

Multi-modal Synthesis of Regular Expressions

Qiaochu Chen
University of Texas at Austin
Austin, Texas, USA
qchen@cs.utexas.edu

Xinyu Wang
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA
xwangsd@umich.edu

Xi Ye
University of Texas at Austin
Austin, Texas, USA
xiye@cs.utexas.edu

Greg Durrett
University of Texas at Austin
Austin, Texas, USA
gdurrett@cs.utexas.edu

Isil Dillig
University of Texas at Austin
Austin, Texas, USA
isil@cs.utexas.edu

Abstract

In this paper, we propose a multi-modal synthesis technique for automatically constructing regular expressions (*regexes*) from a combination of examples and natural language. Using multiple modalities is useful in this context because natural language alone is often highly ambiguous, whereas examples in isolation are often not sufficient for conveying user intent. Our proposed technique first parses the English description into a so-called *hierarchical sketch* that guides our programming-by-example (PBE) engine. Since the hierarchical sketch captures crucial hints, the PBE engine can leverage this information to both prioritize the search as well as make useful deductions for pruning the search space.

We have implemented the proposed technique in a tool called REGEL and evaluate it on over three hundred regexes. Our evaluation shows that REGEL achieves 80% accuracy whereas the NLP-only and PBE-only baselines achieve 43% and 26% respectively. We also compare our proposed PBE engine against an adaptation of ALPHAREGEX, a state-of-the-art regex synthesis tool, and show that our proposed PBE engine is an order of magnitude faster, even if we adapt the search algorithm of ALPHAREGEX to leverage the sketch. Finally, we conduct a user study involving 20 participants and show that users are twice as likely to successfully come up with the desired regex using REGEL compared to without it.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Theory of computation** → *Regular languages*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, United Kingdom

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7613-6/20/06 · \$15.00

<https://doi.org/10.1145/3385412.3385988>

Keywords: Program Synthesis, Programming by Natural Languages, Programming by Example, Regular Expression

ACM Reference Format:

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 International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, United Kingdom*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3385412.3385988>

1 Introduction

As a convenient mechanism for matching patterns in text data, regular expressions (or *regexes*, for short) have found numerous applications ranging from search and replacement to input validation. In addition to being heavily used by programmers, regular expressions have also gained popularity among computer end-users. For example, many text editors, word processing programs, and spreadsheet applications now provide support for performing search and replacement using regexes. However, despite their potential to dramatically simplify various tasks, regular expressions have a reputation for being quite difficult to master.

Due to the practical importance of regexes, prior research has proposed methods to automatically generate regular expressions from high-level user guidance. For example, several techniques generate regexes from natural language descriptions [25, 30, 49], while others synthesize regexes from positive and negative examples [18, 27, 44]. While these techniques have made some headway in regex synthesis, existing NLP-based techniques have relatively low accuracy even for stylized English descriptions [30], whereas example-based synthesizers impose severe restrictions on the kinds of regular expressions they can synthesize (e.g., restrict the use of Kleene star [18, 44] or consider only a binary alphabet [27]).

A central premise of this work is that both modalities of information, namely examples and natural language, are complementary and simultaneously useful for synthesizing regular expressions. As evidenced by numerous regex-related questions posted on online forums, most users communicate their intent using a combination of natural language and positive/negative examples. In particular, a common pattern is that users typically describe the high-level task using natural

language, but they also give positive and negative examples to clarify any ambiguities present in that description.

Motivated by this observation, this paper presents a new *multi-modal* synthesis algorithm that utilizes both examples and English text to generate the target regex. The key idea underlying our method is to parse the natural language description into a *hierarchical sketch* (or *h-sketch* for short) that is used to guide a programming-by-example (PBE) engine. Since hierarchical sketches capture key hints present in the English description, they make it much easier for our PBE technique to find regexes that match the user’s intent. Furthermore, because the hierarchical nature of these sketches closely reflects the compositional structure of the natural language they are derived from, it is feasible to obtain the basic scaffolding of the target regex using non-data-hungry NLP techniques like *semantic parsing* [47, 48].

In order to effectively use the hints derived from natural language, our technique leverages a new PBE algorithm for the regex domain. In particular, our PBE technique uses the hints provided by the h-sketch to both prioritize its search and also perform useful deductive reasoning. In addition, our PBE technique leverages so-called *symbolic regular expressions* to group similar regexes during the search process and uses an SMT solver to concretize them.

We have implemented the proposed approach in a tool called `REGEL`¹ and compare it against relevant baselines on over 300 regexes collected from two different sources. Our evaluation demonstrates the advantages of multi-modal synthesis compared to both `DEEPREGEX`, a state-of-the-art NLP tool, as well as a pure PBE approach. In particular, `REGEL` can find the intended regex in 80% of the cases, whereas the pure PBE and NLP baselines can synthesize only 26% and 43% of the benchmarks respectively. In our evaluation, we also compare `REGEL`’s PBE engine against an adaptation of `ALPHAREGEX`, a state-of-the-art PBE tool for regex synthesis, and demonstrate an order of magnitude improvement in terms of sketch completion time. Finally, we perform a user study targeting real-world regex construction tasks and show that users are twice as likely to construct the intended regex using `REGEL` than without it.

In summary, this paper makes the following contributions:

- We describe a new multi-modal synthesis technique for generating regexes from examples and natural language.
- We introduce *hierarchical sketches* and develop a semantic parser to generate h-sketches from English descriptions.
- We present a new PBE engine for regular expression synthesis that (1) leverages hints in the h-sketch to guide both the search and deduction, and (2) utilizes the concept of *symbolic regexes* to further prune the search space.

- We evaluate our technique on over 300 regexes and empirically demonstrate its advantages against multiple baselines on two different data sets.
- We conduct a user study and run statistical significance tests to evaluate the benefits of `REGEL` to prospective users.

2 Overview

In this section, we give a high-level overview of our technique with the aid of a motivating example. Consider the task of writing a regular expression to match strings that correspond to decimal numbers of the form $x.y$ where x (resp. y) is an integer with at most 15 (resp. 3) digits. Furthermore, this regex should accept strings that correspond to 15 digit integers (i.e., where the $.y$ part is missing).

As posted in a StackOverflow post,² the user explains this task using the following English description \mathcal{L} : “I need a regular expression that validates `Decimal(18, 3)`, which means the max number of digits before comma is 15 then accept at max 3 numbers after the comma.” The user also provides some positive examples \mathcal{E}^+ and negative examples \mathcal{E}^- :

Positive Examples	Negative Examples
123456789.123	1234567891234567
123456789123456.12	123.1234
12345.1	1.12345
123456789123456	.1234

Here, the user’s English description is not only ambiguous, but also somewhat misleading. First, the user means to say “period” instead of “comma”, and, second, it is not clear from the description whether a pure integer such as “123” should be allowed. On the other hand, the string examples alone are also not sufficient for completely understanding user intent. For instance, by looking at the examples in isolation, it is difficult to tell whether digit 0 is allowed or not.

To synthesize the target regex based on the user’s description and examples, our method first uses a semantic parser [7] to “translate” the natural language description into a *hierarchical sketch* (*h-sketch*) that captures the high-level structure of the target regex. Given the English description \mathcal{L} , our semantic parser generates a *ranked* list of such h-sketches, one of which is given below:

$$\text{Concat}\left(\square\{\langle \text{num} \rangle, \langle, \rangle\}, \square\{\text{RepeatRange}(\langle \text{num} \rangle, 1, 3), \langle, \rangle\}\right) \quad (1)$$

In this h-sketch, the symbol \square denotes an unknown regex, and the notation $\square\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ indicates that the unknown regex \square should contain *at least* one of the components (“hints”) $\mathcal{S}_1, \dots, \mathcal{S}_n$ as a leaf node. Thus, looking at this h-sketch, we can make the following observations:

1. Since the top-level operator is `Concat`, the regular expression is of the form `Concat(R_1, R_2)`.
2. R_1 should contain *either* a digit (i.e., $\langle \text{num} \rangle$) or a comma (i.e., \langle, \rangle) as a component.

¹Stands for Regular Expression Generation from Examples and Language.

²<https://stackoverflow.com/questions/19076566/need-regular-expression-that-validate-decimal-18-3>

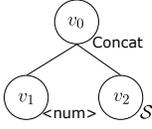


Figure 1. A partial regex example where \mathcal{S} represents the h-sketch $\square_2\{\langle, \rangle, \text{RepeatRange}(\langle \text{num} \rangle, 1, 3)\}$.

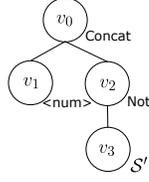


Figure 2. A partial regex expanded from Figure 1 where \mathcal{S}' stands for $\square_1\{\langle, \rangle, \text{RepeatRange}(\langle \text{num} \rangle, 1, 3)\}$.

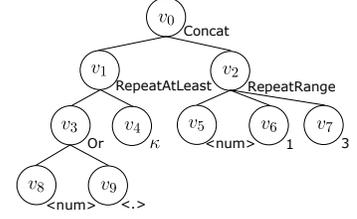


Figure 3. A symbolic regex example.

3. R_2 should contain *either* a 1-3 digit number (i.e., $\text{RepeatRange}(\langle \text{num} \rangle, 1, 3)$) *or* a comma.

While this sketch is far from perfect, it still contains useful sub-regexes that do indeed occur in the target regex.

Given a hierarchical sketch \mathcal{S} like the one from Eq. 1, our PBE engine tries to find a regex that is both a valid completion of \mathcal{S} and also consistent with the provided examples. From a high level, the synthesizer performs top-down sketch-guided enumerative search over *partial regexes* represented as abstract syntax trees (ASTs). For instance, Figure 1 shows an example partial regex where nodes are labeled with h-sketches, operators, or character classes. At every step, the synthesizer picks a node labeled with a sketch and decides how to expand that node. For instance, Figure 2 shows an expansion of the partial regex from Figure 1 where the node v_2 has been instantiated with the Not operator which now has a new child v_3 labeled with a new h-sketch \mathcal{S}' .³

The synthesis engine underlying REGEL leverages two ideas that help make it practical. First, similar to prior work [27], REGEL uses lightweight deductive reasoning to prune away infeasible partial regexes by constructing over- and under-approximations. However, with our h-sketches, we are able to construct these approximations using hints obtained from the natural language and therefore perform more precise reasoning. Specifically, given a partial regex P , our PBE engine uses the h-sketch to construct a pair of regular expressions $\langle o, u \rangle$ such that (1) o accepts every string that *any* completion of P can match, and (2) u accepts only those strings that *every* completion of P accepts. For instance, the under-approximation for the partial regex from Figure 2 is:

$$\text{Concat}(\langle \text{num} \rangle, \text{Not}(\text{Or}(\langle, \rangle, \text{RepeatRange}(\langle \text{num} \rangle, 1, 3)))) \quad (2)$$

Since this regex recognizes the negative example “12345678912345467”, *any* completion of the partial regex from Figure 2 must also recognize this negative example. Thus, we can reject this partial regex without compromising completeness.

