Synthesizing Program Input Grammars

Osbert Bastani
Stanford University, USA
obastani@cs.stanford.edu

Rahul Sharma
Microsoft Research, India
rahsha@microsoft.com

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

Percy Liang
Stanford University, USA
pliang@cs.stanford.edu

Abstract

We present an algorithm for synthesizing a context-free grammar encoding the language of valid program inputs from a set of input examples and blackbox access to the program. Our algorithm addresses shortcomings of existing grammar inference algorithms, which both severely overgeneralize and are prohibitively slow. Our implementation, GLADE, leverages the grammar synthesized by our algorithm to fuzz test programs with structured inputs. We show that GLADE substantially increases the incremental coverage on valid inputs compared to two baseline fuzzers.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords grammar synthesis; fuzzing

1. Introduction

Documentation of program input formats, if available in a machine-readable form, can significantly aid many software analysis tools. However, such documentation is often poor; for example, the specifications of Flex [61] and Bison [20] input syntaxes are limited to informal documentation. Even when detailed specifications are available, they are often not in a machine-readable form; for example, the specification for ECMAScript 6 syntax is 20 pages in Annex A of [15], and the specification for Java class files is 268 pages in Chapter 4 of [45].

In this paper, we study the problem of automatically synthesizing grammars representing program input languages. Such a grammar synthesis algorithm has many potential applications. Our primary motivation is the possibility of using synthesized grammars with grammar-based fuzzers [23, 28, 38]. For example, such inputs can be used to find bugs in real-world programs [24, 39, 48, 67], learn abstractions [41], predict performance [30], and aid dynamic analysis [42]. Beyond fuzzing, a grammar synthesis algorithm could be used to reverse engineer input formats [29], in particular, network protocol message formats can help security analysts discover vulnerabilities in network programs [8, 35, 36, 66]. Synthesized grammars could also be used to whitelist program inputs, thereby preventing exploits [49, 50, 58].

Approaches to synthesizing program input grammars typically examine executions of the program, and then generalize these observations to a representation of valid inputs. These approaches can be either *whitebox* or *blackbox*. Whitebox approaches assume that the program code is available for analysis and instrumentation, for example, using dynamic taint analysis [29]. Such an approach is difficult when only the program binaries are available or when parts of the code (e.g., libraries) are missing. Furthermore, these techniques often require program-specific configuration or tuning, and may be affected by the structure of the code. We consider the blackbox setting, where we only require the ability to execute the program on a given input and observe its corresponding output. Since the algorithm does not examine the program's code, its performance depends only on the language of valid inputs, and not on implementation details.

A number of existing language inference algorithms can be adapted to this setting [14]. However, we found them to be unsuitable for synthesizing program input grammars. In particular, L-Star [3] and RPNI [44], the most widely studied algorithms [6, 12, 13, 19, 62], were unable to learn or approximate even simple input languages such as XML, and furthermore do not scale even to small sets of seed inputs. Surprisingly, we found that L-Star and RPNI perform poorly even on the class of regular languages they target.

The problem with these algorithms is that despite having theoretical guarantees, they depend on assumptions that do not hold in the setting of learning program input grammars. For example, they typically avoid overgeneralizing by relying on an "oracle" to provide negative examples that are used by the algorithm to identify and remove overly general portions of the language. However, these oracles are not available in our setting—e.g., L-Star obtains such examples from an equivalence oracle, and RPNI obtains them "in the limit". They likewise assume that positive examples exercising all interesting behaviors are provided by this oracle. In our setting, the needed positive and negative examples are difficult to find, and existing algorithms consistently overgeneralize (e.g., return Σ^*) or undergeneralize (e.g., return $\emptyset$). Additionally, despite having polynomial running time, they can be very slow on our problem instances. To the best of our knowledge, other existing grammar inference algorithms are either impractical [14, 33] or make assumptions similar to L-Star and RPNI [31].

This paper presents the first practical algorithm for synthesizing program input grammars in the blackbox setting. Our algorithm synthesizes a context-free grammar $\hat{C}$ encoding the language L_* of valid program inputs, given

- A small set of *seed inputs* $E_{\text{in}} \subseteq L_*$ (i.e., examples of valid inputs). Typically, seed inputs are readily available—in our evaluation, we use small test suites that come with programs or examples from documentation.
- Blackbox access to the program executable to answer *membership queries* (i.e., whether a given input is valid).

Our algorithm adopts a high-level design commonly used by language learning algorithms (e.g., RPNI)—it starts with the language containing exactly the given positive examples, and then incrementally generalizes this language, using negative examples to avoid overgeneralizing. Our algorithm avoids the shortcomings of existing algorithms in two ways:

- It considers a much richer set of potential generalizations, which addresses the issue of omitted positive examples.
- It generates negative examples on the fly to avoid overgeneralizing, which addresses the issue of omitted negative examples.

In particular, our algorithm constructs a series of increasingly general languages using *generalization steps*. Each step first proposes a number of candidate languages that generalize the current language, and then uses carefully crafted membership queries to reject candidates that overgeneralize. Our algorithm considers candidates that (i) add repetition and alternation constructs characteristic of regular expressions, (ii) induce recursive productions characteristic of context-free grammars, in particular, parentheses matching grammars, and (iii) generalize constants in the grammar.

We implement our approach in a tool called GLADE,[1]. We conduct an extensive empirical evaluation of GLADE (Section 8), and show that GLADE substantially outperforms both L-Star and RPNI, even when restricted to synthesizing regular expressions. Furthermore, we show that GLADE successfully synthesizes input grammars for real programs, which can be used to fuzz test those programs. In particular, GLADE automatically synthesizes a program input grammar, and then uses the synthesized grammar in conjunction with a standard grammar-based fuzzer (described in Section 8.3) to generate new test inputs. Many fuzzing applications require valid inputs, for example, differential testing [67]. We show that when restricted to generating valid inputs, GLADE increases line coverage compared to both a naïve fuzzer and a production fuzzer afl-fuzz [68]. Our contributions are:

- We introduce an algorithm for synthesizing program input grammars from seed inputs and blackbox program access (Section 3). Our algorithm first learns regular properties such as repetitions and alternations (Section 4), and then learns recursive productions characteristic of matching parentheses grammars (Section 5).
- We implement our grammar synthesis algorithm in a tool called GLADE, and show that GLADE outperforms two widely studied language learning algorithms, L-Star and RPNI, in our application domain (Section 8.2).
- We use GLADE to fuzz test programs, showing that it increases the number of newly covered lines of code using valid inputs by up to 6× compared to two baseline fuzzers (Section 8.3).

2. Problem Formulation

Suppose we are given a program that takes inputs in Σ^*, where Σ is the input alphabet (e.g., ASCII characters). We let $L_* \subseteq \Sigma^*$ denote the *target language* of valid program inputs; typically, L_* is a highly structured subset of Σ^*. Our goal is to synthesize a language $\hat{L}$ approximating L_* from blackbox program access and seed inputs $E_{\text{in}} \subseteq L_*$. We represent blackbox program access as an oracle $\mathcal{O}$ such that $\mathcal{O}(\alpha) = \mathbb{I}[\alpha \in L_*]$ (here, $\mathbb{I}$ is the indicator function, so $\mathbb{I}[\mathcal{C}]$ is 1 if $\mathcal{C}$ is true and 0 otherwise). In particular, we run the program on input $\alpha \in \Sigma^*$, and conclude that α is a valid input (i.e., $\alpha \in L_*$) if the program does not print an error message. Access to the oracle is crucial to avoid overgeneralizing, e.g., rejecting $\hat{L} = \Sigma^*$, whereas the seed inputs give a starting point from which to generalize.

As a running example, suppose the program input language is the XML-like grammar C_{XML} shown in Figure 1. We use $+$ to denote alternations and $*$ (the Kleene star) to denote repetitions. Terminals that are part of regular expressions or context-free grammars are highlighted in blue. Given seed input α_{XML} and oracle $\mathcal{O}_{\text{XML}}$, our goal is to synthesize a language $\hat{L}$ approximating $L_* = \mathcal{L}(C_{\text{XML}})$.

Ideally, we would learn L_* exactly, i.e., $\hat{L} = L_*$, but it is impossible to guarantee exact learning [25]. Instead, we want $\hat{L}$ to be a good approximation of L_*. To measure the

[1] GLADE stands for Grammar Learning for AutomateD Execution, and is available at `https://github.com/obastani/glade`.

- Target language $\mathcal{L}(C_{\text{XML}})$, where the context-free grammar C_{XML} has terminals $\Sigma_{\text{XML}} = \{\texttt{a}, ..., \texttt{z}, \texttt{<}, \texttt{>}, \texttt{/}\}$, start symbol A_{XML}, and production

$$A_{\text{XML}} \to (\texttt{a} + ... + \texttt{z} + \texttt{<a>} A_{\text{XML}} \texttt{</a>})^*$$

- Oracle $\mathcal{O}_{\text{XML}}(\alpha) = \mathbb{I}[\alpha \in \mathcal{L}(C_{\text{XML}})]$
- Seed inputs $E_{\text{XML}} = \{\alpha_{\text{XML}}\}$, where $\alpha_{\text{XML}} = \texttt{<a>hi</a>}$

Figure 1. A context-free language $\mathcal{L}(C_{\text{XML}})$ of XML-like strings, along with an oracle $\mathcal{O}_{\text{XML}}$ for this language and a seed input α_{XML}.

approximation quality, we require probability distributions over L_* and $\hat{L}$. In Section 8.1, we define the distributions we use in detail. Briefly, we convert the context-free grammar into a *probabilistic context-free grammar*, and use the distribution induced by sampling strings in this probabilistic grammar. Then, we measure the quality of $\hat{L}$ as follows:

DEFINITION 2.1. Let $\mathcal{P}_{L_*}$ and $\mathcal{P}_{\hat{L}}$ be probability distributions over L_* and $\hat{L}$, respectively. The *precision* of $\hat{L}$ is $\Pr_{\alpha \sim \mathcal{P}_{\hat{L}}}[\alpha \in L_*]$ and the *recall* of $\hat{L}$ is $\Pr_{\alpha \sim \mathcal{P}_{L_*}}[\alpha \in \hat{L}]$ (here, $\alpha \sim \mathcal{P}$ denotes a random sample from $\mathcal{P}$).

For high precision, a randomly sampled string $\alpha \sim \mathcal{P}_{\hat{L}}$ must be valid with high probability, i.e., $\alpha \in L_*$. For high recall, $\hat{L}$ must contain a randomly sampled valid string $\alpha \sim \mathcal{P}_{L_*}$ with high probability. Both are desirable: $\hat{L} = \{\alpha_{\text{in}}\}$ has perfect precision but typically low recall, whereas $\hat{L} = \Sigma^*$ has perfect recall but typically low precision. Finally, while the synthesized language $\hat{L}$ is context-free, it is often possible for $\hat{L}$ to approximate L_* with high precision and recall even if L_* is not context-free (e.g., L_* is context-sensitive).

3. Overview

In this section, we give an overview of our grammar synthesis algorithm (summarized in Algorithm 1). We consider the case where E_{in} consists of a single seed input $\alpha_{\text{in}} \in L_*$; an extension to multiple seed inputs is given in Section 6.1. Our algorithm starts with the language $\hat{L}_1 = \{\alpha_{\text{in}}\}$ containing only the seed input, and constructs a series of languages

$$\{\alpha_{\text{in}}\} = \hat{L}_1 \Rightarrow \hat{L}_2 \Rightarrow ...,$$

where $\hat{L}_{i+1}$ results from applying a *generalization step* to $\hat{L}_i$. On one hand, we want the languages to become successively larger (i.e., $\hat{L}_i \subseteq \hat{L}_{i+1}$); on the other hand, we want to avoid overgeneralizing (ideally, the newly added strings $\hat{L}_{i+1} \setminus \hat{L}_i$ should be contained in L_*). Our framework returns the current language $\hat{L}_i$ if it is unable to generalize $\hat{L}_i$ in any way. Figure 2 shows the series of languages constructed by our algorithm for the example in Figure 1. Steps R1-R9 (detailed in Section 4) generalize the initial language $\hat{L}_1 = \{\alpha_{\text{XML}}\}$ by adding repetitions and alternations. Steps C1-C2 (detailed in Section 5) add recursive productions.

We now describe generalization steps at a high level.

Algorithm 1 Our grammar synthesis algorithm. Given seed input $\alpha_{\text{in}} \in L_*$ and oracle $\mathcal{O}$ for L_*, it returns an approximation of L_*.

```
procedure LEARNLANGUAGE($\alpha_{\text{in}}, \mathcal{O}$)
    $\hat{L}_{\text{current}} \leftarrow \{\alpha_{\text{in}}\}$
    while true do
        $M \leftarrow$ CONSTRUCTCANDIDATES($\hat{L}_{\text{current}}$)
        $\tilde{L}_{\text{chosen}} \leftarrow \emptyset$
        for all $\tilde{L} \in M$ do
            $S \leftarrow$ CONSTRUCTCHECKS($\hat{L}_{\text{current}}, \tilde{L}$)
            if CHECKCANDIDATE($S, \mathcal{O}$) then
                $\tilde{L}_{\text{chosen}} \leftarrow \tilde{L}$
                break
            end if
        end for
        if $\tilde{L}_{\text{chosen}} = \emptyset$ then
            return $\hat{L}_{\text{current}}$
        end if
        $\hat{L}_{\text{current}} \leftarrow \tilde{L}_{\text{chosen}}$
    end while
end procedure
procedure CHECKCANDIDATE($S, \mathcal{O}$)
    for all $\alpha \in S$ do
        if $\mathcal{O}(\alpha) = 0$ then
            return false
        end if
    end for
    return true
end procedure
```

Candidates. The ith generalization step first constructs *candidate* languages $\tilde{L}_1, ..., \tilde{L}_n$, with the goal of choosing $\hat{L}_{i+1}$ to be the candidate that increases recall the most without sacrificing precision. To ensure candidates can only increase recall, we consider *monotone* candidates $\tilde{L} \supseteq \hat{L}_i$. Furthermore, the candidates are ranked from most preferable ($\tilde{L}_1$) to least preferable ($\tilde{L}_n$). Figure 2 shows the candidates considered for our running example. They are listed in order of preference, with the top candidate being the most preferred. In steps R1-R9, the candidates add a single repetition or alternation to the current regular expression; in steps C1-C2, the candidates try to equate nonterminals in the current context-free grammar.

