Synthesizing Program Input Grammars

Osbert Bastani
Stanford University
obastani@cs.stanford.edu

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Percy Liang
Stanford University
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 programs with highly structured inputs. We show that GLADE consistently increases the incremental coverage compared to two baseline fuzzers.

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 [53] 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 [40].

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 [22, 26, 34]. For example, such inputs can be used to find bugs in real-world programs [23, 35, 43, 58], learn abstractions [37], predict performance [27], and aid dynamic analysis [38]. Beyond fuzzing, a grammar synthesis algorithm could be used to reverse engineer network protocol message formats that can help security analysts discover vulnerabilities in network programs [8, 31, 32, 57]. Synthesized grammars could also be used to whitelist program inputs, thereby preventing exploits [44, 45, 50].

Approaches to synthesizing program input grammars can be *static* or *dynamic*. A static algorithm would analyze the program source code to generate a representation of valid inputs. Such an approach is difficult when only the program binaries are available or when part of the source code (e.g., libraries) is missing. Moreover, static techniques often require program-specific configuration or tuning, and can scale poorly to large programs [18]. In contrast, a dynamic algorithm treats the program as a *blackbox*, only requiring the ability to execute the program on a given input and observe its corresponding output. Since the synthesis algorithm does not look at the program source code at all, its complexity only depends on the target grammar of valid inputs and not the size of the program.

Our main contribution is a new algorithm for synthesizing program input grammars in this blackbox setting. More precisely, our algorithm synthesizes context-free grammars representing valid program inputs using the program executable to answer *membership queries* (i.e., is a given input valid) along with a small set of *seed inputs* (i.e., examples of valid inputs). Seed inputs are often readily available; in our evaluation, we use small test suites that come with the program or examples from the documentation.

A number of approaches to inferring grammars have been proposed, which have been successfully applied to problems in a range of fields, including robotics, linguistics, and biology [14]. However, we found them to be unsuitable for our problem of learning program input grammars. We observe that existing algorithms tend to substantially overgeneralize. Additionally, despite having polynomial running time, they can be very slow on our problem instances. In particular, we found that two widely used algorithms [6, 12, 13, 19, 54], the L-Star algorithm [3] and RPNI [39], were unable to learn or approximate even simple input languages such as XML, and furthermore did not scale even to small sets of seed inputs. Furthermore, we show that the poor performance of L-Star and RPNI are not because they are restricted to learning regular languages.

There are many negative results on learning context-free languages, and approaches for synthesizing context-free grammars are necessarily heuristic [29]. Even so, existing approaches for doing so are primarily theoretical [28, 49], and have been found to suffer a number of shortcomings. For example, [49] requires an enormous number of queries, and [28] is prohibitively expensive and unpredictable; see [29] for a discussion. To the best of our knowledge, this paper describes the first *practical* technique to learn context-free grammars from positive examples and membership queries. As shows in the evaluation (Sections 8 and 9), we have successfully applied our algorithm to learn input formats for real programs.

We propose a novel framework for grammar synthesis, which iteratively constructs a series of increasingly general languages by performing a sequence of *generalization steps*. Each step first proposes a number of candidate languages that generalize the current language, and then uses membership queries to try to identify and reject candidates that overgeneralize. The framework specifies properties that the generalization steps should satisfy; any implementation satisfying these properties leads to a grammar synthesis algorithm. In particular, our grammar synthesis algorithm implements two kinds of generalization steps tailored to learning program input grammars that occur in practice. The first kind learns

- Context-free grammar C_{XML} that generates the target language $\mathcal{L}(C_{\text{XML}})$, which has terminals $\Sigma_{\text{XML}} = \{\texttt{a}, ..., \texttt{z}, \texttt{<}, \texttt{>}, \texttt{/}\}$, start symbol A_{XML}, and productions

$$A_{\text{XML}} \to \texttt{a} + ... + \texttt{z}$$
$$A_{\text{XML}} \to \texttt{<a>} A_{\text{XML}} \texttt{}$$
$$A_{\text{XML}} \to A^*_{\text{XML}}.$$

- Oracle

$$\mathcal{O}_{\text{XML}}(\alpha) = \begin{cases} 1 \text{ if } \alpha \in \mathcal{L}(C_{\text{XML}}) \\ 0 \text{ otherwise.} \end{cases}$$

- Seed inputs $E_{\text{XML}} = \{\alpha_{\text{XML}}\}$, where

$$\alpha_{\text{XML}} = \texttt{<a>hi} \in \mathcal{L}(C_{\text{XML}}).$$

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}.

repetition and alternation constructs characteristic of regular expressions. The second kind learns recursive productions characteristic of context-free grammars, including parenthesis matching and related constructs.

In general, it is impossible to guarantee exact learning given only a membership oracle, even for regular languages [3]. We have a completeness result that the first phase of our algorithm can at least search the entire space of regular expressions, but the search necessarily relies on heuristics. As discussed, synthesizing context-free grammars faces even greater theoretical obstacles, so we design heuristics that learn context-free properties that are important in our application domain.

