Repairing Regular Expressions for Extraction

NARIYOSHI CHIDA, NTT Social Informatics Laboratories / Waseda University, Japan
TACHIO TERAUCHI, Waseda University, Japan

While synthesizing and repairing regular expressions (regexes) based on Programming-by-Examples (PBE) methods have seen rapid progress in recent years, all existing works only support synthesizing or repairing regexes for membership testing, and the support for extraction is still an open problem. This paper fills the void by proposing the *first* PBE-based method for synthesizing and repairing regexes for extraction.

Our work supports regexes that have *real-world extensions* such as backreferences and lookarounds. The extensions significantly affect the PBE-based synthesis and repair problem. In fact, we show that there are unsolvable instances of the problem if the synthesized regexes are not allowed to use the extensions, i.e., there is no regex without the extensions that correctly classify the given set of examples, whereas every problem instance is solvable if the extensions are allowed. This is in stark contrast to the case for the membership where every instance is guaranteed to have a solution expressible by a pure regex without the extensions.

The main contribution of the paper is an algorithm to solve the PBE-based synthesis and repair problem for extraction. Our algorithm builds on existing methods for synthesizing and repairing regexes for membership testing, i.e., the enumerative search algorithms with SMT constraint solving. However, significant extensions are needed because the SMT constraints in the previous works are based on a non-deterministic semantics of regexes. Non-deterministic semantics is sound for membership but not for extraction, because which substrings are extracted depends on the deterministic behavior of actual regex engines. To address the issue, we propose a new SMT constraint generation method that respects the deterministic behavior of regex engines. For this, we first define a novel formal semantics of an actual regex engine as a deterministic big-step operational semantics, and use it as a basis to design the new SMT constraint generation method. The key idea to simulate the determinism in the formal semantics and the constraints is to consider *continuations* of regex matching and use them for disambiguation. We also propose two new search space pruning techniques called *approximation-by-pure-regex* and *approximation-by-backreferences* that make use of the extraction information in the examples. We have implemented the synthesis and repair algorithm in a tool called R3 (Repairing Regex for extRaction) and evaluated it on 50 regexes that contain real-world extensions. Our evaluation shows the effectiveness of the algorithm and that our new pruning techniques substantially prune the search space.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; • **Theory of computation** → **Regular languages**; **Operational semantics**.

Additional Key Words and Phrases: Regular Expression, Programming by Example, Program Repair

ACM Reference Format:
Nariyoshi Chida and Tachio Terauchi. 2023. Repairing Regular Expressions for Extraction. *Proc. ACM Program. Lang.* 7, PLDI, Article 173 (June 2023), 31 pages. https://doi.org/10.1145/3591287

1 INTRODUCTION

Regular expressions (regexes) play a critical role for *membership testing* and *extraction* in modern programming languages and software development. For example, they are heavily used for validating user inputs [O'Hara 2022; OWASP 2022], inspecting packets [Services 2022; Snort 2022],

Authors' addresses: Nariyoshi Chida, nariyoshichidamm@gmail.com, NTT Social Informatics Laboratories / Waseda University, Japan; Tachio Terauchi, Waseda University, Japan, terauchi@waseda.jp.

2475-1421/2023/6-ART173
https://doi.org/10.1145/3591287

and extracting data from (unstructured) texts [Li et al. 2008; Luckie et al. 2019] in the frameworks for building Web applications [Angular 2022; Django 2022], etc. However, despite the practical importance of regexes, writing or repairing regexes is an error-prone task [Michael et al. 2019; Wang et al. 2020].

Recent works have addressed this problem by synthesizing or repairing regexes based on *Programming-by-Examples* (PBE) methods which have received much attention in recent years [Chen et al. 2020; Chida and Terauchi 2022b; Ferreira et al. 2021; Lee et al. 2016; Li et al. 2021, 2020; Pan et al. 2019; Zhang et al. 2020]. The PBE-based methods allow the users to write or repair regexes automatically from examples that reflect the user's intention. Yet, while methods for synthesizing or repairing regexes for membership testing exist, *none of them can handle extraction.* Additionally, there are several works for generating a regex for extraction from examples [Bartoli et al. 2014, 2016] based on genetic algorithms. However, they do not guarantee the correctness of the repair, that is, the synthesized regex is not guaranteed to conform to the given set of examples.

This paper fills the void by proposing the *first* PBE-based method for synthesizing and repairing regexes for extraction. We first formally define a PBE-based repair problem called the *extraction-regex-repair* problem.[1] The problem takes a (possibly incorrect) regex, a set of *positive examples*, and a set of *negative examples*. Like in previous PBE works for membership, a negative example is a string to be rejected. However, unlike the previous works, a positive example consists of not only a string to be accepted, but also the information of substrings to be extracted from the string. The goal of the repair problem is to find a regex that is consistent with the given set of examples and is syntactically close to the given original one. Like in the previous PBE works, the syntactic closeness is used to bias the synthesis to one that is close to the user's intention [Chida and Terauchi 2022b; Pan et al. 2019], that is, the original regex may not be correct but is assumed to be *close* to the one that the user intended. Our work handles regexes with *real-world* extensions such as backreferences and lookarounds [Friedl 2006]. The extensions significantly affect the repair problem. In fact, we show that there are unsolvable problem instances if repaired regexes are not allowed to use the extensions, i.e., there is no regex without the extensions that correctly classify the given set of examples, whereas every problem instance is solvable if the extensions are allowed. This is in stark contrast to the case for the membership repair problem in which every instance is guaranteed to have a solution expressible by a pure regex without the extensions.

At a high level, our algorithm for solving the extraction-regex-repair problem builds on the approach used in the previous works on PBE-based repair methods for membership which conducts enumerative search by repeatedly generating and solving SMT constraints that encode assertions of the form "the given *template* regex can be instantiated to one that conforms with the given set of examples", coupled with certain *pruning* techniques to reduce the search space. A template regex is a regex that contains holes to be filled. However, there are following key challenges that did not exist in the previous works that only considered membership.

The first challenge comes from the fact that regex engines in the real world run in a deterministic manner by prioritizing the behavior of operators that potentially have an ambiguity such as the union operator and the Kleene-star operator. The previous works that only considered membership *do not* face this challenge because the determinism of regex engines does not affect the matching results in terms of membership testing.[2] However, in the case of extraction, it *does* affect the result because which substrings are extracted depends on the deterministic behavior of the regex engine. To address the issue, we propose a new SMT constraint generation method that respects

[1] Synthesis is a special case of the repair problem in which the pre-repair regex is ϵ.

[2] Technically, the membership result can be affected in some rare cases when substrings captured in positive lookarounds are backreferenced [Chida and Terauchi 2022b; Loring et al. 2019], but the issue is not addressed in the previous PBE works.

the deterministic behavior of regex engines. For this, we first define a novel formal semantics of an actual regex engine as a deterministic big-step operational semantics, and use it as a basis to design the new SMT constraint generation method. The key idea to simulate the determinism in the formal semantics and the constraints is to consider *continuations* of regex matching and use them for disambiguation. There are several variants of determinism in actual regex engines that often give different extraction results (even though many are equivalent for just membership), and in this work, we choose that of the ECMAScript 2023 language specification, that is, our semantics and constraints emulate the behavior of the regex engines of JavaScript. However, the high-level idea of our approach is applicable to other regex engines.

The second challenge is the large search space. The previous PBE-based methods for membership reduced the search space by pruning useless templates based on over- and under-approximations [Chen et al. 2020; Chida and Terauchi 2022b; Lee et al. 2016; Pan et al. 2019]. That is, by creating from the template regexes that over- or under-approximate the strings accepted by possible instantiations of the template, one can cheaply detect when a template is impossible to be instantiated to a regex that correctly classifies the examples. Although we can use the same approximations in our method for extraction, it is *not* enough to reduce the search space as we shall show in our evaluation. The increase in the search space is inevitable since, in the case of extraction, not only do we need to find a regex that correctly classifies the examples in terms of membership only (i.e., only whether the example string is accepted or rejected), but we also need to find places in the regex to insert capturing groups to correctly reflect the extraction information in the examples. To address this challenge, we propose two new over-approximation techniques called *approximation-by-pure-regex* and *approximation-by-backreferences*. The key idea in these techniques is to modify the example string by embedding the extraction information in it so that its membership result against a certain regex created from the template can be used to detect whether or not the template can be instantiated to become a regex that correctly reflects the extraction information in the example.

We have implemented our algorithm as a tool called R3 (Repairing Regex for extRaction) and evaluated it on 50 regexes including ones that contain real-world extensions. To evaluate the impact of our new pruning, we compare R3 to the baseline: R3$_{base}$, which does not use our new pruning techniques. Our evaluation shows that R3 can repair regexes efficiently in a real-world situation using few examples, and our new pruning technique substantially improves the performance.

In summary, our paper makes the following contributions.

- We give a novel deterministic formal semantics of regexes that is consistent with an actual regex engine for both membership and extraction (Section 3). Our semantics follows the ECMAScript 2023 language specification and supports real-world extensions such as backreferences and lookarounds.
- We define the *extraction-regex-repair problem*, which is the first PBE problem of repairing a regex for extraction task (Section 4). We show that real-world extensions such as lookarounds are *essential* to ensure the existence of the solutions of the problem, in that there exist instances of the problem for which there exist no solutions without the use of extensions while every instance is solvable when the extensions are allowed.
- We give an algorithm for solving the extraction-regex-repair problem (Section 5). Our algorithm is based on an enumerative search and uses an SMT solver to find a solution. We build on our novel formal semantics to design the new SMT constraint generation method that respects the deterministic behavior of the regex engine (Section 5.2). Additionally, we propose new pruning techniques to reduce the search space (Section 5.3). We emphasize that the SMT constraint generation method and the pruning techniques are *significantly* different from those of prior works that only considered membership due to extraction-specific challenges

such as deterministic semantics of regex engines and utilizing the extraction information in examples. Such challenges are not addressed in any state-of-the-art regex synthesizers and repairers for membership. Our work addresses the new challenges by novel ideas such as the use of continuation regexes in SMT constraints and using backreferences in pruning.
- We evaluate our method on 50 regexes including ones that contain real-world extensions (Section 6). We show that R3 can find a solution efficiently, and our new pruning techniques improve the performance substantially.

2 OVERVIEW

We overview the extraction-regex-repair problem and our PBE method that solves the problem with an example. The example is inspired by regexes from Stack Overflow[3] and Blog posts[4].

Motivating Example. The user wants to extract a text of a leaf element from an XML document using a regex. If an XML document has two or more leaf elements, the user wants to extract the leftmost one. For example, the user wants to extract the text b from the XML document $<a>b</a>$, c from $<a><b>c</b></a>$, and d from $<a><b>d</b><c>e</c></a>$. For this purpose, the user prepared the regex $.^{*?}(?<=<(.^{*})_1>)(.^{*?})_2(?=<[/]\backslash 1>).^{*}$. The regex uses the positive lookbehind $(?<=<(.^{*})_1>)$ and the positive lookahead $(?=<[/]\backslash 1>)$ to find the opening and closing tags, respectively, and extracts the text of the tags by the 2nd capturing group $(.^{*?})_2$. Note that the positive lookbehind contains the capturing group $(.^{*})_1$ which is backreferenced in the positive lookahead by the operator $\backslash 1$ to ensure that the opening and closing tag names match. Unfortunately, the regex does not extract texts correctly due to the *deterministic behavior* of actual regex engines. For example, for the input $<a><b>c</b></a>$, the regex extracts $<b>c</b>$ rather than c.

Repair Problem. Our tool R3 can help the user to repair the regex automatically by solving the extraction-regex-repair problem. The instance of the repair problem is a (possibly incorrect) regex, a set of *positive* examples, and a set of *negative* examples. A solution of the instance is a *correct* regex, i.e., a regex that is consistent with all examples, and is syntactically close to the given regex. A positive example is a string to be accepted with the information of substrings to be extracted. We use $\llparenthesis$ and $\rrparenthesis_i$ to denote the information of a substring to be extracted by the ith capturing group. For example, the positive example $a\llparenthesis a\rrparenthesis_1 a$ means that the regex should accept the string aaa and extract the second character a. Note that a positive example specifies not only substrings to be extracted but also the positions of the substrings, reflecting the behavior of the extraction operations in actual regex engines. A negative example is a string to be rejected. Here, suppose that the user prepared the positive examples $<a><\llparenthesis a\rrparenthesis_1>\llparenthesis a\rrparenthesis_2</a></a>$ and $<\llparenthesis b\rrparenthesis_1>\llparenthesis c\rrparenthesis_2</b>$, and the negative example $<a>a</b>$. Note that, in a PBE scenario, the number of examples should be small for usability. A possible way for a user to prepare such examples is to first prepare membership examples (i.e., by only considering which strings should be accepted or rejected), following the methods recommended in prior PBE works for membership such as consulting the RegExLib website [RegExLib 2022], and then add to the positive examples information about the positions of the substrings to be extracted.

Repair Method. Once the user inputs the examples with the (possibly incorrect) regex, R3 generates a candidate template of a solution (*Template Generation*), and then checks whether the candidate template can be instantiated into the solution (*Searching Template Instances*). R3 iteratively performs these steps until it finds a solution.

(*Template Generation*) R3 generates *templates*, which are regexes containing *holes*. Roughly, a hole $\square$ is a placeholder that is to be replaced with some concrete regex. For the running example, R3

[3] https://stackoverflow.com/questions/68248512/regex-lookahead-lookbehind

[4] http://blog.gtiwari333.com/2011/12/htmlxml-tag-parsing-using-regex-in-java.html

starts with the initial template set to be the input regex $.^{*?}(?<=<(.^{*})_1>)(.^{*?})_2(?=<[/]\backslash 1>).^{*}$. Since the regex does not correctly classify the examples (namely, $<a><\llparenthesis a \rrparenthesis_1>\llparenthesis a \rrparenthesis_2</a></a>$),[5] R3 replaces some subexpressions with holes and expands the holes by replacing them with templates that may contain holes. After some iterations, R3 finds the template $.^{*?}(?<=<(\square^{*})_1>)(\square^{*?})_2(?=<[/]\backslash 1>).^{*}$.

(*Searching Template Instances*) Then, R3 checks whether the template can be instantiated to a regex that correctly classifies the examples by replacing its holes with sets of characters. For this, R3 generates and solves an SMT constraint. The constraint generation builds on that proposed for membership [Chida and Terauchi 2022b; Pan et al. 2019]. That is, for each example, we generate a formula over variables representing the holes in the template encoding a constraint of the form "there is an assignment to the holes that makes the template into a regex that correctly classifies the example", and take the conjunction of the formulas over the examples. We make two extensions to this to generate constraints for our extraction problem: bookkeeping of extracted substring indexes for positive examples, and simulating the deterministic behavior of the regex engine. While the former is a relatively straightforward extension, the latter is more subtle and we next give an overview of how it is done. The key idea is to use *continuations* to encode in the SMT constraint the deterministic behavior of the regex engine.[6] For example, for the template $.^{*?}(?<=<(\square^{*})_1>)(\square^{*?})_2(?=<[/]\backslash 1>).^{*}$, the number of iterations of the lazy Kleene star $\square^{*?}$ depends on the result of the matching of the continuation $(?=<[/]\backslash 1>).^{*}$. That is, if the remaining substring of the example under consideration matches the continuation then we must not iterate the Kleene star at all, and we encode such facts as SMT constraints, i.e., we encode conditions of the form "if the continuation is instantiated to match the remaining substring then ... else ...".

If the SMT constraint is satisfiable, then R3 returns the obtained regex as the repair result. Otherwise, R3 continues the search process by generating more templates as described above. R3 also performs *template pruning* to filter out useless templates. Template pruning is also used in template-based synthesis and repair methods for membership [Chen et al. 2020; Chida and Terauchi 2022b; Lee et al. 2016; Pan et al. 2019], and its high-level idea is simple: if a template cannot be instantiated into a regex that is consistent with some example even if we replace the holes in the template with arbitrary expressions, then there is no need to consider the template or any template that can be obtained by expanding the holes of the template. However, the details of the pruning process are quite subtle, and we propose two novel pruning techniques that make use of the extraction information in positive examples, called *approximation-by-pure-regex* and *approximation-by-backreferences*. Briefly, both techniques over-approximate the problem of detecting whether the template can be pruned to the membership problem of whether a certain regex created from the template accepts certain input strings created from the positive examples. The former technique creates a pure regex (i.e., one without backreferences or lookarounds) while the latter technique does a more accurate but more expensive detection by creating a regex with backreferences. The details are found in Section 5.