The second idea underlying our synthesis algorithm is to introduce *symbolic regexes* to prune large parts of the search space. In particular, our regex DSL has several constructs (e.g., RepeatRange) that take integer constants as arguments, but explicitly enumerating possible values of these integer constants during synthesis can be quite inefficient. To deal

$$\begin{aligned} r := & c \mid \epsilon \mid \emptyset \\ & \mid \text{StartsWith}(r) \mid \text{EndsWith}(r) \mid \text{Contains}(r) \mid \text{Not}(r) \\ & \mid \text{Optional}(r) \mid \text{KleeneStar}(r) \\ & \mid \text{Concat}(r_1, r_2) \mid \text{Or}(r_1, r_2) \mid \text{And}(r_1, r_2) \\ & \mid \text{Repeat}(r, k) \mid \text{RepeatAtLeast}(r, k) \mid \text{RepeatRange}(r, k_1, k_2) \end{aligned}$$

Figure 4. Regex DSL. Here, $k \in \mathbb{Z}^+$ and c is a character class with this challenge, our algorithm introduces a so-called *symbolic integer* κ that represents *any* integer value. Now, given a *symbolic regex* with symbolic integers, our method generates an SMT formula ϕ over the symbolic integers $\kappa_1, \dots, \kappa_n$ such that κ_i can be instantiated with constant c_i only if c_1, \dots, c_n is a model of ϕ . For instance, consider the symbolic regex from Figure 3. By looking at each of the sub-regexes of Figure 3, we can make the following deductions:

- Since v_3 's arguments (an Or node) are both single characters, any string matched by v_3 must have length 1.
- Because RepeatAtLeast concatenates at least κ copies of its first argument, the length of any string matched by v_1 is at least κ .
- Finally, the length of any string matched by v_0 must be at least $\kappa + 1$ because v_0 's first (resp. second) argument has length at least κ (resp. 1).

Now, since there is a positive example (namely, 12345.1) of length 7, this gives us the constraint $\kappa + 1 \leq 7$ (i.e., $\kappa \leq 6$) on the symbolic integer κ . Thus, rather than enumerating all possible integers, our approach instead generates an SMT formula and solves for possible values of the symbolic integers. However, because the generated SMT formula Φ over-approximates—rather than precisely encodes—regex semantics, not every model of ϕ corresponds to a regex that is consistent with the examples. Thus, our approach uses SMT solving to prune infeasible symbolic regexes rather than directly solving for the unknown constants (e.g., as is done in SKETCH [28] and its variants [8, 19, 24, 43]).

Using these ideas, our synthesis algorithm is able to synthesize the following correct regex:

$$\text{Concat}(\text{RepeatRange}(\langle \text{num} \rangle, 1, 15), \text{Optional}(\text{Concat}(\langle, \rangle, \text{RepeatRange}(\langle \text{num} \rangle, 1, 3))))$$

3 Regex Language

Following prior work [30], we express regular expressions in the simple DSL shown in Figure 4.⁴ While most constructs in

³In Figure 1 and Figure 2, the notation \square_k indicates that the depth of the unknown regex is at most k . Thus, when we derive the new sketch for node v_3 , we use the same sketch labeling v_2 but with depth 1 instead of 2.

⁴The precise semantics of this DSL are provided in the Appendix under the extended version of this paper[10].

$$\begin{aligned}
\mathcal{S} &:= \square_d\{\bar{\mathcal{S}}\} && \text{(constrained hole)} \\
&| f(\bar{\mathcal{S}}) && \text{(operator without symbolic integer)} \\
&| g(\mathcal{S}, \bar{\kappa}) && \text{(operator with symbolic integer)} \\
&| r && \text{(regex)}
\end{aligned}$$

Figure 5. Syntax of hierarchical sketch language where r is a concrete regex and κ_i is a symbolic integer.

this DSL are just syntactic sugar for standard regular expressions, the And and Not operators may require performing intersection and complement at the automaton level. However, any “program” in our DSL is expressible as a standard regex, and, furthermore, several regex libraries [1, 2] already directly support some forms of And and Not. In what follows, we briefly go over the regex constructs shown in Figure 4.

Character class. A character class c is either a single character (e.g., $\langle a \rangle$, $\langle 1 \rangle$, \langle , \rangle) or a predefined family of characters. For instance, the character class $\langle \text{num} \rangle$ matches any digit $[\emptyset-9]$, $\langle \text{let} \rangle$ matches any letter $[\text{a-zA-Z}]$, and $\langle \text{cap} \rangle$ and $\langle \text{low} \rangle$ match upper and lower case letters respectively. We also have a character class $\langle \text{any} \rangle$ that matches any character, $\langle \text{alphanum} \rangle$ matches alphanumeric characters, and $\langle \text{hex} \rangle$ matches hexadecimal characters.

Containment. The DSL operator $\text{StartsWith}(r)$ (resp. $\text{EndsWith}(r)$) evaluates to true on string s if there is a prefix (resp. suffix) of s that matches r . Similarly, $\text{Contains}(r)$ evaluates to true on s if any substring of s matches r .

Concatenation. The operator $\text{Concat}(r_1, r_2)$ evaluates to true on string s if s is a concatenation of two strings s_1, s_2 that match r_1, r_2 respectively.

Logical operators. The operator $\text{Not}(r)$ matches a string s if s does not match r . Similarly, $\text{And}(r_1, r_2)$ (resp. $\text{Or}(r_1, r_2)$) matches s if s matches both (resp. either) s_1 and (resp. or) s_2 . The construct $\text{Optional}(r)$ is syntactic sugar for $\text{Or}(\epsilon, r)$.

Repetition. The construct $\text{Repeat}(r, k)$ matches string s if s is a concatenation of exactly k strings s_1, \dots, s_k where each s_i matches r . $\text{RepeatRange}(r, k_1, k_2)$ matches string s if there exists some $k \in [k_1, k_2]$ such that $\text{Repeat}(r, k)$ matches s . Finally, $\text{RepeatAtLeast}(r, k)$ is just syntactic sugar for $\text{RepeatRange}(r, k, \infty)$, and $\text{KleeneStar}(r)$ is equivalent to $\text{Or}(\epsilon, \text{RepeatAtLeast}(r, 1))$. Note that operators in the Repeat family require every integer value k to be a positive number.

4 Hierarchical Sketches

In this section, we present the syntax and semantics of hierarchical sketches (h-sketches) that we derive from the natural language. Intuitively, an h-sketch represents a *family* of regexes that conform to a high-level structure.

As shown in Figure 5, our h-sketch language extends our regex DSL by allowing a “constrained hole” construct. A constrained hole, denoted $\square_d\{\bar{\mathcal{S}}\}$, is an unknown regex that is parametrized with a positive integer d and a set of nested h-sketches $\bar{\mathcal{S}}$. Specifically, regex r belongs to the space of

$$\begin{aligned}
\llbracket r \rrbracket &= \{r\} \\
\llbracket f(\bar{\mathcal{S}}) \rrbracket &= \{f(\bar{r}) \mid \forall_{i \in |\bar{\mathcal{S}}|} r_i \in \llbracket \mathcal{S}_i \rrbracket\} \\
\llbracket g(\mathcal{S}, \bar{\kappa}) \rrbracket &= \{g(r, \bar{k}) \mid r \in \llbracket \mathcal{S} \rrbracket, \forall_{i \in |\bar{\kappa}|} k_i \in \mathbb{N}\} \\
\llbracket \square_d\{\bar{\mathcal{S}}\} \rrbracket &= \begin{cases} \bigcup_{i \in |\bar{\mathcal{S}}|} \llbracket \mathcal{S}_i \rrbracket & d = 1 \\ \bigcup_{i \in |\bar{\mathcal{S}}|} \llbracket \mathcal{S}_i \rrbracket & d > 1 \\ \bigcup_{f \in \mathcal{F}_n} \bigcup_{1 \leq i \leq n} \llbracket f(\underbrace{l, \dots, l}_{i-1 \text{ times}}, \underbrace{\square_{d-1}\{\bar{\mathcal{S}}\}, l, \dots, l}_{n-i \text{ times}}) \rrbracket & \text{where } l = \square_{d-1} C \cup \{\bar{\mathcal{S}}\} \\ \bigcup_{g \in \mathcal{G}_n} \llbracket g(\square_{d-1}\{\bar{\mathcal{S}}\}, \bar{\kappa}) \rrbracket \end{cases}
\end{aligned}$$

Figure 6. Semantics of h-sketches. $g \in \mathcal{G}_n$ (resp. $f \in \mathcal{F}_n$) is an n -ary operator in (resp. outside of) the Repeat family.

regexes defined by $\square_d\{\bar{\mathcal{S}}\}$ if one of the “leaf” nodes of r conforms to \mathcal{S}_i and r has depth at most d (when \mathcal{S}_i is viewed as a “leaf node”). Observe that constrained holes can be arbitrarily nested, which is why these sketches are *hierarchical*.

In addition to constrained holes, h-sketches can also contain operators in our regex DSL. For example, an h-sketch can be of the form $f(\bar{\mathcal{S}})$ where f denotes a DSL operator outside of the Repeat family (e.g., Concat). Semantically, $f(\bar{\mathcal{S}})$ represents the set of regexes $f(\bar{r})$ where we have $r_i \in \llbracket \mathcal{S}_i \rrbracket$. Our h-sketches can also be of the form $g(\mathcal{S}, \bar{\kappa})$ where g is a construct in the Repeat family and κ ’s are so-called *symbolic integers*. The set of programs defined by $g(\mathcal{S}, \bar{\kappa})$ includes all programs of the form $g(r, \bar{k})$ where we have $r \in \llbracket \mathcal{S} \rrbracket$ and k_i is any positive integer. Finally, our h-sketch language also includes *concrete* regular expressions (without holes), and the semantics provided in Figure 6 summarize this discussion.

Example 4.1. The program $\text{Concat}(\langle \text{num} \rangle, \text{Contains}(\langle , \rangle))$ is in the language of the h-sketch $\text{Concat}(\square_1\{\langle , \rangle, \langle \text{num} \rangle\}, \square_2\{\langle , \rangle, \text{RepeatRange}(\langle \text{num} \rangle, 1, 3)\})$.

Remark. While constrained holes in Figure 5 are explicitly parametrized by an integer d to facilitate defining h-sketch semantics, the sketches produced by our semantic parser do not have this explicit integer d . Instead, d should be thought of as a configurable parameter that determines the depth of the search tree explored by the PBE engine.