We implement our approach in a tool called GLADE,[1] which uses our algorithm to synthesize a context-free grammar $\hat{C}$ approximating the target program input language L_*, given blackbox access to the program and seed inputs $E_{\text{in}} \subseteq L_*$. We show that GLADE consistently outperforms both the L-Star and RPNI algorithms, even when restricted to learning regular languages.

Next, we show that the grammars synthesized by GLADE can be used to fuzz 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 9.1) to generate new test inputs. We show that GLADE increases line coverage compared to both a naïve fuzzer and the production fuzzer afl-fuzz [59]. Our contributions are:

- We introduce a framework for synthesizing grammars from seed inputs and blackbox program access (Section 3).
- We instantiate this framework to obtain a grammar synthesis algorithm. Our algorithm first learns regular properties such as repetitions and alternations (Section 4), and then learns recursive productions characteristic of context-free grammars (Section 5).
- We implement our grammar synthesis algorithm in GLADE, and show that GLADE outperforms two widely used language learning algorithms, L-Star and RPNI in our application domain (Section 8).
- We describe how GLADE fuzzes programs, and show that GLADE outperforms two baseline fuzzers (Section 9).

2. Problem Formulation

Suppose we are given a program that takes inputs in Σ^*, where Σ is the input alphabet (e.g., ASCII characters). Furthermore, only a small subset of the inputs are valid; we let $L_* \subseteq \Sigma^*$ denote the language of valid program inputs, which we call the *target language*. Our goal is to approximately learn L_* from blackbox program access and seed inputs $E_{\text{in}} \subseteq L_*$. We represent blackbox access to the program as having an oracle $\mathcal{O}$ such that $\mathcal{O}(\alpha) = \mathbb{I}[\alpha \in L_*]$.[2] In particular, we run the program on input $\alpha \in \Sigma^*$, and conclude that α is a valid input (i.e., $\alpha \in L_*$) as long as the program does not print an error message.

The *language learning problem* is to learn a context-free language $\hat{L}$ that approximates L_* (which may or may not be context-free, see below) given an oracle $\mathcal{O}$ and seed inputs E_{in} (we discuss the notion of approximation below). Access to an oracle is crucial to avoid overgeneralizing; without $\mathcal{O}$, we cannot reject learning the language $\hat{L} = \Sigma^*$. Conversely, the seed inputs E_{in} are crucial since they give the algorithm a starting point from which to generalize. The language learning problem has been studied before in other, theoretical settings (see [29] for a discussion), but to the best of our knowledge, we are the first study the application where L_* is a program input language.

For example, suppose the program is a parser whose input grammar is the XML-like grammar C_{XML} shown in Figure 1. We use $+$ to denote alternations and $*$ (the Kleene star) to denote repetitions. For clarity, we use the color blue to highlight terminals that are part of regular expressions or context-free grammars. Given α_{XML} as a seed input and $\mathcal{O}_{\text{XML}}$ as the oracle, an instance of the language learning problem is to learn a context-free language $\hat{L}$ approximating the language $L_* = \mathcal{L}(C_{\text{XML}})$ of XML-like strings.

Ideally, we would learn L_* exactly (i.e., $\hat{L} = L_*$). However, in general, it is impossible to guarantee exact learning (even if L_* is context-free). Instead, we want $\hat{L}$ to be a "high quality" approximation of L_*. To formalize the quality of an approximation, we assume the ability to sample random strings $\alpha \sim L_*$ from the target language and $\alpha \sim \hat{L}$ from the learned language. Our languages of interest are generated by context-free grammars; we describe how to sample such languages in Section 7. Then, we define the quality of our synthesized grammar using the standard notions of precision and recall:

DEFINITION 2.1. Let L_* be the target language and $\hat{L}$ be the learned language. The *precision* of $\hat{L}$ is $\Pr_{\alpha \sim \hat{L}}[\alpha \in L_*]$ and the *recall* of $\mathcal{L}(\hat{C})$ is $\Pr_{\alpha \sim L_*}[\alpha \in \hat{L}]$.[3]

The precision measures the accuracy of sampling from $\hat{L}$; i.e., the number of sampled strings α that the learned language believes to be valid ($\alpha \in \hat{L}$) and are actually valid ($\alpha \in L_*$). On the other hand, the recall measures the accuracy of using $\hat{L}$ as an oracle; i.e., the number of strings α that are actually valid ($\alpha \in L_*$) and the learned language also believes to be valid ($\alpha \in \hat{L}$). Both are desirable; on one hand, the language $\hat{L} = \{\alpha\}$ (for some $\alpha \in L_*$) has perfect precision but typically low recall, whereas the language $\hat{L} = \Sigma^*$ has perfect recall but typically low precision (assuming most strings are not in L_*).

