L(R, v_2)$.

In the case that $R = (R_1)^c$, $s \in L(R, v_1) \implies s \notin L(R_1, v_1) \implies s \notin L(R_1, v_2) \implies s \in L((R_1)^c, v_2) \implies s \in L(R, v_2)$.

A.5 Proof of Theorem 5 (Sec 5)

The proofs for the empty set and empty are immediate through inspection. The proof for the character expression and string index is also immediate as it must evaluate to a real character, which is not the empty string. By the construction of the predicates in $n(\overline{w_i})$ and $n(\overline{w_i}[j\ :])$, it is ensured that ϵ is only in the language for valuations that will set the string variable or slice to the empty string. In all other cases, the predicate would fail and return $\varnothing$. For the recursive cases, through induction, the nullability of the expression would depend on the nullability of its sub-components. Specifically in the complement case:

$$L(n(R^c), v) = L(\epsilon \cap n(R)^c, v) = L(\epsilon) \cap L(n(R)^c, v) = \{\epsilon\} \cap L(n(R), v)^c$$

If $\epsilon \in L(R^c, v)$, then $\epsilon \notin L(R, v)$, so $L(n(R), v)$ will not contain ϵ. Thus $L(n(R), v)^c$ will, which means the language will evaluate to $\{\epsilon\}$ If $\epsilon \notin L(R^c, v)$, then $\epsilon \in L(R, v)$, so $L(n(R), v)$ will contain ϵ. Thus $L(n(R), v)^c$ will not contain ϵ, and the language will evaluate to $\varnothing$. Finally, in the If-Then-Else case,

$$\begin{aligned}
L(n(\mathbf{IF}\,(\phi, R_1, R_2)\,, v) &= L(\mathbf{IF}\,(\phi, n(R_1), n(R_2))\,, v) \\
= L(\mathbf{IF}\left(\phi, \left\{\begin{array}{ll}\epsilon, & \text{if } \epsilon \in L(R_1, v) \\ \varnothing, & \text{if } \epsilon \notin L(R_1, v)\end{array}\right\}, \left\{\begin{array}{ll}\epsilon, & \text{if } \epsilon \in L(R_2, v) \\ \varnothing, & \text{if } \epsilon \notin L(R_2, v)\end{array}\right\}\right), v) & \\
= \left\{\begin{array}{ll}\epsilon, & \text{if } \epsilon \in L(R_1, v) \wedge P(\phi, v) \\ \varnothing, & \text{if } \epsilon \notin L(R_1, v) \wedge P(\phi, v) \\ \epsilon, & \text{if } \epsilon \in L(R_2, v) \wedge \neg P(\phi, v) \\ \varnothing, & \text{if } \epsilon \notin L(R_2, v) \wedge \neg P(\phi, v)\end{array}\right\} & \\
= \left\{\begin{array}{ll}\epsilon, & \text{if } \epsilon \in L(\mathbf{IF}\,(\phi, R_1, R_2)\,, v) \\ \varnothing, & \text{if } \epsilon \notin L(\mathbf{IF}\,(\phi, R_1, R_2)\,, v)\end{array}\right\} &
\end{aligned}$$

A.6 Proof of Theorem 6 (Sec 5)

For the empty string and the empty set cases, the correctness of the derivative is trivially true. In the case of the character expression and string index, the construction of the predicate of $d(R, x)$ ensures that the only strings in the language of the derivative will have started with the character expression x in the original USR. Similarly, in the case of the string variable, there will only be

strings in the language of the derivative if it begins with the character expression x. Additionally, the output is the rest of the string, so through inspection, it is clear that the derivative is correct. Similar reasoning can be applied to the string slice case as well. For the recursive cases, we can use induction upon the subexpressions of the USR. The derivatives also follow naturally from the definition of the derivative of classical regular expressions. In the If-Then-Else case, Let $z \in L(d(\textbf{IF}\,(\phi, R_1, R_2), i), v)$. Then,

$$z \in L(\textbf{IF}\,(\phi, d(R_1, i), d(R_2, i)), v) \Rightarrow$$
$$L'(x,v) \cdot z = \left\{ \begin{array}{ll} L'(x,v) \cdot z \in L(d(R_1, i), v) & \text{if } L(\phi, v) \\ L'(x,v) \cdot z \in L(d(R_2, i), v) & \text{else} \end{array} \right\}$$

By induction on the base cases:

$$= \left\{ \begin{array}{ll} z' \in L(R_1, v) & \text{if } L(\phi, v) \\ z' \in L(R_2, v) & \text{else} \end{array} \right\} \in L(\textbf{IF}\,(\phi, R_1, R_2), v)$$

A.7 Simple Sub Rules (Sec 5)

The following are the complete set of rules for simple substitutions.

$$R[\emptyset] = R$$
$$\epsilon[f] = \epsilon$$
$$\sigma[f] = \sigma$$
$$\overline{w_i}[f] = \begin{cases} f(\overline{w_i}), \overline{w_i} \in \text{Dom}(f) \\ \overline{w_i}, \text{o.w.} \end{cases}$$
$$\overline{w_i}[j][f] = \begin{cases} \emptyset, f(\overline{w_i}) = \epsilon \\ \overline{w_i}[j-1], f(\overline{w_i}) = x\overline{w_i}, j > 0 \\ x, f(\overline{w_i}) = x\overline{w_i}, j = 0 \\ x, f(\overline{w_i}) = x\overline{w_i}, j = 0 \\ \emptyset, f(\overline{w_i}) = x, j > 0 \\ \sigma_{j+1}, f(\overline{w_i}) = \sigma_1 ... \sigma_n \overline{w_i}, j < n \\ \overline{w_i}[j-n], f(\overline{w_i}) = \sigma_1 ... \sigma_n \overline{w_i}, j \geq n \\ \sigma_{j+1}, f(\overline{w_i}) = \sigma_1 ... \sigma_n, j < n \\ \emptyset, f(\overline{w_i}) = \sigma_1 ... \sigma_n, j \geq n \end{cases}$$

$$\overline{w_i}[j:][f] = \begin{cases} \epsilon, f(\overline{w_i}) = \epsilon, j = 0 \\ \emptyset, f(\overline{w_i}) = \epsilon, j > 0 \\ \overline{w_i}[(j-1):], f(\overline{w_i}) = x\overline{w_i}, j > 0 \\ x\overline{w_i}, f(\overline{w_i}) = x\overline{w_i}, j = 0 \\ x, f(\overline{w_i}) = x\overline{w_i}, j = 0 \\ \epsilon, f(\overline{w_i}) = x\overline{w_i}, j = 1 \\ \emptyset, f(\overline{w_i}) = x, j > 0 \\ \sigma_{j+1}...\sigma_n\overline{w_i}, f(\overline{w_i}) = \sigma_1...\sigma_n\overline{w_i}, j < n \\ \overline{w_i}[(j-n):], f(\overline{w_i}) = \sigma_1...\sigma_n\overline{w_i}, j \geq n \\ \sigma_{j+1}...\sigma_n, f(\overline{w_i}) = \sigma_1...\sigma_n, j < n \\ \epsilon, f(\overline{w_i}) = \sigma_1...\sigma_n, j = n \\ \emptyset, f(\overline{w_i}) = \sigma_1...\sigma_n, j \geq n \end{cases}$$

$$c_i[f] = \begin{cases} f(c_i), c_i \in \mathrm{Dom}(f) \\ c_i, \text{o.w.} \end{cases}$$