5 Regex Synthesis from H-Sketches

In this section, we describe our synthesis algorithm that generates a regex from an h-sketch \mathcal{S} and a set of positive and negative examples, \mathcal{E}^+ and \mathcal{E}^- . The output of the synthesis procedure is either \perp which indicates an unsuccessful synthesis attempt or a regex r such that:

$$(1) r \in \llbracket \mathcal{S} \rrbracket \quad (2) \forall s \in \mathcal{E}^+. \llbracket r \rrbracket_s = \text{true} \quad (3) \forall s \in \mathcal{E}^-. \llbracket r \rrbracket_s = \text{false}$$

Our synthesis procedure is given in Figure 7. At a high-level, SYNTHESIZE maintains a worklist of *partial regexes* and

keeps growing this worklist by expanding the *abstract syntax tree* (AST) representation of a partial regex.

Definition 5.1. (Partial regex) A *partial regex* P is a tree (V, E, A) where V is a set of vertices, E is a set of directed edges, and A is a mapping from each node $v \in V$ to a label ℓ , which is either (1) a DSL construct (e.g., character class or operator), (2) a symbolic integer κ , or (3) a hierarchical sketch \mathcal{S} .

In the remainder of this section, we use the term *symbolic regex* to denote a partial regex where all of the node labels are either DSL constructs or symbolic integers (not an h-sketch), and we use the term *concrete regex* to denote a partial regex where all node labels are DSL constructs. Thus, every concrete regex corresponds to a program written in the regex DSL from Figure 4. Given a partial regex P , we write $\text{IsConcrete}(P)$ to indicate that P is a concrete regex and $\text{IsSymbolic}(P)$ to indicate that P is a symbolic (but not concrete) regex. Finally, we refer to any node whose corresponding label is an h-sketch as an *open node*.

Example 5.2. The partial regex shown in Figure 3 is a symbolic (but not concrete) regex. On the other hand, the partial regexes from Figures 1 and 2 are neither symbolic nor concrete because the nodes labeled with \mathcal{S} are *open*.

Notation. Given a partial regex P represented as an AST, we write $\text{Edges}(P)$ to denote the set of all edges in P , $\text{Root}(P)$ to denote the root node, and $\text{Subtree}(P, v)$ to denote the subtree of P rooted at node v . Given a node v , we write $v : \ell$, to denote that the label of v is ℓ . Adding a node $v : \ell$ to P is denoted as $P[v \triangleleft \ell]$ (in case v already exists in P , it updates v 's label to be ℓ). Furthermore, adding multiple nodes $v_1 : \ell_1, \dots, v_n : \ell_n$ is denoted as $P[v_1 \triangleleft \ell_1, \dots, v_n \triangleleft \ell_n]$, and we assume that $(v_1, v_2), \dots, (v_1, v_n)$ are added as edges to P if it does not already contain them.

With this notation in place, we now explain the `SYNTHESIZE` procedure from Figure 7 in more detail. The algorithm first initializes the worklist to be the singleton $\{P_0\}$, where P_0 is a partial regex with a single node v_0 labeled with the input sketch \mathcal{S} (line 2). The loop in lines 3–15 dequeues one of the partial regexes P from the worklist and processes it based on whether it is concrete, symbolic, or neither. If it is concrete (line 5), we return P as a solution if it is consistent with the examples (line 6).

On the other hand, if P is symbolic (line 7), we invoke a procedure called `INFERCONSTANTS` (described in Section 5.2) that instantiates the symbolic integers in P with integer constants (line 8). As mentioned in Section 2, `INFERCONSTANTS` should be viewed as merely a way of *pruning* infeasible programs, so the regexes produced by `INFERCONSTANTS` are *not* guaranteed to satisfy the examples. Thus, the regexes produced by `INFERCONSTANTS` still have to be checked for consistency with the examples in future iterations.

Lines 10–15 of the `SYNTHESIZE` algorithm deal with the case where the dequeued partial regex is neither concrete

```

1: procedure SYNTHESIZE( $\mathcal{S}, \mathcal{E}^+, \mathcal{E}^-$ )
   input: an h-sketch  $\mathcal{S}$ , positive and negative examples  $\mathcal{E}^+, \mathcal{E}^-$ 
   output: a regex consistent with  $\mathcal{S}, \mathcal{E}^+$  and  $\mathcal{E}^-$ , or  $\perp$ 
2:    $P_0 := (v_0, \emptyset, [v_0 \triangleleft \mathcal{S}]);$   $worklist := \{P_0\};$ 
3:   while  $worklist \neq \emptyset$  do
4:      $P := worklist.remove();$ 
5:     if  $\text{IsConcrete}(P)$  then
6:       if  $\text{IsCorrect}(P, \mathcal{E}^+, \mathcal{E}^-)$  then return  $P;$ 
7:     else if  $\text{IsSymbolic}(P)$  then
8:        $worklist := worklist \cup \text{INFERCONSTANTS}(P, \mathcal{E}^+, \mathcal{E}^-);$ 
9:     else
10:       $(v, \mathcal{S}) := \text{SelectOpenNode}(P);$ 
11:       $worklist' := \text{EXPAND}(P, v, \mathcal{S});$ 
12:      for all  $P' \in worklist'$  do
13:        if  $\text{INFEASIBLE}(P', \mathcal{E}^+, \mathcal{E}^-)$  then
14:           $worklist'.remove(P');$ 
15:         $worklist := worklist \cup worklist';$ 
16:   return  $\perp;$ 

```

Figure 7. Synthesis algorithm for generating a regex from an h-sketch and a set of positive/negative examples.

$$\frac{n = |\overline{\mathcal{S}}| \quad \Pi = \bigcup_{i=1}^n \{P[v \triangleleft \mathcal{S}_i]\}}{v : \square_1\{\overline{\mathcal{S}}\} \vdash P \rightsquigarrow \Pi} \quad (1)$$

$$\frac{\begin{aligned} \Pi_1 &= \bigcup_{i=1}^{|\overline{\mathcal{S}}|} \{P[v \triangleleft \mathcal{S}_i]\} & \ell &= \square_{d-1}\{\overline{\mathcal{S}}\} & \ell' &= \square_{d-1}C \cup \{\overline{\mathcal{S}}\} \\ \Pi_2 &= \bigcup_{j=1}^{|\overline{\mathcal{V}}|} \{P[v \triangleleft f, v_j \triangleleft \ell, \forall_{i \neq j} v_i \triangleleft \ell'] \mid f \in \mathcal{F}_{|\overline{\mathcal{V}}|}, \overline{v} \text{ fresh}\} \\ \Pi_3 &= \{P[v \triangleleft g, v_0 \triangleleft \ell, \forall_{i \in [1, |\overline{\mathcal{V}}|]} v_i \triangleleft \kappa_i] \mid g \in \mathcal{G}_{|\overline{\mathcal{V}}|}, \overline{v}, \overline{\kappa} \text{ fresh}\} \end{aligned}}{v : \square_{d \geq 1}\{\overline{\mathcal{S}}\} \vdash P \rightsquigarrow \Pi_1 \cup \Pi_2 \cup \Pi_3} \quad (2)$$

$$\frac{\overline{v} \text{ fresh} \quad n = |\overline{\mathcal{S}}| \quad \Pi = \{P[v \triangleleft f, \forall_{i \in [1, n]} v_i \triangleleft \mathcal{S}_i]\}}{v : f(\overline{\mathcal{S}}) \vdash P \rightsquigarrow \Pi} \quad (3)$$

$$\frac{\overline{v} \text{ fresh} \quad \Pi = \{P[v \triangleleft g, v_0 \triangleleft \mathcal{S}, \forall_{i \in [1, |\overline{\mathcal{K}}|]} v_i \triangleleft \kappa_i]\}}{v : g(\mathcal{S}, \overline{\kappa}) \vdash P \rightsquigarrow \Pi} \quad (4)$$

Figure 8. Inference rules for `EXPAND`. In rule (2), C denotes all character classes in the DSL, \mathcal{G}_i (resp. \mathcal{F}_i) denotes Repeat (resp. non-Repeat) constructs with arity i .

nor symbolic (i.e., P has at least one open node). In this case, we pick one of the open nodes v in P and expand it according to the hints contained in the h-sketch labeling v . Specifically, the `EXPAND` function from line 11 is described in Figure 8 using inference rules of the form $v : \mathcal{S} \vdash P \rightsquigarrow \Pi$. The meaning of this judgement is that we obtain a new set of partial regexes Π by expanding node v according to h-sketch \mathcal{S} . Intuitively, given a node v labeled with sketch $\square_d\{\overline{\mathcal{S}}\}$, the inference rules enforce that *at least* one descendant of v must correspond to a regex in the languages of $\overline{\mathcal{S}}$.

Next, given each expansion P' of P , we check whether P' is consistent with the provided examples via the call at line 13 to the `INFEASIBLE` function (discussed in detail in Section 5.1). Observe that the worklist only contains partial

$$\frac{\text{Root}(P) = v : \mathcal{S} \quad \vdash \mathcal{S} \rightsquigarrow \langle o, u \rangle}{\vdash P \rightsquigarrow \langle o, u \rangle} \quad (1)$$

$$\frac{\text{Root}(P) = v : (f \in \mathcal{F}_n \setminus \{\text{Not}\}) \\ (v, v_i) \in \text{Edges}(P) \quad \vdash \text{Subtree}(P, v_i) \rightsquigarrow \langle o_i, u_i \rangle}{\vdash P \rightsquigarrow \langle f(\overline{o_N}), f(\overline{u_N}) \rangle} \quad (2)$$

$$\frac{\text{Root}(P) = v : \text{Not} \\ (v, v_1) \in \text{Edges}(P) \quad \vdash \text{Subtree}(P, v_1) \rightsquigarrow \langle o_1, u_1 \rangle}{\vdash P \rightsquigarrow \langle \text{Not}(u_1), \text{Not}(o_1) \rangle} \quad (3)$$

$$\frac{\text{Root}(P) = v : (g \in \mathcal{G}_n) \quad (v, v_i : \ell_i) \in \text{Edges}(P) \\ \vdash \text{Subtree}(P, v_i) \rightsquigarrow \langle o_i, u_i \rangle \quad \forall i \geq 2. \ell_i \in \mathbb{N}}{\vdash P \rightsquigarrow \langle g(o_1, \bar{\ell}), g(u_1, \bar{\ell}) \rangle} \quad (4)$$

$$\frac{\text{Root}(P) = v : (g \in \mathcal{G}_n) \quad (v, v_i : \ell_i) \in \text{Edges}(P) \\ \vdash \text{Subtree}(P, v_i) \rightsquigarrow \langle o_i, u_i \rangle \quad \exists i \geq 2. \text{SymInt}(\ell_i)}{\vdash P \rightsquigarrow \langle \text{RepeatAtLeast}(o_1, 1), \perp \rangle} \quad (5)$$

Figure 9. Inference rules for APPROXIMATE. \mathcal{G}_n (resp. \mathcal{F}_n) denotes arity n operators in (resp. not in) the Repeat family.

regexes that are consistent with the examples according to the abstract semantics given in Section 5.1.