Checks. To ensure high precision, we want to avoid overgeneralizing. Ideally, we want to select a candidate that is *precision-preserving*, i.e., $\tilde{L} \setminus \hat{L}_i \subseteq L_*$. In other words, all strings added to the candidate $\tilde{L}$ (compared to the current language $\hat{L}_i$) are contained in the target language L_*. However, we only have access to a membership oracle for L_*, so it is typically impossible to prove that a given candidate $\tilde{L}$ is precision-preserving—we would have to check $\mathcal{O}(\alpha) = 1$ for every $\alpha \in \tilde{L} \setminus \hat{L}_i$, but this set is often infinite.

Instead, we carefully choose a finite number of heuristic *checks* $S \subseteq \tilde{L} \setminus \hat{L}_i$. Then, our algorithm rejects $\tilde{L}$ if $\mathcal{O}(\alpha) = 0$ for any $\alpha \in S$. Alternatively, if all checks pass (i.e., $\mathcal{O}(\alpha) = 1$), then $\tilde{L}$ is *potentially precision-preserving*. Since the candidates are ranked in order of preference, we choose the first potentially precision-preserving candidate. Figure 2 shows examples of checks our algorithm constructs.

Step	Language	Candidates	Checks
R1	$[\texttt{<a>hi</a>}]_{\text{rep}}$	$\star\ ([\texttt{<a>hi</a>}]_{\text{alt}})^*$	$\{\epsilon$ ✓, `<a>hi</a><a>hi</a>` ✓$\}$
		$([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}}$	$\{$`<a>hi</a` ✗, `<a>hi</a<a>hi</a>` ✗$\}$
		…	…
		$\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{</a>}]_{\text{rep}}$	$\{$`<a></a>` ✓, `<a>hihi</a>` ✓$\}$
		…	…
R2	$([\texttt{<a>hi</a>}]_{\text{alt}})^*$	$([\texttt{<}]_{\text{rep}} + [\texttt{a>hi</a>}]_{\text{alt}})^*$	$\{$`<` ✗, `a>hi</a>` ✗$\}$
		…	…
		$\star\ ([\texttt{<a>hi</a>}]_{\text{rep}})^*$	$\emptyset$
R3	$([\texttt{<a>hi</a>}]_{\text{rep}})^*$	$(([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}})^*$	$\{$`<a>hi</a` ✗, `<a>hi</a<a>hi</a>` ✗$\}$
		…	…
		$\star\ (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{</a>}]_{\text{rep}})^*$	$\{$`<a></a>` ✓, `<a>hihi</a>` ✓$\}$
		…	…
R4	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{</a>}]_{\text{rep}})^*$	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*([\texttt{</a>}]_{\text{alt}})^*)^*$	$\{$`<a>hi` ✗, `<a>hi</a></a>` ✗$\}$
		…	…
		$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{</a}([\texttt{>}]_{\text{alt}})^*)^*$	$\{$`<a>hi</a` ✗, `<a>hi</a>>` ✗$\}$
		$\star\ (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{</a>})^*$	$\emptyset$
R5	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{</a>})^*$	$\star\ (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{</a>})^*$	$\{$`<a>h</a>` ✓, `<a>i</a>` ✓$\}$
		$(\texttt{<a>}([\texttt{hi}]_{\text{rep}})^*\texttt{</a>})^*$	$\emptyset$
R6	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{</a>})^*$	$\star\ (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{</a>})^*$	$\emptyset$
R7	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{</a>})^*$	$\star\ (\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{</a>})^*$	$\emptyset$
R8	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{</a>})^*$	$\star\ (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>})^*$	$\emptyset$
R9	$(\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>})^*$	–	–
C1	$\left(\begin{matrix} A'_{\text{R1}} \to (\texttt{<a>}A'_{\text{R3}}\texttt{</a>})^* \\ A'_{\text{R3}} \to (\texttt{h}+\texttt{i})^* \end{matrix}, \{(A'_{\text{R1}}, A'_{\text{R3}})\}\right)$	$\star \left(\begin{matrix} A \to (\texttt{<a>}A\texttt{</a>})^* \\ A \to (\texttt{h}+\texttt{i})^* \end{matrix}, \emptyset\right)$	$\{$`hihi` ✓, `<a><a>hi</a><a>hi</a></a>` ✓$\}$
		$\left(\begin{matrix} A'_{\text{R1}} \to (\texttt{<a>}A'_{\text{R3}}\texttt{</a>})^* \\ A'_{\text{R3}} \to (\texttt{h}+\texttt{i})^* \end{matrix}, \emptyset\right)$	$\emptyset$
C2	$\left(\begin{matrix} A \to (\texttt{<a>}A\texttt{</a>})^* \\ A \to (\texttt{h}+\texttt{i})^* \end{matrix}, \emptyset\right)$	–	–

Figure 2. The generalization steps taken by our algorithm given seed input α_{XML} and oracle $\mathcal{O}_{\text{XML}}$. The initial language $\{\alpha_{\text{XML}}\}$ is generalized to a regular expression in steps R1-R9. The resulting regular expression is translated to a context-free grammar, which is further generalized in steps C1-C2. The candidates at each step are shown in order of preference, with the most preferable on top (ellipses indicate omitted candidates). Checks for each candidate are shown; a green check mark ✓ indicates that the check passes and a red cross ✗ indicates that it fails. A star $\star$ is shown next to the selected candidate.

4. Phase One: Regular Expression Synthesis

We describe the first phase of generalization steps, which generalize the seed input into a regular expression.

4.1 Candidates

In phase one, the current language is represented by a regular expression annotated with extra data: substrings of terminals $\alpha = \sigma_1 ... \sigma_k$ may be enclosed in square brackets, i.e., $[\alpha]_\tau$, where $\tau \in \{\text{rep}, \text{alt}\}$. These annotations indicate that the bracketed substring in the current regular expression can be generalized by adding either a repetition (if $\tau = \text{rep}$) or an alternation (if $\tau = \text{alt}$). The seed input α_{in} is automatically annotated as $[\alpha_{\text{in}}]_{\text{rep}}$. Then, each generalization step selects a single bracketed substring $[\alpha]_\tau$ and generates candidates based on *decompositions* of α (i.e., an expression of α as a sequence of substrings $\alpha = \alpha_1 ... \alpha_k$):

- **Repetitions:** If generalizing $P[\alpha]_{\text{rep}}Q$, for each decomposition $\alpha = \alpha_1\alpha_2\alpha_3$ such that $\alpha_2 \neq \epsilon$, generate
$$P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q.$$
- **Alternations:** If generalizing $P[\alpha]_{\text{alt}}Q$, for each decomposition $\alpha = \alpha_1\alpha_2$, where $\alpha_1 \neq \epsilon$ and $\alpha_2 \neq \epsilon$, generate
$$P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q.$$

In both cases, the candidate $P\alpha Q$ is also generated. For example, in Figure 2, step R1 selects $[\texttt{<a>hi</a>}]_{\text{rep}}$ and applies the repetition rule.

The candidates are monotonic (proven in Appendix A.1):

PROPOSITION 4.1. Each candidate constructed in phase one of our algorithm is monotone.

We briefly describe the intuition behind these rules. In particular, we define a *meta-grammar*[2] $\mathcal{C}_{\text{regex}}$, which is a context-free grammar whose members $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ are regular expressions. The terminals of $\mathcal{C}_{\text{regex}}$ are $\Sigma_{\text{regex}} = \Sigma \cup \{+, *\}$, where $+$ denotes alternations and $*$ denotes repetitions. The nonterminals are $\mathcal{V}_{\text{regex}} = \{T_{\text{rep}}, T_{\text{alt}}\}$, where T_{rep} corresponds to repetitions (and is also the start symbol) and T_{alt} corresponds to alternations. The productions are

$$T_{\text{rep}} ::= \beta \mid T_{\text{alt}}^* \mid \beta T_{\text{alt}}^* \mid T_{\text{alt}}^* T_{\text{rep}} \mid \beta T_{\text{alt}}^* T_{\text{rep}}$$
$$T_{\text{alt}} ::= T_{\text{rep}} \mid T_{\text{rep}} + T_{\text{alt}}$$

where $\beta \in \Sigma^* - \{\epsilon\}$ ranges over nonempty substrings of α_{in}.

Consider the series of regular expressions $R_1 \Rightarrow ... \Rightarrow R_n$ in phase one. For each regular expression, we can replace each bracketed substring $[\alpha]_\tau$ with the nonterminal T_τ.

[2] We use the term *meta-grammar* to distinguish $\mathcal{C}_{\text{regex}}$ from the context-free grammars we synthesize.

Doing so produces a derivation in $\mathcal{C}_{\text{regex}}$, for example, steps R1-R9 in Figure 2 correspond to the derivation:

$$\begin{array}{ll}
[\texttt{<a>hi</a>}]_{\text{rep}} & T_{\text{rep}} \\
\Rightarrow ([\texttt{<a>hi</a>}]_{\text{alt}})^* & \Rightarrow T_{\text{alt}}^* \\
\Rightarrow ([\texttt{<a>hi</a>}]_{\text{rep}})^* & \Rightarrow T_{\text{rep}}^* \\
\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{</a>}]_{\text{rep}})^* & \Rightarrow (\texttt{<a>}T_{\text{alt}}^*T_{\text{rep}})^* \\
\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{</a>})^* & \Rightarrow (\texttt{<a>}T_{\text{alt}}^*\texttt{</a>})^* \\
\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{</a>})^* & \Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{alt}})^*\texttt{</a>})^* \\
\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{</a>})^* & \Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{rep}})^*\texttt{</a>})^* \\
\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{</a>})^* & \Rightarrow (\texttt{<a>}(T_{\text{rep}} + \texttt{i})^*\texttt{</a>})^* \\
\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>})^* & \Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>})^*
\end{array}$$

In fact, this correspondence goes backwards as well:

PROPOSITION 4.2. For any derivation $T_{\text{rep}} \overset{*}{\Rightarrow} R$ in $\mathcal{C}_{\text{regex}}$ (where $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$), there exists $\alpha_{\text{in}} \in \mathcal{L}(R)$ such that R can be derived from α_{in} via a series of generalization steps

$$\{\alpha_{\text{in}}\} = R_1 \Rightarrow ... \Rightarrow R_n = R$$

We give a proof in Appendix B.1. Furthermore, $\mathcal{L}(\mathcal{C}_{\text{regex}})$ almost contains every regular expression:

PROPOSITION 4.3. For any regular language L_*, there exist $R_1, ..., R_m \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ such that $L_* = \mathcal{L}(R_1 + ... + R_m)$.

We give a proof in Appendix B.2. In other words, phase one can synthesize almost any regular language L_*, assuming the "right" sequence of generalization steps is taken. Our extension to multiple inputs in Section 6.1 extends this result to any regular language. However, the space of all regular expressions is too large to search exhaustively. We sacrifice completeness for efficiency—our algorithm greedily chooses the first candidate according to the candidate ordering described in Section 4.2.

The productions in $\mathcal{C}_{\text{regex}}$ are unambiguous, so each regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ has a single valid parse tree. This disambiguation allows our algorithm to avoid considering candidate regular expressions multiple times.

4.2 Candidate Ordering

The candidate ordering is a heuristic designed to maximize the generality of the regular expression synthesized at the end of phase one. We use the following ordering for candidates constructed by phase one generalization steps:

- **Repetitions:** If generalizing $P[\alpha]_{\text{rep}}Q$, among

  $$P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q,$$

  we first prioritize shorter α_1, since α_1 is not further generalized. Second, we prioritize longer α_2—for example, in step R3 of Figure 2, if we instead chose candidate $\texttt{<a>}([\texttt{h}]_{\text{alt}})^*[\texttt{i</a>}]_{\text{rep}}$, then we would synthesize $(\texttt{<a>h}^*\texttt{i}^*\texttt{</a>})^*$, which is less general than step R9.

- **Alternations:** If generalizing $P[\alpha]_{\text{alt}}Q$, among

  $$P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q,$$

  we prioritize shorter α_1—for example, in step R5 of Figure 2, if we instead chose candidate $(\texttt{<a>}([\texttt{hi}]_{\text{rep}})^*\texttt{</a>})^*$, then step R6 would instead be $(\texttt{<a>}([\texttt{hi}]_{\text{rep}})^*\texttt{</a>})^*$, which is less general than the one we obtain.

In either case, the final candidate $P\alpha Q$ is ranked last. Note that candidate repetitions and candidate alternations can be ordered independently—each generalization step considers only repetitions (if the chosen bracketed string has form $[\alpha]_{\text{rep}}$) or only alternations (if it has form $[\alpha]_{\text{alt}}$).

4.3 Check Construction

We describe how phase one of our algorithm constructs checks $S \subseteq \tilde{L} \setminus \hat{L}_i$. Each check $\alpha \in S$ has form $\alpha = \gamma\rho\delta$, where ρ is a *residual* capturing the portion of $\tilde{L}$ that is generalized compared to $\hat{L}_i$, and (γ, δ) is a *context* capturing the portion of $\tilde{L}$ which is in common with $\hat{L}_i$. More precisely, suppose the current language is $P[\alpha]_\tau Q$, where $[\alpha]_\tau$ is chosen to be generalized, and the candidate language is $PR_\alpha Q$, i.e., α is generalized to R_α. Then, a residual $\rho \in \mathcal{L}(R_\alpha) \setminus \{\alpha\}$ captures how R_α is generalized compared to the substring α, and a context (γ, δ) captures the semantics of the expressions (P, Q).