When L_* is not context-free, we aim to learn a context-free subset $\hat{L} \subseteq L_*$. For example, suppose we generalize the XML-like grammar in Figure 1 to the grammar $\tilde{C}_{\text{XML}}$ allowing any sequence of letters in the matching tags; then, $\tilde{C}_{\text{XML}}$ includes the (infinitely many) productions

$$\tilde{A}_{\text{XML}} \to \texttt{a} + ... + \texttt{z}$$
$$\tilde{A}_{\text{XML}} \to \texttt{<}\alpha\texttt{>} \tilde{A}_{\text{XML}} \texttt{</}\alpha\texttt{>} \qquad (\forall \alpha \in [\texttt{a} - \texttt{z}]^*)$$
$$\tilde{A}_{\text{XML}} \to \tilde{A}^*_{\text{XML}}.$$

[1] GLADE stands for Grammar Learning for AutomateD Execution.

[2] Here, $\mathbb{I}$ is the indicator function, so $\mathbb{I}[\mathcal{C}]$ is 1 if $\mathcal{C}$ is true and 0 if $\mathcal{C}$ is false.

[3] Here, $\Pr_{\alpha \sim \hat{L}}[\mathcal{C}]$ is the probability that α sampled from $\hat{L}$ satisfies $\mathcal{C}$.

Algorithm 1 Our framework for language learning. Given an input example $\alpha_{\text{in}} \in L_*$ and an oracle $\mathcal{O}$ for L_*, it learns an approximation $\hat{L}$ of the target language L_*. Implementations of CONSTRUCTCANDIDATES and CONSTRUCTCHECKS are specified by the user.

```
procedure LEARNLANGUAGE(α_in, O)
    L̂_current ← {α_in}
    while true do
        M ← CONSTRUCTCANDIDATES(L̂_current)
        L̃_chosen ← ∅
        for all L̃ ∈ M do
            S ← CONSTRUCTCHECKS(L̂_current, L̃)
            if CHECKCANDIDATE(S, O) then
                L̃_chosen ← L̃
                break
            end if
        end for
        if L̃_chosen = ∅ then
            return L̂_current
        end if
        L̂_current ← L̃_chosen
    end while
end procedure
procedure CHECKCANDIDATE(S, O)
    for all α ∈ S do
        if O(α) = 0 then
            return false
        end if
    end for
    return true
end procedure
```

Note that $\tilde{C}_{\text{XML}}$ is not context-free. Our algorithm cannot learn $\mathcal{L}(\tilde{C}_{\text{XML}})$; instead, our goal is to learn a context-free subset such as $\mathcal{L}(C_{\text{XML}})$.

3. Grammar Synthesis Framework

In this section, we describe our framework (summarized in Algorithm 1) for designing algorithms that solve the language learning problem. For now, we consider the case where E_{in} consists of a single seed input $\alpha_{\text{in}} \in L_*$, and describe how to extend our approach to multiple seed inputs in Section 6.2. Our framework 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.

Below, we describe the high-level operation of a generalization step; the specific implementation of generalization steps are tailored to the application domain. In particular, our algorithm considers generalizations that introduce regular and context-free constructs. For example, 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, and steps C1-C2 (detailed in Section 5) add recursive productions, which can capture nonregular behaviors.

Candidates. In the first part of the ith generalization step, our algorithm constructs a set of *candidate* languages $\tilde{L}_1, ..., \tilde{L}_n$, with the goal of choosing $\hat{L}_{i+1}$ to be the candidate $\tilde{L}$ that most increases recall 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 by our algorithm at each generalization step (some are omitted, as indicated by ellipses). The candidates 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, and in steps C1-C2, the candidates try to "merge" nonterminals in the context-free grammar (which can induce recursive behaviors).

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; we cannot prove that a given candidate $\tilde{L}$ is precision-preserving since 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 heuristically choose a finite number of *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.

Summary. The following operations require implementations:

- **Candidates:** Given the current language $\hat{L}_i$, construct candidates $\tilde{L}_1, ..., \tilde{L}_n$ (ordered by preference).
- **Checks:** For each candidate $\tilde{L}$, construct checks $S \subseteq \tilde{L} \setminus \hat{L}_i$.

Given implementations of these two operations, Algorithm 1 learns a language $\hat{L}$ given a single seed input α_{in} and and oracle $\mathcal{O}$.

4. Phase One: Regular Expression Synthesis

In this section, we describe the implementation of the first phase of generalization steps considered by our algorithm. The languages in phase one are represented by regular expressions. A step in this phase constructs candidates by adding a single repetition or alternation to the current language. We show a completeness result that every regular expression can be constructed via a series of phase one generalization steps. For efficiency, we perform a greedy search guided by both a heuristic candidate ordering (designed to ensure generality) and a heuristic construction of checks (designed to avoid overgeneralizing).