$$PQ[f] = P[f]Q[f]$$
$$(P \cup Q)[f] = P[f] \cup Q[f]$$
$$P^*[f] = P[f]^*$$
$$(P \cap Q)[f] = P[f] \cap Q[f]$$
$$P^c[f] = P[f]^c$$
$$\mathbf{IF}\,(\phi, P, Q)\,[f] = \mathbf{IF}\,(\phi[f], P[f], Q[f])$$

$$(C = C)[f] = C[f] = C[f]$$

$$(|\overline{w_i}| = j)[f] = \begin{cases} T, j = 0, (\overline{w_i} \mapsto \epsilon) \in f \\ T, j = n, (\overline{w_i} \mapsto \sigma_1...\sigma_n) \in f \\ (|\overline{w_i}| = j-1), j \geq 1, (\overline{w_i} \mapsto x\overline{w_i}) \in f \\ (|\overline{w_i}| = j-n), n \geq j, (\overline{w_i} \mapsto \sigma_1...\sigma_n\overline{w_i}) \in f \\ F, \text{o.w.} \end{cases}$$

$$(\overline{w_i}[j] = C)[f] = (\overline{w_i}[j][f] = C[f])$$
$$(\overline{w_i}[j] = \overline{w_k}[l]) = (\overline{w_i}[j][f] = \overline{w_k}[l][f])$$
$$(\overline{w_i}[j] = \sigma)[f] = (\overline{w_i}[j][f] = \sigma[f])$$
$$(\phi_1 \wedge \phi_2)[f] = (\phi_1[f] \wedge \phi_2[f])$$
$$(\phi_1 \vee \phi_2)[f] = (\phi_1[f] \vee \phi_2[f])$$
$$(\neg\phi)[f] = \neg(\phi[f])$$

A.8 Proof of Corollary 1 (Sec 5)

Proof. We prove the left and right sides are subsets of each other.

Assume $s \in a^{-1}(L\downarrow)(R)$. By definition of language derivative and language projection, there exists some $v \in \text{Val}$ such that $\sigma s \in L(R, v)$. By correctness of derivative, there exists $(R', f) \in \Delta(R, \sigma)$ such that $v \in \text{Val}_f$ and $s \in L(R', F'_f(v))$. Lastly, by language projection, $s \in L\downarrow(R') \subseteq \bigcup_{(R',f)\in\Delta(R,\sigma)} L\downarrow(R')$.

Now assume $s \in \bigcup_{(R',f)\in\Delta(R,\sigma)} L\downarrow(R')$. Then $s \in L\downarrow(R')$ for some $(R', f) \in \Delta(R, \sigma)$. By language projection, $s \in L(R', v)$ for some v. By correctness of derivative, $\sigma s \in L(R, F_f(v))$. Thus, $\sigma s \in L\downarrow(R)$ and $s \in a^{-1}(L\downarrow(R))$.

A.9 Substitutions Definition and Auxiliary Operations

Definition 7. *A substitution is a set of ordered tuples written using the following grammars:*

$$\begin{aligned}
&\mathit{Subs} ::= \mathbb{1} | \langle \mathit{Pair} \rangle \\
&\mathit{Pair} ::= \mathit{Pair}, \mathit{Pair} | \overline{w_i} \mapsto E_i | \overline{c_i} \mapsto C \\
&\mathit{SubExpr}\ E_i ::= C E_i | \epsilon | \overline{w_i}
\end{aligned}$$

If a pair starts with $\overline{w_i}$ then the corresponding SubExpr must end on $\overline{w_i}$ with the same index or with ϵ. Character variables can only map to character expressions which is a subset of SubExpr.

Semantically, a substitution f restricts the space of all valuations Val to its range Val_f. The identity substitution $\mathbb{1}$ is the substitution with no pairs and does not restrict Val. We recursively define Val_f with substitution in the following way:

$$\text{Val}_f = \begin{cases}
\{v | v(\overline{w_i}) = \epsilon\}, f = \langle \overline{w_i} \mapsto \epsilon \rangle \\
\{v | v(\overline{w_i}) = \sigma_1 v(\overline{c_i}) ... \sigma_n\}, f = \langle \overline{w_i} \mapsto x_1 ... x_n \rangle \\
\{v | v(\overline{w_i}) = \sigma_1 v(\overline{c_i}) ... \sigma_n s, s \in \Sigma^*\}, f = \langle \overline{w_i} \mapsto x_1 ... x_n \overline{w_i} \rangle \\
\{v | v(\overline{c_i}) = \sigma\}, f = \langle \overline{c_i} \mapsto \sigma \rangle \\
\text{Val}_{f_1} \cap \text{Val}_{f_2}, f_1 \cup f_2 = f, |f| > 1
\end{cases}$$

where σ, σ_i are character literals, $\overline{c_i}$ is a character variable, and $\overline{w_i}$ are string variables.

Next, we define simple substitutions which are substitutions that syntactically behave well with USRs.

Definition 8 (Simple Substitutions). *Given a substitution f, f is a simple substitution if it satisfies the following properties:*

- *For any $\langle \overline{c_i} \mapsto x \rangle, \langle \overline{c_j} \mapsto y \rangle \in f$, $i \neq j$.*
- *For any $\langle \overline{w_i} \mapsto e_1 \rangle, \langle \overline{w_j} \mapsto e_2 \rangle \in f$, $i \neq j$.*
- *If $\langle \overline{c_i} \mapsto x \rangle \in f$, then for all $\langle \overline{w_i} \mapsto e \rangle \in f$ and $\langle \overline{c_j} \mapsto y \rangle \in f$, $\overline{c_i}$ is not a variable in e or x.*

The first two properties allows f to be viewed as a function where each ordered pair represents a mapping from a variable to a SubExpr. This allows syntactic substitution of the variables occurring in any USR R and predicate ϕ using f. We denote this syntactic operation with $R[f]$ and $\phi[f]$. More specifically, for every string variable $\overline{w_i}$ or character variable x in the domain f that appears in the syntax of a USR R or predicate ϕ, they are replaced with $f(\overline{w_i})$ or $f(\overline{c_i})$ respectively. When substituting into indexed string variables $\overline{w_i}[j]$, the index is adjusted accordingly or is replaced by a character expression depending on substitution. When substituting into length constraint predicates the length constraint is adjusted accordingly based on the length of the SubExpr.

The third property ensures the order of substituting each variable into a USR or predicate does not cause differing semantics. That is, if a variable is part of the domain of f, in cannot appear in the co-domain of f is any form.

A simple substitution f also induces a function $F_f : \mathrm{Val} \to \mathrm{Val}_f$ and a reverse map $F'_f : \mathrm{Val}_f \to \mathrm{Val}$ such that F_f is surjective and F'_f is injective.

$$F_f(v)(x) := \begin{cases} f(x)[\langle \overline{w_i} \mapsto v(\overline{w_i}) \rangle][\langle \overline{c_i} \mapsto v(\overline{c_i}) \rangle], x \in \mathrm{Dom}(f) \\ v(x), \mathrm{else} \end{cases}$$

$$F'_f(v)(x) := \begin{cases} v(x)[|f(x)| - 1 :], f(a) \text{ ends with } \overline{w_i} \\ v(x), \mathrm{else} \end{cases}$$

Lemma 6 (Correctness of F_f and F'_f). *For any simple substitution f,*

- $F_f(v)(a) \in \mathit{Val}_f$
- $F_f(F'_f(v)) = v$

Proof.