5.1 Pruning infeasible partial regexes

The high-level idea for pruning infeasible partial regexes is quite simple and leverages the same observation made by Lee et al. [27]: Given a partial regex P , we can generate two concrete regexes, o and u , that over- and under-approximate P respectively. Specifically, o and u have the following properties:

- (1) $\forall s. (\exists r \in \llbracket P \rrbracket. \text{Match}(r, s)) \Rightarrow \text{Match}(o, s)$
- (2) $\forall s. \text{Match}(u, s) \Rightarrow (\forall r \in \llbracket P \rrbracket. \text{Match}(r, s))$

Here, we use the notation $r \in \llbracket P \rrbracket$ to denote that r is a valid completion of P . Thus, o matches every string s that *some* completion of P can match and u only matches those strings that *all* completions of P accept. Then, if there is any $e^+ \in \mathcal{E}^+$ that o does not match, we know that P cannot satisfy the examples and can be rejected without sacrificing completeness of our synthesis algorithm. Conversely, if there is any $e^- \in \mathcal{E}^-$ that u matches, we know that P will also match it and can thus be rejected safely. The main novelty of our feasibility checking technique compared to Lee et al. [27] is to leverage the hints inside the h-sketch to compute more precise over- and under-approximations.

Figure 9 describes our approximation procedure using inference rules of the shape $\vdash P \rightsquigarrow \langle o, u \rangle$ indicating that P is over- (resp. under-) approximated by o (resp. u). These rules make use of an auxiliary judgment $\vdash \mathcal{S} \rightsquigarrow \langle o, u \rangle$ (described in Figure 10) that generate over- and under-approximations of hierarchical sketches. In what follows, we explain a subset of these rules.

Approximating holes. The first three rules in Figure 10 describe how to approximate holes in an h-sketch. We differentiate between two cases: If the depth of the hole is exactly 1, then the hole must be filled with an instantiation of one of the h-sketches $\overline{\mathcal{S}}$. Thus, we first recursively compute over- and under-approximations for each \mathcal{S}_i as $\langle o_i, u_i \rangle$. Then, the over-approximation for the hole is obtained by taking the union over all the o_i 's and the under-approximation is obtained by intersecting all the u_i 's (rule 3). The intuition for the latter is that the under-approximation must match only strings that *every* instantiation of \mathcal{S}_i matches; hence, we use intersection. On the other hand, for holes with depth greater than 1, we approximate them as $\langle \top, \perp \rangle$ (rule 2). In principle, we could perform a more precise approximation by instantiating the hole with *every* possible DSL operator and taking the union/intersection of these regexes. However, since the resulting regex would be very large, such an alternative approximation would add a lot of overhead. Furthermore, since holes can be nested inside one another, we can often obtain a useful approximation of the top-level sketch even when we use this less precise approximation for nested holes.

Approximating negation. Rule 3 from Figure 9 and rule 5 from Figure 10 both deal with the negation operator. Because the negation of an over-approximation yields an under-approximation and vice versa, $\text{Not}(\mathcal{S})$ is approximated as $\langle \text{Not}(u), \text{Not}(o) \rangle$ where $\langle o, u \rangle$ is the approximation for \mathcal{S} .

Approximating repetition operators. The last two rules in Figure 9 deal with operators in the Repeat family, which take a regex as their first argument and integers for the remaining arguments. In rule 4, if all of the integer arguments are constants (rather than symbolic integers), then the over- and under-approximations are computed precisely. However, if one of the arguments is a symbolic integer (rule 6), the under-approximation is given by \perp , and the over-approximation is $\text{RepeatAtLeast}(o_1, 1)$ where o_1 is the over-approximation of the first argument. (Note that the second argument is 1 since the integer arguments of all constructs in the Repeat family require *positive* integers.)

Example 5.3. Consider the partial regex from Figure 2. Its over-approximation is $\text{Concat}(\langle \text{num} \rangle, \text{KleeneStar}(\langle \text{any} \rangle))$ and its under-approximation is shown in Eq. 2.

Theorem 5.4. (Correctness of APPROXIMATE in Figure 9)
Given a partial regex P , suppose $\text{APPROXIMATE}(P)$ yields $\langle o, u \rangle$. Then, we have:

- (1) $\forall s. (\exists r \in \llbracket P \rrbracket. \text{Match}(r, s)) \Rightarrow \text{Match}(o, s)$
- (2) $\forall s. \text{Match}(u, s) \Rightarrow (\forall r \in \llbracket P \rrbracket. \text{Match}(r, s))$

5.2 Solving Symbolic Regexes with SMT

Recall that our method uses symbolic regexes to avoid explicit enumeration of integer constants that appear inside Repeat constructs. In this section, we explain how to “solve” for these symbolic integers using SMT-based reasoning.

$$\begin{array}{c}
\frac{\vdash \mathcal{S} \rightarrow \langle o, u \rangle}{\vdash \square_1 \{\mathcal{S}\} \rightarrow \langle o, u \rangle} \quad (1) \qquad \frac{d > 1}{\vdash \square_d \{\overline{\mathcal{S}}\} \rightarrow \langle \top, \perp \rangle} \quad (2) \\
\frac{\vdash \mathcal{S}_1 \rightarrow \langle o, u \rangle \quad \vdash \square_1 \{\mathcal{S}_2, \dots, \mathcal{S}_{|\overline{\mathcal{S}}|}\} \rightarrow \langle o', u' \rangle}{\vdash \square_1 \{\overline{\mathcal{S}}\} \rightarrow \langle \text{Or}(o, o'), \text{And}(u, u') \rangle} \quad (3) \\
\frac{f \in \mathcal{F}_n \setminus \{\text{Not}\} \quad \vdash \mathcal{S}_i \rightarrow \langle o_i, u_i \rangle}{\vdash f(\overline{\mathcal{S}}) \rightarrow \langle f(\overline{o}), f(\overline{u}) \rangle} \quad (4) \\
\frac{\vdash \mathcal{S} \rightarrow \langle o, u \rangle}{\vdash \text{Not}(\mathcal{S}) \rightarrow \langle \text{Not}(u), \text{Not}(o) \rangle} \quad (5) \qquad \frac{}{\vdash r \rightarrow \langle r, r \rangle} \quad (6) \\
\frac{g \in \mathcal{G}_n \quad \vdash \mathcal{S} \rightarrow \langle o, u \rangle}{\vdash g(\overline{\mathcal{S}}, \overline{\kappa}) \rightarrow \langle \text{RepeatAtLeast}(o, 1), \perp \rangle} \quad (7)
\end{array}$$

Figure 10. Inference rules for over- and under-approximating h-sketches. r denotes a concrete regex.

```

1: procedure INFERCONSTANTS( $P_0, \mathcal{E}^+, \mathcal{E}^-$ )
   input: a symbolic regex  $P_0$ , examples  $\mathcal{E}^+, \mathcal{E}^-$ .
   output: a set of concrete regular expressions  $\Pi$ .
2:    $(\phi_0, x_0) := \text{Encode}(P_0)$ ;  $\psi_0 := (\bigwedge_{s \in \mathcal{E}^+} \phi_0[\text{len}(s)/x_0])$ ;
3:    $\text{worklist} := \{(P_0, \psi_0)\}$ ;  $\Pi := \emptyset$ ;
4:   while  $\text{worklist} \neq \emptyset$  do
5:      $(P, \phi) := \text{worklist.remove}()$ ;
6:     if UNSAT( $\phi$ ) then continue;
7:      $\sigma := \text{Model}(\phi)$ ;  $\kappa := \text{ChooseSymInt}(P)$ ;
8:      $P' := P[\kappa \leftarrow \sigma[\kappa]]$ ;
9:      $\text{worklist} := \text{worklist} \cup \{(P, \phi \wedge \kappa \neq \sigma[\kappa])\}$ ;
10:    if IsConcrete( $P'$ ) then  $\Pi := \Pi \cup \{P'\}$ ;
11:    else
12:      if  $\neg \text{INFEASIBLE}(P, \mathcal{E}^+, \mathcal{E}^-)$  then
13:         $\text{worklist} := \text{worklist} \cup \{(P', \phi[\kappa \leftarrow \sigma[\kappa]])\}$ ;
14:    return  $\Pi$ ;

```

Figure 11. Algorithm for INFERCONSTANTS.

Figure 11 shows the INFERCONSTANTS procedure for obtaining a set of concrete regexes from a given symbolic regex P . The high-level idea underlying this algorithm is as follows: We first infer a constraint ϕ on the values of symbolic integers $\kappa_1, \dots, \kappa_n$ using the *length* of the strings that appear in the examples. However, this constraint is over-approximate in the sense that every concrete regex must satisfy ϕ but not every model of ϕ corresponds to a concrete regex that satisfies the examples. Thus, given a candidate assignment to one of the κ 's (obtained from a model of ϕ), we use the INFEASIBLE procedure discussed in the previous section to check whether this (partial) assignment is feasible. If so, we then continue and repeat the same process for the remaining κ_i 's until we have found a full assignment for all symbolic integers that appear in P .

SMT Encoding. Before explaining the INFERCONSTANTS algorithm in more detail, we first explain how to generate

a constraint for a given symbolic regex. Our encoding is described in Figure 12 using inference rules $P \hookrightarrow (\phi, x)$. This judgement means, for any instantiation of P to match a string s , the symbolic integers in P must satisfy $\phi[\text{len}(s)/x]$. As presented in the first rule, our encoding uses a function Φ , shown also in Figure 12, that generates a constraint for a given regex from constraints on its sub-regexes. Specifically, it takes as input a DSL construct op , a variable x that refers to the length of the string matched by the top-level regex, and constraints ϕ_1, \dots, ϕ_k for the sub-regexes (where the length of the string matched by i 'th sub-regex is x_i).

For instance, consider the encoding for the `StartsWith`(r) construct: If the length of the string matched by r is x_1 (which is constrained according to ϕ_1), then any string matched by `StartsWith`(r) will be at least as long as x_1 . Thus, we have:

$$\Phi(\text{StartsWith}, x, x_1, \phi_1) = \exists x_1. (x \geq x_1 \wedge \phi_1)$$

Observe that x_1 is existentially quantified in the formula because it is a “temporary” variable that refers to the length of the string matched by the sub-regex. Since the other cases in the definition of the Φ function are similar and follow the semantics of the DSL operators, we do not discuss them in detail but just highlight two cases for `Not` and `RepeatAtLeast`.

The encoding for the `Not` operator is *true* regardless of the sub-regex because inferring anything more precise would require us to track *sufficient* (rather than necessary) conditions for accepting a string, which is not feasible to do using the *length* of the string alone.

The encoding for the `Repeat` family of constructs introduces non-linear multiplication. For instance, consider the symbolic regex `RepeatAtLeast`(r, κ) where the constraint on the sub-regex r is (ϕ_1, x_1) . Since r is repeated at least κ times, the length of the string matched by this regex is at least $x_1 \cdot \kappa$, which introduces non-linear constraints. Thus, while the formulas generated by the ENCODE procedure are technically in Peano (rather than Presburger) arithmetic, we found that the Z3 SMT solver can efficiently handle the type of non-linear constraints we generate.