We may want to choose $\gamma \in \mathcal{L}(P)$ and $\delta \in \mathcal{L}(Q)$. However, P and Q may not be regular expressions. For example, on step R5 in Figure 2, $P =$ "$(\texttt{<a>}$", $\alpha =$ "$\texttt{hi}$", and $Q =$ "$\texttt{</a>})^*$" (the expressions are quoted to emphasize the placement of parentheses). Instead, P and Q form a regular expression when sequenced together, possibly with a string α' in between, i.e., $P\alpha'Q$. We want contexts (γ, δ) such that

$$\gamma\alpha'\delta \in \mathcal{L}(P\alpha'Q) \qquad (\forall \alpha' \in \Sigma^*). \tag{1}$$

Then, the constructed check $\alpha = \gamma\rho\delta$ satisfies

$$\gamma\rho\delta \in \mathcal{L}(P\rho Q) \subseteq \mathcal{L}(PR_\alpha Q),$$

where the first inclusion follows from (1) and the second inclusion follows since $\rho \in \mathcal{L}(R_\alpha)$. We discard α such that $\alpha \in \mathcal{L}(\hat{L}_i)$ to obtain valid checks $\alpha \in \tilde{L} \setminus \hat{L}_i$.

Next, we explain the construction of residuals and contexts. Our algorithm generates residuals as follows:

- **Repetitions:** For current language $P[\alpha]_{\text{rep}}Q$ and candidate $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, generate residuals $\alpha_1\alpha_3$ and $\alpha_1\alpha_2\alpha_2\alpha_3$.
- **Alternations:** For current language $P[\alpha]_{\text{alt}}Q$ and candidate $P(\alpha_1 + \alpha_2)Q$, generate residuals α_1 and α_2.

Next, our algorithm associates a context (γ, δ) with each bracketed string $[\alpha]_\tau$. The context for the initial bracketed string $[\alpha_{\text{in}}]_{\text{rep}}$ is (ϵ, ϵ). After each generalization step, contexts for new bracketed substrings are generated:

- **Repetitions:** For current language $P[\alpha]_{\text{rep}}Q$, where $[\alpha]_{\text{rep}}$ has context (γ,δ), and candidate $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, the context generated for the new bracketed substring $[\alpha_2]_{\text{alt}}$ is $(\gamma\alpha_1,\alpha_3\delta)$, and for $[\alpha_3]_{\text{rep}}$ is $(\gamma\alpha_1\alpha_2,\delta)$.
- **Alternations:** For current language $P[\alpha]_{\text{alt}}Q$, where $[\alpha]_{\text{alt}}$ has context (γ,δ), and candidate $P([\alpha_1]_{\text{rep}}+[\alpha_2]_{\text{alt}})Q$, the context generated for the new bracketed substring $[\alpha_1]_{\text{rep}}$ is $(\gamma,\alpha_2\delta)$, and for $[\alpha_2]_{\text{alt}}$ is $(\gamma\alpha_1,\delta)$.

For example, on step R3, the context for $[\texttt{<a>hi</a>}]_{\text{rep}}$ is (ϵ,ϵ). The residuals for candidate $(([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}})^*$ are `<a>hi</a` and `<a>hi</a>>`; since the context is empty, these residuals are also the checks, and they are rejected by the oracle, so the candidate is rejected. On the other hand, the residuals (and checks) for the chosen candidate $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{</a>}]_{\text{rep}})^*$ are `<a></a>` and `<a>hihi</a>`, which are accepted by the oracle. For the new bracketed string $[\texttt{hi}]_{\text{alt}}$, the algorithm constructs the context $(\texttt{<a>},\texttt{</a>})$, and for the new bracketed string $[\texttt{</a>}]_{\text{rep}}$, the algorithm constructs the context $(\texttt{<a>hi},\epsilon)$.

Similarly, on step R5, the context for $[\texttt{hi}]_{\text{alt}}$ is $(\texttt{<a>},\texttt{</a>})$. The residuals constructed for the chosen candidate $(\texttt{<a>}([\texttt{h}]_{\text{rep}}+[\texttt{i}]_{\text{alt}})^*\texttt{</a>})^*$ are `h` and `i`, so the constructed checks are `<a>h</a>` and `<a>i</a>`. Our algorithm constructs the context $(\texttt{<a>},\texttt{i</a>})$ for the new bracketed string $[\texttt{h}]_{\text{rep}}$ and the context $(\texttt{<a>h},\texttt{</a>})$ for the new bracketed string $[\texttt{i}]_{\text{alt}}$.

We have the following result:

PROPOSITION 4.4. The contexts constructed by phase one generalization steps satisfy (1).

We give a proof in Appendix A.2, which ensures that the constructed checks are valid (i.e., belong to $\tilde{L}\setminus\hat{L}_i$).

4.4 Computational Complexity

Let n be the length of the seed input α_{in}. In phase one, our algorithm considers at most $O(n^2)$ repetition candidates (since each of the n^2 substrings of α_{in} is considered at most once), and $O(n^3)$ alternation candidates (since at most $O(n)$ alternation candidates are considered per discovered repetition). Examining each candidate takes constant time (assuming each query to $\mathcal{O}$ takes constant time), so the complexity of phase one is $O(n^3)$. In our evaluation, we show that our algorithm is quite scalable.

5. Phase Two: Recursive Properties

The second phase of generalization steps learn recursive properties of program input languages that cannot be represented using regular expressions. Consider the regular expression $(\texttt{<a>}(\texttt{h}+\texttt{i})^*\texttt{</a>})^*$ obtained at the end of phase one in Figure 2, which can be written as $\hat{R}_{\text{XML}}=(\texttt{<a>}R_{\text{hi}}\texttt{</a>})^*$, where $R_{\text{hi}}=(\texttt{h}+\texttt{i})^*$. Since every regular language is also context-free, we can begin by translating $\hat{R}_{\text{XML}}$ to the context-free grammar

$$\{A_{\text{XML}}\to(\texttt{<a>}A_{\text{hi}}\texttt{</a>})^*,\ A_{\text{hi}}\to(\texttt{h}+\texttt{i})^*\}.$$

Then, we can equate the nonterminals A_{XML} and A_{hi} to obtain the context-free grammar $\hat{C}_{\text{XML}}$:

$$\{A\to(\texttt{<a>}A\texttt{</a>})^*,\ A\to(\texttt{h}+\texttt{i})^*\},$$

which does not overgeneralize, since $\mathcal{L}(\hat{C}_{\text{XML}})\subseteq\mathcal{L}(C_{\text{XML}})$. Furthermore, $\mathcal{L}(\hat{C}_{\text{XML}})$ is not regular, as it contains the language of matching tags `<a>` and `</a>`.

In general, phase two of algorithm first translates the synthesized regular expression $\hat{R}$ into a context-free grammar $\hat{C}$. Then, each generalization step considers equating a pair (A,B) of nonterminals in $\hat{C}$, where A and B correspond to *repetition subexpressions* of $\hat{R}$, which are subexpressions R of $\hat{R}$ of the form $R=R_1^*$. The restriction to equating repetition subexpressions is empirically motivated—in practice, recursive constructs can typically also be repeated, e.g., in matching parentheses grammars, so constraining the search space reduces the potential for imprecision without sacrificing recall. In our example, A_{XML} corresponds to repetition subexpression $\hat{R}_{\text{XML}}$, and A_{hi} corresponds to repetition subexpression R_{hi}, so our algorithm considers equating A_{XML} and A_{hi}.

In the remainder of this section, we first describe how we translate regular expressions to context-free grammars, and then describe phase two candidates and checks.

5.1 Translating $\hat{R}$ to a Context-Free Grammar

Our algorithm translates the regular expression $\hat{R}$ to a context-free grammar $\hat{C}=(V,\Sigma,P,T)$ such that $\mathcal{L}(\hat{R})=\mathcal{L}(\hat{C})$ and subexpressions in $\hat{R}$ correspond to nonterminals in $\hat{C}$. Intuitively, the translation follows the derivation of $\hat{R}$ in the meta-grammar $\mathcal{C}_{\text{regex}}$ (described in Section 4.1). First, the terminals in $\hat{C}$ are the program input alphabet Σ. Next, the nonterminals V of $\hat{C}$ correspond to generalization steps, additionally including an auxiliary nonterminal for steps that generalize repetition nodes:

$$V=\{A_i\mid\text{step }i\}\cup\{A_i'\mid\text{step }i\text{ generalizes }P[\alpha]_{\text{rep}}Q\}.$$

The start symbol is A_1. Finally, the productions are generated according to the following rules:

- **Repetition:** If step i generalizes current language $P[\alpha]_{\text{rep}}Q$ to $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, we generate productions

  $$A_i\to\alpha_1A_i'A_k,\quad A_i'\to\epsilon+A_i'A_j,$$

  where j is the step that generalizes $[\alpha_2]_{\text{alt}}$ and k is the step that generalizes $[\alpha_3]_{\text{rep}}$. Intuitively, these productions are equivalent to the "production" $A_i\to\alpha_1A_j^*A_k$.
- **Alternation:** If step i generalizes $P[\alpha]_{\text{alt}}Q$ to $P([\alpha_1]_{\text{rep}}+[\alpha_2]_{\text{alt}})Q$, we include production $A_i\to A_j+A_k$, where j is the step that generalizes $[\alpha_1]_{\text{rep}}$ and k is the step that generalizes $[\alpha_2]_{\text{alt}}$.

For example, Figure 3 shows the result of the translation algorithm applied to the generalization steps in the first phase

Step	Chosen Generalization	Productions	Language $\mathcal{L}(\hat{C}, A_i)$
R1	$[\texttt{<a>hi</a>}]_{\text{rep}}^{\text{R1}} \Rightarrow ([\texttt{<a>hi</a>}]_{\text{alt}}^{\text{R2}})^*$	$\{A_{\text{R1}} \to A'_{\text{R1}},\ A'_{\text{R1}} \to \epsilon + A'_{\text{R1}} A_{\text{R2}}\}$	$(\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>})^*$
R2	$[\texttt{<a>hi</a>}]_{\text{alt}}^{\text{R2}} \Rightarrow [\texttt{<a>hi</a>}]_{\text{rep}}^{\text{R3}}$	$\{A_{\text{R2}} \to A_{\text{R3}}\}$	$\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>}$
R3	$[\texttt{<a>hi</a>}]_{\text{rep}}^{\text{R3}} \Rightarrow \texttt{<a>}([\texttt{hi}]_{\text{alt}}^{\text{R5}})^*[\texttt{</a>}]_{\text{rep}}^{\text{R4}}$	$\{A_{\text{R3}} \to \texttt{<a>}A'_{\text{R3}} A_{\text{R4}},\ A'_{R3} \to \epsilon + A'_{R3} A_{R5}\}$	$\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{</a>}$
R4	$[\texttt{</a>}]_{\text{rep}}^{\text{R4}} \Rightarrow \texttt{</a>}$	$\{A_{\text{R4}} \to \texttt{</a>}\}$	$\texttt{<a>}$
R5	$[\texttt{hi}]_{\text{alt}}^{\text{R5}} \Rightarrow [\texttt{h}]_{\text{rep}}^{\text{R8}} + [\texttt{i}]_{\text{alt}}^{\text{R6}}$	$\{A_{\text{R5}} \to A_{\text{R8}} + A_{\text{R6}}\}$	$\texttt{h} + \texttt{i}$
R6	$[\texttt{i}]_{\text{alt}}^{\text{R6}} \Rightarrow [\texttt{i}]_{\text{rep}}^{\text{R7}}$	$\{A_{\text{R6}} \to A_{\text{R7}}\}$	$\texttt{i}$
R7	$[\texttt{i}]_{\text{rep}}^{\text{R7}} \Rightarrow \texttt{i}$	$\{A_{\text{R7}} \to \texttt{i}\}$	$\texttt{i}$
R8	$[\texttt{h}]_{\text{alt}}^{\text{R8}} \Rightarrow \texttt{h}$	$\{A_{\text{R8}} \to \texttt{h}\}$	$\texttt{h}$
R9	–	–	–

Figure 3. The productions added to $\hat{C}_{\text{XML}}$ corresponding to each generalization step are shown. The derivation shows the bracketed subexpression $[\alpha]_\tau^i$ (annotated with the step number i) selected to be generalized at step i, as well as the subexpression to which $[\alpha]_\tau^i$ is generalized. The language $\mathcal{L}(\hat{C}, A_i)$ (i.e., strings derivable from A_i) equals the subexpression in $\hat{R}$ that eventually replaces $[\alpha]_\tau^i$. As before, steps that select a candidate that strictly generalizes the language are bolded (in the first column).

of Figure 2 to produce a context-free grammar $\hat{C}_{\text{XML}}$ equivalent to $\hat{R}_{\text{XML}}$. Here, steps R1 and R3 handle the semantics of repetitions, step R5 handles the semantics of the alternation, steps R2 and R6 only affect brackets so they are identities, and steps R4, R7, and R8 are constant expressions. Furthermore, $\mathcal{L}(\hat{C}, A_i)$ is the language of strings matched by the subexpression that eventually replaces the bracketed substring $[\alpha]_\tau$ generalized on step i; this language is shown in the last column of Figure 3.

The auxiliary nonterminals A'_i correspond to repetition subexpressions in $\hat{R}$—if step i generalizes $[\alpha]_{\text{rep}}$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, then $\mathcal{L}(\hat{C}, A'_i) = \mathcal{L}(R^*)$, where R is the subexpression to which $[\alpha_2]_{\text{alt}}$ is eventually generalized. In our example, A'_{R1} corresponds to $\hat{R}_{\text{XML}} = (\texttt{<a>}(\texttt{h}+\texttt{i})^*\texttt{</a>})^*$, and A'_{R3} corresponds to $R_{\text{hi}} = (\texttt{h}+\texttt{i})^*$.

For conciseness, we redefine $\hat{C}_{\text{XML}}$ to be the equivalent context-free grammar with start symbol A'_{R1} and productions

$$A'_{\text{R1}} \to (\texttt{<a>}A'_{\text{R3}}\texttt{</a>})^*, \quad A'_{\text{R3}} \to (\texttt{h} + \texttt{i})^*$$

where the Kleene star implicitly expands to the productions described in the repetition case.