- Statement 1: The result is immediate by construction if the constraints in f do not contain mappings from variables. In the cases where f contains constraints involving variables, x, each case in the construction of f follows the definition of Val_f to ensure $F_f(v)(a) \in \mathrm{Val}_f$.
- Statement 2: In the first two cases of the construction of F'_f, it removes a prefix if f contains a mapping that adds a prefix to a string variable a. Otherwise, in the final case of the construction of F'_f, it maps identically as the input. The forward map, F_f, reattaches the removed prefixes while mapping that do not attaches prefixes remain the same. Thus, $F_f(F'_f(v)) = v$.

With simple substitutions defined we proceed to the desired properties of a simple substitution. The following lemma is an equality relationship between the languages of a USR substituted by a simple substitution, $L(R[f], v \in \mathrm{Val})$ and the languages of the same USR with no substitution but with a restricted set of valuations, $L(R, v \in \mathrm{Val}_f)$.

Lemma 7 (Correctness of Simple Substitutions). *Let f be an arbitrary simple substitution such that $Val_f \neq \varnothing$. The following statements are true.*

- *For all $v \in Val$, and for all predicates ϕ, $L(\phi[f], v) = L(\phi, F_f(v))$.*
- *For all $v' \in Val_f$, and for all predicates ϕ, $L(\phi[f], F'_f(v')) = L(\phi, v')$.*
- *For all $v \in Val$, and that for all USRs R, $L(R[f], v) = L(R, F_f(v))$.*
- *For all $v' \in Val_f$, and for all USRs R, $L(R[f], F'_f(v')) = L(R, v')$.*

Proof. We only need to prove statements 1 and 3 due to the fact that $F_f(F'_f(v')) = v'$ where $v \in \mathrm{Val}_f$. To see this, we prove statement 2 assuming statement 1 is true, the same argument follows for statements 3 and 4. Let $v' \in \mathrm{Val}_f$ be arbitrary. Then, $B(\phi, v') = B(\phi, F_f(F'_f(v'))) = B(\phi[f], F'_f(v'))$.

We proceed to prove statement 1 by structural induction on ϕ. We start from the base cases. When there are no variables involved valuation is irrelevant so the statement is true. When $\phi = (\overline{c_i} = \overline{c_j})$, the statement holds since we are just changing the substitution to happen on v. For predicates involving indexed string variables, the f will adjust the index. On the other hand, $F_f(v)$ will append to the strings in the image of v making the index in the predicate refer to elsewhere in $\overline{w_i}$. The change of index and appending will match up so that indexed string variables refer to the same position. For predicates involving length constraints, the argument is similar to indexed string variables in that the length constraint is adjusted on $\phi[f]$ while the length of the string is adjusted in $F_f(\overline{w_i})$ and these adjustments match up. For the recursive cases, the inductive hypothesis combined with the definition of logical operators prove them.

Next, we prove statement 3 by structural induction on R. For the bases cases where R does not contain variables, the proof is immediate. When $R = \overline{w_i}$ or $\overline{c_i}$, we prove by cases over every possible pair in f. The language contains a single string and it can be shown by definition of F_f that $F_f(v)(\overline{w_i})$ is that string. When $R = \overline{w_i}[j]$ or $\overline{w_i}[j:]$, the adjusted indices when applying substitutions to $wi[j]$ will ensure the string in the language will be the same as the string with the non-adjusted index but with modified valuation. For the recursive cases, applying definition of language combined with inductive hypothesis immediately yields the desired result through a chain of language equalities.

Next, we prove that all simple substitutions correspond to satisfiable constraints on the space of valuations which helps ensure the derivative that we define later only yields USRs that are reachable.

Lemma 8. *If f is a simple substitution, then Val_f is non-empty.*

Proof. Since f is simple, we have F_f and $F_f(v) \in \mathrm{Val}_f$ as desired.

In section 5.3, we define a Unify algorithm which transforms substitutions into equivalent simple substitutions or signals failure if it is impossible. Combined

with the satisfiability lemma of simple substitutions above, Unify effectively helps us filter out unsatisfiable substitutions. Using Unify we proceed to define the merge operation which combines the restrictions of two substitutions into one substitution when the combined restrictions are still satisfiable.

Definition 9 (Merge). *Given two substitutions f, g, the merge of f, g is defined* $f\#g = Unify(f \cup g)$

Lemma 9 (Correctness of Merge).

- *If* $f\#g \neq \bot$, *then* $Val_{f\#g} = Val_f \cap Val_g$
- *If* $f\#g = \bot$, *then* $Val_f \cap Val_g = \emptyset$

Proof. If $f\#g \neq \bot$, by property of Unify and definition of constraining the space of valuations, $\text{Val}_{f\#g} = \text{Val}_{f\cup g} = \text{Val}_f \cap \text{Val}_g$. If $f\#g = \bot$, then $\text{Unify}(f \cup g) = \bot$. By property of Unify and definition of constraining the space of valuations, $\text{Val}_{f\cup g} = \text{Val}_f \cap \text{Val}_g = \emptyset$.

The following merge property shows that if the membership of a string in a language does not depend on the valuation, then neither will the addition of substitutions affects its membership.

Lemma 10 (Merge Properties). *Let f_1, f_2 be simple substitutions such that $f_1\#f_2 \neq \bot$. For any USR R, if a string $s \in L(R[f_1], v)$ for all $v \in Val$, then $s \in L(R[f_1\#f_2], v)$ for all v. The same holds for f_2.*

Proof. Since $s \in L(R[f_1], v)$ for all $v \in \text{Val}$, by correctness of substitutions, $s \in L(R, v')$ for all $v' \in \text{Val}_{f_1}$. By correctness of merge, $\text{Val}_{f_1\#f_2} \subseteq \text{Val}_{f_1}$ so $s \in L(R, v'')$ for all $v'' \in \text{Val}_{f_1\#f_2}$. Since for all v, there exists v'' such that $L(R[f_1\#f_2], v) = L(R, v'')$, $s \in L(R[f_1\#f_2], v)$ for all v. The argument is identical for f_2.

Building on top of merge, we can define the complements of sets of substitutions. Since our simple substitution must have the ability of syntactic substitution onto USRs, complement must be defined by enumeration over all other substitutions that go against the complemented substitution.

Definition 10 (Complement of Sets of Substitutions). *Given a finite collection of substitutions T, its complement is a finite collection of substitutions defined recursively.*

$$T^c = \begin{cases} \{\mathbb{1}\}, T = \varnothing \\ \bigcup_{\gamma \in f} G_\gamma, T = \{f\} \\ \{g_1\#g_2 | g_1 \in T_1^c, g_2 \in T_2^c, g_1\#g_2 \neq \bot\}, T = T_1 \cup T_2, |T_1| \geq 1, |T_2| \geq 1 \end{cases}$$

where G_γ is a set of substitutions constructed by enumerating over every substitution that conflicts with $\gamma \in f$. If $\gamma = \overline{c_i} \mapsto \sigma$ where $\sigma \in \Sigma$, then $G_\gamma = \{\langle \overline{c_i} \mapsto \sigma' \rangle | \sigma' \neq \sigma\}$. For $\gamma = \overline{c_i} \mapsto \overline{c_j}$, $G_\gamma = \{\langle \overline{c_i} \mapsto \sigma, \overline{c_j} \mapsto \sigma' \rangle | \sigma' \neq \sigma\}$. If $\gamma = \overline{w_i} \mapsto x_1...x_n$ where each x_j is a character expression, then G_γ will be the set of substitutions that map $\overline{w_i}$ to any string less than n length, greater than n length or of n length but with at least one difference. To make a difference, character variables are enumerated over the alphabet and substituted into $x_1...x_n$. $\overline{w_i}$ is mapped to $x_1...x_n$ with at least 1 differing symbol at some x_j. The $\gamma = \overline{w_i} \mapsto x_1...x_n\overline{w_i}$ case is similar where the greater than n case is removed and $\overline{w_i}$ is mapped to $x_1...x_n\overline{w_i}$ with a differing symbol at some x_j.