Example 5.5. Consider the following symbolic regex:

$$\text{Concat} \left(\text{Repeat}(\text{Or}(\langle . \rangle, \langle \text{num} \rangle), \kappa_1), \right. \\
\left. \text{RepeatAtLeast}(\text{RepeatRange}(\langle \text{num} \rangle, 1, 3), \kappa_2) \right) \quad (3)$$

Using the rules presented in Figure 12, we generate the following constraint ϕ :

$$\begin{aligned}
\phi &= \exists x_1, x_2. (x_0 = x_1 + x_2) \wedge \phi_1 \wedge \phi_2 && \text{(Concat)} \\
\phi_1 &= \exists x_3, x'_3. (x_1 \geq x_3 * \kappa_1 \wedge x_1 \leq x'_3 * \kappa_1) && \text{(Repeat)} \\
&\quad \wedge \phi_3 \wedge \phi_3[x'_3/x_3] \wedge (1 \leq \kappa_1 \leq \text{MAX}) \\
\phi_3 &= (x_3 = 1 \vee x_3 = 1) && \text{(Or)} \\
\phi_2 &= \exists x_4. (x_2 \geq x_4 * \kappa_2) \wedge \phi_4 \wedge (1 \leq \kappa_2 \leq \text{MAX}) && \text{(AtLeast)} \\
\phi_4 &= 1 \leq x_4 \leq 3 && \text{(Range)}
\end{aligned}$$

$$\frac{\text{Root}(P) = v : \text{op} \quad \text{arity}(\text{op}) = n \quad (v, v_i) \in \text{Edges}(P) \\ \text{Subtree}(P, v_i) \hookrightarrow (\phi_i, x_i) \quad x \text{ fresh}}{P \hookrightarrow (\Phi(\text{op}, x, \bar{x}, \bar{\phi}), x)} \quad (1)$$

$$\frac{x \text{ fresh} \quad \text{Root}(P) = v : (c \in C)}{P \hookrightarrow (x = 1, x)} \quad (2)$$

$$\frac{\text{Root}(P) = v : (\kappa \in \text{SymInt}(P))}{P \hookrightarrow (1 \leq \kappa \leq \text{MAX}, \kappa)} \quad (3)^5$$

$$\begin{aligned} \Phi(\text{StartsWith}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x \geq x_1 \wedge \phi_1) \\ \Phi(\text{EndsWith}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x \geq x_1 \wedge \phi_1) \\ \Phi(\text{Contains}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x \geq x_1 \wedge \phi_1) \\ \Phi(\text{Not}, x, \bar{x}, \bar{\phi}) &= \text{true} \\ \Phi(\text{Optional}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x = 0 \vee x = x_1) \wedge \phi_1 \\ \Phi(\text{KleeneStar}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x = 0 \vee x \geq x_1) \wedge \phi_1 \\ \Phi(\text{Concat}, x, \bar{x}, \bar{\phi}) &= \exists x_1, x_2. (x = x_1 + x_2) \\ &\quad \wedge \phi_1 \wedge \phi_2 \\ \Phi(\text{Or}, x, \bar{x}, \bar{\phi}) &= \exists x_1, x_2. (x = x_1 \vee x = x_2) \\ &\quad \wedge \phi_1 \wedge \phi_2 \\ \Phi(\text{And}, x, \bar{x}, \bar{\phi}) &= \exists x_1, x_2. (x = x_1 \wedge x = x_2) \\ &\quad \wedge \phi_1 \wedge \phi_2 \\ \Phi(\text{Repeat}, x, \bar{x}, \bar{\phi}) &= \exists x_1, x'_1. (x \geq x_1 x_2 \wedge x \leq x'_1 x_2) \\ &\quad \wedge \phi_1 \wedge \phi_1[x'_1/x_1] \wedge \phi_2 \\ \Phi(\text{RepeatAtLeast}, x, \bar{x}, \bar{\phi}) &= \exists x_1. (x \geq x_1 x_2) \wedge \phi_1 \wedge \phi_2 \\ \Phi(\text{RepeatRange}, x, \bar{x}, \bar{\phi}) &= \exists x_1, x'_1. (x \geq x_1 x_2 \wedge x \leq x'_1 x_3) \\ &\quad \wedge \phi_1 \wedge \phi_1[x'_1/x_1] \wedge \phi_2 \wedge \phi_3 \end{aligned}$$

Figure 12. Inference rules for ENCODE.

Note that the top-level constraint ϕ can be simplified to the following formula by performing quantifier elimination:

$$(x_0 \geq \kappa_1 + \kappa_2) \wedge (1 \leq \kappa_1 \leq \text{MAX}) \wedge (1 \leq \kappa_2 \leq \text{MAX}) \quad (4)$$

Using SMT encoding for inference. Now that we have a way to encode symbolic regexes using SMT, we can describe the INFERCONSTANTS algorithm from Figure 11 in more detail. Given a symbolic regex P_0 , the algorithm first generates the SMT encoding ϕ_0 for P_0 using the ENCODE function (i.e., Figure 12). Here, ϕ_0 contains free variables $\kappa_1, \dots, \kappa_n$ as well as a variable x_0 that refers to the length of the input string. Now, since every $s \in \mathcal{E}^+$ should match the synthesized regex, we can obtain a constraint on the symbolic integers by instantiating x_0 with $\text{len}(s)$ for every $s \in \mathcal{E}^+$ and taking their conjunction. Thus, formula ψ_0 from line 2 gives us a constraint on the symbolic integers used in P .

Next, the loop in lines 5–13 populates a set Π of concrete regexes that can be obtained by instantiating the symbolic integers in P_0 with constants. Towards this goal, it maintains a worklist of symbolic regexes that are made increasingly more concrete in each iteration.

Specifically, the worklist contains pairs (P, ϕ) where P is a symbolic regex and ϕ is a constraint on the symbolic integers used in P – initially, the worklist just contains (P_0, ψ_0) . Then, in each iteration, we remove from the worklist a symbolic regex P and its constraint ϕ and make an assignment to one of the symbolic integers κ used in P . To this end, we first query the SMT solver to get a model σ of ϕ . However, since ϕ is over-approximate, instantiating the symbolic integers in P with σ may not yield a concrete regex that satisfies the examples. Thus, we pick one of the symbolic integers κ in P and check whether $\sigma[\kappa]$ is infeasible using the method described in Section 5.1 (line 12).⁶ If the resulting symbolic regex cannot be proven infeasible, we then add the partially concretized symbolic program $P' = P[\kappa \leftarrow \sigma[\kappa]]$ to the worklist, together with its corresponding constraint $\phi[\kappa \leftarrow \sigma[\kappa]]$ (line 13). However, in addition, we also keep the original symbolic regex P since there may be other valid assignments to κ beyond just $\sigma[\kappa]$ (line 9). Finally, to ensure that the solver does not keep yielding the same assignment to κ , we strengthen its constraint by adding the “blocking clause” $\kappa \neq \sigma[\kappa]$ (also line 9). Upon termination, the set Π contains every feasible concrete regex that can be obtained by instantiating the original symbolic regex P_0 .

Example 5.6. Consider the simplified constraint ϕ from Eq. 4. After instantiating x_0 with the length of each positive example from Section 2 and taking their conjunction, we obtain the following formula ψ_0 :

$$\begin{aligned} &(\kappa_1 + \kappa_2 \leq 13) \wedge (\kappa_1 + \kappa_2 \leq 7) \wedge (\kappa_1 + \kappa_2 \leq 18) \wedge (\kappa_1 + \kappa_2 \leq 15) \\ &\wedge (1 \leq \kappa_1 \leq \text{MAX}) \wedge (1 \leq \kappa_2 \leq \text{MAX}) \end{aligned}$$

This formula is equivalent to the following much simpler constraint:

$$\psi_0 = (\kappa_1 + \kappa_2 \leq 7) \wedge (1 \leq \kappa_1 \leq \text{MAX}) \wedge (1 \leq \kappa_2 \leq \text{MAX}) \quad (5)$$

Now, suppose the solver returns the model $[\kappa_1 \mapsto 1, \kappa_2 \mapsto 1]$ to Eq. 5. Thus, we first assign 1 to κ_1 in the partial regex from Eq. 3, which yields:

$$\begin{aligned} &\text{Concat}(\text{Repeat}(\text{Or}(\langle \text{num} \rangle, \langle . \rangle), 1), \\ &\text{RepeatAtLeast}(\text{RepeatRange}(\langle \text{num} \rangle, 1, 3), \kappa_2)) \end{aligned}$$

We can prove that this partial regex is inconsistent with the examples from Section 2 because no instantiation of κ_2 yields a regex that matches the positive example “123456789.123”. Observe that ignoring the assignment to κ_2 allows us to prune 6 regexes at a time instead of just one.

Theorem 5.7. (Correctness of INFERCONSTANTS in Figure 11) *Given a partial regex P , positive examples \mathcal{E}^+ and negative examples \mathcal{E}^- , suppose that INFERCONSTANTS returns Π . Then, for any concrete regex $r \in \llbracket P \rrbracket$ that is consistent with \mathcal{E}^+ and \mathcal{E}^- , we have $r \in \Pi$.*

⁶Alternatively, we could plug in the whole assignment σ and check whether the resulting regex is consistent with the examples. However, our proposed method is preferable over this alternative because a partial assignment to a subset of the variables often results in an infeasible partial regex and allows us to prune significantly more programs.

⁵MAX is the maximum integer constant in the DSL. We set MAX to the length of the longest example in the implementation.

1	\$CC	<num>	→	number	1	\$ROOT	(IdentityFn arg:0)	→	(\$SKETCH)
2	\$CC	<num>	→	digit	2	\$SKETCH	(SketchFn arg:0, arg:1, ...)	→	(\$PROGRAM, \$PROGRAM, ...)
3	\$CONST	<,>	→	comma	3	\$SKETCH	(ConcatFn arg:0 arg:2)	→	(\$SKETCH \$OP.CONCAT \$SKETCH)
4	\$OP.CONCAT	op.concat	→	before	4	\$PROGRAM	(RepeatRangeFn val:1 arg:0 arg:2)	→	(\$OP.REPEATRANGE \$INT \$PROGRAM)
5	\$OP.CONCAT	op.concat	→	then	5	\$PROGRAM	(RepeatFn arg:0 arg:1)	→	(\$INT \$PROGRAM)
6	\$OP.RPTRANGE	op.rptrange	→	at max	6	\$PROGRAM	(IdentityFn arg:0)	→	(\$CC)
7	\$INT	{integer}	→	{Integer}	7	\$PROGRAM	(IdentityFn arg:0)	→	(\$CONST)

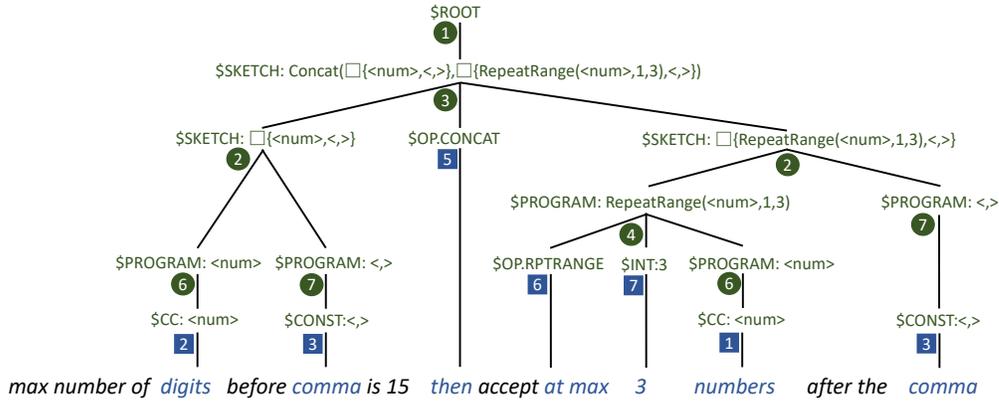


Figure 13. Examples of rules and the parse tree for one possible derivation generated from the given description.