For the template $.^{*?}(?<=<(\square^{*})_1>)(\square^{*?})_2(?=<[/]\backslash 1\ >).^{*}$, R3 finds that the generated constraint is satisfiable, and replaces both holes with the set of characters $[a-z]$ (which is a regex that matches any lowercase Latin alphabet) to return the repaired regex $.^{*?}(?<=<([a-z]^{*})_1)>)([a-z]^{*?})_2(?=</\backslash 1>).^{*}$, which is consistent with all the examples.

[5] In general, R3 checks if the current template can be instantiated to a regex that correctly classifies the examples, by the process described in *Searching Template Instances* (note that a concrete regex is a template that does not have holes).

[6] The idea to use continuations is inspired by the work of Sakuma et al. [2012] (cf. Section 7 for details).

3 REGEX

In this section, we define the syntax and semantics of regexes. As remarked before, unlike the case of membership, the deterministic behavior of the union and Kleene-star operators affect the semantics of regexes. In this paper, we chose to the semantics of regexes used in JavaScript for two reasons. First, JavaScript is one of the most popular programming languages for building Web applications, especially with regexes for extraction. Second, JavaScript has a language specification that defines the syntax and semantics including that of regexes in a semi-formal fashion as the ECMAScript language specification. We follow the latest version of the specification (as of November 2022), i.e., ECMAScript 2023 [ECMA International 2022].

Notation. Throughout the paper, we use the following notation. We write $\mathbb{N}$ for the set of natural numbers, and $[i]$ for the set $\{1, 2, \cdots, i\}$ for $i \in \mathbb{N}$. For a sequence l, we write $|l|$ for its length, $l[i]$ (for $0 \leq i < |l|$) for its ith element, $l[i..j]$ for the sub-sequence from the ith element to the jth element for $0 \leq i \leq j < |l|$, and $l[i..j)$ for $l[i..j-1]$. We fix a finite alphabet Σ. Then, we write $a, b \in \Sigma$ for a character; $x, y \in \Sigma^*$ for a sequence of characters (i.e., *string*); ϵ for the empty string. For a function f, we use $dom(f)$ to denote the domain of f. For a (partial) function f, we write $f[\alpha \mapsto \beta]$ for the (partial) function that maps α to β and behaves as f for all other arguments.

3.1 Syntax and Informal Semantics

The syntax of regexes is defined as follows:

$$r \quad ::= \quad [C] \mid \epsilon \mid rr \mid r|r \mid r^* \mid r^{*?} \mid (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}r) \mid (?{<}!r)$$

where $C \subseteq \Sigma$ and $i \in \mathbb{N} \backslash \{0\}$. The operator $[C]$ is the *set of characters*, which matches a character in C. We write a for $[\{a\}]$, . for $[\Sigma]$, and $[\hat{}C]$ for $[\Sigma \setminus C]$. The operators ϵ and $r_1 r_2$ are the empty string and the concatenation, respectively, and the semantics of the operators are standard. The operator $r_1|r_2$ is the *deterministic* union, which first attempts to match r_1, and if the matching fails, then it attempts to match r_2. The operators r^* and $r^{*?}$ are *greedy* and *lazy* Kleene stars, respectively, which attempt to match r as many (resp. few) as possible. The precedence order of the operators is defined as follows: the (greedy and lazy) Kleene star, the concatenation, and the union. The left one has a higher precedence. From these operators, we can construct the other operators defined in the ECMAScript language specification as syntactic sugars: the greedy and lazy Kleene plus r^+ and $r^{+?}$ as $r^* r$ and $r^{*?} r$, respectively, the greedy and lazy optional operator $r?$ and $r??$ as $r|\epsilon$ and $\epsilon|r$, respectively, and the greedy and lazy bounded repetition $r\{i, j\}$ and $r\{i, j\}?$, where $i \leq j$, as $r_1?r_2? \cdots r_{j-i}?r_{j-i+1} \cdots r_j$ and $r_1??r_2?? \cdots r_{j-i}??r_{j-i+1} \cdots r_j$, respectively. We often refer to regexes that only consist of the above operators as *pure regexes.*

The remaining operators are *real-world extensions*. The operator $(r)_i$ is the *capturing group*, which attempts to match r, and if the attempt succeeds, stores the matched substring into a storage called *environment* with the index i. Precisely, an environment is a mapping Γ from capturing group indexes to substring positions such that for each capturing group index i, $\Gamma(i)$ is the substring position stored at i. The captured substring positions are returned as extraction results.[7] While regex engines often allow capturing groups to be implicitly indexed by their positions, for readability and without loss of generality, we assume that every capturing group in our paper has an explicit index, i.e., all capturing groups in our paper are the so-called *named capturing groups*. However, as is often the case in real regex engines (and inevitably so when the indexes are implicit), we require that every capturing group in a regex has a unique index, i.e., we assume the *no-label-repetition* convention [Berglund and van der Merwe 2017]. The operator $\backslash i$ is the *backreference*,

[7] Technically, only the last captured substring positions are returned if the capturing group is matched multiple times, and capturing groups in negative lookaheads or lookbehinds are not subject to extraction (cf. Section 3.2).

which refers to the substring stored in the environment with the index i, and then attempts to match the substring. Note that, as in real regex engines, capturing groups serve the dual purpose of returning the captured substring positions as extraction results and allowing backreferences to refer to the captured substrings. The operators $(?=r)$ and $(?!r)$ are positive and negative lookaheads, respectively, which attempt to match (resp. not match) r without any character consumption. The operators $(?<=r)$ and $(?<!r)$ are positive and negative lookbehinds, respectively, which attempt to match (resp. not match) r toward the left without any character consumption.

3.2 Formal Semantics

We now define the formal semantics of regexes. Traditionally, the semantics of pure regexes is defined by induction on the structure of the pure regexes in a non-deterministic manner. In our case, it is difficult to use the same approach due to the real-world extensions and the determinism. As a result, we define the semantics of regexes by a big-step semantics that models the behavior of an actual regex engine that follows the ECMAScript 2023 language specification. Later in Section 5, we extend the semantics to generate constraints for ensuring the correctness of repaired regexes.

We define the semantics as the *deterministic matching relation* $(r, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')$, where r is a regex, r_c is a *continuation regex*, w is an input string, p and p' are positions on the input string, Γ and Γ' are *environments*, d is a *direction* of the matching, and l is a *flag* to indicate whether the evaluation is in lookarounds or not. The continuation regex is a regex that needs to be evaluated to finishing the matching. We use the continuation regex in the evaluation of the union and Kleene-star operators to disambiguate the choice in the evaluation. For $k \in \mathbb{N}$, an environment $\Gamma : [k] \longrightarrow \mathbb{N} \times \mathbb{N}$ is a function that maps an index to a pair of integers that denotes the position of a substring captured by the capturing-group operator with the index. The direction is either forward or backward, meaning the position moves from left to right or from right to left, respectively. We start an evaluation with the forward direction, and set the direction to backward when entering lookbehinds. The flag l is either true or false, which indicates whether the matching is in lookarounds. We start an evaluation with the flag with false, and set the flag to true when entering lookarounds. The judgement $(r, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')$ states that the regex r attempts to match the input string w from the position p with the environment Γ, the continuation regex r_c, the direction d, and the flag l, and changes the position to p' and the environment to Γ'. Additionally, we write $(r, r_c, w, p, \Gamma, d, l) \Downarrow$ failed to denote the failure of the matching.

We now show the rules of the semantics. For space, we only describe some representative rules shown in Figure 1. The full rules are given in Appendix E. The rules for the set-of-characters operator say that, when the direction is forward, the matching succeeds if the character $w[p]$ is in the set of characters C. When the direction is backward, it looks back and checks the character $w[p-1]$. The rules for the concatenation operator $r_1 r_2$ evaluate these expressions from left to right if the direction is forward and from right to left if the direction is backward. The forward rules for the union operator $r_1|r_2$ first evaluate the concatenation of the subexpression r_1 and the continuation regex r_c to check whether or not the consequent matching succeeds. If the matching of the concatenation consumes all the remaining characters, it means that the whole matching can be succeeded if we choose r_1.[8] From this, we then evaluate the subexpression r_1. Otherwise, we evaluate the subexpression r_2. The backward rules for the union (shown in Appendix E) are analogous but with the reverse order of r_1 and r_c.

The rules for the greedy-Kleene-star operator first decide whether the iteration should be terminated by using the continuation regex. To this end, we evaluate the concatenation $r\langle r^* : p\rangle r_c$.

[8] Technically, this behavior is only for the case l = false, i.e., when the union is not in a lookaround. When in a lookaround, it is sufficient for the concatenation to just consume some prefix of the remaining characters, as stipulated by the rule.

Set of Characters

$$\frac{d = \text{forward} \quad p < |w| \quad w[p] \in C}{([C], r_c, w, p, \Gamma, d, l) \Downarrow (p + 1, \Gamma)} \qquad \frac{d = \text{backward} \quad 0 \leq p - 1 \quad w[p-1] \in C}{([C], r_c, w, p, \Gamma, d, l) \Downarrow (p - 1, \Gamma)}$$

Concatenation and Union

$$\frac{\begin{array}{c} d = \text{forward} \quad (r_1, r_2 r_c, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \\ (r_2, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2) \end{array}}{(r_1 r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)} \quad \frac{\begin{array}{c} d = \text{backward} \quad (r_2, r_c r_1, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \\ (r_1, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2) \end{array}}{(r_1 r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{d = \text{forward} \quad (r_1 r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \quad (\neg l \wedge p_1 = |w|) \vee l \quad (r_1, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{\begin{array}{c} d = \text{forward} \quad (r_1 r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \quad \tau = \text{failed} \vee (\neg l \wedge \tau = (p'', \Gamma'') \text{ where } p'' \neq |w|) \\ (r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma') \end{array}}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

Greedy Kleene Star with Guards

$$\frac{\begin{array}{c} d = \text{forward} \quad (r\langle r^* : p\rangle r_c, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \quad p_1 = |w| \vee l \\ (r, \langle r^* : p\rangle r_c, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_2, \Gamma_2) \quad (r^*, r_c, w, p_2, \Gamma_2, d, l) \Downarrow (p_3, \Gamma_3) \end{array}}{(r^*, r_c, w, p, \Gamma, d, l) \Downarrow (p_3, \Gamma_3)}$$

$$\frac{p' = p}{(\langle r : p'\rangle, r_c, w, p, \Gamma, d, l) \Downarrow \text{failed}} \qquad \frac{p' \neq p \quad (r, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}{(\langle r : p'\rangle, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

Lazy Kleene Star with Guards

$$\frac{\begin{array}{c} d = \text{forward} \quad (r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \quad \tau = \text{failed} \vee (\neg l \wedge \tau = (p', \Gamma') \text{ where } p' \neq |w|) \\ (r\langle r^{*?} : p\rangle r_c, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_3, \Gamma_3) \quad p_3 = |w| \vee l \\ (r, \langle r^{*?} : p\rangle r_c, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \quad (r^{*?}, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2) \end{array}}{(r^{*?}, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

Capturing Group, Backreference, and Positive Lookbehind

$$\frac{d = \text{forward} \quad (r\$_i, r_c, w, p, \Gamma[i \mapsto (p, \bot)], d, l) \Downarrow (p', \Gamma')}{((r)_i, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

$$\frac{\Gamma(i) = (p', p'') \quad (w[p'..p''), r_c, w, p, \Gamma, d, l) \Downarrow (p''', \Gamma')}{(\backslash i, r_c, w, p, \Gamma, d, l) \Downarrow (p''', \Gamma')}$$

$$\frac{i \notin \mathit{dom}(\Gamma) \vee \Gamma(i) = (p', \bot)}{(\backslash i, r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)} \quad \frac{d = \text{forward} \quad \Gamma(i) = (p, \bot)}{(\$_i, r_c, w, p', \Gamma, d, l) \Downarrow (p, \Gamma[i \mapsto (p, p')])} \quad \frac{(r, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \Downarrow (p', \Gamma')}{((?{<}{=}r), r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma')}$$

Fig. 1. Selected evaluation rules for regexes.

Here, the expression $\langle r^* : p\rangle$ is the Kleene-star operator with the *guard* operator that is used to avoid non-terminating evaluation due to *problematic regexes* [Frisch and Cardelli 2004; Sakuma et al. 2012], i.e., regexes of the form r^* or $r^{*?}$ where r can match ϵ. A guard operator $\langle r : p\rangle$ checks whether or not the current position is the same as the position p. If so, then the evaluation of the guard operator fails. Otherwise, it evaluates the regex r. We return to the explanation of the rules for evaluating the greedy-Kleene-star operator. If the matching of the concatenation $r\langle r^* : p\rangle r_c$ consumes all the remaining characters, i.e., the whole matching can be succeeded even if we choose to iterate the Kleene star, then we evaluate r by $(r, \langle r^* : p\rangle r_c, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_2, \Gamma_2)$, and continue the iteration from $(r^*, r_c, w, p_2, \Gamma_2, d, l)$. Here, *reset*$(r, \Gamma)$ is an environment $\{(i, (p', p'')) \in \Gamma \mid i \text{ is not an index of a capturing group in } r\}$. This means that, for each iteration, the Kleene-star operator returns the environment back to the state before we evaluate it. This behavior is consistent with the ECMAScript 2023 language specification. Otherwise, i.e., the matching of the concatenation fails or does not consume all the remaining characters, then we cannot iterate the Kleene-star operators and therefore stop the iteration by returning the matching result (p_1, Γ_1).[9] The rules for the lazy-Kleene-star operator are similar to those for the greedy-Kleene-star operator. The main difference is that the lazy-Kleene-star operator first checks whether the whole matching can succeed without the iteration by $(r_c, \epsilon, w, p, \Gamma, d, l)$. If it succeeds, then the lazy-Kleene-star operator stops the iteration. Otherwise, it tries to iterate in a similar way as the greedy-Kleene-star operator.

The rules for the capturing-group operator $(r)_i$ evaluate the regex r with the *close symbol* $\$_i$ that denotes the end of the ith capturing group. Evaluating $\$_i$ closes the ith capturing group, and sets the end position of the substring captured by the ith capturing group. We use the close symbol to evaluate backreferences in the continuation regex correctly. The rule for the backreference operator $\backslash i$ refers to the pair of indexes in the environment $\Gamma(i) = (p', p'')$, and evaluates the substring represented by the indexes, i.e., $w[p'..p'']$. If the environment Γ does not have i in the domain (i.e., $i \notin dom(\Gamma)$) or the ith capturing group is not closed yet (i.e., $\Gamma(i) = (p', \bot)$), the backreference $\backslash i$ is evaluated as the empty string ϵ, e.g., the regex $\backslash 1(a)_1$ exactly matches the string a. In this case, the backreference is called *unassigned backreference*. The semantics is consistent with the ECMAScript 2023 specification that treats unassigned backreference as ϵ.

The rule for the positive lookbehind operator (?<=r) sets the direction to backward and the flag to true, and then evaluates the regex r. After evaluating the regex r, it resets the position p' to p. Note that we do not reset the environment. This means that substrings captured in positive lookbehinds can be used from outside of the positive lookbehinds. Additionally, the continuation in the hypothesis is set to ϵ to reflect the facts that a lookbehind only concerns whether the input string can be scanned backward from the current position to match just the expression in the lookbehind (i.e., r) and that once we move outside of the lookbehind we do not backtrack to inside of it (i.e., even in the event of a match failure in the outer continuation r_c of the conclusion).

Next, we provide some examples. In Examples 3.1 and 3.3, we omit the information of directions and flags because they are always forward and false, respectively. Additionally, in Example 3.1, we also omit the information of environments because they are always $\emptyset$. From this, the evaluations of Example 3.1 return a position instead of a pair of a position and an environment.

Example 3.1. The matching of $(a|aa)a$ on aaa is:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \in \{a\}}{(a, a, aaa, 0) \Downarrow 1} \quad \dfrac{a \in \{a\}}{(a, \epsilon, aaa, 1) \Downarrow 2}}{(aa, \epsilon, aaa, 0) \Downarrow 2} \quad 2 \neq |aaa| \quad \dfrac{\dots}{(aa, a, aaa, 0) \Downarrow 2}}{(a|aa, a, aaa, 0) \Downarrow 2} \quad \dfrac{a \in \{a\}}{(a, \epsilon, aaa, 2) \Downarrow 3}}{((a|aa)a, \epsilon, aaa, 0) \Downarrow 3.}$$

Since the whole matching of the regex on the string succeeds, the regex accepts the string.