Lemma 11 (Correctness of Complement). *Given a set of substitutions $F = \{f_1, ..., f_m\}$ along with its complement $F^c = \{f'_1, ..., f'_n\}$, the following two statements hold.*

- *$Val_{f_i} \cap Val_{f'_j} = \varnothing$ for any $f_i \in F$ and $f'_j \in F^c$*
- *$\bigcup_{f_i \in F} Val_{f_i} \cup \bigcup_{f'_i \in F^c} Val_{f'_i} = Val$*

Proof. When $F = \emptyset$, the first statement is vacuously true and the second statement is true because $\text{Val}_{\langle\rangle} = \text{Val}$. When F is non-empty, we prove by induction on the number of elements in F. We prove statements hold in the case $F = \{f_1\}$.

For the first statement, by inspection of the construction of G_γ, if $g \in G_\gamma$, then $\langle\gamma\rangle \# g = \bot$. Each $\gamma \in f_1$ and each $f'_i \in G_\gamma$ for some γ so $f_1 \# f'_i = \bot$ for every i as desired. For the second statement, by inspection of the construction of G_γ and definition of constraining Val, $v \in \text{Val}_g$ for some $g \in G_\gamma$ if and only if $v \notin \text{Val}_{\langle\gamma\rangle}$. By definition of F^c, $f'_i \in G_\gamma$ for some $\gamma \in f_1$ so any v must be in either Val_{f_1} or $\text{Val}_{f'_i}$

Now show the statements hold when $F = \{f_1, ..., f_{k+1}\}$ assuming they hold when F has less than or equal to k elements. Write $F = G \cup H$ where $G = \{f_1, ..., f_k\}$ and $H = \{f_{k+1}\}$. Let $G^c = \{g_1, ..., g_p\}$ and $H = \{h_1, ..., h_q\}$. For the first statement, we want to show $\text{Val}_{f_i} \cap \text{Val}_{g_j \# h_m} = \emptyset$ for any f_i, g_j, h_m in their respective sets. By inductive hypothesis

- If $i \leq k$, then $\text{Val}_{f_i} \cap \text{Val}_{g_j} = \emptyset$ for any j
- If $i = k + 1$, then $\text{Val}_{f_i} \cap \text{Val}_{h_m} = \emptyset$ for any m

By correctness of merge we have $\text{Val}_{f_i} \cap \text{Val}_{g_j \# h_m} = \text{Val}_{f_i} \cap \text{Val}_{g_j} \cap \text{Val}_{h_m} = \emptyset$. Since $g_j \# h_m \in F^c$ when not bottom, the first statement holds.

For the second statement, consider an arbitrary $v \in \text{Val}$. We want to show that either $v \in \text{Val}_{f_i}$ or $v \in \text{Val}_{g_j \# h_m}$ for some g_j and h_m By inductive hypothesis v must be in one of the spaces:$\text{Val}_{f_i}, \text{Val}_{g_j}, \text{Val}_{h_m}$ We split the proof into cases. If $v \in \text{Val}_{f_i}$ for any i we are done. If $v \in \text{Val}_{g_j}$ and $v \in \text{Val}_{h_m}$ then $v \in \text{Val}_{g_j \# h_m}$.

In order to support all the features of a USR, we additionally define the $SFromP$ function to convert the predicates in if-then-else expressions into a set of simple substitutions. The SFromP function is defined recursively as follows:

$\text{SFromP}(\text{F}) = \emptyset$
$\text{SFromP}(\text{T}) = \{\mathbb{1}\}$
$\text{SFromP}(\sigma_1 = \sigma_2) = \begin{cases} \{\mathbb{1}\}, \sigma_1 = \sigma_2 \\ \emptyset, \text{o.w.} \end{cases}$
$\text{SFromP}(\overline{c_i} = \sigma) = \{\langle \overline{c_i} \mapsto \sigma \rangle\}$
$\text{SFromP}(|\overline{w_i}| = j) = \{\langle \overline{w_i} \mapsto \sigma_1 \cdots \sigma_j \rangle | \sigma_k \in \Sigma\}$
$\text{SFromP}(\overline{w_i}[j] = \sigma) = \{\langle \overline{w_i} \mapsto \sigma_1 \cdots \sigma_{j-1} \sigma \overline{w_i} \rangle | \sigma_k \in \Sigma\}$
$\text{SFromP}(\overline{w_i}[j] = \overline{c_k}) = \{\langle \overline{w_i} \mapsto \sigma_1 \cdots \sigma_{j-1} \overline{c_k} \overline{w_i} \rangle | \sigma_k \in \Sigma\}$
$\text{SFromP}(\overline{w_i}[j] = \overline{w_k}[l]) = \{\langle \overline{w_i} \mapsto \sigma_1 \cdots \sigma_j \overline{w_i}, \overline{w_k} \mapsto \sigma'_1 \cdots \sigma'_l \overline{w_k} \rangle | \sigma_j = \sigma'_l, \sigma_k \in \Sigma\}$
$\text{SFromP}(\phi_1 \wedge \phi_2) = \{f \# g | f \in \text{SFromP}(\phi_1), f \in \text{SFromP}(\phi_2)\}$
$\text{SFromP}(\phi_1 \vee \phi_2) = \text{SFromP}(\phi_1) \cup \text{SFromP}(\phi_2)$
$\text{SFromP}(\neg\phi) = \text{SFromP}(\phi)^c$

Lemma 12 (Correctness of SFromP). *Let ϕ be any predicate.*

- *If $f \in SFromP(\phi)$, then for all $v \in Val_f$, $B(\phi, v) = True$.*
- *If $B(\phi, v) = True$, then there exists $f \in SFromP(f)$ such that $v \in Val_f$*

Proof (Proof sketch). Structurally induct over predicates and use definition of B to prove. The complement case uses correctness of complement.

We now have the machinery to define the nullability of a USR via substitutions. Unlike classical nullability, substitution nullability returns the simple substitutions needed to guarantee the USR contains the empty string regardless of valuation when the substitution is applied. Or, when viewed from the perspective of restricting valuations, the substitution restricts the space of valuations such that the languages of the USR contains the empty string.