5.2 Candidates and Ordering

The candidates considered in phase two of our algorithm are *merges*, which are (unordered) pairs of nonterminals (A'_i, A'_j) in $\hat{C}$, where i and j are generalization steps of phase one. Recall that these nonterminals correspond to repetition subexpressions in $\hat{R}$. In particular, associated to $\hat{C}$ is the set M of all such pairs of nonterminals. In Figure 2, the regular expression $\hat{R}_{\text{XML}}$ on step R9 is translated into the context-free grammar $\hat{C}_{\text{XML}}$ on step C1, with its corresponding set of merges M_{XML} containing just $(A'_{\text{R1}}, A'_{\text{R3}})$.

Each phase two generalization step selects a pair $(A'_i, A'_j) \in M$ and considers two candidates (in order of preference):

- The first candidate $\tilde{C}$ equates A'_i and A'_j by introducing a fresh nonterminal A and replacing all occurrences of A'_i and A'_j in $\hat{C}$ with A.
- The second candidate equals the current language $\hat{C}$.

In either case, the selected pair is removed from M. The candidates are monotone since equating two nonterminals can only enlarge the generated language.

For example, in step C1 of Figure 2, the candidate $(A'_{\text{R1}}, A'_{\text{R3}})$ is removed from M_{XML}; the first candidate is constructed by equating A'_{R1} and A'_{R3} in $\hat{C}_{\text{XML}}$ to obtain

$$\tilde{C}_{\text{XML}} = \{A \to (\texttt{<a>}A\texttt{</a>})^*,\ A \to (\texttt{h} + \texttt{i})^*\},$$

where $\mathcal{L}(\tilde{C}_{\text{XML}})$ is not regular. The chosen candidate is $\hat{C}'_{\text{XML}} = \tilde{C}_{\text{XML}}$, since the checks (described in Section 5.3) pass. On step C2, M is empty, so our algorithm returns $\hat{C}'_{\text{XML}}$. In particular, $\hat{C}'_{\text{XML}}$ equals $\mathcal{L}(C_{\text{XML}})$, except the characters $\texttt{a} + \ldots + \texttt{z}$ are restricted to $\texttt{h} + \texttt{i}$. In Section 6.2, we describe an extension that generalizes characters in $\hat{C}'_{\text{XML}}$.

Finally, we formalize the intuition that equating $(A'_i, A'_j) \in M$ corresponds to merging repetition subexpressions:

PROPOSITION 5.1. Let regular expression $\hat{R}$ translate to context-free grammar $\hat{C}$. Suppose that nonterminal A_i in $\hat{C}$ corresponds to repetition subexpression R, so $\hat{R} = PRQ$, and A_j to R', so $\hat{R} = P'R'Q'$. Let $\tilde{C}$ be obtained by equating A_i and A_j in $\hat{C}$. Then, $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ (and symmetrically, $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$).

In other words, equating $(A'_i, A'_j) \in M$ merges R and R' in $\hat{R}$. We give a proof in Appendix C.1.

5.3 Check Construction

Consider the candidate $\tilde{C}$ obtained by merging $(A'_i, A'_j) \in M$ in the current language $\hat{C}$, where A'_i corresponds to repetition subexpression R and A'_j to R'. Suppose that step i generalizes $P[\alpha]_{\text{rep}}Q$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, and step j generalizes $[\alpha']_{\text{rep}}$ to $\alpha'_1([\alpha'_2]_{\text{alt}})^*[\alpha'_3]_{\text{rep}}$. Note that $([\alpha_2]_{\text{alt}})^*$ is eventually generalized to the repetition subexpression R in $\hat{R}$, and $([\alpha'_2]_{\text{alt}})^*$ is eventually generalized to R' in $\hat{R}$.

Our algorithm constructs the check $\gamma\rho'\delta$, where $\rho' = \alpha'\alpha' \in \mathcal{L}(R')$ is a residual for R', and (γ, δ) is the context for $([\alpha_2]_{\text{alt}})^*$. This check satisfies

$$\gamma\rho'\delta \in \mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C}),$$

where the first inclusion follows by the property (1) for contexts described in Section 4.3, and the second inclusion follows from Proposition 5.1. A similar argument to Proposition 4.4 shows that this context satisfies property (1).

The check $\gamma\rho'\delta$ tries to ensure that R' can be substituted for R without overgeneralizing, i.e., $\mathcal{L}(PR'Q) \subseteq L_*$. Our algorithm similarly generates a second check trying to ensure that R can be substituted for R', i.e., $\mathcal{L}(P'RQ) \subseteq L_*$.

For example, in Figure 2, the context for the repetition subexpression $\hat{R}_{\text{XML}} = (\texttt{<a>}(\texttt{h}+\texttt{i})^*\texttt{</a>})^*$ is (ϵ, ϵ), and the residual for R_{hi} is `hihi`, so the constructed check is `hihi`. Similarly, the context for R_{hi} is (`<a>`, `</a>`) and the residual for $\hat{R}_{\text{XML}}$ is `<a>hi</a><a>hi</a>`, so the constructed check is `<a><a>hi</a><a>hi</a></a>`.

5.4 Learning Matching Parentheses Grammars

To demonstrate the expressive power of merges, we show that they can represent the following class of generalized matching parentheses grammars:

DEFINITION 5.2. A *generalized matching parentheses grammar* is a context-free grammar $C = (V, \Sigma, P, S_1)$, with

$$V = \{S_1, ..., S_n, R_1, ..., R_n, R'_1, ..., R'_n\}$$

and productions

$$S_i \to (R_i(S_{i_1} + ... + S_{i_{k_i}})^* R'_i)^*,$$

where for $1 \leq i \leq n$, R_i, R'_i are regular expressions over Σ.

In other words, R_i and R'_i are pairs of matching parentheses, except that they are allowed to be regular expressions, e.g., XML tags. They may also match the empty string ϵ, e.g., to permit unmatched open parentheses. Then, the valid matched parentheses strings matched by the grammars $S_{i_1}, ..., S_{i_{k_i}}$ can occur between R_i and R'_i. In particular, the XML-like grammar shown in Figure 1 is a generalized matching parentheses grammar, where the "parentheses" are `<a>` and `</a>`. We have the following result:

PROPOSITION 5.3. For any generalized matching parentheses grammar C, there exists a regular expression R and merges M over R such that letting C' be the grammar obtained by transforming R into a context-free grammar and performing the merges in M, we have $\mathcal{L}(C) = \mathcal{L}(C')$.

In other words, phase two of our algorithm at least allows us to learn the common class of generalized matching parentheses grammars. We give a proof in Appendix D.

5.5 Computational Complexity

The complexity of phase two is $O(n^4)$, where n is the length of the seed input α_{in}, since each pair of repetition subexpressions is a merge candidate, and as shown in Section 4.4, there are at most $O(n^2)$ repetition candidates. Therefore, the overall complexity is $O(n^4)$.

6. Extensions

In this section, we discuss two extensions to our algorithm.

6.1 Multiple Seed Inputs

Given multiple seed inputs $E_{\text{in}} = \{\alpha_1, ..., \alpha_n\}$, our algorithm first applies phase one separately to each α_i to synthesize a corresponding regular expression $\hat{R}_i$. Then, it combines these into a single regular expression $\hat{R} = \hat{R}_1 + ... + \hat{R}_n$ and applies phase two to $\hat{R}$. Repetition subexpressions in different components $\hat{R}_i$ of $\hat{R}$ may be merged. A useful optimization is to construct $\hat{R}$ incrementally—if we have $\alpha_i \in \mathcal{L}(\hat{R}_1 + ... + \hat{R}_{i-1})$, then α_i can be skipped.

6.2 Character Generalization

After phase one, we include a *character generalization* phase that generalizes terminals in the synthesized regular expression $\hat{R}$. At each generalization step, the algorithm selects a terminal string $\alpha = \sigma_1...\sigma_k$ in $\hat{R}$, i.e., $\hat{R} = P\alpha Q$, and a terminal σ_i in α, and a different terminal $\sigma \in \Sigma$ such that $\sigma \neq \sigma_i$, and considers two candidates. First, $\tilde{R} = P\sigma_1...\sigma_{i-1}(\sigma + \sigma_i)\sigma_{i+1}...\sigma_k Q$ replaces σ_i with $(\sigma_i + \sigma)$. Second, the current language $\hat{R}$. Each such generalization is considered exactly once in this phase.

For the first candidate, we construct residual $\rho = \sigma$. Every terminal string α in $\hat{R}$ was added by generalizing $[\alpha'_{\text{rep}}]$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, where $\alpha = \alpha_1$. Supposing that the context for $[\alpha'_{\text{rep}}]$ is (γ, δ), we construct context $(\gamma\sigma_1...\sigma_{i-1}, \sigma_{i+1}...\sigma_k\alpha_3\delta)$. The generated checks are $\gamma\rho\delta$.

For example, in the regular expression $\hat{R}_{\text{XML}}$ output by phase one in Figure 2, our algorithm considers generalizing each terminal in `<a>`, `h`, `i`, and `</a>` to every (different) terminal $\sigma \in \Sigma$. Generalizing `<` to `a` is ruled out by the check `aa>hi</a>`. Alternatively, `h` is generalized to `a` since the generated checks `<a>ai</a>` and `<a>a</a>` pass. Eventually, $\hat{R}_{\text{XML}}$ generalizes to

$$\hat{R}'_{\text{XML}} = (\texttt{<a>}((\texttt{a} + ... + \texttt{z}) + (\texttt{a} + ... + \texttt{z}))^*\texttt{</a>})^*,$$

which phase two generalizes to the grammar $\hat{C}'_{\text{XML}}$:

$$\left\{ \begin{array}{l} A \to (\texttt{<a>}A\texttt{</a>})^*, \\ A \to ((\texttt{a} + ... + \texttt{z}) + (\texttt{a} + ... + \texttt{z}))^* \end{array} \right\}.$$

In particular, $\mathcal{L}(\hat{C}'_{\text{XML}}) = \mathcal{L}(C_{\text{XML}})$.

7. Discussion

***Phases of* GLADE.** We have described GLADE as proceeding in three phases, but the distinction is primarily for purposes of clarity. More precisely, the character generalization phase can equivalently be performed at any time. Phase two (the merging phase) depends on phase one to identify candidate repetition subexpressions to merge, but these phases could be interleaved if desired.

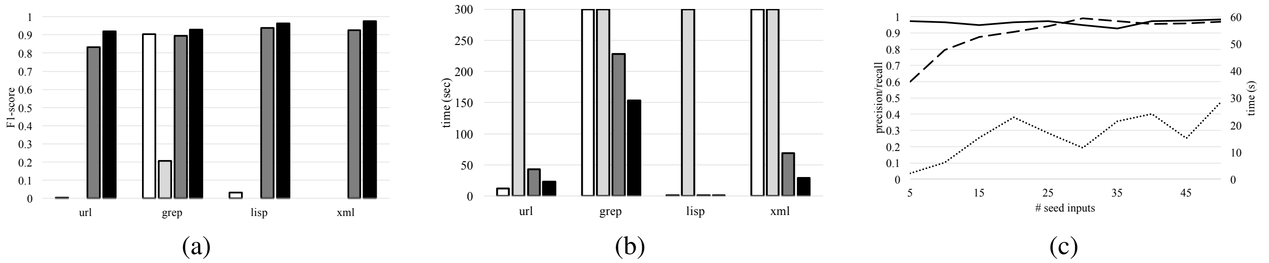


Figure 4. We show (a) the F_1 score, and (b) the running time of L-Star (white), RPNI (light grey), GLADE omitting phase two (dark grey), and GLADE (black) for each of the four test grammars C. The algorithms are trained on 50 random samples from the target language $L_* = \mathcal{L}(C)$. In (c), for the XML grammar, we show how the precision (solid line), recall (dashed line), and running time (dotted line) of GLADE vary with the number of seed inputs $|E_{\text{in}}|$ (between 0 and 50). The y-axis for precision and recall is on the left-hand side, whereas the y-axis for the running time (in seconds) is on the right-hand side.

Limitations. The greedy search strategy is necessary for GLADE to efficiently search the space of languages. However, the cost of greediness is that suboptimal grammars may be synthesized (i.e., only generating a subset of the target language), even if all selected candidates are precise. For example, consider extending the XML grammar shown in Figure 1 with the production

$$A_{\text{XML}} \rightarrow \texttt{<a/>}.$$

Given the seed input

$$\alpha_{\text{in}} = \texttt{<a><a/></a>},$$

phase one of GLADE synthesizes the regular expression

$$(\texttt{<a}(\texttt{><a/})^*\texttt{></a>})^*,$$

which is a valid subset of L_{XML}. However, in phase two of GLADE, the two repetition nodes

$$(\texttt{><a/})^* \quad \text{and} \quad (\texttt{<a}(\texttt{><a/})^*\texttt{></a>})^*$$

cannot be merged, since the check `><a/` is invalid. Ideally, GLADE would instead synthesize the regular expression

$$(\texttt{<a>}(\texttt{<a/>})^*\texttt{</a>})^*,$$

in phase one, in which case the two repetition nodes

$$(\texttt{<a/>})^* \quad \text{and} \quad (\texttt{<a>}(\texttt{<a/>})^*\texttt{</a>})^*$$

are successfully merged in phase two. GLADE fails to do so because of the greedy nature of phase one. If GLADE is instead provided with the seed inputs

$$\{\texttt{<a/>},\ \texttt{<a>hi</a>}\},$$

then it would successfully recover the target language.

Intuitively, the greedy strategy employed by GLADE works best when the target language has fewer nondeterministic constructs (as is the case with many program input languages in practice, e.g., to ensure efficient parsing). Such grammars are less likely to have multiple incompatible candidates at each generalization step, ensuring that GLADE rarely makes suboptimal choices.

8. Evaluation

We implement our grammar synthesis algorithm in a tool called GLADE, which synthesizes a context-free grammar $\hat{C}$ given an oracle $\mathcal{O}$ and seed inputs $E_{\text{in}} \subseteq L_*$. In our first experiment, we compare GLADE to widely studied language inference algorithms, and in our second experiment, we evaluate the ability of GLADE to learn useful approximations of real program input grammars for a fuzzing client. We note that the only grammar used to guide the design our algorithm is the XML grammar, and no other grammar was used for this purpose. GLADE is implemented in Java, and all experiments are run on a 2.5 GHz Intel Core i7 CPU.