Example 3.2. The matching of ..(?<=$(.)_1$) on ab is:

[9] This behavior is only for the case the Kleene star is not in a lookaround. The difference in the behavior when in a lookaround is similar to that for the union.

$$\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{a \in \Sigma}{(., \$_1, aa, 2, \Gamma_0, \text{backward}, \text{true}) \Downarrow (1, \Gamma_0)} \qquad \dfrac{\Gamma_0(1) = (\bot, 2)}{(\$_1, \epsilon, aa, 1, \Gamma_0, \text{backward}, \text{true}) \Downarrow (1, \Gamma_1)}}{(\$_1 ., \epsilon, aa, 2, \Gamma_0, \text{backward}, \text{true}) \Downarrow (1, \Gamma_1)}}{((.)_1, \epsilon, aa, 2, \emptyset, \text{backward}, \text{true}) \Downarrow (1, \Gamma_1)}}{((?<=(.)_1), \epsilon, aa, 2, \emptyset, \text{forward}, \text{false}) \Downarrow (2, \Gamma_1)}}{(..(?<=(.)_1), \epsilon, aa, 0, \emptyset, \text{forward}, \text{false}) \Downarrow (2, \Gamma_1)}$$

where $\Gamma_0 = \{(1, (\bot, 2))\}$ and $\Gamma_1 = \{(1, (1, 2))\}$. The regex accepts the string, and extracts the second character a by the 1st capturing group.

Example 3.3. The matching of $(.^*)_1\backslash 1$ on aa is:

$$\dfrac{A \qquad B}{((.^*)_1\backslash 1, \epsilon, aa, 0, \emptyset) \Downarrow (2, \Gamma_1)}$$

where $\Gamma_1 = \{(1, (0, 1))\}$ and A and B are subderivations. The subderivation A is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{(.\langle .^* : 0\rangle \$_1\backslash 1, \epsilon, aa, 0, \Gamma_0) \Downarrow (2, \Gamma_1)} \qquad \dfrac{a \in \Sigma}{(., \langle .^* : 0\rangle \$_1\backslash 1, aa, 0, \Gamma_0) \Downarrow (1, \Gamma_0)} \qquad \cdots}{(.^*, \$_1\backslash 1, aa, 0, \Gamma_0) \Downarrow (1, \Gamma_0)} \qquad \dfrac{\Gamma_0(1) = (0, \bot)}{(\$_1, \backslash 1, aa, 1, \Gamma_0) \Downarrow (1, \Gamma_1)}}{(.^*\$_1, \backslash 1, aa, 0, \Gamma_0) \Downarrow (1, \Gamma_1)}}{((.^*)_1, \backslash 1, aa, 0, \emptyset) \Downarrow (1, \Gamma_1)}$$

where $\Gamma_0 = \{(1, (0, \bot))\}$.

The subderivation B is:

$$\dfrac{\Gamma_1(1) = (0, 1) \qquad \dfrac{a \in \{a\}}{(a, \epsilon, aa, 1, \Gamma_1) \Downarrow (2, \Gamma_1)}}{(\backslash 1, \epsilon, aa, 1, \Gamma_1) \Downarrow (2, \Gamma_1).}$$

The regex accepts the input string, and extracts first character a by the 1st capturing group.

Example 3.4. Recall the motivating example from Section 2. We show a subderivation of the matching of the repaired regex on the input string $w = <a><a>a</a></a>$. For the subexpression $r = ([a-z]^{*?})_2$, the continuation regex $r_c = (?=<[/]\backslash 1\ >).^*$, the position $p = 6$, the environment $\Gamma = \{(1, (4, 5))\}$, the direction $d = \text{forward}$, and the flag $l = \text{false}$, the subderivation is:

$$\dfrac{\dfrac{\dfrac{(\$_2 r_c, \epsilon, w, p, \Gamma_0, d, l) \Downarrow \text{failed} \qquad A \qquad B \qquad C}{([a-z]^{*?}, \$_2 r_c, w, p, \Gamma_0, d, l) \Downarrow (p+1, \Gamma_0)} \qquad (\$_2, r_c, w, p+1, \Gamma_0, d, l) \Downarrow (p+1, \Gamma_1)}{([a-z]^{*?}\$_2, r_c, w, p, \Gamma_0, d, l) \Downarrow (p+1, \Gamma_1)}}{(r, r_c, w, p, \Gamma, d, l) \Downarrow (p+1, \Gamma_1)}$$

where $\Gamma_0 = \Gamma[2 \mapsto (p, \bot)]$ and $\Gamma_1 = \Gamma[2 \mapsto (p, p{+}1)]$. The subderivation A derives $([a{-}z]\langle [a{-}z]^{*?} : p\rangle \$_2 r_c, \epsilon, w, p, \mathit{reset}([a{-}z], \Gamma_0), d, l) \Downarrow (|w|, \Gamma_1)$, B derives $([a{-}z], \langle [a{-}z]^{*?} : p\rangle \$_2 r_c, w, p, \mathit{reset}([a{-}z], \Gamma_0), d, l) \Downarrow (p+1, \Gamma_0)$, and C derives $([a-z]^{*?}, \$_2 r_c, w, p+1, \Gamma_0, d, l) \Downarrow (p+1, \Gamma_0)$. Therefore, the regex r consumes and captures the pth character of w, i.e., the middle character a of w.

Finally, we define the language of regexes for extraction called *capturing language*, a terminology borrowed from [Loring et al. 2019]. A capturing language of a regex is a set of pairs of an input string and the environment containing the information about the substrings extracted from the input string when matched against the regex.

Definition 3.5 (Capturing Language). The *capturing language* $\mathcal{L}_c(r)$ of a regex r is defined as $\mathcal{L}_c(r) = \{(w, \Gamma) \mid (r, \epsilon, w, 0, \emptyset, \text{forward}, \text{false}) \Downarrow (|w|, \Gamma)\}$.

Additionally, we define the (non-capturing) *language* of r by $\mathcal{L}(r) = \{w \mid \exists \Gamma.(w, \Gamma) \in \mathcal{L}_c(r)\}$.

Validation of the formal semantics. In order to validate our semantics, we have implemented a regex engine based on our semantics, and evaluated it on the data sets used in our evaluation and regexes that involve JavaScript-specific behavior taken from Davis et al. [2019]. In all cases, we have confirmed that the behavior of our regex engine is consistent with that of actual JavaScript regex engines.

4 REPAIR PROBLEM

In this section, we define our PBE regex repair problem for extraction called *extraction-regex-repair problem*. There are two types of examples: *positive examples* and *negative examples*. For $k \in \mathbb{N}$, a *positive example* is a pair $(w, \Gamma) \in \Sigma^* \times ([k] \to \mathbb{N} \times \mathbb{N})$ for some $k \geq 1$ that should belong to the capturing language of the to-be synthesized regex. That is, w is a string to be accepted and Γ denotes the indexes of the substrings to be extracted from w. For readability, we use $\llparenthesis_i$ and $\rrparenthesis_i$ (or $\llparenthesis$ and $\rrparenthesis_i$ if there is no danger of ambiguity) to denote a substring to be extracted by the ith capturing group in a positive example. For example, we write $\llparenthesis_1 a \llparenthesis_2 b \rrparenthesis_2 c \rrparenthesis_1$ to denote a positive example (abc, Γ) where $\Gamma = \{(1, (0, 3)), (2, (1, 2))\}$. A *negative example* is $w \in \Sigma^*$ that should not belong to the language of the to-be synthesized regex, i.e., a string to be rejected.

Before we define the repair problem, we recall the notion of *distance* from prior works on PBE-based repair of regexes [Chida and Terauchi 2022b; Pan et al. 2019]. The notion is used to quantify the quality of a repair. That is, a regex of a short distance from the pre-repair regex is deemed to be of high quality, justified by the assumption that the pre-repair regex may not be correct but is close to the one the user intended.

Definition 4.1 (Distance [Chida and Terauchi 2022b; Pan et al. 2019]). For subtrees $r_1, \ldots, r_n$ of a regex r, the *edit* $r[r'_1/r_1, \ldots, r'_n/r_n]$ replaces each r_i with r'_i. The cost of an edit is the sum of the number of nodes in r_i and r'_i (for $i \in [n]$). Given two regexes r_1 and r_2, the *distance* between r_1 and r_2, $D(r_1, r_2)$, is the minimum cost of an edit that rewrites r_1 to r_2.

We now define the extraction-regex-repair problem.

Definition 4.2 (Extraction-Regex-Repair Problem). Given a regex r_1, a set of positive examples $\mathcal{E}^+$ and a set of negative examples $\mathcal{E}^-$ satisfying satisfying $(w, \Gamma_1), (w, \Gamma_2) \in \mathcal{E}^+ \Rightarrow \Gamma_1 = \Gamma_2$ and $\mathcal{E}^- \cap \{w \mid (w, \_) \in \mathcal{E}^+\} = \emptyset$, the *extraction-regex-repair problem* is the problem of synthesizing a regex r_2 that is consistent with the examples (i.e., $\mathcal{E}^+ \subseteq \mathcal{L}_c(r_2)$ and $\mathcal{E}^- \cap \mathcal{L}(r_2) = \emptyset$) and the distance from r_1 is minimal (i.e., $D(r_1, r_2) \leq D(r_1, r_3)$ for any r_3 consistent with the examples).

In what follows, we show some interesting results regarding the extraction-regex-repair problem: (1) there are unsolvable problem instances if repaired regexes are not allowed to use the extensions (i.e., pure regexes), whereas (2) every problem instance is solvable if the extensions are allowed. For (1), we show that there are *non-trivial* unsolvable problem instances, i.e., there are unsolvable problem instances even if positive examples do not have *overlaps* in its extraction such as $(abc, \{(1, (0, 2)), (2, (1, 3))\})$, represented as $\llparenthesis_1 a \llparenthesis_2 b \rrparenthesis_1 c \rrparenthesis_2$.[10]

Theorem 4.3. *There exist extraction-regex-repair-problem instances without overlaps that do not have a pure regex solution.*

Proof. Let $\Sigma = \{a, b\}$ for simplicity. Then, we show that the instance $r_1 = \epsilon, \mathcal{E}^+ = \{(ab, \Gamma_0), (ba, \Gamma_1)\}$ where $\Gamma_0 = \{(1, (0, 1))\}$ and $\Gamma_1 = \{(1, (1, 2))\}$, and $\mathcal{E}^- = \{\epsilon, a, b, aa, bb\}$ does not have a pure regex solution. Roughly, if there exists a solution, then the form of the solution must be $r'(r_a)_1 r''$ where r_a is a regex that matches the character a, and r' and r'' need to match the empty string ϵ and the

[10] It is trivial to see that no pure regex can have such an extraction result. By contrast, it is an interesting consequence of Theorem 4.4 that the repair problem is solvable even with such overlaps when the extensions are allowed.

Algorithm 1: The repair algorithm

```
Input: a regex $r$, a set of positive examples $\mathcal{E}^+$, and a set of negative examples $\mathcal{E}^-$
Output: a regex that satisfies the consistency condition and the minimal distance criteria
Q ← { $r$ }
while Q is not empty do
    t ← Q.pop()
    if $\{w \mid (w, \_) \in \mathcal{E}^+\} \subseteq \mathcal{L}(t_\top)$, $\mathcal{E}^- \cap \mathcal{L}(t_\bot) = \emptyset$, and IsFeasible($t$, $\mathcal{E}^+$, $.^*$) then
        if $\{w \mid (w, \_) \in \mathcal{E}^+\} \subseteq \mathcal{L}(t_{\top,\Sigma})$ and IsFeasible($t$, $\mathcal{E}^+$, $.$) then
            $\Phi$ ← generateConstraintForExtraction($t, \mathcal{E}^+, \mathcal{E}^-$)
            if $\Phi$ is satisfiable then
                return solution($t, \Phi$)
        Q.push(expandHoles($t$))
    Q.push(addOrReduceHoles($t$))
```

character b to be a solution. However, the regex accepts the negative example $a \in \mathcal{E}^-$, which is a contradiction. The full proof appears in Appendix A. □

Theorem 4.4. *The extraction-regex-repair problem always has a solution.*

Proof. We construct a regex that only accepts the positive examples by using lookarounds, and therefore it satisfies the consistency condition. Then, the result follows from the fact that the existence of a regex consistent with the examples implies the existence of a minimal such one. The full proof appears in Appendix B. □

These results indicate that the real-world extensions play a crucial role in the repair problem. Finally, we show that the extraction-regex-repair problem is NP-hard. We prove this by a reduction from the set-cover problem which is NP-complete [Karp 1972]. For this, we consider the decision version of the extraction-regex-repair problem. That is, given a regex r_1, sets of positive and negative examples, and $k \in \mathbb{N}$, the decision version of the extraction-regex-repair problem is the problem of synthesizing a regex r_2 that is consistent with the examples and satisfies $D(r_1, r_2) \leq k$. The proof is in Appendix C.

Theorem 4.5. *The extraction-regex-repair problem is NP-hard.*

5 REPAIR ALGORITHM

We now describe our algorithm for solving the repair problem. At a high level, our algorithm is based on the enumerative search with an SMT solver introduced in the previous works for membership [Chida and Terauchi 2022b; Pan et al. 2019]. The main new ideas introduced in this paper are the SMT constraint generation that respects the deterministic semantics and the new pruning techniques that make use of the extraction information in the positive examples. We first give an overview of our algorithm in Section 5.1, and then present the details of the new ideas in Section 5.3 and 5.2.

5.1 Overview

Algorithm 1 shows the algorithm. Our algorithm takes as input an instance of the extraction-regex-repair problem, i.e., a regex r, a set of positive examples $\mathcal{E}^+$, and a set of negative examples $\mathcal{E}^-$. It returns a regex that satisfies the condition required for the solution of the problem, i.e., it is one that is consistent with the examples and the distance from r is minimal.

Initializing a template. Our algorithm maintains a priority queue Q to store *template* regexes. A template regex is a regex with the operator called *hole* denoted as $\square$. More specifically, the syntax of the template t is defined as: $t ::= [C] \mid \epsilon \mid tt \mid \cdots \mid (?{<}!t) \mid \square$, i.e., the only difference between the syntax of regexes and templates is the existence of the hole. Roughly, a hole is a placeholder that is to be replaced with some concrete regex. At the beginning, our algorithm initializes the priority queue Q to contain the given regex r as a template at Line 1. The priority queue Q ranks its elements by the *distance* so that the template that is closer to the given regex r in terms of the distance has a higher priority. This ensures that the returned regex satisfies the minimality criteria, i.e., it is of minimal distance among ones that are consistent with the examples.

Template Pruning by Approximations. Next, we retrieve a template that has the highest priority from the priority queue Q at Line 3. Then, we apply *template pruning* at Line 4. The test $\{w \mid (w, \_) \in \mathcal{E}^+\} \subseteq \mathcal{L}(t_\top), \mathcal{E}^- \cap \mathcal{L}(t_\perp) = \emptyset$ is analogous to the pruning used in the prior works on membership [Chen et al. 2020; Chida and Terauchi 2022b; Lee et al. 2016; Pan et al. 2019], and works by building an over- and under-approximating regexes $t_\top$ and $t_\perp$. The regexes satisfy the properties $\mathcal{L}(r') \subseteq \mathcal{L}(t_\top)$ for any regex r' obtainable by filling the holes of t by arbitrary regexes. Therefore, if the test fails, then we can safely prune the template and any template obtainable by expanding the holes of the template (cf. **Expanding, Adding, or Reducing Holes** below). The construction of $t_\top$ and $t_\perp$ can be found in Appendix D.

However, the above pruning is not enough for our purpose because it does not consider extraction. For example, consider a template $t = a(b\square)_1$ and a positive example $(\!| abc |\!)_1$. The above pruning technique cannot discard the template because the over-approximated regex is $t_\top = a(b.^*)_1$ and $\{abc \mid (abc, (1, 3)) \in \mathcal{E}^+\} \subseteq \mathcal{L}(t_\top)$. However, t cannot become a regex consistent with the example because any regex obtained from it consumes the first character a without extraction.

Thus, for positive examples, we apply new template pruning techniques for extraction introduced in this paper, denoted as IsFeasible in Algorithm 1. The idea of our pruning is to construct a regex and an input string from a template and a positive example by embedding the information of extraction. We defer the details of our pruning technique for extraction to Section 5.3.