4.1 Candidates

The regular expressions in this phase carry extra structure. In particular, substrings of terminals $\alpha = \sigma_1...\sigma_k$ may be enclosed in square brackets $[\alpha]_\tau$. This extra structure keeps track of how candidates may generalize the regular expression. Essentially, each candidate adds either a single repetition (if $\tau = \text{rep}$) or a single alternation (if $\tau = \text{alt}$) to a bracketed string α. More precisely, the seed input α_{in} is automatically bracketed as $[\alpha_{\text{in}}]_{\text{rep}}$. Then, each generalization step selects a single bracketed substring $[\alpha]_\tau$ and constructs the following candidates:

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

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

Figure 2. This example shows the generalization steps taken by our grammar synthesis algorithm given seed input α_{XML} and oracle $\mathcal{O}_{\text{XML}}$. The steps occur in two phases. First, the initial language $\{\alpha_{\text{XML}}\}$ is generalized to a regular expression (steps R1-R9). The resulting regular expression is translated to a context-free grammar, which is further generalized (steps C1-C2). The generalization candidates considered at each step are shown in order of preference with the most preferable at the top (with ellipses indicating candidates that are omitted), along with the checks for each candidate. A green check mark ✓ indicates that a check α is accepted (i.e., $\mathcal{O}_{\text{XML}}(\alpha) = 1$) and a red cross ✗ indicates that it is rejected (i.e., $\mathcal{O}_{\text{XML}}(\alpha) = 0$). A star $\star$ is shown next to the candidate selected at each step (i.e., the first one for which all checks pass).

candidate

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

In both cases, the candidate $P\alpha Q$ is also generated. For example, in Figure 2, step R1 selects $[\texttt{<a>hi}]_{\text{rep}}$ and applies the repetition rule, and step R5 selects $[\texttt{hi}]_{\text{alt}}$ and applies the alternation rule. Intuitively, the new bracketed substrings in the constructed candidates are labeled so that the repetition rule and the alternation rule are applied alternatingly.

The repetition and alternation rules satisfy the monotonicity property (proven in Appendix A.1):

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

4.2 Completeness of the Search Space

The repetition and alternation rules are designed so that any regular language can be synthesized using a specific sequence of generalization steps applied to a specific set of seed inputs E_{in} (together with the extension to multiple seed inputs described in Section 6.2). In particular, series of generalization steps correspond to derivations in a context-free grammar $\mathcal{C}_{\text{regex}}$ describing regular expressions (i.e., $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ is a regular expression). To distinguish it from the grammars that we are synthesizing, we call $\mathcal{C}_{\text{regex}}$ a *meta-grammar*. 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 $\beta \in \Sigma^*$ is a nonempty substring of a seed input $\alpha \in E_{\text{in}}$.

Then, for a given series of regular expressions constructed in phase one of our algorithm, replacing each bracketed substring $[\alpha]_\tau$ with the nonterminal T_τ produces a derivation in $\mathcal{C}_{\text{regex}}$. For example, steps R1-R9 in Figure 2 correspond to a derivation in $\mathcal{C}_{\text{regex}}$ as follows:

$[\texttt{<a>hi}]_{\text{rep}}$	T_{rep}
$\Rightarrow ([\texttt{<a>hi}]_{\text{alt}})^*$	$\Rightarrow T^*_{\text{alt}}$
$\Rightarrow ([\texttt{<a>hi}]_{\text{rep}})^*$	$\Rightarrow T^*_{\text{rep}}$
$\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$	$\Rightarrow (\texttt{<a>}T^*_{\text{alt}}T_{\text{rep}})^*$
$\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}T^*_{\text{alt}}\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{alt}})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{rep}})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + \texttt{i})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$

A key result is that 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 a string $\alpha_{\text{in}} \in \mathcal{L}(R)$ such that R can be derived from α_{in} via a sequence of generalization steps.

We give a proof in Appendix B.1. Therefore, for any regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$, we can derive R using a sequence of generalization steps starting from a particular seed input α_{in}.

While $\mathcal{L}(\mathcal{C}_{\text{regex}})$ does not contain every regular expression, it satisfies the following property:

PROPOSITION 4.3. Any regular language L_* can be represented as

$$L_* = \mathcal{L}(R_1 + ... + R_m),$$

where $R_1, ..., R_m \in \mathcal{L}(\mathcal{C}_{\text{regex}})$.

We give a proof in Appendix B.2. As described in Section 6.2, our extension to multiple inputs synthesizes a regular expression $\hat{R}_i$ for each seed input $\alpha_i \in E_{\text{in}}$ and then constructs $\hat{R} = \hat{R}_1 + ... + \hat{R}_m$. Therefore, with this extension, any regular target language L_* can be constructed using specific sequences of generalization steps applied to a particular set of seed inputs E_{in}. Thus, the first phase of our algorithm can search over the space of all regular expressions. 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 4.3.

Furthermore, 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.

4.3 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 $P[\alpha]_{\text{rep}}Q$, our algorithm first prioritizes candidates

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

  with shorter α_1, because α_1 cannot be generalized any further. Second, our algorithm prioritizes candidates with longer α_2, which typically ensures greater generality. For example, in step R3 of Figure 2, if we generalized $[\texttt{<a>hi}]_{\text{rep}}$ to $\texttt{<a>}([\texttt{h}]_{\text{alt}})^*[\texttt{i}]_{\text{rep}}$ instead of $\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}}$, then the regular expression at the end of phase one would be $(\texttt{<a>h}^*\texttt{i}^*\texttt{})^*$, which is less general than the one obtained in step R9 (e.g., it does not include `<a>ih</a>`).
- **Alternations:** If generalizing $P[\alpha]_{\text{alt}}Q$, our algorithm prioritizes candidates

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

  with shorter α_1, since α_1 is successively generalized using the repetition rule rather than the alternation rule, whereas α_2 can be further generalized using the alternation rule. For example, in step R5 of Figure 2, if we generalized $[\texttt{hi}]_{\text{alt}}$ to $[\texttt{hi}]_{\text{rep}}$ instead of $[\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}}$, then on step R6 we would have regular expression $(\texttt{<a>}([\texttt{hi}]_{\text{rep}})^*\texttt{})^*$, which is less general than the one shown in Figure 2.