8.1 Sampling Context-Free Grammars

We describe how we randomly sample a string α from a context-free grammar C. The ability to sample implicitly defines a probability distribution $\mathcal{P}_{\mathcal{L}(C)}$ over $\mathcal{L}(C)$, which we use to measure precision and recall as in Definition 2.1. We also use random samples in our grammar-based fuzzer in Section 8.3. To describe our approach, we more generally describe how to sample $\alpha \sim \mathcal{P}_{\mathcal{L}(C,A)}$ (which is the language of strings that can be derived from nonterminal A using productions in C). To do so, we convert the context-free grammar $C = (V, \Sigma, P, S)$ to a *probabilistic context-free grammar*. For each nonterminal $A \in V$, we construct a discrete distribution $\mathcal{D}_A$ of size $|P_A|$ (where $P_A \subseteq P$ is the set of productions in C for A). Then, we randomly sample $\alpha \sim \mathcal{P}_{\mathcal{L}(C,A)}$ as follows:

- Randomly sample production $(A \rightarrow A_1...A_k) \sim \mathcal{D}_A$.
- If A_i is a nonterminal, recursively sample $\alpha_i \sim \mathcal{P}_{\mathcal{L}(C,A_i)}$; otherwise, if A_i is a terminal, let $\alpha_i = A_i$.
- Return $\alpha = \alpha_1...\alpha_k$.

For simplicity, we choose $\mathcal{D}_A$ to be uniform.

8.2 Comparison to Language Inference

In our first experiment, we show that GLADE can synthesize simple input grammars with much better precision and recall compared to two widely studied language inference

Grammar	Target Language L_*	Synthesized Grammar $\hat{L}$
URL	$A \to \texttt{http}(\epsilon + \texttt{s})\texttt{://}(\epsilon + \texttt{www.})[...]^*\texttt{.}[...]^*$	$A \to \texttt{http://}B^*\texttt{.}C^* + \texttt{https://}B^*\texttt{.}C^*$ $+\texttt{http://www.}B^*\texttt{.}C^* + \texttt{https://www.}B^*\texttt{.}C^*$ $B \to [...]^*$ $C \to [...]^*$
Grep	$A \to ([...] + \texttt{\textbackslash(}A\texttt{\textbackslash)})^*$	$A \to ([...]^* + ((\texttt{\textbackslash(}\texttt{(}A^*)^*\texttt{\textbackslash)}^*\texttt{)})^*)^*$
Lisp	$A \to \texttt{(}[...][...]^*(\text{␣␣}^*([...][...]^* + A))^*\texttt{)}$	$A \to \texttt{(}([...]^*[...]((\text{␣}^*\text{␣}A)^*\text{␣}^*\text{␣})^*)^*[...]^*[...]\texttt{)}$
XML	$A \to \texttt{<a}(\text{␣␣}^*[...][...]^*\texttt{="}[...]^*\texttt{"})^*\texttt{>}(A + [...])^*\texttt{</a>}$	$A \to \texttt{<a}(\text{␣}^*\text{␣}[...]^*[...]\texttt{="}[...]^*\texttt{"})^*B^*\texttt{>}[...]^*\texttt{</a>}$ $B \to \texttt{>}[...]^*\texttt{<a}(\text{␣}^*\text{␣}[...]^*[...]\texttt{="}[...]^*\texttt{"})^*B^*\texttt{>}[...]^*\texttt{</a}$ $+\texttt{>}[...]^*\texttt{<a>}[...]^*\texttt{</a}$

Figure 5. Examples of context-free grammars that are synthesized by GLADE for the given target languages. The symbol ␣ denotes a space. For clarity, character ranges with large numbers of characters are denoted by [...].

algorithms, L-Star [3] and RPNI [44], both implemented using `libalf` [5]. We also compare to a variant of GLADE with phase two omitted, which restricts GLADE to learning regular languages, which shows that the benefit of GLADE is not just its ability to synthesize non-regular properties.

Grammars. We manually wrote four grammars encoding valid inputs for various programs:

- A regular expression for matching URLs [55].
- A grammar for the regular expression accepted as input by GNU Grep [21]
- A grammar for a simple Lisp parser [43], including support for quoted strings and comments.
- A grammar for XML parsers [64], including all XML constructs (attributes, comments, CDATA sections, etc.), except that only a fixed number of tags are included (to ensure that the grammar is context-free).

Methods. For each grammar C, we sampled 50 seed inputs $E_{\text{in}} \subseteq L_* = \mathcal{L}(C)$ using the technique in Section 8.1, and implemented an oracle $\mathcal{O}$ for L_*. Then, we use each algorithm to learn L_* from E_{in} and $\mathcal{O}$. Since the algorithms sometimes cannot scale to all 50 inputs, we incrementally give the seed inputs to the algorithms until they time out (after 300 seconds), and use the last language successfully learned without timing out.

L-Star. Angluin's L-Star algorithm learns a regular language $\hat{R}$ approximating the target language L_*. It takes as input a membership oracle and an *equivalence oracle* $\mathcal{O}_E$; given a candidate regular language $\hat{R}$, $\mathcal{O}_E$ accepts $\hat{R}$ if $\mathcal{L}(\hat{R}) = L_*$, and returns a counterexample otherwise. In our experiments, there is no way to check equivalence with the target language (i.e., the program input language). Instead, we use the variant in [3] where the equivalence oracle $\mathcal{O}_E$ is implemented by randomly sampling strings to search for counter-examples; we accept $\hat{R}$ if none are found after 50 samples.

RPNI. RPNI learns a regular language $\hat{R}$ given both positive examples E_{in} and negative examples E_{in}^-. As negative examples, we sample 50 random strings not in L_*.

Results. We estimate the precision of $\hat{C}$ by $\frac{|E_{\text{prec}} \cap L_*|}{|E_{\text{prec}}|}$, where E_{prec} consists of 1000 random samples from $\mathcal{L}(\hat{C})$, and estimate the recall of $\hat{C}$ by $\frac{|E_{\text{rec}} \cap \mathcal{L}(\hat{C})|}{|E_{\text{rec}}|}$, where E_{rec} consists of 1000 random samples from L_*, and report the F_1-score $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. The F_1 score is a standard metric combining precision and recall—achieving high F_1 score requires both high precision and high recall. We also report the running time of each algorithm, which is timed out at 300 seconds. We average all results over five runs. Figure 4 shows (a) the F_1-score and (b) the running time of each algorithm; (c) shows how the precision, recall, and running time of GLADE vary with the number of samples in E_{in}.

***Performance of* GLADE.** With just the 50 given training examples, GLADE was able to learn each grammar with an F_1-score of nearly 1.0, meaning that both precision and recall were nearly 100%. These results strongly suggest that GLADE learns most of the true structure of L_*. Finally, as can be seen from Figure 4 (c), GLADE performs well even with few samples, and its running time likewise scales well with the number of samples. The performance of GLADE with phase two omitted (i.e., P1 in Figure 4) continues to substantially outperform L-Star and RPNI.

***Phases of* GLADE.** As can be seen in Figure 4 (a), GLADE consistently performs 5-10% better than P1—i.e., the majority of the improvement of GLADE over existing algorithms is due to the active learning strategy, and the remainder is due to the ability to induce context-free grammars.

Furthermore, a consequence of our optimization when using multiple inputs (see Section 6.1), GLADE is actually faster than P1—because GLADE generalizes better than P1, it uses fewer samples in E_{in}, thereby reducing the running time. We performed the same experiment using GLADE with the character generalization phase removed (but including both phases one and two). This variant of GLADE consistently performed similar but slightly worse than P1 both in terms of F_1-score and running time, so we omit results.

Comparison to L-Star and RPNI. L-Star performs well for the Grep grammar, but essentially fails to learn the other grammars, achieving either very small precision or very small recall. RPNI performs even worse, failing to learn any of the languages. L-Star guarantees exact learning only when a true equivalence oracle is available. Similarly, RPNI has an “in the limit” learning guarantee, i.e., for any enumeration of all strings $\alpha_1, \alpha_2, ... \in \Sigma^*$, it eventually learns the correct language. Both of these learning guarantees require following examples:

- **Positive:** Exercise all transitions in the minimal DFA.
- **Negative:** Reject all incorrect generalizations.

These examples are assumed to be provided either by the equivalence oracle (for L-Star) or in the given examples E_{in} and E_{in}^- (for RPNI).

However, in our setting, the equivalence oracle is unavailable to the L-Star algorithm and must be approximated using random sampling, so its theoretical guarantees may not hold. Indeed, random sampling rarely provides the needed examples—for example, in most runs of L-Star, at most two calls to the equivalence oracle found counterexamples. Similarly, for RPNI, the given examples are typically incomplete, so its theoretical guarantees likewise may not hold.

Furthermore, because these algorithms are designed to learn when the guarantees hold, they do not provide any mechanisms for recovering from failure of the assumptions, and instead fail dramatically. For example, if a terminal appears in L_* but not in any seed input in E_{in}, then the language learned by RPNI does not contain any strings with this terminal. In contrast, GLADE incorporates generalization steps that enable it to generalize beyond behaviors in the given examples, and its carefully selected checks often provide the counterexamples needed to avoid overgeneralizing.

Additionally, while polynomial, the running times of L-Star and RPNI are very long. The long running time of L-Star is not because L_* is non-regular, instead, we observe that L-Star algorithm issues a large number of membership queries on each of its iterations. In our setting, L-Star often could not even learn a four state automaton.

Examples. Figure 5 shows examples of grammars synthesized by GLADE for the target language shown and a small set of representative seed inputs. The target languages are substantially simplified fragments of the grammars used in this experiment (to ensure clarity); the synthesized grammars are correspondingly simplified.

The structure of a synthesized grammar sometimes differs from the structure of the grammar defining the target language, even if they generate the same language. Such discrepancies occur because GLADE obtains no information about the internal representation of the target language. For example, consider the synthesized XML grammar. In a more natural grammar, the character > at the front of the production for B would instead appear in the production for A, and the corresponding > in the production for A would instead appear at the end of the production for B; however, this modification does not affect the generated language.

Program	Lines of Code	Lines in E_{in}	Time (min.)
sed	2K	3	0.25
flex	6K	15	1.83
grep	12K	4	0.17
bison	13K	14	4.91
xml	123K	7	2.30
ruby	120K	80	229.00
python	128K	267	269.00
javascript	156K	118	113.00

Figure 6. For each program, we show lines of program code, the lines of seed inputs E_{in}, and running time of GLADE.

8.3 Comparison to Fuzzers

For fuzzing applications such as differential testing [67], it is useful to obtain a large number of grammatically valid samples that exercise different functionalities of the given program. GLADE is perfectly suited to automatically generating such inputs. Given blackbox access $\mathcal{O}$ to a program with input language L_* and seed inputs $E_{\text{in}} \subseteq L_*$, GLADE automatically synthesizes a context-free grammar $\hat{C}$ approximating L_*. Then, GLADE uses a standard grammar-based fuzzer that takes as input the synthesized grammar $\hat{C}$ and the seed inputs E_{in}, and randomly generates new inputs $\alpha \in \mathcal{L}(\hat{C})$ that can be used to test the program; we give details below.

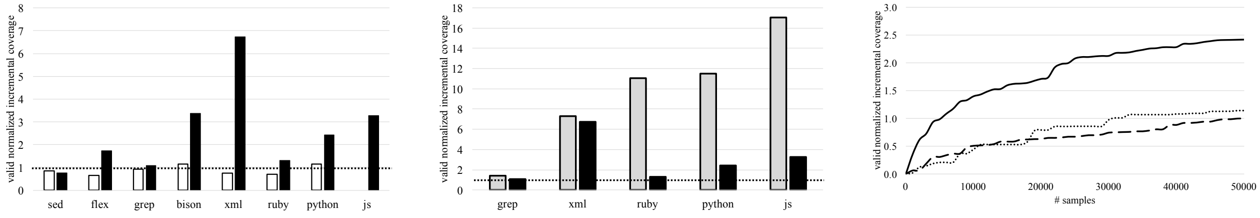
In our application to fuzzing, it is acceptable for $\hat{C}$ to be an approximation—high precision suffices to ensure that most generated inputs are valid, and high recall ensures that most program behaviors have a chance of being executed.

We compare GLADE to two baseline fuzzers (described below) on the task of generating valid test inputs, and show that GLADE consistently performs significantly better.

Grammar-based fuzzer. GLADE first synthesizes a context-free grammar $\hat{C}$ approximating the target language L_* of valid program inputs. Our grammar-based fuzzer, based on standard techniques [28], takes as input the synthesized context-free grammar $\hat{C}$ and the seed inputs E_{in}. To generate a single random input, our grammar-based fuzzer first uniformly selects a seed input $\alpha \in E_{\text{in}}$ and constructs the parse tree for α according to $\hat{C}$. Second, it performs a series of n modifications to α, where n is chosen uniformly between 0 and 50. A single modification is performed as follows:

- Randomly choose a node N of the parse tree of α.
- Decompose $\alpha = \alpha_1\alpha_2\alpha_3$ where α_2 is represented by the subtree with root N.
- Letting A be the nonterminal labeling N, randomly sample $\alpha' \sim \mathcal{P}_{\mathcal{L}(C,A)}$, and return $\alpha_1\alpha'\alpha_3$.

Afl-fuzz. Our first baseline fuzzer is a production fuzzer developed at Google [68], and is widely used due to its minimal setup requirements and state-of-the-art quality. It systematically modifies the input example (e.g., bit flips, copies, deletions, etc.). Unlike GLADE, afl-fuzz requires that the program be instrumented to obtain branch coverage for



(a) (b) (c)

Figure 7. In (a) we show the normalized incremental coverage restricted to valid samples for the naïve fuzzer (black dotted line), afl-fuzz (white), and GLADE (black). In (b), we show the same metric for the naïve fuzzer (black dotted line) and GLADE (black); grey represents either a handwritten fuzzer (for Grep and the XML parser) or a large test suite (for Python, Ruby, and Javascript). In (c), we compare the valid normalized incremental coverage of GLADE (solid) to the naïve fuzzer (dashed) and afl-fuzz (dotted) as the number of seed inputs varies (all values are normalized by the final coverage of the naïve fuzzer).

each execution—it uses this information to identify when an input α causes the program to execute new paths. It adds such inputs α to a worklist, and iteratively applies its fuzzing strategy to each input in the worklist. This monitoring allows it to incrementally discover deeper code paths. To run afl-fuzz on multiple inputs E_{in}, we fuzz each input $\alpha \in E_{\text{in}}$ in a round-robin fashion.