Constraint Solving for Finding a Solution. If the pruning by approximations passes, then we check whether the template can be turned into a regex that is consistent with the examples by replacing its holes with some sets of characters. For this, our algorithm generates an SMT constraint at Line 6. As remarked before, the constraint generation builds on that proposed for membership [Chida and Terauchi 2022b; Pan et al. 2019], and works by generating an SMT formula ϕ_e constructed for each example $e \in \mathcal{E}^+ \cup \mathcal{E}^-$ that encodes the condition for correctly classifying e. In more detail, ϕ_e is over propositional variables v_i^a where $i \in \mathbb{N}$ denotes the ith hole in the template and $a \in \Sigma$. The variable v_i^a is set to true if and only if the set of characters that fills the ith hole contains the character a. The final constraint is the conjunction $\bigwedge_{e \in \mathcal{E}^+ \cup \mathcal{E}^-} \phi_e$, and if the constraint is satisfiable, we obtain from the solution a regex that correctly classifies all examples. As remarked before, one of the key innovations of our work is a method for generating a constraint that respects the deterministic semantics of an actual regex engine. For this, we design the constraint generation algorithm by following our novel formal semantics presented Section 3.2. We defer further details of the constraint generation to Section 5.2. Once we find a solution, we apply the following generalization technique to the solution: we add characters to the sets of characters that were obtained by replacing holes in the template as long as the additions do not violate the consistency of the examples.

Expanding, Adding, or Reducing Holes. If the SMT constraint is unsatisfiable, then we continue to explore by enumerating templates. At Line 9, we *expand* holes in the template by replacing a hole with an expression all of whose immediate subexpressions are holes. At Line 10,

Hole

$$\frac{d = \text{forward} \qquad \square \text{ is the } i\text{th hole.} \qquad p < |w|}{(\square, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p+1, \Gamma, v_i^{w[p]})\}, \{(\bot, \bot, \neg v_i^{w[p]})\})}$$

$$\frac{d = \text{backward} \qquad \square \text{ is the } i\text{th hole.} \qquad 0 \leq p - 1}{(\square, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p-1, \Gamma, v_i^{w[p-1]})\}, \{(\bot, \bot, \neg v_i^{w[p-1]})\})}$$

Set of Characters

$$\frac{d = \text{forward} \qquad p < |w| \qquad w[p] \in C}{([C], t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p+1, \Gamma, \text{true})\}, \emptyset)} \qquad \frac{d = \text{backward} \qquad 0 \leq p - 1 \qquad w[p-1] \in C}{([C], t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p-1, \Gamma, \text{true})\}, \emptyset)}$$

Union

$$\frac{\begin{array}{c} d = \text{forward} \qquad (t_1 t_c, \epsilon, w, p, \Gamma, d, l) \dashrightarrow (S_1, F_1) \\ S_1' = ite(l, S_1, \{(p', \Gamma', \phi') \in S_1 \mid p' = |w|\}) \qquad F_1' = F_1 \cup ite(l, \emptyset, \{(\bot, \bot, \phi') \mid (p', \_, \phi') \in S_1, p' \neq |w|\}) \\ \phi_{S_1} = \bigvee_{(\_,\_,\phi') \in S_1'} \phi' \qquad \phi_{F_1} = \bigvee_{(\bot,\bot,\phi') \in F_1'} \phi' \\ (t_1, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_2, F_2) \qquad (t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_3, F_3) \end{array}}{\begin{array}{c} (t_1 | t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p', \Gamma', \phi_{S_1} \land \phi') \mid (p', \Gamma', \phi') \in S_2\} \cup \{(p', \Gamma', \phi_{F_1} \land \phi') \mid (p', \Gamma', \phi') \in S_3\}, \\ \{(\bot, \bot, \phi_{F_1} \land \phi') \mid (\bot, \bot, \phi') \in F_3\}) \end{array}}$$

Capturing Group, Backreference, and Positive Lookbehind

$$\frac{\begin{array}{c} d = \text{forward} \\ (t\$_i, t_c, w, p, \Gamma[i \mapsto (p, \bot)], d, l) \dashrightarrow (S, F) \end{array}}{((t)_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)} \qquad \frac{d = \text{forward} \qquad \Gamma(i) = (p', \bot)}{(\$_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma[i \mapsto (p', p)], \text{true})\}, \emptyset)}$$

$$\frac{\Gamma(i) = (p', p'') \quad (w[p'..p''), t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)}{(\backslash i, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)} \qquad \frac{i \notin dom(\Gamma) \lor \Gamma(i) = (p', \bot)}{(\backslash i, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \text{true})\}, \emptyset)}$$

$$\frac{(t, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \dashrightarrow (S, F)}{((?{<}{=}t), t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma', \phi') \mid (\_, \Gamma', \phi') \in S\}, F)}$$

Fig. 2. Selected rules for generating an SMT constraint.

we *add* a hole to the template by replacing a leaf subexpression (i.e., a set of characters $[C]$, the empty string ϵ, or a backreference $\backslash i$) by a hole. Additionally, we *reduce* an expression having a hole as an immediate subexpression into a hole. For example, we have $\mathsf{expandHoles}(\square) = \{\square\square, \square|\square, \square^*, \square^{*?}, (\square)_i, (?{=}\square), (?!\square), (?{<}{=}\square), (?{<}!\square)\}$. And for the template $t = (\square|\square)a\backslash i$, we have $\mathsf{addOrReduceHoles}(t) = \{\square a \backslash i, (\square|\square)\square\backslash i, (\square|\square)a\square\}$. This part is essentially identical to that of the previous works [Chida and Terauchi 2022b; Pan et al. 2019], and we refer to them for further information.

5.2 SMT Constraint Generation

We describe the SMT constraint generation. As remarked before, for each example, we construct an SMT formula that encodes the condition that the regex obtained from a satisfying assignment to the formula correctly classifies the example. For this, we define constraint generation rules that derive judgements of the form $(t, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)$. The rules are designed based on the rules of our novel formal semantics of regexes (cf. Section 3.2) and have a similar form. Namely, t is a template regex, t_c is a continuation template regex, w is an input string, p is a position on the input string, Γ is an environment, d is a direction, and l is a flag. S and F are sets of *results of succeeded or failed matching with constraints*. A result of succeeded (resp. failed) matching with constraints is a tuple (p', Γ', ϕ), meaning that the matching succeeds (resp. fails) at position p' with the environment Γ'

if the constraint ϕ is satisfiable. In the case of failed matching, $p' = \bot$ and $\Gamma' = \bot$. For space, we show selected rules in Figure 2. The full rules are given in Appendix F. In the figure, the function *ite* is defined by: $\mathit{ite}(\mathrm{true}, A, B) = A$ and $\mathit{ite}(\mathrm{false}, A, B) = B$.

The main difference from the rules of the formal semantics is that the constraint generation rules return (a pair of) *sets* of matching results each paired with an SMT formula, whereas the semantics just returns a single matching result. This is so because the constraint generation rules simulate all possible (deterministic) runs of the regex on the example string that can be obtained by changing the sets of characters to fill the holes, pair each such result with the constraint formula that must be satisfied for the run that yields the result to be actually happen, and return the set of such pairs (divided into two sets: one for the successful runs, i.e., S, and one for the failing runs, i.e., F).

For example, in the rules for holes, the matching can succeed (resp. fail) at position p by assigning (resp. not assigning) the character $w[p]$ in the set of characters to fill the hole. Thus, in the case of $d = \mathrm{forward}$, the rule returns as the successful result $(p + 1, \Gamma, v_i^{w[p]})$ and as the failing result $(p - 1, \Gamma, v_i^{w[p-1]})$, meaning that the hole can be filled with the set of characters $[C]$ such that $w[p] \in C$ (resp. $w[p-1] \notin C$) to succeed (resp. fail) the matching. The rules for the other operators follow the corresponding rules of the formal semantics, and similarly to the rules for holes, take care of the fact that there can be multiple matching results due to the choice of how the holes are filled. For example, like the corresponding rule of the formal semantics, the forward rule for the union operator $t_1|t_2$ first evaluates the concatenation of the subexpression t_1 and the continuation template regex t_c. However, unlike the corresponding rule of the formal semantics, the rule of the concatenation $t_1 t_c$ returns sets of results of succeeded and failed matching with constraints. Therefore, the rule next evaluates both cases of succeeded and failed matching. In the case of the succeeded matching, the rule evaluates the subexpression t_1, and obtains the sets of succeeded and failed matching results with constraints, i.e., S_2 and F_2, respectively. Similarly, in the case of the failed matching, the rule evaluates the subexpression t_2, and obtains the sets of succeeded and failed matching results with constraints, i.e., S_3 and F_3, respectively. Finally, from these results, the rule constructs the results for $t_1|t_2$. Specifically, the results of succeeded matching of $t_1|t_2$ consist of the case that the matchings of both $t_1 t_c$ and t_1 succeed and the case that the matching of $t_1 t_c$ fails but the matching of t_2 succeeds. Hence, they consist of $(p', \Gamma', \phi_{S_1} \wedge \phi')$, where $(p', \Gamma', \phi') \in S_2$ and ϕ_{S_1} is the constraint for succeeding in the matching of $t_1 t_c$, and $(p', \Gamma', \phi_{F_1} \wedge \phi')$, where $(p', \Gamma', \phi') \in S_3$ and ϕ_{F_1} is the constraint for failing in the matching of $t_1 t_c$. Also, the results of failed matching of $t_1|t_2$ consist of the case that the matching of both $t_1 t_c$ and t_2 fails. Hence, they consist of $(\bot, \bot, \phi_{F_1} \wedge \phi')$ where $(\bot, \bot, \phi') \in F_3$ and ϕ_{F_1} is the constraint for failing in the matching of $t_1 t_c$.

Finally, for a template t, a set of positive examples $\mathcal{E}^+$, and a set of negative examples $\mathcal{E}^-$, we define the SMT constraints as $\Phi = \Phi^+ \wedge \Phi^-$ where Φ^+ and Φ^- are constraints for positive and negative examples, respectively. The constraints Φ^+ and Φ^- are defined as follows.

- $\Phi^+ = \bigwedge_{e=(w,\Gamma)\in\mathcal{E}^+} \mathrm{encode}^+(t, e)$ where $\mathrm{encode}^+(t, e) = \bigwedge_{(|w|,\Gamma,\phi)\in S} \phi$ and
- $\Phi^- = \bigwedge_{w\in\mathcal{E}^-} \neg\mathrm{encode}^-(t, w)$ where $\mathrm{encode}^-(t, w) = \bigwedge_{(|w|,\_,\phi)\in S} \phi$

with $(t, \epsilon, w, 0, \emptyset, \mathrm{forward}, \mathrm{false}) \dashrightarrow (S, F)$.

As an example of the SMT constraint generation, consider a template $t = (a|aa)_1(\square|\epsilon)$, a set of positive examples $\mathcal{E}^+ = \{\llparenthesis aa \rrparenthesis_1\}$, and a set of negative examples $\mathcal{E}^- = \{ab\}$. Then, the constraint for the positive example is $\Phi^+ = \neg v_1^a$ and the constraint for the negative example is $\Phi^- = \neg v_1^b$. Therefore, the constraint for the examples is $\Phi = \neg v_1^a \wedge \neg v_1^b$. Additionally, consider the template $.^{*?}(?\!<\!=\!<\!(\square^*)_1\!>)(\square^{*?})_2(?\!=\!<\![/]\backslash 1\ >).^*$ from Section 2. The constraints for the subexpression $t = (\square^{*?})_2$ on the example $w = {<}a{>}{<}\llparenthesis a \rrparenthesis_1{>}\llparenthesis a \rrparenthesis_2{<}/a{>}{<}/a{>}$ from the position $p = 6$ with the continuation template regex $t_c = (?\!=\!<\![/]\backslash 1\ >).^*$, the environment $\Gamma = \{(1, (4, 5))\}$, the direction $d = \mathrm{forward}$, and the flag $l = \mathrm{true}$ are constructed by the judgement $(t, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)$ where $S =$

$\{(p, \Gamma \cup \{(2,(6,6))\}, \neg v_2^a), (p+1, \Gamma \cup \{(2,(6,7))\}, v_2^a)\}$ and $F = \emptyset$. Here, for example, the second element $(p+1, \Gamma \cup \{(2,(6,7))\}, v_2^a)$ of S indicates that, if the hole of $(\square^{*?})_2$ is replaced with the set of characters $[C]$ where $a \in C$, then $(([C]^{*?})_2, \texttt{(?=<[/]\textbackslash 1 >).}^*, w, p, \Gamma, d, l) \Downarrow (p+1, \Gamma \cup \{(2,(6,7))\})$.

5.3 Over-Approximation for Extraction

In this section, we introduce two novel pruning techniques for extraction. We first introduce the pruning technique that exploits backreferences, namely *approximation-by-backreferences*, to ensure the correctness of extraction in Section 5.3.1. Although the pruning technique can support real-world extensions, it sometimes suffers from *regular expression denial of service* (ReDoS) [Davis et al. 2018] that stops the repair procedure for a long time. To avoid ReDoS, we introduce *ReDoS-free* pruning technique for extraction, namely *approximation-by-pure-regex*, in Section 5.3.2. Although the second pruning technique does not support real-world extensions, we can mitigate the ReDoS issue by using it before we use the first pruning technique.

5.3.1 Approximation by Backreferences. We introduce our first pruning technique for extraction *approximation-by-backreferences*. Given a template t and a set of positive examples $\mathcal{E}^+$, the over-approximation checks whether there exists a *completion* r' of t, i.e., a regex obtained from t by replacing the holes in t with some regex, such that for each capturing group that appears in the template t, r' correctly extracts substrings from the positive examples. For this, it constructs a regex r from the template t and input strings w_e from each positive example $e \in \mathcal{E}^+$ such that $\forall e \in \mathcal{E}^+. ((\exists r' \in \mathcal{R}(t). (e \in \mathcal{L}_c(r')) \Rightarrow w_e \in \mathcal{L}(r))$, where $\mathcal{R}(t)$ is the set of valid completions.

We first show the construction for the input string. The idea of the construction is to append the information of substrings to be extracted by the ith capturing group with the delimiter $\#_i$. The information consists of a string of the positive example and two delimiters @. We represent a position of a substring to be extracted by surrounding the substring with the delimiters @, e.g., the string $a@a@b$ means that the second a should be extracted from the string aab. Formally, given a template t and a positive example $e = (w', \Gamma)$, let $I = \{i_1, i_2, \cdots, i_n\}$ be the set of indexes of capturing groups in t, then we construct an input string w as follow: $w = @w'\#_1 w_{i_1} \#_2 w_{i_2} \# \cdots \#_n w_{i_n}$, where, for $j \in [n]$, $\#_j, @ \notin \Sigma$ and, for $i \in I$, $w_i = w'[0..p_l)@w'[p_l..p_r)@w'[p_r..|w'|)$ with $\Gamma(i) = (p_l, p_r)$ if $i \in dom(\Gamma)$, and otherwise $w_i = \epsilon$. That is, for the case of $i \in I$ and $i \in dom(\Gamma)$, w_i means that the substring $w'[p_l..p_r)$ between the delimiters @ should be extracted by the ith capturing group, and the prefix $w'[0..p_l)$ and the suffix $w'[p_r..|w'|)$ are used to indicate the position of $w[p_l..p_r)$. For the case of $i \notin dom(\Gamma)$, w_i means that the ith capturing group does not extract any substring. The first character @ is prepended to handle the delimiters @ in the regex. For example, consider a set of positive examples $\mathcal{E}^+ = \{a(\!|\,|\!)_2 d\}$. For the positive example $a(\!|\,|\!)_2 d = (ad, \{(2,(1,1))\})$, the input string is $@w'\#_1 w_1 \#_2 w_2 = @ad\#_1\#_2 a@@d$. Here, $w_1 = \epsilon$ indicates that the 1st capturing group does not extract any substring and $w_2 = a@@d$ indicates that the 2nd capturing group extracts the empty string ϵ.

Next, we show the construction of the regex. To verify the correctness of extraction, we construct two types of expressions: $r_{\mathit{assert1},I}$ and $r_{\mathit{assert2},I}$, where I is a set of indexes of capturing groups. The expressions $r_{\mathit{assert1},I}$ and $r_{\mathit{assert2},I}$ are used for verifying the correctness of extraction at the end of the whole matching and positive lookarounds, respectively. First, the expression $r_{\mathit{assert1},I}$ is defined as follows.