6 From English Text to H-Sketches

In this section, we describe a technique for generating hierarchical sketches from English text. While there are many NLP techniques that can be used to solve this problem (including currently-popular *seq2seq* models), we frame it as an instance of semantic parsing and build our sketch generator on top of the SEMPRES framework [7]. As mentioned briefly in Section 1, we choose semantic parsing over deep learning techniques because it does not require as much labeled training data. However, our general synthesis methodology and the PBE algorithm are both agnostic to the NLP technique used for parsing English text into an h-sketch.

6.1 Background on semantic parsing

Semantic parsing is used for converting natural language to a formal representation, such as SQL [46, 47], lambda calculus [9], or natural logic [31]. This formal representation is often referred to as a *logical form*, and semantic parsers use a context-free grammar (CFG) to translate natural language to the target logical form. However, since natural language is highly complex and often very ambiguous, there are many possible logical forms that can be obtained from a given natural language description. Thus, modern semantic parsers also incorporate a machine learning model to score different parses for a given utterance. However, as mentioned earlier, these techniques still do not require as much labeled training data as other methods based on deep learning.

In the context of this work, logical forms correspond to hierarchical sketches, so our CFG needs to parse a given English utterance into an h-sketch. In the remainder of this section, we first give an overview of REGEL’s CFG (Section 6.2) and then discuss how to produce a *ranked* list of h-sketches using a machine learning model (Section 6.3).

6.2 Grammar-based sketch composition

Following standard convention, we specify our grammar rules in the following format:

<target category> *<target derivation>* → *<source sequence>*

Such a rule maps *<source sequence>* to a *<target derivation>* with category *<target category>*. Rules of the semantic parser can be further categorized into two groups, namely lexical rules and compositional rules. Examples of both types of rules are provided in Figure 13. A lexical rule maps a word in the sentence to base concepts in the DSL, including character class (e.g., lexical rule 1) and operator (e.g., lexical rule 4). A compositional rule combines one or more base components and builds larger h-sketches. For instance, as shown in Figure 13, compositional rule 2 is applied to generate a sketch $\square\{\langle\text{num}\rangle, \langle,\rangle\}$, labeled with category *\$SKETCH*, from a sequence of two derivations, $\langle\text{num}\rangle$ and \langle,\rangle , both labeled with *\$PROGRAM*, via the semantic function *SketchFn*. Here, we use category *\$SKETCH* to denote sketches containing holes and category *\$PROGRAM* to mark concrete regexes.

Given a set of pre-defined grammar rules and a natural language description \mathcal{L} , the semantic parser generates a list of possible derivations for \mathcal{L} . Each derivation can be mapped to an h-sketch deterministically, and, in general, multiple derivations of the same sentence can map to the same h-sketch. We construct the derivations for a given sentence recursively in a bottom-up fashion using dynamic programming. More specifically, we first apply lexical rules to generate derivations for any span (i.e., sequence of words) that they match. Then, the derivations of larger spans are constructed by applying compositional rules to derivations built over non-overlapping constituent spans. As the final output, we take derivations spanning the whole sentence that are labeled with a designated *\$ROOT* category.

Example 6.1. To build intuition, Figure 13 demonstrates the parsing process for the English phrase “*the max number of digits before comma is 15 then accept at max 3 numbers*”. Note that our parser allows skipping arbitrary words; thus, not every span in the description is used for building this derivation. Finally, we do not require applying every rule from Figure 13 when constructing this derivation, such as lexical rule 4 and compositional rule 5. Also observe that our grammar does not uniquely define an h-sketch for a given sentence. In particular, we can also obtain the following alternative h-sketch from the same text:

$$\text{Concat}(\square\{\langle \text{num} \rangle\}, \square\{\langle , \rangle, \text{Repeat}(\langle \text{num} \rangle, 3)\}) \quad (6)$$

6.3 Learning feature weights

Since there are many different h-sketches for an given English sentence, we need a way of scoring derivations so that h-sketches that are more consistent with the utterance are assigned a higher score. Towards this goal, our parser leverages a discriminative log-linear model using a set of *features* extracted from natural language. Specifically, given a derivation d from the set of possible derivations $D(\mathcal{L})$ for a description \mathcal{L} , we extract a feature vector $\phi(\mathcal{L}, d) \in \mathbb{R}^b$. The features are local to individual rules and are chosen to capture lexical, compositional, and semantic characteristics of the derivation and its sub-derivations. REGEL leverages two feature sets, namely *rule features* and *span features*, both of which are inherited from the SEMPRES framework. Concretely, a rule feature indicates whether a particular rule is fired during the derivation, and a span feature tracks the number of consecutive words that are used when generating a particular category in the derivation.

Given these extracted feature vectors, the probability that a derivation d is the intended sketch is given by:

$$P(d|\mathcal{L}) = \frac{\exp(\theta^\top \phi(\mathcal{L}, d))}{\sum_{d' \in D(\mathcal{L})} \exp(\theta^\top \phi(\mathcal{L}, d'))}$$

where $\theta \in \mathbb{R}^b$ is the vector of parameters to be learned. We learn these parameters with supervision from labeled training data, which consists of pairs (\mathcal{L}_i, h_i^*) where \mathcal{L}_i is the English description and h_i^* is a corresponding sketch label. During learning, we maximize the log probability of the system generating h_i^* regardless of derivation. In particular, given N training samples, our objective function is:

$$\max_{\theta} \log \sum_i^N \sum_{d: \text{sketch}(d)=h_i^*} P(d|\mathcal{L}_i)$$

Intuitively, the model increases the weight assigned to features for derivations that exactly match the annotated sketch.

In practice, $D(\mathcal{L})$ is a very large set of derivations, exponential with respect to the number of active lexical rules in the span. Therefore, we use beam search to find the approximate highest-scoring derivation. That is, instead of keeping all possible derivations for a span, we only keep a set of top- m derivations $D_m(\mathcal{L})$ according to their probabilities

and discard the rest. During training, we maximize the likelihood of the correct derivation with respect to this set; that is, normalizing over $D_m(\mathcal{L})$ rather than $D(\mathcal{L})$.

7 Implementation

We have implemented our synthesis algorithm in a new tool called REGEL⁷. In addition to the natural language description and positive/negative examples, REGEL takes two additional inputs, namely a time budget t and a parameter k that controls how many results to show to the user. The output of REGEL consists of up to k regexes that satisfy the examples. Note that the actual number of regexes returned by REGEL may be less than k due to the time budget.

REGEL is written in Java and leverages a number of other existing tools. First, our semantic parser is built on top of the SEMPRES framework [7] and leverages its existing functionalities, such as the linguistic pre-processor. Second, REGEL makes use of the Z3 SMT solver [12] for inferring possible values of the symbolic integers (recall Section 5.2). Finally, REGEL uses the Brics automaton library [33] for checking whether a string is matched by a regex.

The internal workflow of REGEL is as follows: First, the semantic parser generates up to 500 derivations for the given utterance and ranks them using the machine learning model. Then, we take the top 25 sketches produced by the parser and run 25 instances of the PBE engine *in parallel* to find a completion of each sketch that is consistent with the given examples. Then, given a value of k that can be specified by users, we wait for up to k PBE engine instances to complete their task and return the synthesized regexes for those tasks that terminate within the given time budget t .

Eliminating membership queries. For every concrete regex r explored by our synthesis algorithm, we need to check whether r matches all positive examples and rejects all negative ones. Thus, REGEL ends up issuing many regular language membership queries, some of which are quite expensive in practice. To reduce this overhead, our implementation uses various heuristics to eliminate unnecessary membership queries. For example, if we have determined that the regex `Contains(r)` does not match one of the positive examples, then we know that `StartsWith(r)` will also not match at least one of the examples. Similarly, if we have determined that the regex `RepeatAtLeast(r , 2)` does not match a positive example, we can conclude `RepeatAtLeast(r , k)` will not match the examples for any value of $k \geq 2$. Our implementation uses such “subsumption” heuristics to eliminate some of the redundant membership queries.

Eliminating redundant sketches. During semantic parsing, duplicate tokens in a span lead to many redundant derivations. We eliminate these duplicate sketches during beam search and keep the generated derivations non-identical.

⁷REGEL is publicly available at <https://github.com/utopia-group/regel>

8 Data Sets for Evaluation

To conduct our experiments, we collected two data sets, one of which is an adapted version of a data set used in DEEPREGEX [30] and another much more challenging data set curated from StackOverflow.

DeepRegex data set. As mentioned earlier, DEEPREGEX is a tool for generating regexes directly from natural language [30]. However, to evaluate our technique on the DEEPREGEX data set, we need positive and negative examples in addition to the English description. Thus, to adapt this data set to our setting, we took 200 benchmarks from this data set and asked users to provide positive and negative examples⁸. On average, each benchmark in this adapted DEEPREGEX data set contains 4 positive and 5 negative examples.

StackOverflow data set. To evaluate REGEL on more realistic string matching tasks encountered by real-world users, we collected a set of *much more challenging* benchmarks from StackOverflow. Specifically, we searched StackOverflow using relevant keywords, such as “*regex*”, “*regular expression*”, “*text validation*” etc. and retained *all* benchmarks that contain *both* an English description as well as positive *and* negative examples. Using this methodology, we obtained a total of 122 regex-related tasks and generated the ground-truth by directly converting the answer on StackOverflow to our DSL.

Training for each data set. As described in Section 6.3, our semantic parser is parametrized by a vector θ that is used for assigning scores to each possible derivation. Because these parameters are learned using supervision from labeled training data, we need training data for each data set in the form of pairs of English sentences and their corresponding h-sketches. However, since the original data sets are not annotated with hierarchical sketches, we had to construct the h-sketches used for training ourselves.