Naïve fuzzer. We implement a second baseline fuzzer, which is not grammar aware. It randomly selects a seed input $\alpha \in E_{\text{in}}$ and performs n random modifications to α, where n is chosen randomly between 0 and 50. A single modification of α consists of randomly choosing an index i in $\alpha = \sigma_1 ... \sigma_k$, and either deleting the terminal σ_i or inserting a randomly chosen terminal $\sigma \in \Sigma$ before σ_i.

Programs. We set up each fuzzer on eight programs that include front-ends of language interpreters (Python, Ruby, and Mozilla's Javascript engine SpiderMonkey), Unix utilities that take structured inputs (Grep, Sed, Flex, and Bison), and an XML parser. We were unable to setup afl-fuzz for Javascript, showing that even production fuzzers can have setup difficulties when they require code instrumentation. For interpreters (e.g., the Python interpreter), we focus on fuzzing just the parser (e.g., the Python parser) since the input grammar of the interpreter contains elements such as variable and function names, use-before-define errors, etc., that are out of scope for our grammar synthesis algorithm. To fuzz the parser, we "wrap" the input inside a conditional statement, which ensures that the input is never executed. For example, we convert the Python input (`print 'hi'`) to the input (`if False: print 'hi'`). Then, syntactically incorrect inputs are rejected, but inputs that are syntactically correct but possibly have runtime errors are accepted.

Seed inputs. To fuzz a program, we use a small number of seed inputs $E_{\text{in}} \subseteq L_*$ that capture interesting semantics of the target language L_*. These seed inputs were obtained either from documentation and tutorials or from small test suites that came with the program.

Methods. Coverage is difficult to interpret because a large amount of code in each program is unreachable due to configuration, test code that cannot be executed, and other unused functionality. Therefore, we use a relative measure of coverage to evaluate performance. As before, all results are averaged over five runs.

For each program and fuzzer, we generate 50,000 samples $E \subseteq \Sigma^*$ by running the fuzzer on the program. First, we restrict E to valid inputs, i.e., $E \cap L_*$. In particular, the *valid coverage* of E, computed using gcov, is

$$\frac{\#(\text{lines covered by } E \cap L_*)}{\#(\text{lines coverable})}.$$

Next, the *valid incremental coverage* of E is the percentage of code covered by valid inputs in E, ignoring those already covered by the seed inputs E_{in} (thereby measuring the ability to discover inputs that execute new code paths):

$$\frac{\#(\text{lines covered by } E \cap L_* \text{ but not covered by } E_{\text{in}})}{\#(\text{lines coverable but not covered by } E_{\text{in}})}.$$

Finally, to enable comparison across programs, the *valid normalized incremental coverage* normalizes the incremental coverage by a baseline E_{base}:

$$\frac{\text{valid incremental coverage of } E}{\text{valid incremental coverage of } E_{\text{base}}}.$$

In particular, we use samples from the naïve fuzzer as E_{base}.

Results. In Figure 6, we show various statistics for the eight programs we use and for the corresponding seed inputs E_{in}. We also show the time GLADE needed to synthesize an approximation of the program input grammar. In Figure 7 (a), we show the valid normalized incremental coverages of the various fuzzers. In (b), for five of our programs, we show a proxy for the "upper bound" in coverage that is achievable—for Grep and the XML parser, we show the valid normalized incremental coverage achieved by our handwritten grammars, and for Python, Ruby, and Javascript, we show the valid normalized incremental coverage of a large test suite (each more than 100,000 lines of code). In (c), we show how coverage varies with the number of samples for Python.

Comparison to baselines. As can be seen from Figure 7 (a), GLADE (black) is effective at generating valid inputs that exercise new code paths, significantly outperforming both the naïve fuzzer (black dotted line) and afl-fuzz (white) except on Grep (where it only performs slightly better) and Sed (where it actually performs slightly worse). Since these programs have a relatively simple input format, using a grammar-based fuzzer is understandably less effective. For the remaining six programs, our grammar-based fuzzer performs between 1.3 and 7 times better than the naïve fuzzer.

Comparison to proxy for the upper bound. Figure 7 (b) compares GLADE (black bars) to a proxy for the upper bound of coverage, i.e., handwritten grammars or large test suites (grey bars). For Grep, both GLADE and the naïve fuzzer achieve coverage close to the handwritten grammar. For the XML parser, GLADE significantly outperforms the naïve fuzzer, achieving coverage close to the handwritten grammar. For Python and Javascript, GLADE is able to recover a significantly larger fraction of the upper bound compared to the naïve fuzzer. However, a sizable gap remains, which is expected since the test suites are very large (each having at least 100,000 lines of code) and are specifically designed to test the respective programs. We provided fewer seed inputs for Ruby, which explains why GLADE outperformed the naïve fuzzer by a smaller amount (about 30%).

Coverage over time. Figure 7 (c) shows how the valid normalized incremental coverage varies with the number of samples. GLADE (solid) quickly finds a number of high-coverage inputs that the other fuzzers cannot, and continues to find more inputs that execute new lines of code.

Examples. The synthesized grammars are too large to show. Instead, as an example, a fragment of the synthesized XML grammar is

$$\begin{aligned} A &\rightarrow \texttt{<a}\text{\textvisiblespace}^{*}\text{\textvisiblespace}[\ldots]^{*}[\ldots]\texttt{="}[\ldots]^{*}\texttt{"}B^{*}\texttt{>}[\ldots]^{*}\texttt{</a>}\\ B &\rightarrow \texttt{>}[\ldots]^{*}\texttt{<a}\text{\textvisiblespace}^{*}\text{\textvisiblespace}[\ldots]^{*}[\ldots]\texttt{="}[\ldots]^{*}\texttt{"}B^{*}\texttt{>}[\ldots]^{*}\texttt{</a}\\ &\quad + \texttt{>}[\ldots]^{*}\texttt{<a>}[\ldots]^{*}\texttt{</a}. \end{aligned}$$

This grammar is identical to the synthesized XML grammar shown in Figure 5, except that attributes cannot be repeated. In particular, GLADE learns that attributes cannot be repeated since XML semantics requires that different attributes have different names—for example, the input string `<a a="" a=""></a>` is invalid. Therefore, repeating the attribute would lead to overgeneralization, so this construct is rejected by GLADE. Indeed, this constraint on attribute names is not a context-free property, so as expected, GLADE learns a subset of the XML input language.

Figure 8 shows an example of a valid sample from the grammar synthesized by GLADE for the XML parser. As can be seen, the sample contains many XML constructs, including nested tags, attributes, comments, and processing instructions.

```
<a>
  \%
  <a QE="{>_">
    C
    <a    _="#">
      ">q(+_[s:?>^0+
      <a  _eD="{@">
        :"<a>. q</a>1+%
      </a>
      y<!--       y-->y
    </a>
    _<a>x</a>y
  </a>
  xy<?q  xy?>xy<?xV <?By_![?>x
</a>
```

Figure 8. An example of a valid sample from the grammar synthesized by GLADE for the XML parser. For clarity, the string has been formatted with additional whitespace.

9. Related Work

Mining input formats. The work most closely related to our own is [29], which uses dynamic taint analysis to trace the flow of each input character, and uses this information to reconstruct the input grammar. More broadly, there has been work on reverse engineering network protocol message formats [8, 35, 36, 66], though these papers focus on learning and understanding the structure of given inputs rather than learning a grammar; for example, [8] looks for variables representing the internal parser state to determine the protocol, and [35] constructs syntax trees for given inputs. All of these techniques rely on static and dynamic analysis methods intended to reverse engineer parsers of specific designs.

In contrast, our approach is fully blackbox and depends only on the language accepted by the program, not the specific design of the program's parser. In addition, our approach can be used when the program cannot be instrumented, for instance, to learn the input format for a remote program. Finally, the programs we consider have more complex input formats than most previously examined programs.

Learning theory. There has been a line of work in learning theory (often referred to as *grammar induction* or *grammar inference*) aiming to learn a grammar from either examples or oracles (or both); see [14] for a survey. The most well known algorithms are L-Star [3] and RPNI [44]. These algorithms have a number of applications including model checking [19], model-assisted fuzzing [12, 13], verification [62], and specification inference [6]. To the best of our knowledge, our work is the first to focus on the application of learning common program input languages from positive examples and membership oracles.

Additionally, [33] discusses approaches to learning context-free grammars, including from positive examples and a membership oracle. As they discuss, these algorithms are often either slow [54] or do not generalize well [32].

Bayesian language learning. A related line of work aims to learn probabilistic grammars from examples alone [56, 57]. These algorithms study a different setting than ours, in particular, they are given access to positive (and sometimes negative) examples, but do not assume access to a membership oracle. These algorithms typically identify frequently occurring patterns that are likely to correspond to nonterminals in the grammar. More precisely, these algorithms are typically Bayesian learning algorithms that operate by putting a prior over the space of grammars, and then computing the most likely grammar conditioned on the given examples. To achieve statistically significant results, these algorithms require a large number of input examples.

In contrast, our algorithm leverages access to the membership oracle, enabling it to use actively generated examples to determine which patterns are actually in the grammar. Therefore, our algorithm works well even when only a few seed inputs are available. While it may be possible to modify existing Bayesian language learning algorithms to fit this setting, to the best of our knowledge, no such active learning variants of these algorithms have been proposed.

Additionally, whereas this literature aims to learn a probabilistic grammar, our grammar synthesis algorithm learns a deterministic grammar. The difference is how we measure approximation quality—in particular, even though our definitions of precision and recall require distributions over L_* and $\hat{L}$, they still measure the approximation quality of $\hat{L}$ deterministically, i.e., the predicates $\alpha \in L_*$ and $\alpha \in \hat{L}$ are binary rather than probabilistic.

Blackbox fuzzing. Numerous approaches to automated test generation have been proposed; we refer to [2] for a survey. Approaches to fuzzing (i.e., random test case generation) broadly fall into two categories: whitebox (i.e., statically inspect the program to guide test generation) and blackbox (i.e., rely only on concrete program executions). Blackbox fuzzing has been used to test software for several decades; for example, [51] randomly tests COBOL compilers and [48] generated random inputs to test parsers. An early application of blackbox fuzzing to find bugs in real-world programs was [39], who executed Unix utilities on random byte sequences to discover crashing inputs. Subsequently, there have been many approaches using blackbox fuzzing with dynamic analysis to find bugs and security vulnerabilities [17, 40, 59]; see [60] for a survey. Finally, afl-fuzz [68] is almost blackbox, requiring only simple instrumentation to guide the search.

Whitebox fuzzing. Approaches to whitebox fuzzing [4, 24] typically build on *dynamic symbolic execution* [9–11, 22, 52]; given a concrete input example, these approaches use a combination of symbolic execution and dynamic execution to construct a constraint system whose solutions are inputs that execute new program branches compared to the given input. It can be challenging to scale these approaches to large programs [18]. Therefore, approaches relying on more imprecise input have been studied; for example, taint analysis [18], or extracting specific information such as a checksum computation [65].

Grammar-based fuzzing. Many fuzzing approaches leverage a user-defined grammar to generate valid inputs, which can greatly increase coverage. For example, blackbox fuzzing has been combined with manually written grammars to test compilers [37, 67]; see [7] for a survey. Such techniques have also been used to fuzz interpreters; for example, [28] develops a framework for grammar-based testing and applies it to find bugs in both Javascript and PHP interpreters.

Grammar-based approaches have also been used in conjunction with whitebox techniques. For example, [23] fuzzes a just-in-time compiler for Javascript using a handwritten Javascript grammar in conjunction with a technique for solving constraints over grammars, and [38] combines exhaustive enumeration of valid inputs with symbolic execution techniques to improve coverage. In [60], Chapter 21 gives a case study developing a grammar for the Adobe Flash file format. Our approach can complement existing grammar-based fuzzers by automatically generating a grammar.

Finally, there has been some work on inferring grammars for fuzzing [63], but focusing on simple languages such as compression formats. To the best of our knowledge, our work is the first targeted at learning complex program input languages that contain recursive structure, e.g., XML, regular expression formats, and programming language syntax.

Synthesis. Finally, our approach uses machinery related to some of the recent work on programming by example—in particular, a systematic search guided by a meta-grammar. This approach has been used to synthesize string [26], number [53], and table [27] transformations (and combinations thereof [46, 47]), as well as recursive programs [1, 16] and parsers [34]. Unlike these approaches, our approach exploits an oracle to reject invalid candidates.

10. Conclusion

We have presented GLADE, the first practical algorithm for inferring program input grammars, and demonstrated its value in an application to fuzz testing. We believe GLADE may be valuable beyond fuzzing, e.g., to generate whitelists of inputs or to reverse engineer input formats.

Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government. This work was also supported by NSF grant CCF-1160904 and a Google Fellowship.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *Computer Aided Verification*, pages 934–950. Springer, 2013.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.

[5] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *International Conference on Computer Aided Verification*, pages 360–364. Springer, 2010.

[6] M. Botinčan and D. Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 443–456, 2013.

[7] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[8] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[9] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[10] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[12] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, pages 139–154, 2011.

[13] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, pages 623–640, 2013.

[14] C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[15] ECMA International. *Standard ECMA-262: ECMA 2015 Language Specification*. 6 edition, June 2015.

[16] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–239. ACM, 2015.

[17] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68. Seattle, 2000.

[18] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[19] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *International Static Analysis Symposium*, pages 248–264. Springer, 2012.

[20] GNU. Gnu bison. `https://www.gnu.org/software/bison`, 2014.

[21] GNU Grep. `https://www.gnu.org/software/grep/manual`, 2016.

[22] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[23] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[24] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[25] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[26] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330, 2011.

[27] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

[28] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[29] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 720–725. ACM, 2016.

[30] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, pages 883–891, 2010.

[31] H. Ishizaka. Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5(2):151–164, 1990.