- For $I = \{i_1, i_2, \cdots, i_n\}$, $r_{\mathit{assert1},I} = r_{\mathit{assert1},i_1} r_{\mathit{assert1},i_2} \cdots r_{\mathit{assert1},i_n}$, where
- for $j \in [n]$, $r_{\mathit{assert1},i_j} = \#_j \backslash \mathit{prefix}_{i_j} \backslash \mathit{flag}_{i_j} \backslash i_j \backslash \mathit{flag}_{i_j} \backslash \mathit{suffix}_{i_j}$

In the above definition, we use a string as an index of capturing groups instead of integers for readability. For $i_j \in I$ with $j \in [n]$, the expression r_{assert1,i_j} checks whether the extracted substring by the i_jth capturing group is correct, i.e., the expression r_{assert1,i_j} tries to match the substring

$\#_j w_{i_j}$ of the input string constructed above. From this, if w_{i_j} is of form $w'_p@w'_{i_j}@w'_s$, then the backreferences $\backslash \mathit{prefix}_{i_j}$ and $\backslash \mathit{suffix}_{i_j}$ try to match w'_p and w'_s, respectively, the backreferences $\backslash \mathit{flag}_{i_j}$ try to match the delimiter @, and the backreference $\backslash i_j$ tries to match w'_{i_j}. Otherwise, i.e., $w_{i_j} = \epsilon$, the expression tries to match the empty string ϵ. Note that if a backreference is unassigned, then it is handled as the empty string ϵ. As a result, if the matching succeeds, then it means that the i_jth capturing group extracts the correct substring in the correct position. Additionally, we need to check the correctness of extraction in positive lookarounds because once we move outside of positive lookarounds, then we cannot backtrack to the inside of the positive lookarounds, and therefore, we cannot check the correctness of extraction in positive lookarounds by the expression $r_{\mathit{assert1},I}$. From this, before we move outside of positive lookarounds, we check whether the extracted substrings in the positive lookarounds are correct or not by using the expression $r_{\mathit{assert2},I}$.

The expression $r_{\mathit{assert2},I}$ is defined as follows.

- For $I = \{i_1, i_2, \cdots, i_n\}$ $r_{\mathit{assert2},I} = r_{\mathit{assert2},i_1} r_{\mathit{assert2},i_2} \cdots r_{\mathit{assert2},i_n}$, where
- for $j \in [n]$, $r_{\mathit{assert2},i_j} = (?{=}[\Sigma \cup \{\#_1, \#_2, \cdots, \#_{j-1}, \#_{j+1}, \cdots \#_n\}]^* \#_j r_{\mathit{assert1},i_j})$.

Basically, the expression $r_{\mathit{assert2},I}$ behaves like the expression $r_{\mathit{assert1},I}$, but the subexpressions $r_{\mathit{assert2},i}$ is nested by a positive lookahead to change the direction of the matching to forward. Additionally, for $i_j \in I$, we add the Kleene star to move the position to the expression $\#_j r_{\mathit{assert1},i_j}$ for checking the correctness of extraction of the i_jth capturing group.

With them, we construct a regex for the approximation-by-backreferences as follows. For a template t, we use I_t to denote the set of indexes of capturing groups that appear in t. Given a template t, we construct a regex $r = @r_{\mathit{body}} r_{\mathit{assert1},I_t}$, where r_{body} is calculated as follows. First, we construct the over-approximated regex $r_\top$ by the over-approximation for membership (cf. Appendix D). Then, we construct r_{body} by embedding expressions to check the correctness of extraction into the regex $r_\top$ by using the function β, i.e., $r_{\mathit{body}} = \beta(r_\top)$. The definition of the function β is as follows.

$$\begin{array}{rclrclrcl}
\beta([C](\text{resp. } \epsilon)) & = & [C](\text{resp. } \epsilon) & \beta(t_1^*) & = & \beta(t_1)^* & \beta((?{<}!t_1)) & = & (?{<}!\beta(t_1)) \\
\beta(t_1 t_2) & = & \beta(t_1)\beta(t_2) & \beta(t_1^{*?}) & = & \beta(t_1)^{*?} & \beta((?{=}t_1)) & = & (?{=}\beta(t_1) r_{\mathit{assert2},I_{t_1}}) \\
\beta(t_1|t_2) & = & \beta(t_1)|\beta(t_2) & \beta((?!t_1)) & = & (?!\beta(t_1)) & \beta((?{<=}t_1)) & = & (?{<=}r_{\mathit{assert2},I_{t_1}} \beta(t_1)) \\
\end{array}$$
$$\beta((t_1)_i) = (?{<=}(@)_{\mathit{flag}_i}(.^*)_{\mathit{prefix}_i})(\beta(t_1))_i(?{=}(.^*)_{\mathit{suffix}_i})$$

Note that backreferences are eliminated by the approximation for membership. The function β inserts the expression $r_{\mathit{assert2},I_{t_1}}$ into positive lookarounds to verify the correctness of the extraction before the matching moves outside of the positive lookarounds. The main part of the construction by β is the case of capturing groups. In that case, it inserts the positive lookbehind and the positive lookahead before and after the ith capturing group, respectively. The positive lookbehind extracts the delimiter @ with the index flag_i and the substring from the beginning of the input string to the position before starting the ith capturing group with the index prefix_i. The positive lookahead extracts the substring from the position after ending the ith capturing group to the end of the input string with the index suffix_i.

As a result, IsFeasible at Line 4 computes the regex r and the input string w_e for each example $e \in \mathcal{E}^+$ by the procedures described above, and checks whether $w_e \in \mathcal{L}(r)$ for each $e \in \mathcal{E}^+$. If so, there may exist a completion and therefore R3 expands the holes in the template. Otherwise, R3 does not expand the holes. As an example of the approximation, consider a template $t = \square|(b)_1$ and a set of positive examples $\mathcal{E}^+ = \{(\!|b|\!)_1, a\}$. Then, the regex obtained by the over-approximation for membership is $r_\top = .^*|(b)_1$, i.e., the hole $\square$ is replaced with $.^*$, and the regex r obtained by applying the function β to $r_\top$ and attaching @ and r_{assert1,I_t} before and after it is as follows.

$$r \quad = \quad @(.^*|(?{<=}(@)_{\mathit{flag}_1}(.^*)_{\mathit{prefix}_1})(b)_1(?{=}(.^*)_{\mathit{suffix}_1}))\#_1 \backslash \mathit{prefix}_1 \backslash \mathit{flag}_1 \backslash 1 \backslash \mathit{flag}_1 \backslash \mathit{suffix}_1.$$

For the positive examples $\llparenthesis b \rrparenthesis_1$ and a, the input strings are $w_1 = @b\#_1@b@$ and $w_2 = @a\#_1$. Recall that unassigned backreferences in the JavaScript semantics are handled as the empty string ϵ. From this, in the matching of r on w_2, the backreferences in r are evaluated as the empty string ϵ.

Finally, the regex and the input strings obtained by the above construction satisfy the following property.

Proposition 5.1. *Given a template t and a set of positive examples $\mathcal{E}^+$, let r and w_e for each $e \in \mathcal{E}^+$ be the regex and the input strings obtained by the above construction. Then, $\forall e \in \mathcal{E}^+.\ ((\exists r' \in \mathcal{R}(t).\ (e \in \mathcal{L}_c(r')) \Rightarrow w_e \in \mathcal{L}(r))$.*

We use a variant of the approximation-by-backreferences at Line 5 of Algorithm 1. The variant uses . instead of .* when we apply the approximation for membership.

5.3.2 Approximation by Pure Regex. Next, we introduce our second pruning technique *approximation-by-pure-regex*. At a high level, the idea is the same as our first pruning technique, i.e., we convert a template and a positive example to a regex and an input string, respectively, to have the information of substrings to be extracted. However, the second pruning technique takes a different approach to the conversion. The conversion used in the second pruning technique eliminates real-world extensions in a template by approximating them to obtain a pure regex. From this, we can use nonbacktracking regex engines, i.e., automata-based regex engines such as Google RE2 [Google [n. d.]] which are known as ReDoS-free regex engines [Cox 2007], for membership. Note that the regex-engine-dependent behavior does not affect results of the matching in terms of membership and there is no automata-based regex engine that fully supports real-world extensions such as backreferences and lookarounds[11] (and therefore we cannot use automata-based regex engines for the first pruning technique).

We first describe the conversion from a positive example to an input string. Let $e = (w, \Gamma)$ be a positive example and I be the set of indexes of capturing groups that appear in the template. Then, for each index $i \in dom(\Gamma) \cap I$, we insert the delimiters $\#_i$ in the position described by $\Gamma(i)$, i.e., we simply replace $\llparenthesis_i$ and $\rrparenthesis_i$ with $\#_i$. For example, we convert the positive example $\llparenthesis_1 \llparenthesis_2 a \rrparenthesis_2 b \rrparenthesis_1 c$ to the input string $\#_1\#_2 a\#_2 b\#_1 c$. The delimiters indicate the positions of substrings to be extracted.

Next, we consider the conversion from a template to a pure regex. Before we describe the details of the conversion, we first explain the procedure to eliminate backreferences by approximating them, which is also used in the approximation for membership. For eliminating backreferences, we first replace backreferences $\backslash i$ that are always evaluated as unassigned backreferences and those that can be evaluated as both assigned and unassigned references with the empty string ϵ and $(\backslash i|\epsilon)$, respectively. For example, we convert the template $((\backslash 2\square)_1(\backslash 1b)_2)^*$ to the template $((\epsilon\square)_1(\backslash 1b)_2)^*$ because the Kleene-star operator in JavaScript resets substrings extracted in the Kleene-star operator for each iteration (cf. Section 3), and therefore the backreference $\backslash 2$ is always evaluated as an unassigned reference. Additionally, for example, we convert the template $((\square)_1|a)\backslash 1$ to the template $((\square)_1|a)(\backslash 1|\epsilon)$ because the backreference $\backslash 1$ may refer to the expression of the 1st capturing group (i.e., $\square$) or it may be an unassigned reference (i.e., ϵ). By the replacements, the relations between backreferences and the corresponding capturing groups form a directed acyclic graph (DAG). Therefore, we eliminate backreferences by approximating them in a topological order starting from the backreference whose corresponding capturing group does not have a backreference. For this, for backreferences $\backslash i$ and the corresponding capturing groups $(t)_i$, we replace $\backslash i$ with t' where t' is an expression obtained by removing capturing groups in t. For

[11] Actually, automata models that are equivalent to regexes with real-world extensions are unknown even for regexes with backreferences and (positive and negative) lookaheads [Chida and Terauchi 2022a].

example, we convert the template $(\square)_1(\backslash 1a)_2\backslash 2$ to the template $(\square)_1(\square a)_2\square a$, and the template $(?{=}((\square)_2\backslash 2)_1)\backslash 1$ to the template $(?{=}((\square)_2\square)_1)\square\square$.

Now, we describe the conversion. The conversion first eliminates the real-world extensions to obtain a pure template regex. That is, it eliminates backreferences by approximation described above, then eliminates lookarounds by simply replacing them with the empty string ϵ, and eliminates capturing groups by replacing them with expressions with delimiters. We describe the elimination of capturing groups. Since nested structures of capturing groups also form a DAG, we replace capturing groups in a topological order starting from the capturing group that does not have a capturing group in the expression. Specifically, let t be a template that does not have a capturing group as a subexpression. We replace a capturing group $(t)_i$ that is (resp. not) in the Kleene-star operator with the expression $(\#_i t \#_i | t)$ (resp. $\#_i t \#_i$). We replace capturing groups that are in the Kleene-star operator with the form of the union operator because the capturing groups only extract a substring at the last iteration. For example, we convert the template $(a)_1^*$ to the template $(\#_1 a \#_1 | a)^*$. We perform this procedure repeatedly until all capturing groups are eliminated. Finally, the conversion uses the over-approximation for membership to obtain a pure regex from a pure regex template. As a result, for a template t and positive examples $e \in \mathcal{E}^+$, IsFeasible at Line 4 constructs a pure regex r and an input string w from t and e by the above procedure, respectively, and checks whether $w \in \mathcal{L}(r)$. If there exists an input string w such that $w \notin \mathcal{L}(r)$, then we do not expand the holes.

6 EVALUATION

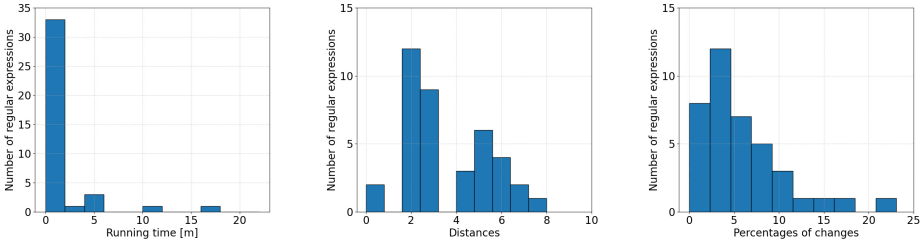
We implemented our algorithm as a tool called R3 in Java with Z3 [De Moura and Bjørner 2008] as the SMT solver. We conducted an experimental evaluation that is designed to answer the following research questions: **(RQ1)** Can R3 repair a regex effectively? **(RQ2)** Can R3 find a high-quality regex? **(RQ3)** Are the new pruning techniques useful for repairing a regex for extraction? To answer **(RQ1)**, we measured the running time of R3 using instances described in Section 6.1. To answer **(RQ2)**, we measured the quality of the repair based on the metrics introduced in the previous works [Chida and Terauchi 2022b; Pan et al. 2019]. Finally, to answer **(RQ3)**, we compared the performance of R3 with and without our pruning techniques. In what follows, we refer to R3 without the new pruning techniques as $\mathrm{R3}_{base}$, and use $\mathrm{R3}_{hybrid}$ to denote the hybrid of R3 and $\mathrm{R3}_{base}$ that returns the regex returned by the faster of the two. The experiments were run on a machine with Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz with the time limit of 30 minutes. In this evaluation, we only consider ASCII symbols, i.e., we only use regexes over the ASCII alphabet.

Validation. Our repair algorithm guarantees to generate only correct regexes, i.e., they are consistent with all examples. We have validated that all regexes generated by R3 in this evaluation are indeed consistent with all examples by running JavaScript's regex engines.

6.1 Data Set

There is no standard benchmark for repairing regexes for extraction. Therefore, to conduct our experiments, we prepared a data set collected from public GitHub projects.

GitHub data set. We collected incorrect and correct regexes for extraction from regex-related commits of public GitHub projects. Specifically, we looked for commits that contain a change that repairs regexes for extraction, and if so, collected the regexes before and after the repair as the incorrect and correct regexes, respectively. When such commits provide examples of correct and incorrect usage, we use them as examples for our repair methods. For commits that do not provide examples, we prepared the examples based on the correct and incorrect regexes manually. The examples were prepared by inspecting the differences between before and after the change of the



(a) Running times. (b) Distances. (c) Percentages of changes.

Fig. 3. Histograms of the results of R3.

regexes in the commits. We prepared at least 10 positive and negative examples, respectively, for each commit. From these examples, we use at most 5 positive and 5 negative examples for each instance to be used for our repair algorithms. Note that, for usability, PBE methods should find a solution from a small number of examples. The remaining examples are used as *left-out example sets* for evaluating the quality of repaired regexes (cf. *RQ2: Quality*).

We took two approaches to find the commits. First, we checked all commits in the list collected by Wang et al. [2020] who studied real-world bugs of regexes in Apache, Mozilla, Facebook, and Google GitHub projects. As a result, we obtained 23 regexes. Second, inspired by the approach of Wang et al. [2020] who searched regex-related commits, we used GitHub Advanced Search with keywords such as "regular expression", "regex", and "regexp" with the keyword "bug". We checked the commits in the order of keyword relevance until we obtained 27 regexes, ignoring duplicates. Consequently, we collected 50 regexes. The average and maximum sizes of the regexes (measured as the number of AST nodes) are 38.8 and 199, respectively. Additionally, the average and maximum numbers of capturing groups are 1.9 and 13, respectively. Finally, 24 (resp. 5) regexes have a (resp. lazy) Kleene star, 32 (resp. 2) regexes have a (resp. lazy) Kleene plus, 9 regexes have a union operator, 17 regexes have an optional operator, 1 regex has a positive lookahead, and 1 regex has a positive lookbehind.