Suppose we want to unify the substitution

$$\langle \overline{w_1} \mapsto 123\overline{w_1}, \overline{w_i} \mapsto x\overline{w_1}, x \mapsto y, y \mapsto z \rangle$$

From phase one we obtain the set $S_{\overline{w_1}} = \{123\overline{w_1}, x\overline{w_1}\}$. On the first character $\text{Union}(1, x)$ is run. On second character, $\overline{w_1}$ is encountered on the element $x\overline{w_1}$ so the element is removed. $123\overline{w_1}$ is the only element and there are no other $S_{\overline{w_i}}$ sets so stringsubs $= \langle \overline{w_1} \mapsto 123\overline{w_1} \rangle$.
From phase two we need to run the following unions:

$$\text{Union}(x, y), \text{Union}(y, z)$$

Algorithm 1 Satisfiability algorithm.

```
1: visited: a set of USRs
2: queue: a queue for USRs
3: freshvar: assumes a fresh variable is created whenever used
4: procedure SATISFIABLE(R)
5:     Add R to queue
6:     while current ← queue.front do
7:         Append R to visited
8:         if NULLABLE(current) = ∅ then
9:             deriv ← DERIVATIVE(current, freshvar)
10:            if deriv = ∅ then
11:                return False
12:            else
13:                for all subexpr in deriv do
14:                    if subexpr ∉ visited then
15:                        Add subexpr to queue
16:                return False
17:            Add current to visited
18:        else
19:            return True
```

In our Union-Find structure, we have the equivalence class $\{x, y, z, 1\}$ where 1 is the representative. Thus, CharSubs $= \langle x \mapsto 1, y \mapsto 1, z \mapsto 1 \rangle$.
From phase 3 substituting charsubs into strsubs does nothing so our final result is

$$\langle \overline{w_1} \mapsto 123\overline{w_1}, x \mapsto 1, y \mapsto 1, z \mapsto 1 \rangle$$

A.10 Correctness and Time Complexity of Satisfiability

Lemma 13 (Correctness of Satifiability Algorithm). *For an arbitrary USR R, if there exists $v \in Val$ such that $L(R, v) \neq \emptyset$ then Satisfiable$(R) = T$.*

Proof. $\Rightarrow$ Suppose that there exists a $v \in$ Val such that $L(R, v) \neq \emptyset$. Then there exists some shortest string s of length n where $s \in L(R, v)$. Because this is the shortest string, if $n = 0$, then the USR will be non-empty and the algorithm will immediately terminate. Otherwise, by the correctness of the Antimirov derivative, there exists a $(P, f) \in \Delta(R, x)$ such that $s[1:] \in L(P, F'_f(v))$. Thus, the algorithm will take the derivative and iterate over all of the. Recursively, this process will continue until we have a derivative $R^{(n)}$ which accepts the string $s[n:] = \epsilon$. In this instance, by the definition of nullable, $N(R^{(n)})$ will finally be non-empty, so the algorithm will return true. Thus, if there is a string accepted by the USR, the algorithm will successfully identify so.
$\Leftarrow$ Suppose that the algorithm returns True at some point. This is only possible if the USR R is nullable or if recursively, another derivative returns true when running the algorithm. Let us assume that the nth derivative has a non-empty nullable. Then, there exists a $v \in$ Val such that $\epsilon \in L(P, v)$. By the correctness

of the Antimirov derivative, $F_f(v)(x) \in L(P_1, F_f(v))$, where P_1 is the USR that called the satisfiable algorithm on this derivative. Thus, this algorithm will also return True. Recursively, we can continue back to the original USR R, which will thus be satisfied by $F_{f_1}(v)F_{f_2}(v)\dots F_{f_n}(v) \in L(R, F_f(v))$. This means that there is a $v \in \text{Val}$ such that $L(R, v) \neq \emptyset$. Therefore, the algorithm is correct.

A.11 Correctness and Time Complexity of Unify

Lemma 14 (Correctness of Unify). *For any substitution f, let $g = Unify(f)$.*

- *If $g \neq \bot$, then g is simple and $Val_f = Val_g$.*
- *If $g = \bot$, then $Val_f = \emptyset$.*

Proof. We proceed by cases on g.

- In the $g = \bot$ case, the algorithm must have terminated due to either unioning differing literals (Line 27, Line 33) or there are maps from a single $\overline{w_i}$ to differing length strings (Line 21). In the former case, it would be possible to create two separate chain of equivalences from $\overline{w_i}$ or c_i to differing literals using the mappings in f. One variable cannot be mapped to differing literal so f is impossible to satisfy making $\text{Val}_f = \emptyset$. In the latter case, it is impossible to map $\overline{w_i}$ to two strings of differing length so $\text{Val}_f = \emptyset$.
- In the $g \neq \bot$ case, we first prove g is simple. $g = \text{strsubs} \cup \text{charsubs}$ by line 40. Each pair in charsubs has the form $x, \text{Find}(x)$ such that $\text{Find}(x) \neq x$ by lines 35-36, thus charsubs is a simple substitution. By line 16 all except one mapping for each string variable is kept, thus the first component of every pair in strsubs is unique. Since charsubs was applied onto the second component of each pair in strsubs (Line 40), there does not exist any SubExpr that contain a character variables in the first component of any pair in charsubs. With these observations, g is simple. Next, we prove $\text{Val}_f = \text{Val}_g$ by showing if a valuation v satisfies all constraints in f, then it satisfies all constraints in g and vice versa.
  - Let $v \in \text{Val}_f$,
    - If $(c_i, \sigma) \in g$ or $(c_i, c_j) \in g$, then by the union calls in lines 33 and 27, we can create a chain of equalities from $v(c_i)$ to $v(\sigma)$ or $v(c_j)$ using the constraints in f. Thus, $v(c_i) = \sigma$ or $v(c_j)$ as desired.
    - If $(\overline{w_i}, \epsilon) \in g$, then $(\overline{w_i}, \epsilon) \in f$ due to ϵ being length 0 so $v(\overline{w_i}) = \epsilon$.
    - If $(\overline{w_i}, x\overline{w_i}) \in g$, then x was only used in union calls with other character variables or was not used in any union calls. With no union calls, $(\overline{w_i}, x\overline{w_i}) \in f$ as well. If there were union calls, then pairs of form $(\overline{w_i}, y\overline{w_i}) \in f$ and a chain of equality can be created to x and $v(\overline{w_i}) = v(x)s$ for some $s \in \sigma^*$.
    - If $(\overline{w_i}, x) \in g$, the argument is the same as the $(\overline{w_i}, x\overline{w_i}) \in g$ case.

Algorithm 2 Unify algorithm.

`Sub`: a type for substitutions (array of ordered pairs) (variables $f, g, \ldots$)
`Lit`: a type for character literals (variables $a, b, c, \ldots$)
`StrVar`: a type for string variables (variables $\overline{w_1}, \overline{w_2}, \cdots$)
`CharVar`: a type for character variables (variables $x, y, z, \cdots$)
`SubExpr`: a type for SubExprs (variables $R, R' \cdots$)
`EqExpr`: a type for sets of SubExprs (array) (Variable S)
`StrEqClass`: a map from `StrVar` to `EqExpr`
`StrSubs`: a substitution containing ordered pairs whose first component is a `StrVar`
`CharSubs`: a substitution containing ordered pairs whose first component is a `CharVar`
`UF`: a union-find data structure over `V` where we assume that characters are prioritized over strings for canonicalization. UF.Union can be given a set to union all elements in the set
procedure UNIFY(`Sub`(f))
    for all (`StrVar`($\overline{w_i}$), `SubExpr`(R)) in `Sub`(f) **do**
        Append R to `StrEqClass`($\overline{w_i}$)
    for all `EqExpr`(S) non empty in `StrEqClass`($\overline{w_i}$) **do**
        $i \leftarrow 0$
        while $|S| > 1$ **do**
            UnionSet$\leftarrow \emptyset$
            for all R in S **do**
                if $R[i]$ is undefined **then**
                    for all $R' \neq R$ in S **do**
                        if $R'[i]$ is not undefined and $R'[i] \neq$ `StrVar`($\overline{w}$) **then return** $\perp$
                    break
                else if $R[i]$ is a `StrVar` and $|S| > 1$ **then**
                    Remove R from S
                else
                    Append $R[i]$ to UnionSet
                $i \leftarrow i + 1$
            if `UF`.Union(UnionSet)=$\perp$ **then return** $\perp$
    for all $(\overline{w}, S)$ in `StrEqClass` **do**
        $R \leftarrow R' \in S$ ▷ S is a singleton at this point
        Append $(\overline{w}, R)$ to `StrSubs`
    for all (`CharVar`(x), `SubExpr`(R)) in `Sub`(f) **do**
        Append x to CharSet
        if `UF.Union`$(x, R) = \perp$ **then return** $\perp$
    for all x in CharSet **do**
        if $x \neq$ `UF.Find`(x) **then**
            Append (x, `UF.Find`(x)) to `CharSubs`
    for all $(\overline{w}, R)$ in `StrSubs` **do**
        for all (x, R') in `CharSubs` **do**
            if x in R **then**
                Replace x with R' in R.
    return `StrSubs` $\cup$ `CharSubs`