[32] B. Knobe and K. Knobe. A method for inferring context-free grammars. *Information and Control*, 31(2):129–146, 1976.

[33] L. Lee. Learning of context-free languages: A survey of the literature. *Techn. Rep. TR-12-96, Harvard University*, 1996.

[34] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 565–574. ACM, 2015.

[35] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 83–93. ACM, 2008.

[36] Z. Lin, X. Zhang, and D. Xu. Reverse engineering input syntactic structure from program execution and its applications. *Software Engineering, IEEE Transactions on*, 36(5):688–703, 2010.

[37] C. Lindig. Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12. ACM, 2005.

[38] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 134–143. ACM, 2007.

[39] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[40] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54. ACM, 2006.

[41] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. pages 373–386, 2012.

[42] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[43] P. Norvig. `http://norvig.com/lispy.html`, 2010.

[44] J. Oncina and P. García. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5(99-108):15–20.

[45] Oracle America, Inc. *The Java^{TM}Virtual Machine Specification*. 7 edition, July 2011.

[46] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–418, 2014.

[47] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126. ACM, 2015.

[48] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[49] M. Rinard. Acceptability-oriented computing. pages 221–239, 2003.

[50] M. C. Rinard. Living in the comfort zone. pages 611–622, 2007.

[51] R. L. Sauder. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 317–323. ACM, 1962.

[52] K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[53] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[54] R. J. Solomonoff. A new method for discovering the grammars of phrase structure languages. In *Information Processing*. Unesco, Paris, 1960.

[55] Stack Overflow. `http://stackoverflow.com/questions/3809401/what-is-a-good-regular-expression-to-match-a-url`, 2010.

[56] A. Stolcke. *Bayesian learning of probabilistic language models*. PhD thesis.

[57] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. *Grammatical inference and applications*, pages 106–118, 1994.

[58] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[59] M. Sutton and A. Greene. The art of file format fuzzing. In *Blackhat USA conference*, 2005.

[60] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[61] The Flex Project. Flex: The fast lexical analyzer. `http://flex.sourceforge.net`, 2008.

[62] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *International Conference on Formal Engineering Methods*, pages 274–289. Springer, 2004.

[63] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning. Experiences with model inference assisted fuzzing. In *WOOT*, 2008.

[64] W3C. `https://www.w3.org/TR/2008/REC-xml-20081126`, 2008.

[65] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[66] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna. Automatic network protocol analysis. In *NDSS*, volume 8, pages 1–14, 2008.

[67] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[68] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2015.

A. Properties of Phase One

We prove the desired properties discussed in Section 3 for the generalization steps proposed in Section 4. First, we prove Proposition 4.1, which says that the candidates in phase one are monotone. Next, we prove Proposition 4.4, which says that the contexts constructed by phase one satisfy (1); as discussed in Section 4.3, this result implies that the corresponding checks α constructed in phase one are valid (i.e., $\alpha \in \tilde{L} \setminus \hat{L}_i$).

A.1 Proof of Proposition 4.1

There are two cases:

- **Repetitions:** Every candidate has form (omitting bracketed substrings) $R' = P\alpha_1\alpha_2^*\alpha_3Q$, where the current language is $R = P\alpha Q$ and $\alpha = \alpha_1\alpha_2\alpha_3$. Since $\alpha \in \mathcal{L}(\alpha_1\alpha_2^*\alpha_3)$, it is clear that
$$\mathcal{L}(R) = \mathcal{L}(P\alpha Q) \subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha_3Q) = \mathcal{L}(R').$$
- **Alternations:** Every candidate has form (omitting bracketed substrings) $R' = P(\alpha_1 + \alpha_2)Q$, where the current language is $R = P\alpha Q$ and $\alpha = \alpha_1\alpha_2$. Note that an bracketed expression $[\alpha]_{\text{alt}}$ always occurs within a repetition, so the candidate has form
$$\begin{aligned} R' &= ...(... + (\alpha_1 + \alpha_2) + ...)^*... \\ &= ...(... + (\alpha_1 + \alpha_2)^* + ...)^*..., \end{aligned}$$
so since $\alpha \in (\alpha_1 + \alpha_2)^*$, we have
$$\mathcal{L}(R) = \mathcal{L}(P\alpha Q) \subseteq \mathcal{L}(P(\alpha_1 + \alpha_2)^*Q) = \mathcal{L}(R').$$

The result follows. □

A.2 Proof of Proposition 4.4

We prove by induction. The initial context (ϵ, ϵ) for $[\alpha_{\text{in}}]_{\text{rep}}$ clearly satisfies (1). Next, assume that the context (γ, δ) for the current language satisfies (1). There are two cases:

- **Repetitions:** Suppose that the current language is $R = P[\alpha]_{\text{rep}}Q$ and the candidate is $R' = P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{alt}}$. Then, the context constructed for $[\alpha_2]_{\text{alt}}$ is $(\gamma', \delta') = (\gamma\alpha_1, \alpha_3\delta)$. Also, let $P' =$ "$P\alpha_1($" and $Q' =$ "$)^*\alpha_3Q$", so $R' = P'\alpha_2Q'$. Then, for any $\alpha' \in \Sigma^*$, we have
$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha_1\alpha'\alpha_3\delta &\in \mathcal{L}(P\alpha_1\alpha'\alpha_3Q) \\ &\subseteq \mathcal{L}(P\alpha_1(\alpha')^*\alpha_3Q) \\ &= \mathcal{L}(P'\alpha'Q'), \end{aligned}$$
where the first inclusion follows by applying (1) to the context (γ, δ) with $\alpha_1\alpha'\alpha_3 \in \Sigma^*$. Therefore, the context (γ', δ') satisfies (1). Similarly, the context constructed for $[\alpha_3]_{\text{rep}}$ is $(\gamma', \delta') = (\gamma\alpha_1\alpha_2, \delta)$. Also, let $P' = P\alpha_1\alpha_2^*$ and $Q' = Q$, so $R' = P'\alpha_3Q'$. Then, for any $\alpha' \in \Sigma^*$, we have
$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha_1\alpha_2\alpha' &\in \mathcal{L}(P\alpha_1\alpha_2\alpha'Q) \\ &\subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha'Q) \\ &= \mathcal{L}(P'\alpha'Q'), \end{aligned}$$
where the first inclusion follows by applying (1) to the context (γ, δ) with $\alpha_1\alpha_2\alpha' \in \Sigma^*$. Therefore, the context (γ', δ') satisfies (1).
- **Alterations:** Suppose that the current language is $R = P[\alpha]_{\text{alt}}Q$ and the candidate is $R' = P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$. Then, the context constructed for $[\alpha_1]_{\text{rep}}$ is $(\gamma', \delta') = (\gamma, \alpha_2\delta)$. Also, let $P' =$ "$P($" and $Q' =$ "$+\alpha_2)Q$", so $R' = P'\alpha_2Q'$. Then, for any $\alpha' \in \Sigma^*$, we have
$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha'\alpha_2\delta &\in \mathcal{L}(P\alpha'\alpha_2Q) \\ &= \mathcal{L}(P(\alpha' + \alpha_2)^*Q) \\ &= \mathcal{L}(P(\alpha' + \alpha_2)Q) \\ &= \mathcal{L}(P'\alpha'Q'), \end{aligned}$$
where the inclusion follows by applying (1) to the context (γ, δ) with $\alpha'\alpha_2 \in \Sigma^*$, and the equality on the third line follows as in the proof of Proposition 4.1 (in Section A.1).

The claim follows. □

B. Expressiveness of Phase One

In this section, we prove the expressiveness results discussed in Section 4.1.

B.1 Correspondence to Derivations in $\mathcal{C}_{\text{regex}}$

In this section, we prove Proposition 4.2, which says that derivations in $\mathcal{C}_{\text{regex}}$ can be transformed to series of generalization steps in phase one of our algorithm. In particular, consider the derivation of a regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$:
$$T_{\text{rep}} = \eta_1 \Rightarrow ... \Rightarrow \eta_n = R.$$
We prove that for each i, there is a series of generalization steps
$$R_i \Rightarrow R_{i+1} \Rightarrow ... \Rightarrow R_n = R$$
such that each R_j (for $i \leq j \leq n$) maps to η_j in the way defined in Section 4.1 (i.e., by replacing $[\alpha]_\tau$ with T_τ); we express this mapping as $\eta_j = \overline{R_j}$. The result follows since for $i = 1$, we get $[\alpha]_{\text{rep}} = R_1 \Rightarrow ... \Rightarrow R_n = R$, so we can take $\alpha_{\text{in}} = \alpha$.

We prove by (backward) induction on the derivation. The base case $i = n$ is trivial, since $\eta_n \in \mathcal{L}(\mathcal{C}_{\text{regex}})$, so we can take $R_n = \eta_n$ since $\overline{R_n} = R_n = \eta_n$. Now, suppose that we have a series of generalization steps $R_{i+1} \Rightarrow ... \Rightarrow R_n = R$ that satisfies the claimed property. It suffices to show that we can construct R_i such that $R_i \Rightarrow R_{i+1}$ is a generalization step and $\overline{R_i} = \eta_i$. Consider the following cases for the step $\eta_i \Rightarrow \eta_{i+1}$ in the derivation:

- Step $\mu T_{\text{rep}}\nu \Rightarrow \mu\beta T_{\text{alt}}^{*}T_{\text{rep}}\nu$: Then, we must have
$$R_{i+1} = P\alpha_1[\alpha_2]_{\text{alt}}[\alpha_3]_{\text{rep}}Q,$$
where $\overline{P} = \mu$, $\overline{Q} = \nu$, and $\alpha_1 = \beta$. Also, since R_{i+1} is valid, we have $\alpha_1, \alpha_2, \alpha_3 \neq \epsilon$. Therefore, we can take
$$R_i = P[\alpha]_{\text{rep}}Q,$$
where $\alpha = \alpha_1\alpha_2\alpha_3 \neq \epsilon$. The remaining productions for T_{rep} are similar. In particular, the assumption that $\beta \neq \epsilon$ in these derivations is needed to ensure that $\alpha \neq \epsilon$.
- Step $\mu T_{\text{alt}}\nu \Rightarrow \mu(T_{\text{rep}} + T_{\text{alt}})\nu$: Then, we must have
$$R_{i+1} = P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q,$$
where $\overline{P} = \mu$ and $\overline{Q} = \nu$. Also, since R_{i+1} is valid, we have $\alpha_1, \alpha_2 \neq \epsilon$. Therefore, we can take
$$R_i = P[\alpha]_{\text{alt}}Q,$$
where $\alpha = \alpha_1\alpha_2 \neq \epsilon$. The remaining production for T_{alt} is similar.

The result follows. □

B.2 Expressiveness of $\mathcal{C}_{\text{regex}}$

In this section, we prove Proposition 4.3, which says that any regular language can be expressed as $\mathcal{L}(R_1 + ... + R_m)$, where $R_1, ..., R_m \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ are regular expressions that can be synthesized by phase one of our algorithm.

We slightly modify $\mathcal{C}_{\text{regex}}$, by introducing a new nonterminal T_{regex}, taking T_{regex} to be the start symbol, and adding productions
$$T_{\text{regex}} ::= \bar{\epsilon} \mid T_{\text{alt}} \mid \bar{\epsilon} + T_{\text{alt}},$$
where $\bar{\epsilon} \in \Sigma_{\text{regex}}$ is a newly introduced terminal denoting the regular expression for the empty language. This modification has two effects:

- Now, regular expressions $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ can have top-level alternations.
- Furthermore, the top-level alternation can explicitly include the empty string $\bar{\epsilon}$ (e.g., $R = \bar{\epsilon} + \mathtt{a}$).

As described in Section 4.1, the first modification can be addressed by using multiple inputs (see Section 6.1), which allows our algorithm to learn top-level alternations. The second modification can be addressed by including a seed input $\bar{\epsilon} \in E_{\text{in}}$, in which case phase one of our algorithm synthesizes $\bar{\epsilon}$ (since there is nothing for it to generalize).

Now, let the context-free grammar $\tilde{\mathcal{C}}_{\text{regex}}$ be a standard grammar for regular expressions:
$$T ::= \beta \mid TT \mid T + T \mid T^{*}. \tag{2}$$

It suffices to show that for any $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$, there exists $R' \in \mathcal{L}(\tilde{\mathcal{C}}_{\text{regex}})$ such that $\mathcal{L}(R) = \mathcal{L}(R')$ (which we express as $R \equiv R'$).

First, we prove the result for $\mathcal{C}_{\text{regex}}^{\epsilon}$, which is identical to $\mathcal{C}_{\text{regex}}$ except that we allow $\beta = \epsilon$. Let $R \in \mathcal{L}(\tilde{\mathcal{C}}_{\text{regex}})$. Suppose that either $R = S_1 + S_2$, $R = S_1S_2$, or $R = \beta$. We claim that we can express R as
$$\begin{aligned} R &\equiv X_1 + ... + X_n \\ X_i &= Y_{i,1}...Y_{i,k_i} \qquad (1 \le i \le n) \end{aligned} \tag{3}$$
where either $Y_{i,j} = \beta$ or $Y_{i,j} = W_{i,j}^{*}$ for each i and j. Consider two possibilities:

- Suppose R can be expressed in the form (3), but $Y_{i,j} = Z_1 + Z_2$. Then
$$\begin{aligned} X_i &= Y_{i,1}...Y_{i,j}...Y_{i,k_i} \\ &= Y_{i,1}...(Z_1 + Z_2)...Y_{i,k_i} \\ &\equiv Y_{i,1}...Z_1...Y_{i,k_i} + Y_{i,1}...Z_2...Y_{i,k_i} \end{aligned}$$
which is again in the form (3).
- Suppose R has the form (3), but $Y_{i,j} = Z_1Z_2$. Then
$$X_i = Y_{i,1}...Y_{i,j}...Y_{i,k_i} = Y_{i,1}...Z_1Z_2...Y_{i,k_i}$$
which is again in the form (3).

Note that either $R = S_1 + S_2$ or $R = S_1S_2$, so R starts in the form (3). Therefore, we can repeatedly apply the above two transformations until $Y_{i,j} = \beta$ or $Y_{i,j} = W_{i,j}^{*}$ for every i and j. This process must terminate because the parse tree for R is finite, so the claim follows.