Table 1. Solved instances.

	instances	running time (seconds)		
	Solved (50)	**min**	**med**	**max**
R3	39 (78%)	0.1	1.7	1011.7
R3$_{base}$	36 (72%)	0.1	3.2	536.0
R3$_{hybrid}$	40 (80%)	0.1	1.8	1011.7

RQ1: Efficiency. Table 1 reports the results of the number of solved instances within the time limit (**Solved**) and the minimum, median, and maximum running times in seconds (**min**, **med**, and **max**, respectively). Additionally, we plot the number of regexes against the running times. Figure 3a shows the histogram. The times in the histogram are grouped every 2 minutes. As the table shows, R3 repaired 39 instances (78%) within the time limit. We inspected the unsolved instances, and found that R3 could not repair the instances that (i) require large changes from the original and (ii) involve highly-nested Kleene stars which burden the constraint generation process. These findings agree with the existing works that are based on the enumerative search algorithms with SMT constraint solving [Chida and Terauchi 2022b; Pan et al. 2019]. For example, R3 could not find the repair of the regex $.^*$(mochitest(-debug|-e10s|-devtools-chrome)?|reftest|$\cdots$|jittest)([0-9]$^+$)$_1$ within the time limit. The intended solution is $.^*$-([0-9]$^+$)$_1$\$, which is of a large distance from the original. As the plot shows, majority of the instances (74.3%) can be repaired within 5 seconds. In summary, *R3 can repair a real-world regex efficiently.*

RQ2: Quality. As mentioned by Pan et al. [2019], repairs that are similar to the original ones are often considered good in PBE because they are similar to what the user intended, and the prior PBE works [Chida and Terauchi 2022b; Pan et al. 2019] used the edit distance between from the original as a metric of repair quality. A large change indicates low quality as such repairs may be far from what the user intended. As Figure 3b shows, all repaired regexes were repaired within the distance 10. Figure 3c shows the percentages of changes from the original. As the figure shows, we observe that most repairs are close to the original regexes with the average ratio of change being 5.8%, and about 87.2% of regexes were repaired within 10% of changes.

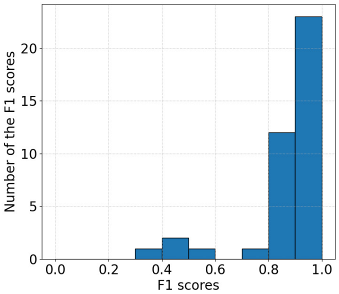


Fig. 4. F1 scores.

Additionally, following the approach used by Pan et al. [2019], we also measured the quality of repaired regexes with respect to left-out example sets. That is, by using the examples not used in the repair algorithms, we measured the F1 score. The F1 score is defined as follows: for the repaired regex r, we estimate the precision of r by $TP/(TP + FP)$ where TP (resp. FP) is the number of positive (resp. negative) examples that are correctly (resp. incorrectly) classified by r. Additionally, we estimate the recall of r by $TP/(TP + FN)$ where FN is the number of positive examples that are incorrectly classified by r. Then, the F1 score of r is defined as $(2 \times \text{recall} \times \text{precision})/(\text{recall} + \text{precision})$. The F1 score is between 0 and 1, and the high F1 score implies the high quality of the repaired regex. Figure 4 shows the histogram. As the figure shows, more than 87.5% of the repaired regexes have the F1 scores of at least 0.8. We inspected the repaired regexes with low F1 scores, and found that R3 could not find a repair with a high F1 score when the repair requires large changes from the original. In summary, *R3 can produce regexes that have high-similarity and generalize well to left-out example sets, and therefore of high-quality.*

RQ3: Impact of Over-Approximation for Extraction. To understand the impact of our new pruning techniques, we compared R3 against R3$_{base}$. Figure 5 plots their running times. The points above the diagonal means that R3 is faster than R3$_{base}$. The points on the border (colored in blue) indicate that the tool could not find a repair within the timeout. As the plot shows, R3 is faster than R3$_{base}$ in 24 instances, and R3 is slower than R3$_{base}$ in 16 instances. However, in 12 instances of the 16 instances, the difference of the running times between R3 and R3$_{base}$ were within 0.2 seconds, whereas such is true for only 2 of the former 24 instances. We inspected the 4 instances on which R3$_{base}$ was non-trivially faster, and found that, in these cases, R3 was suffering from ReDoS (cf. Section 5.3). A possible way to address the issue is to simply set a time limit on the approximation process. Additionally, in 4 instances, R3 solved the instance within the timeout while R3$_{base}$ could not. R3 achieved 21.9× speedups on average and more than 445.9× in the largest case. In summary, *the new pruning techniques significantly improve the running times.*

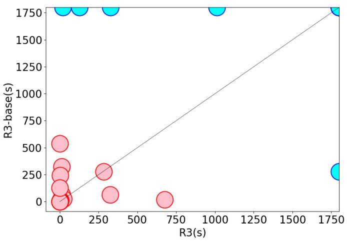


Fig. 5. Effect of the pruning techniques.

7 RELATED WORK

Synthesizing regexes for membership. There is much work on synthesizing or repairing a regex from examples [Angluin 1978; Brāzma 1993; Fernau 2009; Rebele et al. 2018]. Our work is most closely related to the recent works on synthesizing or repairing a regex from examples [Chida and

Terauchi 2022b; Lee et al. 2016; Li et al. 2020; Pan et al. 2019] or from examples and natural language descriptions [Chen et al. 2020; Li et al. 2021]. However, all these works only support for membership, and the support for extraction is out of scope. As shown in our paper, supporting extraction requires considering deterministic semantics which incurs non-trivial extensions to the prior methods that only considered membership. Additionally, all except for Chida and Terauchi [2022b] only support pure regexes and do not consider real-world extensions such as lookarounds and backreference. Chida and Terauchi [2022b] support backreferences and some forms of lookarounds but not general lookbehinds, and as remarked above, they do not consider extraction and use a non-deterministic semantics.

Generating regexes by genetic algorithms. Bartoli et al. [2014, 2016] introduced a genetic-programming based algorithm for generating a regex for extraction from examples. However, as mentioned in Section 1, they do not guarantee the correctness of the repair. Additionally, the extraction considered in their work is different from ours. That is, our work considers extraction from matching a regex against the whole given string and extracting the (positions of) substrings captured by the capturing groups in the regex, whereas their work considers extracting the substrings of the given string that each matches a regex without capturing groups.

Formal semantics of regexes. Our paper's novel formal semantics of regexes follows the ECMAScript 2023 language specification, and while its purpose in this paper is to be used as a basis for a repair algorithm, it may be of independent interest. Our semantics is inspired by those in the previous works [Chida and Terauchi 2022b; Sakuma et al. 2012]. Namely, Chida and Terauchi [2022b] give a big-step semantics for regexes including some real-world extensions, and Sakuma et al. [2012] give a deterministic semantics of Perl regexes using continuations and monads. However, as remarked above, the semantics of Chida and Terauchi [2022b] is non-deterministic and hence is unsuitable to describe extraction, and that of Sakuma et al. [2012] only considers pure regexes and does not consider real-world extensions. Other prior works on formal semantics of regexes include the work by Loring et al. [2019] who provide a semantics that follows the ECMAScript 2015 language specification. However, their goal is dynamic symbolic execution that involves counterexample-guided refinement, and adopting their semantics to a PBE repair problem may be difficult. Furthermore, they do not support lookbehinds. Finally, Chen et al. [2022] give a semantics of JavaScript's regexes by using prioritized streaming string transducers. However, they do not consider extensions such as backreferences and lookarounds.

8 CONCLUSION

We have presented the *first* PBE-based method for repairing regexes for extraction. Our method supports real-world extensions such as general lookarounds and backreferences, and we have shown that the extensions are important for the existence of a solution to the repair problem, and that the repair problem is NP-hard. Our algorithm for solving the repair problem makes two significant extensions to the prior methods that only considered membership: handling of deterministic behavior and the new pruning techniques to reduce the search space. To realize the former, we have also presented a novel formal semantics that the ECMAScript 2023 language specification. We have implemented the algorithm as a tool called R3 and experimentally evaluated it on a real-world data set. The evaluation has shown that R3 can repair real-world regexes successfully and efficiently.

ACKNOWLEDGMENTS

We thank anonymous reviewers for useful comments. This work was supported by JSPS KAKENHI Grant Numbers JP17H01720, JP18K19787, JP20H04162, JP20K20625, and JP22H03570.

REFERENCES

Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337–350. https://doi.org/10.1016/S0019-9958(78)90683-6

Angular. 2022. Angular: The modern web developer's platform. https://angular.io/ [Online; accessed 10-November-2022].

A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio. 2014. Automatic Synthesis of Regular Expressions from Examples. *Computer* 47, 12 (dec 2014), 72–80. https://doi.org/10.1109/MC.2014.344

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1217–1230. https://doi.org/10.1109/TKDE.2016.2515587

Martin Berglund and Brink van der Merwe. 2017. Regular Expressions with Backreferences Re-examined. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, Jan Holub and Jan Zdárek (Eds.). Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 30–41. http://www.stringology.org/event/2017/p04.html

Alvis Brādzma. 1993. Efficient Identification of Regular Expressions from Representative Examples. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory* (Santa Cruz, California, USA) *(COLT '93)*. Association for Computing Machinery, New York, NY, USA, 236–242. https://doi.org/10.1145/168304.168340

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving String Constraints with Regex-Dependent Functions through Transducers with Priorities and Variables. *Proc. ACM Program. Lang.* 6, POPL, Article 45 (jan 2022), 31 pages. https://doi.org/10.1145/3498707

Nariyoshi Chida and Tachio Terauchi. 2022a. On Lookaheads in Regular Expressions with Backreferences. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:18. https://doi.org/10.4230/LIPIcs.FSCD.2022.15

Nariyoshi Chida and Tachio Terauchi. 2022b. Repairing DoS Vulnerability of Real-World Regexes. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2060–2077. https://doi.org/10.1109/SP46214.2022.9833597

Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). https://swtch.com/~rsc/regexp/regexp1.html [Online; accessed 10-November-2022].

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/3338906.3338909

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Django. 2022. Django: The Web framework for perfectionists with deadlines. https://www.djangoproject.com/ [Online; accessed 10-November-2022].

ECMA International. 2022. ECMAScript® 2023 Language Specification. https://tc39.es/ecma262/multipage/#sec-intro.

Henning Fernau. 2009. Algorithms for learning regular expressions from positive data. *Information and Computation* 207, 4 (2009), 521–541. https://doi.org/10.1016/j.ic.2008.12.008

Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. 2021. FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 152–169.

Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (3 ed.). O'Reilly, Beijing. https://www.safaribooksonline.com/library/view/mastering-regular-expressions/0596528124/

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 618–629.

Google. [n. d.]. RE2. https://github.com/google/re2 [Online; accessed 10-November-2022].

Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. *SIGPLAN Not.* 52, 3 (oct 2016), 70–80. https://doi.org/10.1145/3093335.2993244

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Honolulu, Hawaii, 21–30. https://aclanthology.org/D08-1003

Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2021. TransRegex: Multi-Modal Regular Expression Synthesis by Generate-and-Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1210–1222. https://doi.org/10.1109/ICSE43902.2021.00111

Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2020. FlashRegex: Deducing Anti-ReDoS Regexes from Examples. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 659–671. https://doi.org/10.1145/3324884.3416556

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 425–438. https://doi.org/10.1145/3314221.3314645

Matthew Luckie, Bradley Huffaker, and k claffy. 2019. Learning Regexes to Extract Router Names from Hostnames. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/3355369.3355589

Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 415–426. https://doi.org/10.1109/ASE.2019.00047

Chris O'Hara. 2022. Validator.js. https://github.com/validatorjs/validator.js/ [Online; accessed 10-November-2022].

OWASP. 2022. Input Validation Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html [Online; accessed 10-November-2022].

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (oct 2019), 29 pages. https://doi.org/10.1145/3360565

Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining*, Dinh Phung, Vincent S. Tseng, Geoffrey I. Webb, Bao Ho, Mohadeseh Ganji, and Lida Rashidi (Eds.). Springer International Publishing, Cham, 67–79.

RegExLib. 2022. https://regexlib.com/ [Online; accessed 10-November-2022].

Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. 2012. Translating regular expression matching into transducers. *Journal of Applied Logic* 10, 1 (2012), 32–51. https://doi.org/10.1016/j.jal.2011.11.003 Special issue on Automated Specification and Verification of Web Systems.

Amazon Web Services. 2022. Regex match rule statement. https://docs.aws.amazon.com/waf/latest/developerguide/waf-rule-statement-type-regex-match.html [Online; accessed 10-November-2022].

Snort. 2022. Snort. https://www.snort.org/ [Online; accessed 10-November-2022].

Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2020. An Empirical Study on Regular Expression Bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 103–113. https://doi.org/10.1145/3379597.3387464

Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

A PROOF OF THEOREM 4.3

Proof. For simplicity, we assume that $\Sigma = \{a, b\}$. We show that the instance $r_1 = \epsilon$, $\mathcal{E}^+ = \{(ab, \Gamma_0), (ba, \Gamma_1)\}$ where $\Gamma_0 = \{(1, (0, 1))\}$ (i.e., the first character a should be extracted from ab) and $\Gamma_1 = \{(1, (1, 2))\}$ (i.e., the second character a should be extracted from ba), and $\mathcal{E}^- = \{\epsilon, a, b, aa, bb\}$ (i.e., except for ab and ba, the length of strings accepted by the solution should be greater than or equal to 3) does not have a pure regex solution. Suppose for a contradiction that the instance

Set of Characters and Empty String

$$\frac{d = \text{forward} \qquad p < |w| \qquad w[p] \in C}{([C], r_c, w, p, \Gamma, d, l) \Downarrow (p+1, \Gamma)} \qquad \frac{d = \text{backward} \qquad 0 \leq p-1 \qquad w[p-1] \in C}{([C], r_c, w, p, \Gamma, d, l) \Downarrow (p-1, \Gamma)}$$

$$(\epsilon, r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)$$

Concatenation

$$\frac{d = \text{forward} \qquad (r_1, r_2 r_c, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \qquad (r_2, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2)}{(r_1 r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{d = \text{backward} \qquad (r_2, r_c r_1, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \qquad (r_1, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2)}{(r_1 r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

Union

$$\frac{d = \text{forward} \qquad (r_1 r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \qquad (\neg l \land p_1 = |w|) \lor l \qquad (r_1, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{\begin{array}{c} d = \text{forward} \qquad (r_1 r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \\ (\neg l \land (\tau = \text{failed} \lor \tau = (p'', \Gamma'') \text{ where } p'' \neq |w|) \lor (l \land \tau = \text{failed}) \\ (r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma') \end{array}}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

$$\frac{d = \text{backward} \qquad (r_c r_1, \epsilon, w, p, \Gamma, d, l) \Downarrow (p_1, \Gamma_1) \qquad (\neg l \land p_1 = |w|) \lor l \qquad (r_1, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (r_c r_1, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \\ (\neg l \land (\tau = \text{failed} \lor \tau = (p'', \Gamma'') \text{ where } p'' \neq |w|) \lor (l \land \tau = \text{failed}) \\ (r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma') \end{array}}{(r_1 | r_2, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

Fig. 6. Rules for pure regexes (except for Kleene stars and repetitions).

has a pure regex solution. Let r_2 be the solution. Since r_2 is consistent with positive examples, it has a capturing group $(r_a)_1$ where r_a matches to the character a, i.e., $a \in \mathcal{L}(r_a)$. The capturing group $(r_a)_1$ does not appear in the Kleene-star operator. To prove this, suppose the capturing group appears in the Kleene-star operator. Then, the Kleene-star operator can iterate exactly once because if it iterates zero times, then the solution r_2 does not extract any substring, which is a contradiction. In addition, if it iterates more than two times, for the string ab, the substring extracted by the capturing group is no longer the first character a, which is a contradiction, because the Kleene-star operator consumes at least one character for each iteration. Therefore, the Kleene-star operator iterates exactly once. However, in this case, we can obtain a regex r_3 that is consistent with examples and $D(r_1, r_3) < D(r_1, r_2)$ by replacing the Kleene-star operator r'^* with the expression r' which is a contradiction. Additionally, the capturing group does not appear in the union operator for the similar reason, i.e., for the union operator, we can obtain the smaller expression that is consistent with examples by replacing the union operator with the subexpression that contains the capturing group. From this, the only operator in which the capturing group appears is the concatenation operator. As a result, r_2 should take the form of $r'(r_a)_1 r''$. Since r_2 is consistent with positive examples, r' and r'' can match to the character b and the empty string ϵ, i.e., $\epsilon, b \in \mathcal{L}(r')$ and $\epsilon, b \in \mathcal{L}(r'')$. However, if so, r_2 can match the string $a \in \mathcal{E}^-$, which is a contradiction. □