  - If $(\overline{w_i}, \sigma_1 ... \sigma_n \overline{w_i}) \in g$ or $(\overline{w_i}, \sigma_1 ... \sigma_n) \in g$ then they are also in f.
  - If $(\overline{w_i}, \sigma \overline{w_i}) \in g$ or $(\overline{w_i}, \sigma) \in g$, then it is either identical to the above case or σ is equal to a character variable in the union find structure and $(\overline{w_i}, x\overline{w_i}) \in f$ or $(\overline{w_i}, x) \in f$ causing $\nu(x) = \sigma$.

  Since every type of constraint in g are satisfied $\nu \in \text{Val}_g$ and $\text{Val}_f \subseteq \text{Val}_g$.

- Let $\nu \in \text{Val}_g$,
  - If $(c_i, \sigma) \in f$, then $\text{Find}(c_i) = \sigma$ so $(c_i, \sigma) \in g$ and $\nu(c_i) = \sigma$.
  - If $(c_i, c_j) \in f$, then $\text{Find}(c_i) = \text{Find}(c_j)$ and the pairs $(c_i, \text{Find}(c_i)), (c_j, \text{Find}(c_j)) \in g$ so $\nu(c_i) = \nu(c_j)$.
  - If $(\overline{w_i}, \epsilon) \in f$, then $(\overline{w_i}, \epsilon) \in g$ due to ϵ being length 0 so $\nu(\overline{w_i}) = \epsilon$.
  - If $(\overline{w_i}, x\overline{w_i}) \in f$ or $(\overline{w_i}, \sigma_1 ... \sigma_n \overline{w_i}) \in f$, then either $(\overline{w_i}, y\overline{w_i}) \in g$ or $(\overline{w_i}, \sigma_1 ... \sigma_m \overline{w_i}) \in g$ where $m > n$. Thus, the constraints in f will be satisfied.
  - If $(\overline{w_i}, x) \in f$ or $(\overline{w_i}, \sigma_1 ... \sigma_n) \in f$, then in the former case the union find structure ensures x changes to an equivalent length 1 string or $(\overline{w_i}, \sigma_1 ... \sigma_n) \in g$ in the latter case.

  Since every type of constraint in f are satisfied $\nu \in \text{Val}_f$ and $\text{Val}_g \subseteq \text{Val}_f$.

Complexity of Unify. We compute complexity based on the number of Union and Find calls required given an input substitution f. Let m be the number of ordered pairs in f and n be the length of the longest SubExpr in the second component of the ordered pairs. In lines 14 to 30, the for loop will go through every element in every S which adds up to m at max if all pairs in f were string variable mappings. The while loop nested within loops based on the longest SubExpr so we will use n. Thus, these lines yield $O(mn)$ Union-Find calls.

In lines 31 to 36, the for loop loops at max up to m if all pairs in f were character variable mappings. Thus, these lines yield $O(m)$ Union-Find calls.

In lines 37 to 40, the outer for loop loops at max m times and the inner for loop loops at max n times. There are $O(mn)$ operations but, these lines do not contribute Union-Find calls.

From the complexity analysis above, Unify has a time complexity of $O(mn)$ Union-Find calls which translates to $O(mn\alpha(mn))$ where α is the inverse Ackermann function. To simplify even further, we can say the length of the input f is N, representing the length of all SubExprs in f concatenated together. In the case where every SubExpr is the same length, $N = mn$ and the Unify algorithm has complexity $O(N\alpha(N))$ which is almost linear.

Algorithm 3 Substitution Difference.

```
1: SubExpr: the SubExprs type (variables R, R')
2: Sub: Type for substitutions (variables f, g, ...)
3: RetSub: Holds the final substitution to be returned
4: procedure SUBSTITUTION DIFFERENCE((Sub,Sub)(f, g))
5:     h ← f#g
6:     RetSub ← g
7:     if h = ⊥ then return ⊥
8:     RetSub ← g
9:     for all (c̄_i, _) ∈ h do
10:        if (c̄_i, R) ∈ RetSub then
11:            Remove (c̄_i, R) from RetSub
12:    for all (w̄_i, R) ∈ h do
13:        if (w̄_i, R') ∈ RetSub then
14:            RetSub∪MATCH(R, R')
       return RetSub
15: procedure MATCH((SubExpr, SubExpr)(R, R'))
16:    if R = ϵ and R' = ϵ then return ∅
17:    else if R = w̄_i then return ⟨w̄_i ↦ R'⟩
18:    else if R' = w̄_i then return ⟨w̄_i ↦ R⟩
19:    if R[0] is a character variable then return ⟨R[0] ↦ R'[0]⟩∪MATCH(R[1 :], R'[1 :])
20:    else if R'[0] is a character variable then return ⟨R'[0] ↦ R[0]⟩∪MATCH(R[1 :], R'[1 :])
21:    else return MATCH(R[1 :], R'[1 :])
```

A.12 Correctness and Time Complexity of Substitution Difference

Lemma 15 (Correctness of Substitution Difference). *Given arbitrary simple substitutions f, g, $f - g = \bot$ if and only if $f\#g = \bot$. Furthermore, when $f - g \neq \bot$, the following statements are true.*

- $R[g][f - g] = R[f\#g]$ *for all regex* R
- $F_g \circ F_{f-g} = F_{f\#g}$

Proof. Only the initial merge on line 5 could cause substitution difference to return bottom, so it is true that $f - g = \bot$ if and only if $f\#g = \bot$. For statement 1, since application of substitutions on USRs is equivalent to replacing every string and character variable in the domain of the substitution, we will only show the statement is true on $R = \overline{w_i}$ and $R = \overline{c_i}$ and the recursive cases will follow by inductive hypothesis.

If $R = \overline{c_i}$, we go through every potential character variable mapping in g. If g does not have any character variable mapping, then any character mapping in f remains in $f - g$ which is also in $f\#g$. If $(\overline{c_i}, \sigma) \in g$, then it must also be in $f\#g$ by definition of merge and is not in $f - g$ by line 11. Thus, only $(\overline{c_i}, \sigma)$ gets applied which is in $f\#g$. If $(\overline{c_i}, \overline{c_j}) \in g$, then it may be identical to the

prior case when f does not contain some $(\overline{c_i} \mapsto \sigma)$ or $(\overline{c_j} \mapsto \sigma)$. In the case f does contain one such mapping, $f\#g$ will contain both a $(\overline{c_i} \mapsto \sigma)$ and $(\overline{c_j} \mapsto \sigma)$ mapping. While $(\overline{c_i} \mapsto \sigma) \notin f - g$, the final composed mapping does contain it since applying g's $(\overline{c_i}, \overline{c_j})$ followed by $(\overline{c_j} \mapsto \sigma)$ still in $f - g$ results in $(\overline{c_i} \mapsto \sigma)$.