Now, we construct $R' \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{alt}})$ such that $R \equiv R'$ by structural induction. First, suppose that either $R = S_1 + S_2$, $R = S_1S_2$, or $R = \beta$. Then we can express R in the form (3). By induction, $W_{i,j} \equiv W'_{i,j}$ for some $W'_{i,j} \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{alt}})$ for every i and j. By the definition of T_{rep}, we have $X_i \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{rep}})$, so by the definition of T_{alt}, we have $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{alt}})$, so the inductive step follows.

Alternatively, suppose $R = S^{*}$. If $S = S_1^{*}$, then $R \equiv S_1^{*}$, so without loss of generality assume $S = S_1 + S_2$, $S = S_1S_2$, or $S = \beta$, so by the previous argument, we have $S \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{alt}})$. Since $T_{\text{alt}} ::= T_{\text{rep}}$ and $T_{\text{rep}} ::= T_{\text{alt}}^{*}$, we have $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon}, T_{\text{alt}})$, so again the inductive step follows. Finally, since $T ::= T_{\text{alt}}$, we have $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}^{\epsilon})$.

Now, we modify the above proof to show that as long as $\epsilon \notin \mathcal{L}(R)$, we have $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$. As before, we proceed by structural induction. Suppose that either $R = S_1 + S_2$, $R = S_1S_2$, or $R = \beta$, so we can express R in the form (3). First, consider the case $Y_{i,j} = \beta$; if $\beta = \epsilon$, we can remove $Y_{i,j}$ from X_i unless $k_i = 1$. However, if $Y_{i,j} = \beta = \epsilon$ and $k_i = 1$, whence $X_i = \epsilon$ so $\epsilon \in \mathcal{L}(R)$, a contradiction; hence, we can always drop $Y_{i,j}$ such that $Y_{i,j} = \epsilon$. For the remaining $Y_{i,j} = \beta$, we have $Y_{i,j} \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$ by the definition of $\mathcal{C}_{\text{regex}}$.

Second, consider the case $Y_{i,j} = Z_{i,j}^*$. Let $Z'_{i,j}$ be a regular expression such that $\mathcal{L}(Z'_{i,j}) = \mathcal{L}(Z_{i,j}) - \{\epsilon\}$, and note that

$$Y_{i,j} = Z_{i,j}^* \equiv (Z'_{i,j})^*.$$

By induction, we know that $Z_{i,j} \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$, so $Y'_{i,j} = (Z'_{i,j})^* \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$ by the definition of $\mathcal{C}_{\text{regex}}$.

For each X_i, we remove every $Y_{i,j} = \beta = \epsilon$ and replace every $Y_{i,j} = Z_{i,j}^*$ with $Y'_{i,j} = (Z'_{i,j})^*$ to produce $X'_i \equiv X_i$. By definition of $\mathcal{C}_{\text{regex}}$, we have $X_i \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$, so $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$ as claimed; now, the case $R = S^*$ follows by the same argument as before.

For any R such that $\epsilon \in \mathcal{L}(R)$, we can write $R = \epsilon + S$ where $\epsilon \notin \mathcal{L}(S)$ and apply the above argument to S. Since $T ::= \epsilon + T_{\text{alt}}$ is a production in $\mathcal{C}_{\text{regex}}$, we have shown that $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ for any regular expression R. $\square$

C. Properties of Phase Two

We prove the desired properties discussed in Section 3 for the generalization steps proposed in Section 5. As discussed in Section 5.2, the candidates constructed in phase two are clearly monotone (since equating nonterminals in a context-free grammar can only enlarge the generated language). We prove Proposition 5.1, which formalizes our intuition about how candidates constructed in phase two merge repetition subexpressions; as discussed in Section 5.3, this result implies that the checks constructed in phase two are valid.

C.1 Proof of Proposition 5.1

In this section, we sketch a proof of Proposition 5.1. In particular, we show that if we merge two nonterminals $(A'_i, A'_j) \in M$ by equating them in the context-free grammar $\hat{C}$ (translated from $\hat{R}$) to obtain $\tilde{C}$, then the repetition subexpressions R in $\hat{R} = PRQ$ (corresponding to A'_i) and R' in $\hat{R} = P'R'Q'$ (corresponding to A'_j) are merged; i.e., $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ and $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$. While we prove the result for the translation $\hat{C}$ of $\hat{R}$, note that (i) subsequent merges can only enlarge the generated language, and (ii) the order in which merges are performed does not affect the final context-free grammar, so the result holds for any step of phase two of our algorithm.

Note that equating two nonterminals $(A'_i, A'_j) \in M$ in $\hat{C}$ is equivalent to adding productions $A'_i \to A'_j$ and $A'_j \to A'_i$ to $\hat{C}$. Therefore, Proposition 5.1 shows that both $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ and $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$. It suffices to show that adding $A'_i \to A'_j$ to $\hat{C}$ results in the context-free grammar $\tilde{C}$ satisfying $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ (intuitively, this is a one-sided merge that only merges $\hat{R}'$ into $\hat{R}$, not vice versa).

We use the fact that our algorithm for translating a regular expression to a context-free grammars works more generally for any regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ derived from T_{rep} in according to the meta-grammar $\mathcal{C}_{\text{regex}}$. In particular, if we consider the series of generalization steps

$$\alpha_{\text{in}} = R_1 \Rightarrow ... \Rightarrow R_n = \hat{R},$$

we get a corresponding derivation

$$T_{\text{rep}}^{(1)} = \eta_1 \Rightarrow ... \Rightarrow \eta_n = \hat{R}$$

in $\mathcal{C}_{\text{regex}}$ as described in Section 4.1. Similarly to the labels on bracketed strings in the series of generalization steps, we label each nonterminal in the derivation with the index at which it is expanded. For example, for the derivation corresponding to the the series of generalization steps in Figure 3 is

$$\begin{aligned}
&T_{\text{rep}}^{(1)} \\
&\Rightarrow (T_{\text{alt}}^{(2)})^* \\
&\Rightarrow (T_{\text{rep}}^{(3)})^* \\
&\Rightarrow (\texttt{<a>}(T_{\text{alt}}^{(5)})^* T_{\text{rep}}^{(4)})^* \\
&\Rightarrow (\texttt{<a>}(T_{\text{alt}}^{(5)})^* \texttt{</a>})^* \\
&\Rightarrow (\texttt{<a>}(T_{\text{rep}}^{(8)} + T_{\text{alt}}^{(6)})^* \texttt{</a>})^* \\
&\Rightarrow (\texttt{<a>}(T_{\text{rep}}^{(8)} + T_{\text{rep}}^{(7)})^* \texttt{</a>})^* \\
&\Rightarrow (\texttt{<a>}(T_{\text{rep}}^{(8)} + \texttt{i})^* \texttt{</a>})^* \\
&\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^* \texttt{</a>})^*
\end{aligned}$$

Now, each nonterminal A_i is associated to step i in the derivation, and we add productions for A_i depending on step i in the derivation (and auxiliary nonterminals A'_i if step i in the derivation expands nonterminal T_{rep} in the meta-grammar):

- Step $\mu T_{\text{rep}}^{(i)} \nu \Rightarrow \mu \beta (T_{\text{alt}}^{(j)})^* T_{\text{rep}}^{(k)} \nu$: We add productions $A_i \to \beta A'_i A_k$ and $A'_i \to \epsilon \mid A'_i A_j$.
- Step $\mu T_{\text{alt}}^{(i)} \nu \Rightarrow \mu (T_{\text{rep}}^{(j)} + T_{\text{alt}}^{(k)}) \nu$: We add production $A_i \to A_j \mid A_k$.

Now, consider step i in the derivation, where productions for A_i and A'_i were added to $\hat{C}$. Then, step i of the derivation has form

$$\mu T_{\text{rep}}^{(i)} \nu \Rightarrow \mu \beta (T_{\text{alt}}^{(j)})^* T_{\text{rep}}^{(k)} \nu.$$

We can assume without loss of generality that we expand $T_{\text{rep}}^{(i)}$ last; i.e., $\mu = \overline{\mu} = P$ and $\nu = \overline{\nu} = Q$ do not contain any nonterminals. Therefore, the derivation has form

$$\begin{aligned}
(\eta_1 = T_{\text{rep}}^{(1)}) &\Rightarrow ... \\
&\Rightarrow (\eta_i = P T_{\text{rep}}^{(i)} Q) \\
&\Rightarrow (\eta_{i+1} = P \beta (T_{\text{alt}}^{(j)})^* T_{\text{rep}}^{(k)} Q) \\
&\Rightarrow ... \\
&\Rightarrow (\eta_n = PRQ).
\end{aligned}$$

Now, note that the following derivation is also in $\mathcal{C}_{\text{regex}}$:

$$\begin{aligned}(\eta_1 = T_{\text{rep}}^{(1)}) &\Rightarrow ... \\ &\Rightarrow (\eta_i = PT_{\text{rep}}^{(i)}Q) \\ &\Rightarrow (\eta'_{i+1} = P\beta'(T_{\text{alt}}^{(j')})^* T_{\text{rep}}^{(k')}Q) \\ &\Rightarrow ... \\ &\Rightarrow \eta'_{n'} = PR'Q\end{aligned}$$

since R' can be derived from T_{rep}. Note that $\hat{R}' = PR'Q$ is exactly the regular expression produced by this derivation. Then, let $\hat{C}'$ be the context-free grammar obtained by applying our translation algorithm to $\hat{R}'$ using this derivation.

Note that $\hat{C}'$ has the same productions as $\hat{C}$, except the productions for A_i in $\hat{C}$ (i.e., all productions added on step i of the derivation and after) have been replaced with productions A_i in $\hat{C}'$ such that $\mathcal{L}(\hat{C}', A_i) = \mathcal{L}(R')$. Since $\mathcal{L}(R') \subseteq \mathcal{L}(\tilde{C}, A_i)$, and the nonterminals involved in the productions for A_i do not occur in $\tilde{C}$, it is clear that adding the productions for A_i in $\hat{C}'$ to $\tilde{C}$ does not modify $\mathcal{L}(\tilde{C})$. By construction, the other productions in $\hat{C}'$ are in $\hat{C}$, so they are also in $\tilde{C}$. Therefore, $\mathcal{L}(\hat{C}') \subseteq \mathcal{L}(\tilde{C})$. The result follows, since $\mathcal{L}(\hat{C}') = \mathcal{L}(\hat{R}') = \mathcal{L}(PR'Q)$. □

D. Expressiveness of Phase Two

In this section, we give a proof sketch of the expressiveness result stated in Proposition 5.3 of Section 5.4. Let C be a generalized matching parentheses grammar. Suppose that nonterminal S_i ($1 \leq i \leq n$) corresponds to production

$$S_i \to R_i(S_{i_1} + ... + S_{i_{k_i}})^* R'_i.$$

First, we need to identify a context such that S_i can occur in a derivation in C; in particular, we want to construct a derivation of the form

$$\begin{aligned}S_0 = S_{i,1} &\Rightarrow R_{i,1}S_{i,2}R'_{i,1} \\ &\Rightarrow R_{i,1}R_{i,2}S_{i,3}R'_{i,2}R'_{i,1} \\ &\Rightarrow ... \\ &\Rightarrow R_{i,1}...R_{i,h_i}S_iR'_{i,h_i}...R'_{i,1}.\end{aligned}$$

To do so, we construct a directed graph G with vertices $\{S_1, ..., S_n\}$ and edges $S_i \to S_j$ whenever the production for S_i has form

$$S_i \to R_i(... + S_j + ...)^* R'_i.$$

In other words, an edge indicates that S_j is contained in a derivation of S_i. Then, we can constructed the desired derivation using a spanning tree rooted at S_1, in particular, by examining the path

$$S_1 = S_{i,1} \to ... \to S_{h_i} \to S_i$$

from S_1 to S_i in this spanning tree. Note that if no path exists, then S_i cannot occur in any derivation of S_1.

Now, for each pair of regular expressions R_i and R'_i ($1 \leq i \leq n$), let $\alpha_i \in \mathcal{L}(R_iR'_i) \subseteq \mathcal{L}(C, S_i)$. Then, let

$$\begin{aligned}X_i &= R_{i,1}...R_{i,h_i}Y_iR'_{i,h_i}...R'_{i,1} \\ Y_i &= (R_i(\alpha^*_{i_1} + ... + \alpha^*_{i_{k_i}})R'_i)^*.\end{aligned}$$

Intuitively, X_i is constructed according to the derivation of S_1 containing S_i, and Y_i is constructed using the production for S_i. In paricular, by construction, $\mathcal{L}(X_i) \subseteq \mathcal{L}(C)$.

Consider the following regular expression:

$$\begin{aligned}X &= X_1 + ... + X_n \\ M &= \{(Y_i, \alpha^*_{j_k}) \mid i = j_k\}.\end{aligned}$$

We claim that translating X and M into a context-free grammar yields a grammar C' such that $\mathcal{L}(C) = \mathcal{L}(C')$. First, we show that each production in C is also in C', which implies that $\mathcal{L}(C) \subseteq \mathcal{L}(C')$. In particular, note that the translation algorithm introduces exactly one nonterminal for each Y_i, since two repetition nodes Y_i and Y_j are never merged together, and every other repetition node in X is merged with a Y_i node. Let S'_i be the nonterminal introduced for Y_i; since each α_{i_j} is merged with Y_{i_j}, the production added to C' is

$$S'_i \to (R_i(S'_{i_1} + ... + S'_{i_{k_i}})^* R'_i)^*,$$

which is equivalent to the production for S_i in C.

Next, we show that $\mathcal{L}(X) \subseteq \mathcal{L}(C)$. First, note that by construction, $\mathcal{L}(X_i) \subseteq \mathcal{L}(C)$ for each $1 \leq i \leq n$, so $\mathcal{L}(X) \subseteq \mathcal{L}(C)$. Second, applying each merge in M does not affect this invariant, since Y_i and $\alpha^*_{j_k}$ can both be replaced with $S_i = S_{j_k}$. Therefore, $\mathcal{L}(C) = \mathcal{L}(C')$. □