Greedy Kleene Star

$$\frac{\begin{array}{c} d = \text{forward} \qquad (r\langle r^* : p\rangle r_c, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \qquad (\neg l \land p_1 = |w|) \lor l \\ (r, \langle r^* : p\rangle r_c, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_2, \Gamma_2) \qquad (r^*, r_c, w, p_2, \Gamma_2, d, l) \Downarrow (p_3, \Gamma_3) \end{array}}{(r^*, r_c, w, p, \Gamma, d, l) \Downarrow (p_3, \Gamma_3)}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (r_c\langle r^* : p\rangle r, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \qquad (\neg l \land p_1 = |w|) \lor l \\ (r, r_c\langle r^* : p\rangle, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_2, \Gamma_2) \qquad (r^*, r_c, w, p_2, \Gamma_2, d, l) \Downarrow (p_3, \Gamma_3) \end{array}}{(r^*, r_c, w, p, \Gamma, d, l) \Downarrow (p_3, \Gamma_3)}$$

Lazy Kleene Star

$$\frac{\begin{array}{c} d = \text{forward} \qquad (r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \qquad (\neg l \land (\tau = \text{failed} \lor \tau = (p', \Gamma') \text{ where } p' \neq |w|)) \lor (l \land \tau = \text{failed}) \\ (r\langle r^{*?} : p\rangle r_c, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_3, \Gamma_3) \qquad (\neg l \land p_3 = |w|) \lor l \\ (r, \langle r^{*?} : p\rangle r_c, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \qquad (r^{*?}, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2) \end{array}}{(r^{*?}, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (r_c, \epsilon, w, p, \Gamma, d, l) \Downarrow \tau \qquad (\neg l \land (\tau = \text{failed} \lor \tau = (p', \Gamma') \text{ where } p' \neq |w|) \lor (l \land \tau = \text{failed}) \\ (r_c\langle r^{*?} : p\rangle r, \epsilon, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_3, \Gamma_3) \qquad (\neg l \land p_3 = |w|) \lor l \\ (r, r_c\langle r^{*?} : p\rangle, w, p, \mathit{reset}(r, \Gamma), d, l) \Downarrow (p_1, \Gamma_1) \qquad (r^{*?}, r_c, w, p_1, \Gamma_1, d, l) \Downarrow (p_2, \Gamma_2) \end{array}}{(r^{*?}, r_c, w, p, \Gamma, d, l) \Downarrow (p_2, \Gamma_2)}$$

Guard

$$\frac{p' = p}{(\langle r : p'\rangle, r_c, w, p, \Gamma, d, l) \Downarrow \text{failed}} \qquad \frac{p' \neq p \qquad (r, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}{(\langle r : p'\rangle, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')}$$

Fig. 7. Rules for Kleene stars.

B PROOF OF THEOREM 4.4

Proof. Let k be the maximum number of indexes of capturing groups in positive examples. We assume that $k \geq 1$. Note that if $k = 0$, then it is immediate since we can obtain the solution by taking the unions of strings in positive examples. We show that we can always construct a regex that satisfies the consistency condition. Let a regex r, a set of positive examples $\mathcal{E}^+ = \{(w_1, \Gamma_1), (w_2, \Gamma_2), \cdots, (w_n, \Gamma_n)\}$, and a set of negative examples $\mathcal{E}^-$ be an input instance of the extraction-regex-repair problem.

Then, we prepare a regex $r' = r_1 r_2 \cdots r_k r_{body}$ where

- for all $i \in [k]$, $r_i = (?{=}(r_{i,\mathit{defined}}|r_{i,\mathit{undefined}}))$,
- $r_{i,\mathit{defined}} = .^*(r_{i,\mathit{defined},1}|r_{i,\mathit{defined},2}|\cdots|r_{i,\mathit{defined},n})_i$,
- for $j \in [n]$, $r_{i,\mathit{defined},j} = \begin{cases} (?{<=}(?{<!}.)w_j[0..p_l))w_j[p_l..p_r)(?{=}w_j[p_r..|w_j|)(?!.))) & \text{if } \Gamma_j(i) = (p_l, p_r) \\ r_{i,\mathit{defined},j} = \emptyset & \text{otherwise} \end{cases}$,
- $r_{i,\mathit{undefined}} = r_{i,\mathit{undefined},1}|r_{i,\mathit{undefined},2}|\cdots|r_{i,\mathit{undefined},n}$,
- for $j \in [n]$, $r_{i,\mathit{undefined},j} = \begin{cases} \emptyset & \text{if } \Gamma_j(i) = (p_l, p_r) \\ (?{<!}.)w_j(?!.) & \text{otherwise} \end{cases}$, and
- $r_{body} = (w_1|w_2|\cdots|w_n)$.

We now show that the regex r' satisfies the consistency condition by showing $\mathcal{L}_c(r') = \mathcal{E}^+$. We first show that for every $(w_j, \Gamma_j) \in \mathcal{E}^+$, $(w_j, \Gamma_j) \in \mathcal{L}_c(r')$. To prove this, we show that $w_j \in \mathcal{L}(r')$, and then, we show that r' extracts correct substrings. Since r' consists of the positive lookaheads r_i and the expression r_{body} whose language is $\{w \mid (w, \_) \in \mathcal{E}^+\}$, r' accepts w_j if all the positive lookaheads r_i succeeds. For each r_i, it succeeds on w_j and correctly extracts the substrings because

Capturing Group

$$\frac{\begin{array}{c} d = \text{forward} \\ (r\$_i, r_c, w, p, \Gamma[i \mapsto (p, \bot)], d, l) \Downarrow (p', \Gamma') \end{array}}{((r)_i, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')} \qquad \frac{d = \text{forward} \qquad \Gamma(i) = (p, \bot)}{(\$_i, r_c, w, p', \Gamma, d, l) \Downarrow (p, \Gamma[i \mapsto (p, p')])}$$

$$\frac{\begin{array}{c} d = \text{backward} \\ (\$_i r, r_c, w, p, \Gamma[i \mapsto (\bot, p)], d, l) \Downarrow (p', \Gamma') \end{array}}{((r)_i, r_c, w, p, \Gamma, d, l) \Downarrow (p', \Gamma')} \qquad \frac{d = \text{backward} \qquad \Gamma(i) = (p', \bot)}{(\$_i, r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma[i \mapsto (p, p')])}$$

Backreference

$$\frac{\Gamma(i) = (p', p'') \qquad (w[p'..p''), r_c, w, p, \Gamma, d, l) \Downarrow (p''', \Gamma')}{(\backslash i, r_c, w, p, \Gamma, d, l) \Downarrow (p''', \Gamma')} \qquad \frac{i \notin dom(\Gamma) \lor \Gamma(i) = (p', \bot)}{(\backslash i, r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)}$$

Lookahead

$$\frac{(r, \epsilon, w, p, \Gamma, \text{forward}, \text{true}) \Downarrow (p', \Gamma')}{((?{=}r), r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma')} \qquad \frac{(r, \epsilon, w, p, \Gamma, \text{forward}, \text{true}) \Downarrow \text{failed}}{((?!r), r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)}$$

Lookbehind

$$\frac{(r, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \Downarrow (p', \Gamma')}{((?{<}{=}r), r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma')} \qquad \frac{(r, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \Downarrow \text{failed}}{((?{<}!r), r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)}$$

Fig. 8. Rules for real-world extensions.

(1) if $\Gamma_j(i) = (p_l, p_r)$, then $r_{i,defined,j} = (?{<}{=}(?{<}!.)w_j[0..p_l))w_j[p_l..p_r)(?{=}w_j[p_r..|w_j|)(?!.))$ by the construction. The subexpression $w_j[p_l..p_r)$ exactly matches the string $w_j[p_l..p_r)$. The positive lookbehind $(?{<}{=}(?{<}!.)w_j[0..p_l))$ asserts that the string from the beginning to the current position, i.e., $p_l - 1$, should be exactly the string $w_j[0..p_l)$ and the positive lookahead $(?{=}w_j[p_r..|w_j|)(?!.))$ asserts that the string from the current position, i.e., p_r, to the end of the string should be exactly the string $w_j[p_r..|w_j|)$. Note that the expressions $(?{<}!.)$ and $(?!.)$ match the beginning and the end of strings, respectively, i.e., we can see them as the syntactic sugars of the *anchor* operators ^ and \$ [Friedl 2006], respectively. As a result, the expression $r_{i,defined,j}$ exactly matches w_j on the correct position. Additionally, since lookarounds do not consume any character, the ith capturing group only extracts $w_j[p_l..p_r)$.

(2) If $i \notin dom(\Gamma_j)$, then $r_{i,defined,j} = \emptyset$ that does not match any word and $r_{i,undefined,j} = (?{<}!.)w_j(?!.)$ that exactly matches the string w_j. Since $r_{i,defined}$ fails due to $r_{i,defined,j}$, r_j does not extract any word, whereas the matching of r_j on w_j succeeds due to $r_{i,undefined,j}$.

Next, we show that for every $(w_j, \Gamma_j) \notin \mathcal{E}^+$, $(w_j, \Gamma_j) \notin \mathcal{L}_c(r')$. The matching of r' on w_j fails regardless of the positive lookaheads r_i because the only subexpression that consumes characters is r_{body}, and it exactly matches the strings in $\{w \mid (w, \_) \in \mathcal{E}^+\}$ by the construction. Hence, $\mathcal{L}_c(r') = \mathcal{E}^+$. This satisfies the consistency condition, and implies the existence of the solution.

□

C PROOF OF THEOREM 4.5

We first review SetCover.

Definition C.1. Given a finite set U, $S \subseteq \mathcal{P}(U)$, where $\mathcal{P}(U)$ denotes the powerset of a set U, and a positive integer k, SetCover is the problem of deciding whether there exists a subset $T = \{T_1, T_2, \cdots, T_n\} \subseteq S$ such that $n \leq k$ and $\bigcup T = U$.

Set of Characters and Empty String

$$\frac{d = \text{forward} \qquad p < |w| \qquad w[p] \in C}{([C], t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p+1, \Gamma, \text{true})\}, \emptyset)} \qquad \frac{d = \text{backward} \qquad 0 \leq p - 1 \qquad w[p-1] \in C}{([C], t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p-1, \Gamma, \text{true})\}, \emptyset)}$$

$$(\epsilon, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \text{true})\}, \emptyset)$$

Hole

$$\frac{d = \text{forward} \qquad \square \text{ is the } i\text{th hole.} \qquad p < |w|}{(\square, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p+1, \Gamma, v_i^{w[p]})\}, \{(\bot, \bot, \neg v_i^{w[p]})\})}$$

$$\frac{d = \text{backward} \qquad \square \text{ is the } i\text{th hole.} \qquad 0 \leq p - 1}{(\square, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p-1, \Gamma, v_i^{w[p-1]})\}, \{(\bot, \bot, \neg v_i^{w[p-1]})\})}$$

Concatenation

$$\frac{d = \text{forward} \qquad (t_1, t_2 t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F) \qquad \forall (p_i, \Gamma_i, \phi_i) \in S.\ (t_2, t_c, w, p_i, \Gamma_i, d, l) \dashrightarrow (S_i, F_i)}{\begin{array}{c}(t_1 t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (\bigcup_{0 \leq i < |S|} \{(p', \Gamma', \phi_i \wedge \phi') \mid (p', \Gamma', \phi') \in S_i\}, \\ F \cup \bigcup_{0 \leq i < |S|} \{(\bot, \bot, \phi_i \wedge \phi') \mid (\bot, \bot, \phi') \in F_i\})\end{array}}$$

$$\frac{d = \text{backward} \qquad (t_2, t_c t_1, w, p, \Gamma, d, l) \dashrightarrow (S, F) \qquad \forall (p_i, \Gamma_i, \phi_i) \in S.\ (t_1, t_c, w, p_i, \Gamma_i, d, l) \dashrightarrow (S_i, F_i)}{\begin{array}{c}(t_1 t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (\bigcup_{0 \leq i < |S|} \{(p', \Gamma', \phi_i \wedge \phi') \mid (p', \Gamma', \phi') \in S_i\}, \\ F \cup \bigcup_{0 \leq i < |S|} \{(\bot, \bot, \phi_i \wedge \phi') \mid (\bot, \bot, \phi') \in F_i\})\end{array}}$$

Union

$$\frac{\begin{array}{c} d = \text{forward} \qquad (t_1 t_c, \epsilon, w, p, \Gamma, d, l) \dashrightarrow (S_1, F_1) \\ S_1' = \mathit{ite}(l, S_1, \{(p', \Gamma', \phi') \in S_1 \mid p' = |w|\}) \qquad F_1' = F_1 \cup \mathit{ite}(l, \emptyset, \{(\bot, \bot, \phi') \mid (p', \_, \phi') \in S_1, p' \neq |w|\}) \\ \phi_{S_1} = \bigvee_{(\_,\_,\phi') \in S_1'} \phi' \qquad \phi_{F_1} = \bigvee_{(\bot,\bot,\phi') \in F_1'} \phi' \\ (t_1, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_2, F_2) \qquad (t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_3, F_3) \end{array}}{\begin{array}{c}(t_1 | t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p', \Gamma', \phi_{S_1} \wedge \phi') \mid (p', \Gamma', \phi') \in S_2\} \cup \{(p', \Gamma', \phi_{F_1} \wedge \phi') \mid (p', \Gamma', \phi') \in S_3\}, \\ \{(\bot, \bot, \phi_{F_1} \wedge \phi') \mid (\bot, \bot, \phi') \in F_3\})\end{array}}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (t_c t_1, \epsilon, w, p, \Gamma, d, l) \dashrightarrow (S_1, F_1) \\ S_1' = \mathit{ite}(l, S_1, \{(p', \Gamma', \phi') \in S_1 \mid p' = |w|\}) \qquad F_1' = F_1 \cup \mathit{ite}(l, \emptyset, \{(\bot, \bot, \phi') \mid (p', \_, \phi') \in S_1, p' \neq |w|\}) \\ \phi_{S_1} = \bigvee_{(\_,\_,\phi') \in S_1'} \phi' \qquad \phi_{F_1} = \bigvee_{(\bot,\bot,\phi') \in F_1'} \phi' \\ (t_1, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_2, F_2) \qquad (t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (S_3, F_3) \end{array}}{\begin{array}{c}(t_1 | t_2, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p', \Gamma', \phi_{S_1} \wedge \phi') \mid (p', \Gamma', \phi') \in S_2\} \cup \{(p', \Gamma', \phi_{F_1} \wedge \phi') \mid (p', \Gamma', \phi') \in S_3\}, \\ \{(\bot, \bot, \phi_{F_1} \wedge \phi') \mid (\bot, \bot, \phi') \in F_3\})\end{array}}$$

Fig. 9. Rules for pure regexes (except for Kleene stars) and holes.

Proof. We give a reduction from SetCover to the repair problem. For this, we create (the decision version of) the repair problem.

- The alphabet $\Sigma = U$;
- The set of positive examples $\mathcal{E}^+ = \{ \llparenthesis a \rrparenthesis_1 \mid a \in \Sigma \}$;
- The set of negative examples $\mathcal{E}^- = \emptyset$;
- The distance bound is $2k$; and
- The pre-repair expression $r = (r_1 | r_2 | \cdots | r_n)_1$ where $r_i = (?{=}[T_i])^k \emptyset [T_i]$ for $i \in [n]$.

Here, r^k is the expression obtained by concatenating r k times.