If $R = \overline{w_i}$, we consider Match to help prove the statement. If g does not contain any string variable mappings, then no match calls are made and any string variable mapping in $f - g$ will be in $f\#g$. This yields $R[g][f - g] = R[f - g] = R[f\#g]$. If g contains some $(\overline{w_i} \mapsto R')$, we need to consider when R' ends with ϵ and when it ends on $\overline{w_i}$. In the former case, $(\overline{w_i} \mapsto R')$ will also be in $f\#g$ so it is not in $f - g$ by the Match recursing down to ϵ's as its argument. This yields $R[g][f - g] = R[g] = R[f\#g]$. In the latter case, it may be identical to the former case. However, when there is a mapping $(\overline{w_i}, R'') \in f\#g$ where R'' is longer than R', Match will cut off the prefix of R'' that is determined by g. Thus, $R[g][f - g]$ will first attach g resulting in R' which ends on $\overline{w_i}$, then attach the portion that excludes g which gets concatenated on. The composed substitution is $f\#g$.

We prove statement 2 by unwrapping the definition of the F function and statement 1. F applies substitutions by appending constraints on the left side of strings in the image of the the input valuation. Since applying substitutions on USRs appends from the front, by reversing the order for F, we get that $F_g \circ F_{f-g} = F_{f\#g}$. Let $(f - g)(\overline{w_i}) = C_n\overline{w_i}$ and $g(\overline{w_i}) = C'_n\overline{w_i}$.

Lemma 16 (Time Complexity of Substitution Difference). *Substitution difference can be computed in $O(mn\alpha(mn))$ time, where m is the total number of pairs in the input substitutions f and g, and n is the length of the longest SubExpr in the pairs.*

Proof. We show all other operation in substitution difference take shorter time than the initial merge on line 5. The first for loop on line 9 traverses the pairs in h and the inclusion check in the loop can be done in $O(1)$. This yields at most $O(m)$ for m pairs in h. The second for loop on line 12 involves the Match function which recurses at most n times down the length of R or R'. The loop loops at most m times. When combined with Match, we have $O(mn)$. Neither are longer than the initial merge that takes $O(mn\alpha(mn))$ so substitution difference has $O(mn\alpha(mn))$ complexity.

A.13 Proof of Theorem 8 (Sec5)

Proof. Assuming $L \downarrow (R) \neq \varnothing$, there exists ν such that $L(R, \nu)$ is non-empty. If $\epsilon \in L(R, \nu)$, then nullability is nonempty by correctness of substitution nullable and the algorithm will output SAT. Let $s \in L(R, \nu)$. By correctness of derivative, $s[1 :] \in L(P, F'_f(\nu'))$ for some $(P, f) \in \Delta(R, x)$ where x is a fresh character variable. $\nu'(x) = s[0]$ and agrees with ν on all other variables. Since the algorithm searches through all pairs on the same derivative and no single application of derivative produces an infinite set, such (P, f) will be visited. The process repeats for P until a nullable term which happens in $|s|$ derivatives returning SAT.

Proving the contrapositive of the second statement, assume the algorithm returns SAT on input R. If the input was nullable, then $\epsilon \in L \downarrow (R)$ and we are done. Suppose a nullable derivative has been found, say $(P, f) \in \Delta(P', x)$, so $\epsilon \in L(P, v)$ for some v. By correctness of derivative, $F_f(v)(x)\epsilon \in L(P', F_f(v))$. If $P' = R$, then $F_f(v)(x) \in L \downarrow (R)$ and we are done. If $P' \neq R$, then P' comes from a previous derivative $(P', f') \in \Delta(P'', x')$. Since we have $F_f(v)(x)\epsilon \in L(P', F_f(v))$, correctness of derivative is be applied again. This process repeats until R is recovered and a string $s \in L \downarrow (R)$ is constructed.

A.14 Proof of Lemma 1 (Sec 5)

Proof. We prove both statements by structural induction on R. Assume $N(R)$ is non-empty or there exists v where $\epsilon \in L(R, v)$.

- If $R = \epsilon$:
  - Statement 1. $\emptyset \in N(R)$. For all v,
    $$L(\epsilon[\emptyset], v) = \{\epsilon\}$$
  - Statement 2. For all v, $L(R, v) = \{\epsilon\}$.
    $$N(R) = \{\emptyset\}$$
    $$\epsilon \in \text{Val}_\emptyset = \text{Val}$$
- If $R = \overline{w_i}$ for some i:
  - Statement 1. $f = \{(\overline{w_i} \mapsto \epsilon)\} \in N(R)$. For all v,
    $$L(\overline{w_i}[f] = \epsilon, v) = \{\epsilon\}$$
  - Statement 2. $\epsilon \in L(R, v)$ only when $v(\overline{w_i}) = \epsilon$.
    $$N(R) = \{f = \{\overline{w_i} \mapsto \epsilon\}\}$$
    $$v \in \text{Val}_f$$
    since $v(\overline{w_i}) = \epsilon$.
- If $R = P \cup Q$:
  - Statement 1. $f \in N(P)$ or $f \in N(Q)$. WLOG assume $f \in N(P)$. By inductive hypothesis, for all v
    $$\epsilon \in L(P[f], v) \subseteq L(P[f] \cup Q[f], v) = L((P \cup Q)[f], v)$$
  - Statement 2. $\epsilon \in L(R, v) = L(P, v) \cup L(Q, v)$. WLOG, assume $\epsilon \in L(P, v)$. By inductive hypothesis, there exists $f \in N(P)$ such that $v \in \text{Val}_f$. Since $f \in N(P)$, $f \in N(P \cup Q) = N(R)$ by definition of nullable.

– If $R = PQ$:

  • Statement 1. $f = f_1 \# f_2$ where $f_1 \in N(P)$ and $f_2 \in N(Q)$. Since $f_1, f_2 \subseteq f$ and by the inductive hypothesis

$$\epsilon \in L(P[f_1], v)$$
$$\epsilon \in L(Q[f_2], v)$$

  for all v. By Lemma 4

$$\epsilon \in L(P[f], v)$$
$$\epsilon \in L(Q[f], v)$$

  for all v. Thus,

$$\epsilon \in L(P[f], v)L(Q[f], v) = L(PQ[f], v)$$

  for all v.

  • Statement 2. $\epsilon \in L(R, v) = L(P, v)L(Q, v)$. By definition of concatenation of sets, $\epsilon \in L(P, v) \cap L(Q, v)$. By inductive hypothesis, There exists $f_1 \in N(P)$ and $f_2 \in N(Q)$ such that $v \in \mathrm{Val}_{f_1}$ and $v \in \mathrm{Val}_{f_2}$. Thus, $v \in \mathrm{Val}_{f_1} \cap \mathrm{Val}_{f_2} = \mathrm{Val}_{f_1 \# f_2}$. By definition of nullable, $f_1 \# f_2 \in N(PQ)$ since $\mathrm{Val}_{f_1} \cap \mathrm{Val}_{f_2}$ is non-empty and the merge is not bottom.

– If $R = P \cap Q$:

  • Statement 1. $f = f_1 \# f_2$ where $f_1 \in N(P)$ and $f_2 \in N(Q)$. Following the same argument as the previous case, by applying the inductive hypothesis and Lemma 4, we obtain

$$\epsilon \in L(P[f], v)$$
$$\epsilon \in L(Q[f], v)$$

  for all v so $\epsilon \in L(P[f], v)L(Q[f], v) = L(PQ[f], v)$ for all v as well.