In general, the optimal h-sketch to use for training is hard to determine. On the one extreme, we can write an h-sketch that is exactly the target regex, but that would lead to poor performance of the semantic parser on the test set. On the other extreme, we can use a sketch that is completely unconstrained but that would be completely unhelpful for the PBE engine. To achieve a reasonable trade-off between these two extremes, we used the following strategy. For the DEEPREGEX dataset where the target regexes are relatively small and simple, we automatically generated the h-sketch by replacing the top-level (root) operator with a hole. For example, if the target regex is `Concat(<num>, <let>)`, our h-sketch used for training would be `□{<num>, <let>}`. While this strategy worked well for the DEEPREGEX dataset, it was not sufficiently fine-grained for the much more difficult StackOverflow benchmarks. Therefore, we manually constructed the h-sketches for the StackOverflow benchmarks by reading

the English description and expressing its high-level structure as an h-sketch. In many cases, our manually-written h-sketch faithfully captures the unambiguous parts of the English description (e.g., letter) but replaces ambiguous (or difficult to parse) fragments with holes.

Settings for each data set. Recall from Section 7 that REGEL is parametrized by two additional inputs t, k that control the time budget and number of results to display. For the easier DEEPREGEX data set, we set a time-out limit of 10 seconds and display only a single result. For the much harder StackOverflow benchmarks, we set the time budget to be 60 seconds and display the top 5 results. For performing comparisons, we use the same values of t and k across all tools and consider the benchmarks to be successfully solved if the intended regex is within the top k results.

9 Experimental Results

In this section, we describe a series of three experiments that are designed to answer the following research questions:

- **Q1:** What is the benefit of multi-modal synthesis? Does our approach work better compared to alternative approaches that use *only* examples or *only* natural language?
- **Q2:** How effective is our proposed PBE technique? In particular, how useful is sketch-guided deduction (Sec. 5.1) and SMT-based solving of symbolic regexes (Sec. 5.2)?
- **Q3:** Is REGEL helpful to users in constructing regular expressions for a given task?

All experiments are conducted on an Intel Xeon(R) E5-1620 v3 CPU with 32GB physical memory.

9.1 Benefits of multi-modal synthesis

To evaluate the benefits of leveraging two different specification modalities, we compare REGEL against two baselines. Our first baseline is DEEPREGEX which directly translates the natural language description into a regex using a sequence-to-sequence model [30]. Our second baseline is a variant of REGEL, henceforth referred to as REGEL-PBE, that only uses positive and negative examples. In particular, REGEL-PBE starts with a completely unconstrained sketch (i.e., single hole) and searches for a regex that satisfies the examples using the same algorithm described in Section 5.⁹

Since PBE tools are meant to be used interactively, we use the following methodology. First, we run both REGEL and REGEL-PBE on the initial examples in the original data set and consider synthesis to be successful if the intended regex is among those returned by the tool. If it is unsuccessful, in the next iteration, we provide two additional examples that are guaranteed to rule out the returned incorrect regex. We continue this process up to a maximum of four iterations.

⁸The details of this data set and the procedure for adapting it to our setting are described in the Appendix of an extended version of this paper[10].

⁹As we show in the next subsection, REGEL-PBE outperforms prior state-of-the-art regex PBE techniques; thus, we take REGEL-PBE as the representative state-of-the-art approach for synthesizing regular expressions purely from examples.

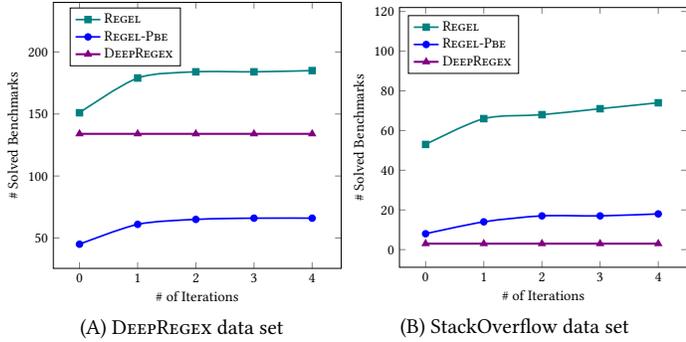


Figure 14. Number of solved benchmarks over iterations.

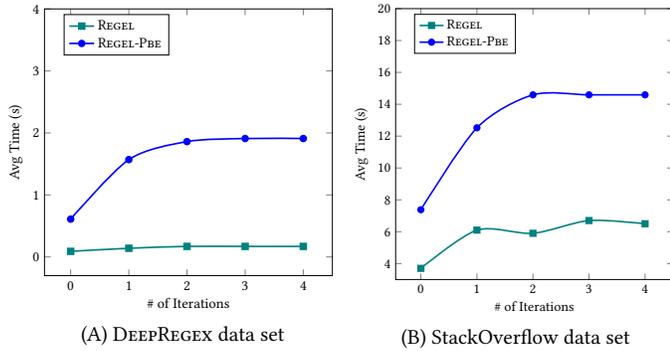


Figure 15. Average running time per solved benchmark over iterations. Time for DEEPREGEX’s *seq2seq* model is negligible.

Our results are summarized in Figures 14 and 15. For each figure, the *x*-axis shows the number of iterations and the *y*-axis shows either the number of benchmarks that can be successfully solved (Figure 14) or the average running time per benchmark (Figure 15). For each figure, (A) shows results for the DEEPREGEX data set and (B) is for StackOverflow. The green line (with squares) corresponds to REGEL, the blue line (with circles) is REGEL-PBE, and the violet line (with triangles) is DEEPREGEX. Because DEEPREGEX only takes natural language as input, the DEEPREGEX line in Figure 14 is flat. Furthermore, since DEEPREGEX does not involve any search, its running time is negligible and not shown in Figure 15.

DeepRegex data set. Let us first focus on the results for the DEEPREGEX data set, shown in Figure 14 (A) and Figure 15 (A). Given the original examples in this data set, REGEL can produce the *intended* regexes for 151 out of 200 benchmarks (75.5% accuracy). Furthermore, REGEL solves up to 185 benchmarks (92.5%) when more examples are available. In comparison, DEEPREGEX solves 134 benchmarks (67%), whereas REGEL-PBE solves at most 66 benchmarks (33%). Furthermore, as illustrated in Figure 15 (A), using the natural language specification also substantially speeds up the PBE engine.

StackOverflow data set. Next, we consider the StackOverflow results shown in Figure 14 (B). As expected, the accuracy is much lower compared to the DEEPREGEX data

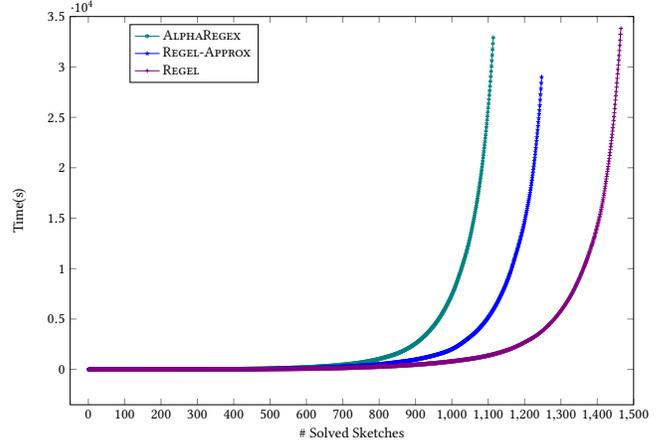


Figure 16. Number of solved sketches within a given time budget. For each StackOverflow benchmark, we take the top 25 sketches generated by the parser (or fewer than 25 if the parser does not generate 25).

set, as the StackOverflow benchmarks are much more challenging.¹⁰ Thus, the two baselines (namely, DEEPREGEX and REGEL-PBE) can only solve 3 (2.4%) and 18 benchmarks (14.7%) respectively, out of 122 benchmarks in total. In contrast, REGEL is able to solve up to 74 benchmarks out of 122 (60.7%).

Failure analysis for StackOverflow. To understand cases where REGEL does not work well, we investigate the StackOverflow benchmarks where REGEL fails to synthesize the intended regex. Among these benchmarks, we notice that many of the failure cases description rely on high-level concepts such as *date*, *range*, etc. that our semantic parser has no knowledge of; therefore, the generated sketch does not precisely capture the English description in most failure cases.

Result 1: Among 322 regex tasks, REGEL solves 80% of the benchmarks but DEEPREGEX solves only 43% and the PBE-baseline solves only 26%.

9.2 Evaluation of PBE engine

In this section, we describe an ablation study that allows us to quantify the impact of the pruning techniques described in Sections 5.1 and 5.2. Specifically, in Figure 16, we plot the number of solved sketches against cumulative running time for REGEL and two other baselines. In this context, a sketch is considered as solved if the PBE engine can find an instantiation of the sketch that is consistent with the examples. We now evaluate the following PBE engines:

- **ALPHA-REGEX:** The plot labeled ALPHA-REGEX is a baseline that implements the pruning techniques described in ALPHA-REGEX [27]. Specifically, we adapt ALPHA-REGEX to perform sketch-guided enumerative search (instead of

¹⁰In particular, the average number of words in a StackOverflow benchmark is 26 whereas DEEPREGEX benchmarks have 12 words on average. Furthermore, the average AST node size of the target regex is 13 for the StackOverflow data set and 5 for the DEEPREGEX data set.

breadth-first search) but use their pruning technique instead of the ideas proposed in Sections 5.1 and 5.2.

- **REGEL-APPROX**: This variant uses the pruning techniques described in Section 5.1 but does not leverage the symbolic regex idea introduced in Section 5.2.
- **REGEL**: This corresponds to the full REGEL system incorporating both ideas from Sections 5.1 and 5.2.

As we can see from Figure 16, both pruning techniques discussed in Sections 5.1 and 5.2 have a significant positive impact on the running time of the synthesizer.

Result 2: For the first 1000 sketches that can be solved by all variants, REGEL is around 10× faster than ALPHAREGEX and 2.5× faster than REGEL-APPROX.

9.3 User study

To further evaluate whether REGEL helps users complete regex-related tasks, we conducted a user study involving 20 participants, 5 of whom are professional software engineers and 15 of whom are computer science students. Each participant was provided with 6 regex tasks randomly sampled from the StackOverflow benchmarks, regardless of whether REGEL can that benchmark or not. Then, we provided each participant with the original task description in the StackOverflow post (including both the English description and the examples) and asked them to solve exactly a (randomly selected) half of the examples using REGEL and the remaining half without REGEL. For both setups, the users had a total of 15 minutes to work on each setting (with REGEL or without REGEL). More details about our user study setup can be found in the appendix in the extended version of the paper [10].

For the setup involving REGEL, participants were just provided with the tool and educated about how to use it, but they were not required to use REGEL in any specific way. Furthermore, while the participants were provided with the original StackOverflow post describing the task, they were free to modify both the English description and the examples as they saw fit.