In either case, the final candidate $P\alpha Q$ is ranked last.

4.4 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 that the current language has the form $P[\alpha]_\tau Q$, where $[\alpha]_\tau$ is the bracketed substring chosen to be generalized, and the candidate language has the form $PR_\alpha Q$, where α has been 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 pair of expressions (P, Q).

Intuitively, we might want to choose $\gamma \in \mathcal{L}(P)$ and $\delta \in \mathcal{L}(Q)$; however, P and Q may not be regular expressions by themselves. For example, on step R5 in Figure 2, we have $P =$ "$\texttt{(<a>}$", $\alpha =$ "$\texttt{hi}$", and $Q =$ "$\texttt{)}^*$" (we quote the expressions to emphasize the placement of parentheses); in this example, P and Q are not regular expressions by themselves.

Instead, P and Q form a regular expression when sequenced together, possibly with a string α' in between (i.e., $P\alpha'Q$). Then, 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 constructs residuals as follows:

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

Next, our algorithm keeps track of a context (γ, δ) for every bracketed string $[\alpha]_\tau$. The context for the initial bracketed substring $[\alpha_{\text{in}}]_{\text{rep}}$ is (ϵ, ϵ). At the end of each generalization step, the contexts (γ, δ) for new bracketed substrings are constructed using the following rules:

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

For example, on step R3, the context for $[\texttt{<a>hi}]_{\text{rep}}$ is (ϵ, ϵ). The residuals constructed 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 $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$ 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 $(\texttt{<a>}, \texttt{})$, and for the new bracketed string $[\texttt{}]_{\text{rep}}$, the algorithm constructs the context $(\texttt{<a>hi}, \epsilon)$.

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

5. Phase Two: Generalizing to Context-Free Grammars

A number of impossibility results highlight the difficulty of learning context-free grammars [14], but we can design heuristics to learn context-free properties commonly found in program input languages such as parenthesis matching. In particular, the second phase of our algorithm *merges* certain subexpressions of the regular expression $\hat{R}$ constructed in the first phase to learn recursive behaviors that cannot be captured by regular languages. For example, consider the regular expression $(\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$ obtained at the end of phase one in Figure 2, which can be written as $\hat{R}_{\text{XML}} = (\texttt{<a>}R_{\text{hi}}\texttt{})^*$, where $R_{\text{hi}} = (\texttt{h} + \texttt{i})^*$. Intuitively, in the (context-free) target language $\mathcal{L}(C_{\text{XML}})$, the expression $\hat{R}_{\text{XML}}$ can recursively appear in place of R_{hi}. More precisely, if we translate $\hat{R}_{\text{XML}}$ to a context-free grammar[4]

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

then we can equate nonterminals A_{XML} and A_{hi} without overgeneralizing; i.e., the context-free grammar $\hat{C}_{\text{XML}}$

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

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

In general, our algorithm first translates the regular expression $\hat{R}$ output by the first phase into a context-free grammar $\hat{C}$. Then, our algorithm considers a second phase of generalization steps that merge certain pairs of nonterminals (A, B) in $\hat{C}$ by equating A and B, thereby allowing us to learn non-regular properties of the target language L_*. In particular, our algorithm considers pairs (A, B) where A and B each correspond to *repetition subexpressions* R of $\hat{R}$, which are subexpressions of $\hat{R}$ with the form $R = R_1^*$. 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}. We begin by describing the translation from a regular expression to a context-free grammar, and then describe the candidates and checks constructed by the generalization steps of phase two.

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 $\hat{R}$ corresponds 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 constructed according to the following rules:

- **Repetition:** If step i generalizes the current language $P[\alpha]_{\text{rep}}Q$ to $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, then we include the productions
$$A_i \to \alpha_1 A'_i A_k, \quad A'_i \to \epsilon \mid A'_i A_j,$$
where j is the step that generalizes $[\alpha_2]_{\text{alt}}$ and k is the step that generalizes $[\alpha_3]_{\text{rep}}$. These two productions handle the semantics of the Kleene star; they are equivalent to the "production" $A_i \to \alpha_1 A_j^* A_k$.
- **Alternation:** If step i generalizes $P[\alpha]_{\text{alt}}Q$ to $P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$, then we include production $A_i \to A_j \mid A_k$, 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 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 $\hat{R}_{\text{XML}} = (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$, 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

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

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 $(A'_{\text{R1}}, A'_{\text{R3}})$.