It is easy to see that this is a polynomial reduction and is a valid instance of the repair problem. We show that the reduction is correct, i.e., the instance of SetCover has a solution iff the instance

Greedy Kleene Star

$$\frac{\begin{array}{c} d = \text{forward} \qquad (t\langle t^* : p\rangle t_c, \epsilon, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S,F) \qquad S_a = \mathit{ite}(l, S, \{(p',\Gamma',\phi') \in S \mid p' = |w|\}) \\ \phi_S = \bigvee_{(\_,\_,\phi')\in S_a} \phi' \qquad \phi_F = \bigvee_{(\_,\_,\phi')\in F\cup(S\setminus S_a)} \phi' \qquad (t, \langle t^* : p\rangle t_c, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S',F') \\ S'_{SAT} = \{(p',\Gamma',\phi') \in S' | \phi' \wedge \phi_S \text{ is satisfiable.}\} \qquad \phi_{F'} = \bigvee_{(\bot,\bot,\phi')\in F'} \phi' \\ \forall (p_i,\Gamma_i,\phi_i) \in S'_{SAT}.\ (t^*, t_c, w, p_i, \Gamma_i, d, l) \dashrightarrow (S_i, F_i) \end{array}}{(t^*, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi_F \vee (\phi_S \wedge \phi_{F'}))\} \cup \bigcup_{0\le i<|S'_{SAT}|} \{(p',\Gamma',\phi_S \wedge \phi_i \wedge \phi') \mid (p',\Gamma',\phi') \in S_i\}, \emptyset)}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (t_c\langle t^* : p\rangle t, \epsilon, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S,F) \qquad S_a = \mathit{ite}(l, S, \{(p',\Gamma',\phi') \in S \mid p' = |w|\}) \\ \phi_S = \bigvee_{(\_,\_,\phi')\in S_a} \phi' \qquad \phi_F = \bigvee_{(\_,\_,\phi')\in F\cup(S\setminus S_a)} \phi' \qquad (t, t_c\langle t^* : p\rangle, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S',F') \\ S'_{SAT} = \{(p',\Gamma',\phi') \in S' \mid \phi' \wedge \phi_S \text{ is satisfiable.}\} \qquad \phi_{F'} = \bigvee_{(\bot,\bot,\phi')\in F'} \phi' \\ \forall (p_i,\Gamma_i,\phi_i) \in S'_{SAT}.\ (t^*, t_c, w, p_i, \Gamma_i, d, l) \dashrightarrow (S_i, F_i) \end{array}}{(t^*, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi_F \vee (\phi_S \wedge \phi_{F'}))\} \cup \bigcup_{0\le i<|S'_{SAT}|} \{(p',\Gamma',\phi_S \wedge \phi_i \wedge \phi') \mid (p',\Gamma',\phi') \in S_i\}, \emptyset)}$$

Lazy Kleene Star

$$\frac{\begin{array}{c} d = \text{forward} \qquad (t_c, \epsilon, w, p, \Gamma, d, l) \dashrightarrow (S,F) \qquad S_a = \mathit{ite}(l, S, \{(p',\Gamma',\phi') \in S \mid p' = |w|\}) \\ \phi_S = \bigvee_{(\_,\_,\phi')\in S_a} \phi' \qquad \phi_F = \bigvee_{(\_,\_,\phi')\in F\cup(S\setminus S_a)} \phi' \qquad (t\langle t^{*?} : p\rangle t_c, \epsilon, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S',F') \\ S'_a = \mathit{ite}(l, S', \{(p',\Gamma',\phi') \in S' \mid p' = |w|\}) \qquad \phi_{S'} = \bigvee_{(\_,\_,\phi')\in S'_a} \phi' \qquad \phi_{F'} = \bigvee_{(\_,\_,\phi')\in F'\cup(S'\setminus S'_a)} \phi' \\ (t, \langle t^{*?} : p\rangle t_c, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S'',F'') \qquad S''_{SAT} = \{(p',\Gamma',\phi') \in S'' \mid \phi' \wedge \phi_F \wedge \phi_{S'} \text{ is satisfiable.}\} \\ \phi_{F''} = \bigvee_{(\bot,\bot,\phi')\in F''} \phi' \qquad \forall (p_i,\Gamma_i,\phi_i) \in S''_{SAT}.\ (t^{*?}, t_c, w, p_i, \Gamma_i, d, l) \dashrightarrow (S_i, F_i) \end{array}}{\begin{array}{c} (t^{*?}, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi_S \vee (\phi_F \wedge \phi_{F'}) \vee (\phi_F \wedge \phi_{S'} \wedge \phi_{F''}))\} \cup \\ \bigcup_{0\le i<|S''_{SAT}|} \{(p',\Gamma',\phi_F \wedge \phi_{S'} \wedge \phi_i \wedge \phi') \mid (p',\Gamma',\phi') \in S_i\}, \emptyset) \end{array}}$$

$$\frac{\begin{array}{c} d = \text{backward} \qquad (t_c, \epsilon, w, p, \Gamma, d, l) \dashrightarrow (S,F) \qquad S_a = \mathit{ite}(l, S, \{(p',\Gamma',\phi') \in S \mid p' = |w|\}) \\ \phi_S = \bigvee_{(\_,\_,\phi')\in S_a} \phi' \qquad \phi_F = \bigvee_{(\_,\_,\phi')\in F\cup(S\setminus S_a)} \phi' \qquad (t_c\langle t^{*?} : p\rangle t, \epsilon, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S',F') \\ S'_a = \mathit{ite}(l, S', \{(p',\Gamma',\phi') \in S' \mid p' = |w|\}) \qquad \phi_{S'} = \bigvee_{(\_,\_,\phi')\in S'_a} \phi' \qquad \phi_{F'} = \bigvee_{(\_,\_,\phi')\in F'\cup(S'\setminus S'_a)} \phi' \\ (t, t_c\langle t^{*?} : p\rangle, w, p, \mathit{reset}(t,\Gamma), d, l) \dashrightarrow (S'',F'') \qquad S''_{SAT} = \{(p',\Gamma',\phi') \in S'' \mid \phi' \wedge \phi_F \wedge \phi_{S'} \text{ is satisfiable.}\} \\ \phi_{F''} = \bigvee_{(\bot,\bot,\phi')\in F''} \phi' \qquad \forall (p_i,\Gamma_i,\phi_i) \in S''_{SAT}.\ (t^{*?}, t_c, w, p_1, \Gamma_1, d, l) \dashrightarrow (S_i, F_i) \end{array}}{\begin{array}{c} (t^{*?}, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi_S \vee (\phi_F \wedge \phi_{F'}) \vee (\phi_F \wedge \phi_{S'} \wedge \phi_{F''}))\} \cup \\ \bigcup_{0\le i<|S''_{SAT}|} \{(p'',\Gamma'',\phi_S \wedge \phi_{S'} \wedge \phi_i \wedge \phi'') \mid (p'',\Gamma'',\phi'') \in S_i\}, \emptyset) \end{array}}$$

Guards

$$\frac{p' = p}{(\langle t : p'\rangle, t_c, w, p, \Gamma, d, l) \dashrightarrow (\emptyset, \{(p, \Gamma, \text{true})\})} \qquad \frac{p' \neq p \qquad (t, t_c, w, p, \Gamma, d, l) \dashrightarrow (S,F)}{(\langle t : p'\rangle, t_c, w, p, \Gamma, d, l) \dashrightarrow (S,F)}$$

Fig. 10. Rules for Kleene stars.

of the repair problem created above has a solution. First, we show the only if direction. Let $T' \subseteq T$ be a solution of the instance of SetCover. Then, the solution of the repair problem is the regex $r' = (r'_1|r'_2|\cdots|r'_n)$ where $r'_i = r_i$ if $T_i \notin T'$ and otherwise $r'_i = ([T_i])^k\epsilon[T_i]$. That is, for all r_i, if $T_i \in T'$, then we replace the empty set $\emptyset$ in r_i with the empty string ϵ and otherwise r_i remains the same. Note that the distance between r and r' is $2|T'| \le 2k$. Additionally, the regex r' is consistent with all the examples. That is, for all $a \in \Sigma$, r' extracts the character a by the 1st capturing group because there exists $T_i \in T'$ such that $a \in T_i$ since T' is a solution of the SetCover instance, and therefore $r'_i = (?{=}[T_i])^k\epsilon[T_i]$ by the construction and $a \in \mathcal{L}(r'_i)$. Also, $\mathcal{E}^-$ is immediate since $\mathcal{E}^- = \emptyset$. Thus, r' is a correct repair.

Next, we show the if direction. For this, we first show that the only meaningful change in the repair is to change the $\emptyset$ in r with ϵ. First, any valid repair of r does not change the capturing group because, to change the capturing group, we need to replace all immediate subexpressions with

Capturing Group

$$\frac{d = \text{forward} \qquad (t\$_i, t_c, w, p, \Gamma[i \mapsto (p, \bot)], d, l) \dashrightarrow (S, F)}{((t)_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)} \qquad \frac{d = \text{forward} \qquad \Gamma(i) = (p', \bot)}{(\$_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma[i \mapsto (p', p)], \text{true})\}, \emptyset)}$$

$$\frac{d = \text{backward} \qquad (\$_i t, t_c, w, p, \Gamma[i \mapsto (p, \bot)], d, l) \dashrightarrow (S, F)}{((t)_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)} \qquad \frac{d = \text{backward} \qquad \Gamma(i) = (p', \bot)}{(\$_i, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma[i \mapsto (p, p')], \text{true})\}, \emptyset)}$$

Backreference

$$\frac{\Gamma(i) = (p', p'') \quad (w[p'..p''), t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)}{(\backslash i, t_c, w, p, \Gamma, d, l) \dashrightarrow (S, F)} \qquad \frac{i \notin dom(\Gamma) \vee \Gamma(i) = (p', \bot)}{(\backslash i, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \text{true})\}, \emptyset)}$$

Lookaheads

$$\frac{(t, \epsilon, w, p, \Gamma, \text{forward}, \text{true}) \dashrightarrow (S, F)}{((?{=}t), t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma', \phi') \mid (\_, \Gamma', \phi') \in S\}, F)}$$

$$\frac{(t, \epsilon, w, p, \Gamma, \text{forward}, \text{true}) \dashrightarrow (S, F)}{((?!t), t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi') \mid (\bot, \bot, \phi') \in F\}, \{(\bot, \bot, \phi') \mid (\_, \_, \phi') \in S\})}$$

Lookbehinds

$$\frac{(t, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \dashrightarrow (S, F)}{((?{<=}t), t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma', \phi') \mid (\_, \Gamma', \phi') \in S\}, F)}$$

$$\frac{(t, \epsilon, w, p, \Gamma, \text{backward}, \text{true}) \dashrightarrow (S, F)}{((?{<!}t), t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \phi') \mid (\bot, \bot, \phi') \in F\}, \{(\bot, \bot, \phi') \mid (\_, \_, \phi') \in S\})}$$

Fig. 11. Rules for real-world extensions.

holes and it immediately violates the distance bound $2k$ since r has $(?{=}r_i)^k$. For the same reason, any valid repair of r must preserve the n union choices, and for each r_i, it is useless to change $(?{=}[T_i])$. Additionally, it is also useless to change $[T_i]$ to some expression whose language contains elements not in T_i due to the k many $(?{=}[T_i])$ preceding it. Note that the changing $(?{=}[T_i])^k$ would exceed the distance bound. Nor, can $[T_i]$ be changed to some expression whose language does not contain elements in T_i because we do not need to exclude any character in Σ. As a result, it is easy to see that the only meaningful change is to change $\emptyset$ with ϵ. Therefore, the solution of the repair problem is of the form $r' = (r'_1|r'_2|\cdots|r'_n)$ obtained from r by replacing some $\emptyset$ with ϵ. From the solutions of the repair problem, we can construct the set $T' = \{T_i \mid r_i \neq r'_i\}$. Then, the set T' is a solution of the instance of SetCover. This is because, since the distance bound is $2k$, the repair changes at most k empty sets. From this, there exists at most k r_i such that $r_i \neq r'_i$ for $i \in [n]$, and therefore $|T'| \leq k$ by the construction. Additionally, for all $a \in \Sigma$, there exists T_i such that $a \in T_i$ by the construction because r' is the solution of the repair problem, and therefore there exists r'_i such that $a \in \mathcal{L}(r_i)$ and $r_i \neq r'_i$. Thus, T' is a correct solution. □

D APPROXIMATION FOR MEMBERSHIP

Given a template t, we construct the over- and under-approximation for membership by the following procedures. First, we eliminate backreferences by approximating them. We use the same procedure to eliminate backreferences describes in Section 5.3.2.

Next, we eliminate holes by approximating them. For this, we use the function α which is defined as follows. Below, $\overline{r} = \emptyset$ if $r = .^*$ and $\overline{r} = .^*$ if $r = \emptyset$.

$$\begin{array}{rcl@{\qquad}rcl}
\alpha([C], r) &=& [C] & \alpha((t)_i, r) &=& (\alpha(t, r))_i \\
\alpha(\epsilon, r) &=& \epsilon & \alpha((?{=}t), r) &=& (?{=}\alpha(t, r)) \\
\alpha(t_1 t_2, r) &=& \alpha(t_1, r)\alpha(t_2, r) & \alpha((?!t), r) &=& (?!\alpha(t, \overline{r})) \\
\alpha(t_1 | t_2, r) &=& \alpha(t_1, r) | \alpha(t_2, r) & \alpha((?{<=}t), r) &=& (?{<=}\alpha(t, r)) \\
\alpha(t^*, r) &=& \alpha(t, r)^* & \alpha((?{<}!t), r) &=& (?{<}!\alpha(t, \overline{r})) \\
\alpha(t^{*?}, r) &=& \alpha(t, r)^{*?} & \alpha(\square, r) &=& r
\end{array}$$

As a result, we obtain an over- (resp. under-)approximated regex $r_\top$ (resp. $r_\bot$) by $\alpha(t', .^*)$ (resp. $\alpha(t', \emptyset)$), where t' is a template obtained from t by applying the first procedure for eliminating backreferences.

E FULL RULES OF THE SEMANTICS

In this section, we show the full version of the rules for the formal semantics of regexes. Figures 6, 7, and 8 show the rules for pure regexes except for Kleene stars, Kleene stars, and real-world extensions, respectively. For the state $(r, r_c, w, p, \Gamma, d, l)$ that is not applicable for any rule in the figures, we assume that $(r, r_c, w, p, \Gamma, d, l) \Downarrow (p, \Gamma)$ if r is the Kleene star, and otherwise, $(r, r_c, w, p, \Gamma, d, l) \Downarrow$ failed. Below, we describe the rules that are not described in Section 3.

The rule for positive lookaheads $(?{=}r)$ first performs the matching of the expression r with the direction $d =$ forward. If the matching succeeds, then it returns the result after resetting the position from p' to p. The rule for negative lookaheads $(?!r)$ is similar to the rule of positive lookaheads, i.e., it first performs the matching of r. However, unlike positive lookaheads, the matching of negative lookaheads succeeds if the matching of r fails. Additionally, negative lookaheads reset not only the position but also the environment. The rule for negative lookbehinds is similar to the rule of negative lookahead. The only difference is the direction, i.e., negative lookbehinds set the direction to $d =$ backward.

F FULL RULES OF THE CONSTRAINT GENERATION RULES

In this section, we give the full version of the inference rules for the SMT constraint generation. Figures 9, 10, and 11 show the rules for pure regexes except for Kleene stars and holes, for Kleene stars, and the real-world extensions, respectively. For the state $(t, t_c, w, p, \Gamma, d, l)$ that is not applicable for any rule in the figure, we evaluate it as $(t, t_c, w, p, \Gamma, d, l) \dashrightarrow (\emptyset, \emptyset)$ if t is not the Kleene star, and otherwise, i.e., t is the Kleene star, $(t, t_c, w, p, \Gamma, d, l) \dashrightarrow (\{(p, \Gamma, \mathrm{true})\}, \emptyset)$.

These rules except for the rule of holes build on the semantics of regexes defined in Section 3. The difference between the semantics and the inference rules is that the inference rules compute results of succeeded and failed matching, and construct SMT constraints based on the results. Additionally, the inference rules have the rule of holes. Since our algorithm tries to replace holes with a set of characters such that the obtained regex is consistent with examples, the rule of holes behaves like the rule of the set-of-characters operator. That is, for $(\square, t_c, w, p, \Gamma, d, l)$, if $p \le |w|$, then we can replace the hole with the sets of character $[C]$ such that $w[p] \in C$ or $w[p] \notin C$. Therefore, we add $(p + 1, \Gamma, v_i^{w[p]})$ to the succeeded result, i.e., the matching at the hole succeeded and therefore the replaced set of characters accepts the character $w[p]$, and $(\bot, \bot, \neg v_i^{w[p]})$ to the failed result, i.e., the matching at the hole failed and therefore the replaced set of characters rejects the character $w[p]$.

Received 2022-11-10; accepted 2023-03-31