Results. In the setup where participants did not have access to REGEL, they correctly solved 28.3% of the benchmarks (i.e., produced the intended regex) in the given time limit. In contrast, when they had access to REGEL, success rate went up to 73.3%. We ran a standard 1-tailed t -test to evaluate whether our results are statistically significant. The p -value for this test is less than 0.0000001. Thus, our user study provides firm evidence that the proposed technique makes it easier for users to write regexes.

Failure case analysis. To gain some insight about failure cases in the user study, we manually inspected those scenarios in which users were not able to successfully use REGEL to derive the correct regex. Overall, we found two main root causes for failure. First, because our tasks are randomly selected from the StackOverflow benchmarks, REGEL times out on some tasks and is unable to produce any regex. In

such cases, solving the benchmark with REGEL is no different from solving the benchmark without REGEL. Another main reason for failure is the inherent ambiguity in the StackOverflow post. That is, even with the provided examples, there may be multiple ways to interpret the question, so the users sometimes take one interpretation over the intended one and therefore select the wrong regex. (Note that users in our study were not provided with follow-up questions and discussions in the original StackOverflow post.)

Disclaimers. While we believe that our user study results provide some preliminary evidence of the potential usefulness of a REGEL-like approach, our results are not intended to be a scientific study of the use of REGEL “in the wild” for the following reasons. First, the majority of the participants in our user study are computer science students from the same university. Second, in order to allow a fair comparison between the two approaches across all participants, our tasks are taken from StackOverflow posts as opposed to real-world tasks that the participants *themselves* want to complete.

Result 3: For the particular setup evaluated in our small user study, REGEL users are 2× more likely to construct the correct regex using REGEL within a given time budget.

10 Related Work

In this section, we review prior work on program synthesis from examples and natural language.

Learning regexes from examples. There is a large body of prior research on learning regular expressions from positive and negative examples [4, 5, 16, 17, 38, 39, 42], including Angluin’s well-known L^* algorithm for active learning of regular expressions [6]. In this setting, a regular language is represented by an oracle that can answer membership queries, check for equivalence, and provide counterexamples. While these algorithms can learn the target language in polynomial time (with respect to the minimal DFA), they tend to require orders of magnitude more examples compared to our approach. For instance, for the simple regex $[A-Za-z]^+$, an implementation of the L^* algorithm asked 679 queries (2 equivalence and 677 membership queries) to synthesize the correct regex whereas REGEL-PBE was able to synthesize the desired regex using 8 examples without the natural language description.

More recent work that is closely related to our approach is ALPHAREGEX [27] which also performs top-down enumerative search and uses over- and under-approximations to prune the search space. However, ALPHAREGEX does not utilize natural language whereas we make use of the natural language to generate hierarchical sketches for both guiding the search and pruning infeasible regexes. Additionally, our method uses symbolic regexes and SMT-based reasoning to further prune the search space. Another related tool is RFIXER, which performs repair on regular expressions [37].

Rather than performing synthesis from scratch, RFIXER modifies a given regex to be consistent with the provided examples and also uses techniques similar to ALPHAREGEX to prune the search space.

Learning regexes from language. There has been recent interest in automatically generating regexes from natural language. For example, Kushman and Barzilay [25] build a dependency parser for translating natural language text queries into regular expressions. Their technique is built on top of a combinatory categorical grammar and utilizes semantic unification to improve training. Other work in this space uses seq2seq models to predict regular expressions from English descriptions [30, 49]. However, these techniques do not utilize examples and attempt to directly translate natural language into a regex rather than a sketch.

Multi-modal synthesis. There has been recent interest in synthesizing string manipulation programs from both natural language and examples. For instance, Manshadi et al. [32] propose a PBE system that leverages natural language in order to deduce the correct program more often and faster. Specifically, they use natural language to construct a so-called *probabilistic version space* and apply this idea to string transformations expressible in a subset of the FlashFill DSL [18]. Raza et al. [41] also use propose combining natural language and examples but do so in a very different way. Specifically, they try to decompose the English description into constituent concepts and then ask the user to provide examples for each concept in the decomposition.

Similar to our approach, there have also been recent proposals to combine natural language and examples using a sketching-based approach. For instance, [35] provides a framework for generating program sketches from any type of specification, which can also involve natural language. Specifically, they first use an LSTM to generate a distribution over program sketches and then try to complete the sketch using a generic sketch completion technique based on breadth-first enumeration. Another related effort in this space is the MARS tool which also utilizes natural language and examples [11]. In contrast to our technique, they derive soft constraints from natural language and utilize a MaxSMT solver to perform synthesis. In addition, MARS targets data wrangling applications rather than regexes.

PBE and sketching. Similar to this work, several recent PBE techniques combine top-down enumerative search with lightweight deductive reasoning to significantly prune the search space [3, 13–15, 26, 36, 45]. Our method also bears similarities to sketching-based approaches [28] in two ways: First, we generate some sort of program sketch from the natural language description. However, in contrast to prior work, our sketches are hierarchical in nature, and the holes in the sketch represent arbitrary regexes rather than constants. Second, we use a constraint-solving approach to infer constants in a symbolic regex. However, compared to most

existing techniques [8, 19, 24, 43], we use constraint solving as a way to rule out infeasible integer constants rather than directly solving for them.

Program synthesis from NL. Beyond regexes, there have also been proposals for performing program synthesis directly from natural language [23, 29, 34]. Such techniques have been used to generate SQL queries [23, 46], “if-this-then-that recipes” [40], spreadsheet formulas [20], bash commands [29], and Java expressions [21]. Our technique is particularly similar to SQLIZER [46] in that we also infer a sketch from the natural language description. However, unlike our approach, SQLIZER does not utilize examples and populates the sketch using a different technique called *quantitative type inhabitation* [22].

11 Conclusions and Future Work

In this paper, we presented a new method, and its implementation in a tool called REGEL, to synthesize regular expressions from a combination of examples and natural language. The key idea underlying our approach is to generate a hierarchical sketch from the English description and use the hints embedded in this sketch to guide both search and deduction. We evaluated our approach on 322 regexes obtained from two different sources and showed that our approach can successfully synthesize the intended regex in 80% of the cases within four user interaction steps. In comparison, a state-of-the-art tool that uses only natural language can solve 43% of these benchmarks and an example-only baseline can solve only 26%. We also performed an evaluation of our PBE engine and showed that REGEL is an order of magnitude faster compared to ALPHAREGEX, a state-of-the-art PBE tool for regex synthesis.

In future work, we are interested in exploring a multi-modal *active learning* approach to synthesizing regular expressions. In our current work, REGEL produces top- k results that satisfy the examples, but it is up to the user to inspect these results and provide more examples as needed. However, we believe it would be beneficial to develop a regex synthesis tool that would ask the user membership queries to disambiguate between multiple different solutions that are consistent with the examples. We are also interested in semantic parsing or other NLP techniques that might generate helpful feedback to users in cases where the generated sketch is too coarse. Finally, we plan to explore the use of the proposed synthesis methodology in application domains beyond regular expressions.

Acknowledgments

We thank our shepherd Manu Sridharan as well as our anonymous reviewers and members of the UToPiA group for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1762299, Grant No. CCF-1712067 and Grant No. CCF-1811865.

References

- [1] 2016. Class: Regexp (Ruby 2.4.0). <https://ruby-doc.org/core-2.4.0/Regexp.html>.
- [2] 2019. Pattern (Java Platform SE 8). <https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>.
- [3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification*. Springer, 934–950.
- [4] R. Alquezar and A. Sanfeliu. 1994. Incremental Grammatical Inference From Positive And Negative Data Using Unbiased Finite State Automata. In *In Proceedings of the ACL 2002 Workshop on Unsupervised Lexical Acquisition*. 291–300.
- [5] Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337 – 350.
- [6] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106.
- [7] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 1533–1544.
- [8] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 775–788.
- [9] Bob Carpenter. 1998. *Type-logical Semantics*. MIT Press, Cambridge, MA, USA.
- [10] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2019. Multi-modal Synthesis of Regular Expressions. arXiv:cs.PL/1908.03316
- [11] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 602–612. <https://doi.org/10.1145/3338906.3338951>
- [12] 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 (TACAS'08/ETAPS'08)*. Springer-Verlag, 337–340.
- [13] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 420–435.
- [14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. New York, NY, USA, 422–436.
- [15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 229–239.
- [16] Laura Firoiu, Tim Oates, and Paul R. Cohen. 1998. Learning Regular Languages from Positive Evidence. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*. 350–355.
- [17] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978), 302 – 320.
- [18] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, 317–330.
- [19] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73.
- [20] Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, 803–814.
- [21] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, 416–432.
- [22] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 27–38. <https://doi.org/10.1145/2491956.2462192>
- [23] Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen-tau Yih, and Xiaodong He. 2018. Natural Language to Structured Query Generation via Meta-Learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, 732–738.
- [24] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224.
- [25] Nate Kushman and Regina Barzilay. 2013. Using Semantic Unification to Generate Regular Expressions from Natural Language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 826–836.
- [26] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 542–553. <https://doi.org/10.1145/2594291.2594333>
- [27] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. ACM, 70–80.
- [28] A Solar Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation.
- [29] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*. European Language Resource Association. <http://aclweb.org/anthology/L18-1491>
- [30] Nicholas Locascio, Karthik Narasimhan, Eduardo De Leon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1918–1923.
- [31] Bill Maccartney. 2009. *Natural Language Inference*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Manning, Christopher D. AAI3364139.
- [32] Mehdi Manshadi, Daniel Gildea, and James Allen. 2013. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI Press, 661–667.
- [33] Anders Møller. 2017. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. <http://www.brics.dk/automaton/>.
- [34] Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. 2016. Learning a Natural Language Interface with Neural Programmer. *CoRR* abs/1611.08945 (2016). arXiv:1611.08945 <http://arxiv.org/abs/1611.08945>
- [35] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. *CoRR* abs/1902.06349 (2019). arXiv:1902.06349 <http://arxiv.org/abs/1902.06349>

- [36] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.
- [37] 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>
- [38] Rajesh Parekh and Vasant Honavar. 1996. An incremental interactive algorithm for regular grammar inference. In *Grammatical Interference: Learning Syntax from Sentences*, Laurent Miclet and Colin de la Higuera (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–249.
- [39] Rajesh Parekh and Vasant Honavar. 2001. Learning DFA from Simple Examples. *Machine Learning* 44, 1 (01 Jul 2001), 9–35. <https://doi.org/10.1023/A:1010822518073>
- [40] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 878–888.
- [41] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *IJCAI*.
- [42] R. L. Rivest and R. E. Schapire. 1989. Inference of Finite Automata Using Homing Sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing (STOC '89)*. ACM, 411–420.
- [43] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, 482–497.
- [44] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data Using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 195–213.
- [45] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 508–521.
- [46] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages.
- [47] John M. Zelle and Raymond J. Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2 (AAAI'96)*. AAAI Press, 1050–1055.
- [48] Luke S. Zettlemoyer and Michael Collins. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorical Grammars. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.
- [49] Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A Semantics-Based Approach for Generating Regular Expressions from Natural Language Specifications. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.