Then, each generalization step of phase two selects a pair (A'_i, A'_j) from 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 is identical to the current language $\hat{C}$.

In either case, the selected pair is removed from M. 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

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

where $\mathcal{L}(\tilde{C}_{\text{XML}})$ is not regular. The second candidate equals $\hat{C}_{\text{XML}}$.

The next language $\hat{C}'_{\text{XML}}$ is selected to be the candidate $\tilde{C}_{\text{XML}}$ since the constructed checks (described in next section) pass. On step C2, M is empty, so there are no further generalizations and

[4] Recall that regular language is context-free.

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

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

our algorithm returns the current language $\hat{C}'_{\text{XML}}$. Note that $\hat{C}'_{\text{XML}}$ is identical to the original grammar C_{XML} in Figure 1 except instead of the characters $\texttt{a} + ... + \texttt{z}$, it is restricted to characters $\texttt{h} + \texttt{i}$. In Section 6.1, we describe an extension to our algorithm that generalizes characters thereby learning C_{XML} exactly.

It is clear that the candidates constructed are monotone, since equating two nonterminals in a context-free grammar can only enlarge the generated language. Finally, we also formalize our intuition about how equating a pair of nonterminals $(A'_i, A'_j) \in M$ corresponds to merging repetition subexpressions:

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

In other words, equating two nonterminals $(A'_i, A'_j) \in M$ merges the corresponding repetition subexpressions R and R' in $\hat{R}$ (and may induce further generalizations as well). We give a proof in Appendix C.1.

5.3 Checks

Our algorithm constructs checks for the candidate $\tilde{C}$ obtained by merging $(A'_i, A'_j) \in M$ in the current language $\hat{C}$. Suppose that A'_i corresponds to repetition subexpression R and A'_j corresponds to repetition subexpression R'. Furthermore, suppose that step i generalizes repetition subexpression $P[\alpha]_{\text{rep}}Q$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$ and step j generalizes repetition subexpression $[\alpha']_{\text{rep}}$ 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 $\hat{R}$.

In this case, there are two different sets of checks. The first set of checks seeks to ensure that R' can be substituted for R without overgeneralizing; i.e., $\mathcal{L}(PR'Q) \subseteq L_*$. The second set of checks ensures that conversely, R can be substituted for R'; i.e., $\mathcal{L}(P'RQ') \subseteq L_*$. We describe the first set of checks; the second set is symmetric. Similar to phase one, our algorithm constructs checks $\gamma\rho'\delta$, where $\rho' \in \mathcal{L}(R')$ is a residual and (γ, δ) is a context for the repetition subexpression $([\alpha_2]_{\text{alt}})^*$. These checks satisfy

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

where the first inclusion follows by the property (1) for contexts described in Section 4.4 and the second inclusion follows from Proposition 5.1. To construct contexts, our algorithm utilizes the contexts (γ, δ) constructed in phase one for $[\alpha]_{\text{rep}}$; then, it constructs context $(\gamma\alpha_1, \alpha_3\delta)$ for R. Furthermore, it constructs a single residual $\rho' = \alpha'\alpha'$. A similar argument to Proposition 4.4 shows that these contexts satisfy property (1).

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

6. Extensions

In this section, we discuss two extensions to Algorithm 1.

6.1 Character Generalization

Recall that phases one and two of our algorithm cannot generalize beyond the characters available in the seed inputs E_{in}. We include a *character generalization* phase between phase one and phase two that generalizes terminals in the regular expression $\hat{R}$ synthesized by phase one. We now describe the candidates and the checks of this phase. This intermediate phase selects a terminal string $\alpha = \sigma_1...\sigma_k$ in $\hat{R}$ (i.e., $\hat{R} = P\alpha Q$), a terminal $\sigma_i \in \Sigma$ in α, and a different terminal $\sigma \in \Sigma$ such that $\sigma \neq \sigma_i$. Then our, algorithm considers two candidates. The first candidate $\tilde{R} = P\sigma_1...\sigma_{i-1}(\sigma + \sigma_i)\sigma_{i+1}...\sigma_k Q$ replaces σ_i with $(\sigma_i + \sigma)$. The second candidate is the current language $\hat{R}$.

For the first candidate, we construct a residual $\rho = \sigma$. To construct contexts, note that α was necessarily added to $\hat{R}$ using the repetition case; i.e., generalizing $[\alpha']_{\text{rep}}$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, where $\alpha = \alpha_1$. Then, our algorithm takes the context (γ, δ) for $[\alpha']_{\text{rep}}$ and constructs context $(\gamma\sigma_1...\sigma_{i-1}, \sigma_{i+1}...\sigma_k\alpha_3\delta)$ Finally, our algorithm constructs checks $\alpha = \gamma\rho\delta$ as before.