  • Statement 2. $\epsilon \in L(R, v) = L(P, v) \cap L(Q, v)$. By definition of intersection, $\epsilon \in L(P, v) \cap L(Q, v)$. The rest is identical to the concatenation case.

– If $R = \mathbf{IF}\,(\phi, P, Q)$: apply Lemma 3 since $N(R)$ is a set of merged substitutions.

– If $R = P^c$:

  • Statement 1. Assume $N(P^c)$ is nonempty and $f \in N(P^c) = N(P)^c$. We prove by contradiction. Assume there exists some valuation v such that $\epsilon \notin L(P^c[f], v)$. By definition of language and substitution semantics, this implies $\epsilon \in L(P, v')$ for some $v' \in \mathrm{Val}_f$. From inductive hypothesis we apply statement 2. Let $N(P) = \{h_1, ..., h_n\}$. Then there exists $h_j \in N(P)$ such that $v' \in \mathrm{Val}_{h_j}$. However, this is a contradiction to the correctness of complement since we found $v' \in \mathrm{Val}_f \cap \mathrm{Val}_{h_j}$ when we should have $\mathrm{Val}_f \cap \mathrm{Val}_{h_j} = \emptyset$.

- Statement 2. We prove by contrapositive. Given arbitrary $v \in \text{Val}$, assume $v \notin \text{Val}_f$ for all $f \in N(P^c) = N(P)^c$. By correctness of complement, there exists $f' \in N(P)$ such that $v \in \text{Val}'_f$. Using the inductive hypothesis to apply statement 1, $\epsilon \in L(P[f'], v')$ for any $v' \in \text{Val}$. By correctness of substitutions, there exists v' such that $L(P[f'], v') = L(P, v)$. Thus, $\epsilon \in L(P, v)$ which means $\epsilon \notin L(P^c, v)$ as desired.

A.15 Proof of Theorem 6 (Sec 5)

Proof. The bases cases can be proven by definition of language of USRs. The recursive cases can be proven by applying the various correctness lemmas associated with each operation.
We proceed by structural induction on R.

- If $R = \sigma$,
  - Statement 1: $\Delta(R, x) = \{(\epsilon, \langle x \mapsto \sigma \rangle)\}$ so $\epsilon \in L(\epsilon, v)$ for all v. $F_f(v)(x) = \sigma$ and $\sigma\epsilon \in L(\sigma, F_f(v))$.
  - Statement 2: $\sigma \in L(\sigma, v)$ for all $v' \in \text{Val}$. We know $\Delta(R, x) = \{(\epsilon, \langle x \mapsto \sigma \rangle)\}$ and $\epsilon \in L(\epsilon, v)$ for all v. Enforcing $v(x) = \sigma$ does make it an element of $\text{Val}_{\langle x \mapsto \sigma \rangle}$.
- If $R = c_i$,
  - Statement 1: $\Delta(R, x) = \{(\epsilon, \langle x \mapsto c_i \rangle)\}$ so $\epsilon \in L(\epsilon, v)$ for all v. $F_f(v)(x) = v(c_i)$ and $F_f(v)(x)\epsilon \in L(\sigma, F_f(v))$.
  - Statement 2: $v(c_i) \in L(c_i, v)$ for all $v \in \text{Val}$. We know $\Delta(R, x) = \{(\epsilon, \langle x \mapsto c_i \rangle)\}$ and $\epsilon \in L(\epsilon, v)$ for all v. Enforcing $v(x) = v(\overline{c_i})$ does make it an element of $\text{Val}_{\langle x \mapsto \overline{c_i} \rangle}$.
- If $R = \overline{w_i}$,
  - Statement 1: $\Delta(R, x) = \{\overline{w_i}, \langle \overline{w_i} \mapsto x\overline{w_i} \rangle\}$ so $L(\overline{w_i}, v) = \{\overline{w_i}\}$. Using the transformation $F_f(v)(\overline{w_i}) = v(x)v\overline{w_i}$ and $F_f(v)(x) = v(x)$. Thus, it is true that $v(x)v\overline{w_i} = F_f(v)(x)v\overline{w_i} \in L(\overline{w_i}, F_f(v)) = \{v(x)v(\overline{w_i})\}$.
  - Statement 2: $L(R, v) = \{v(\overline{w_i})\}$ for all $v \in \text{Val}$. We know $\Delta(R, x) = \{\overline{w_i}, \langle \overline{w_i} \mapsto x\overline{w_i} \rangle\}$. For $v(\overline{w_i}) \neq \epsilon$, construct $v(\overline{w_i}) = v'(\overline{w_i})[1:]$. Thus, $v'(\overline{w_i})[1:] \in L(\overline{w_i}, v)$.
- If $R = R_1 \cup R_2$,
  - Statement 1: $\Delta(R, x) = \Delta(R_1, x) \cup \Delta(R_2, x)$. WLOG, assume $(P, f) \in \Delta(R_1, x)$ and $s \in L(R_1, v)$ for some $v \in \text{Val}$. By inductive hypothesis, $F_f(v)(x)s \in L(R_1, F_f(v)) \subseteq L(R_1 \cup R_2, F_f(v)) = L(R, F_f(v))$.
  - Statement 2: Assume $s \in L(R, v') = L(R_1, v') \cup L(R_2, v')$. WLOG, assume $s \in L(R_1, v')$. By inductive hypothesis, there exists $(P, f) \in \Delta(R_1, x) \subseteq \Delta(R_1 \cup R_2, x) = \Delta(R, x)$ such that $s[1:] \in L(P, v)$ for some $v \in \text{Val}$.

– If $R = R_1 R_2$

  • Statement 1: If $(P, f) \in \Delta(R, x)$, then $(P, f) = (R_1' R_2[f'], f')$ where $(R_1', f') \in \Delta(R_1, x)$ or $(P, f) = (R_2'[g - f], f' \# g)$ where $(R_2', f') \in \Delta(R_2, x)$ and $g \in N(R_1)$.

    ∗ In the former case, assume

$$s \in L(P, v) = L(R_1', v) L(R_2[f'], v)$$

    Split $s = s_1 s_2$ so that $s_1 \in L(R_1', v)$ and $s_2 \in L(R_2[f'], v)$. By inductive hypothesis, $F_{f'}(v)(x) s_1 \in L(R_1, F_{f'}(v))$. By correctness of substitution, $L(R_2[f'], v) = L(R_2, F_{f'}(v))$. Thus, $F_{f'}(v)(x) s_1 s_2 = F_{f'}(v)(x) s \in L(R_1, F_{f'}(v)) L(R_2, F_{f'}(v)) = L(R, F_{f'}(v))$

    ∗ In the latter case, assume

$$s \in L(P, v) = L(R_2'[g - f'], v) = L(R_2', F_{g - f'}(v))$$

    Since $g \in N(R_1)$, for all $v \in \mathrm{Val}$, $\epsilon \in L(R_1[g], v)$. By merge property, $\epsilon \in L(R_1[f' \# g], v) = L(R_1, F_{f' \# g}(v))$ By inductive hypothesis, $F_{f'}(F_{g - f'}(v))(x) s \in L(R_2, F_{f'}(F_{g - f'}(v))) = L(R_2, F_{f' \# g}(v))$. Thus, $F_{f' \# g}(v)(x) s = \epsilon F_{f'}(F_{g - f'}(v))(x) s \in L(R_1, F_{f' \# g}(v)) L(R_2, F_{f' \# g}(v)) = L(R, F_{f' \# g}(v))$

– If $R = R_1^*$, then the proof is similar to the concatenation case using the additional fact that $P[g]^* = P^*[g]$ by definition of substitution.