The character generalization phase of our algorithm considers generalizing every candidate of the form described above. For example, in the regular expression $\hat{R}_{\text{XML}}$ output by phase one in Figure 2, our algorithm considers generalizing every terminal in the terminal strings `<a>`, `h`, `i`, and `</a>` to every (different) terminal $\sigma \in \Sigma$. Generalizing `<` to `a` is ruled out by the check `aa>hi</a>`. Similarly, generalizing `a` to `b` is ruled out by the check `<b>hi</a>`; we do not consider simultaneously generalizing pairs of terminals, so our algorithm is unable to determine that the seed input can be generalized to $\texttt{<}\sigma\texttt{>hi</}\sigma\texttt{>}$ for any $\sigma \in [\texttt{a} - \texttt{z}]$. Our algorithm learns that `h` can be generalized to `a` since the checks `<a>ai</a>` (using context `(<a>, i</a>)`) and `<a>a</a>` (using context `(<a>, </a>)`) pass. Eventually, it generalizes $\hat{R}_{\text{XML}}$ to

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

Finally, phase two of our algorithm subsequently generalizes $\hat{R}'_{\text{XML}}$ to the following grammar $\hat{C}'_{\text{XML}}$:

$$\{A \to (\texttt{<a>}A\texttt{})^*,\ A \to ((\texttt{a} + ... + \texttt{z}) + (\texttt{a} + ... + \texttt{z}))^*\},$$

which is equivalent to the original grammar C_{XML} in Figure 1.

6.2 Multiple Seed Inputs

Given multiple seed inputs $E_{\text{in}} = \{\alpha_1, ..., \alpha_n\}$, our algorithm first applies phase one separately to each seed input α_i to synthesize a corresponding regular expression $\hat{R}_i$, combines these into a single regular expression $\hat{R} = \hat{R}_1 + ... + \hat{R}_n$, and applies phase two to $\hat{R}$. Note that repetition subexpressions in different components $\hat{R}_i$ of $\hat{R}$ can be merged in phase two.

7. Sampling Context-Free Grammars

We describe how we randomly sample a string α from a context-free grammar C, denoted as $\alpha \sim \mathcal{L}(C)$. We use random samples in our grammar-based fuzzer in Section 9.1, and to measure precision and recall for context-free languages as in Definiction 2.1. To describe our approach, we more generally describe how to sample $\alpha \sim \mathcal{L}(C, A)$ (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 $\alpha \sim \mathcal{L}(C, A)$ as follows:

- Randomly sample a production $p = (A \to A_1...A_k) \sim \mathcal{D}_A$.
- If A_i is a nonterminal, recursively sample $\alpha_i \sim \mathcal{L}(C, A_i)$; 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. Evaluation of Language Learning

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 this section, we show that GLADE can synthesize artificial (but representative) context-free grammars from a small number of examples and achieve high precision and recall with respect to the target language. In Section 9, we describe how GLADE can be used to fuzz programs, and compare GLADE to two baseline fuzzers.

We compare the output of GLADE to two widely used algorithms, L-Star [3] and RPNI [39], implemented using `libalf` [5]. We also compare with a variant of GLADE with phase two omitted, which restricts GLADE to learning regular languages. To compare, we use four manually written grammars:

- A matching parentheses grammar with 5 kinds of parentheses.
- A grammar for regular expressions.
- A grammar for Lisp-like programs.
- A context-free subset of the XML grammar.[5]

For each grammar C, we sampled 50 seed inputs $E_{\text{in}} \subseteq \mathcal{L}(C)$ using the technique described in Section 7, and implemented an oracle $\mathcal{O}$ for $L_* = \mathcal{L}(C)$. Then, we use each algorithm to learn L_* from E_{in} and $\mathcal{O}$ (we describe how particular algorithms are implemented in detail below). The 50 seed inputs are added to the learning algorithm using a counter-example guided approach, where we iteratively add inputs in $E_{\text{in}} = \{\alpha_1, ..., \alpha_{50}\}$ to the training set. More precisely, for a given algorithm, we start with training set $\tilde{E}^{(0)}_{\text{in}} = \emptyset$, and learn corresponding language $\hat{L}^{(0)}$. Then, on iteration t (where $1 \leq t \leq 50$), we take

$$\tilde{E}^{(t)}_{\text{in}} = \begin{cases} \tilde{E}^{(t-1)}_{\text{in}} \cup \{\alpha_t\} \text{ if } \alpha_t \notin \hat{L}^{(t-1)} \\ \tilde{E}^{(t-1)}_{\text{in}} \text{ if } \alpha_t \in \hat{L}^{(t-1)}. \end{cases}$$

Finally, we use the algorithm to learn $\hat{L}^{(t)}$ from $\tilde{E}^{(t)}_{\text{in}}$. To ensure the experiments run to completion, we time out each language learning algorithm after five minutes (in particular, the L-Star and RPNI algorithms sometimes did not scale to all 50 examples). We use the last language $\hat{L}^{(t)}$ learned before the step that timed out.

Results. Our results are averaged over five runs of each algorithm. In Figure 4, we show the precision and recall of each algorithm for each grammar. We estimate the precision of $\hat{C}$ by $\frac{|E_{\text{prec}} \cap \mathcal{L}(C)|}{|E_{\text{prec}}|}$, where E_{prec} consists of 100 samples from $\mathcal{L}(\hat{C})$. Similarly, we estimate the recall of $\hat{C}$ by $\frac{|E_{\text{rec}} \cap \mathcal{L}(\hat{C})|}{|E_{\text{rec}}|}$, where E_{rec} consists of 100 samples from $\mathcal{L}(C)$. Furthermore, we show the running time of each algorithm. We also show the number of samples from E_{in} used, which can be less than 50 if the algorithm failed to terminate after 300 seconds with all of E_{in}.

In Figure 5, we compare (a) the F_1-score $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$, and (b) the running times of each of the three algorithms; in (c), we show how the precision, recall, and running time of GLADE varies as with the number of samples in E_{in}. The F_1 score (between zero and one) is a standard metric for balancing precision and recall; a high F_1 score can only be achieved if both precision and recall are high.

***Performance of* GLADE.** With just the 50 given training examples, GLADE was able to learn each grammar with almost 100% recall. Furthermore, for each grammar, at least one of the random trials achieved 100% recall. Getting almost 100% recall strongly suggests that GLADE learned much of the structure of the true grammar. The precision is also nearly 100%. Finally, as can be seen from Figure 5 (c), GLADE performs as well with fewer samples, and its performance likewise scales well with the number of samples.

The performance of GLADE suffers slightly when phase two is omitted (shown as P1 in Figure 4), but nevertheless performs quite well. The performance suffers the most on the matching parentheses language, because this language has the most purely non-regular structure. Overall, GLADE substantially outperforms L-Star and RPNI even when limited to learning regular languages. Additionally, GLADE is actually faster with phase two. This speedup is a consequence of our counter-example driven learning algorithm—because GLADE generalizes better compared to P1, it adds fewer samples to $\tilde{E}^{(t)}_{\text{in}}$ thereby reducing the running time.

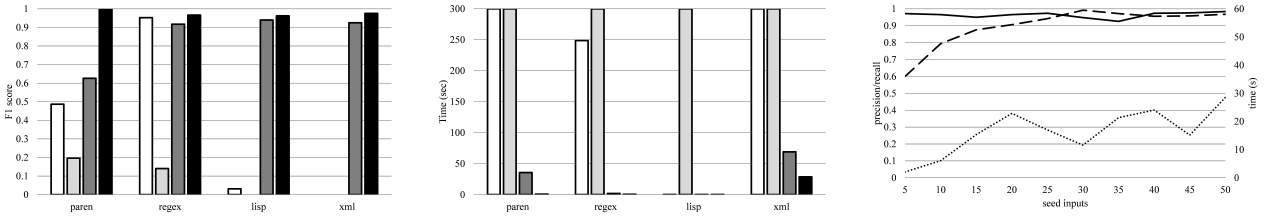
Comparison to L-Star. Angluin's L-Star algorithm [3] learns a regular language $\hat{R}$ approximating C. In addition to using a query oracle $\mathcal{O}$, the L-Star algorithm also uses an *equivalence oracle* $\tilde{O}$, which takes a candidate regular language $\hat{R}$ and either accepts $\hat{R}$ if $\mathcal{L}(\hat{R}) = L_*$ or gives a counter-example. We use the variant in [3] where the equivalence oracle $\tilde{O}$ is implemented by randomly sampling strings to search for counter-examples; we accept the candidate $\hat{R}$ if none are found after 50 samples.

In Figure 5, the L-Star algorithm performs well for the grammar of regular expressions and passably for the matching parentheses grammar. However, for the more complex Lisp-like and XML grammars, it essentially fails to learn the target language, achieving either small precision or small recall. Its poor performance is

[5] In particular, it cannot handle the semantics of matching tags (e.g., `<a></b>` is invalid) and no repeated attributes (e.g., `<a x="y" x="y">` is invalid).

Grammar	Size	Samples				Time (sec)				Precision				Recall			
		L-Star	RPNI	P1	GLADE	L-Star	RPNI	P1	GLADE	L-Star	RPNI	P1	GLADE	L-Star	RPNI	P1	GLADE
paren	22	5	16	50	50	300.00	300.00	35.61	0.72	1.00	0.11	1.00	1.00	0.32	0.98	0.46	1.00
regex	25	36	17	50	50	248.71	300.00	1.80	0.46	0.93	0.08	1.00	0.96	0.97	1.00	0.85	0.98
lisp	110	50	20	50	50	0.04	300.00	0.18	0.14	0.02	0.00	1.00	1.00	1.00	1.00	0.89	0.93
xml	1093	5	8	50	50	300	300.00	69.06	28.59	0.00	0.00	0.98	0.98	0.94	0.56	0.88	0.97

Figure 4. The size of each grammar is defined to be the number of productions. For each algorithm, we show the number of samples used to learn the grammar without timing out (<300 seconds), and the running time (which can be less than 300 if all 50 samples were used). We show the precision and recall of the learned language $\hat{L}$ calculated based on 100 random samples. The algorithm P1 refers to GLADE with phase two omitted.



(a) (b) (c)

Figure 5. 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 artificial grammars C. The algorithms are trained on 50 random samples from the target language $L_* = \mathcal{L}(C)$. Precision is evaluated on 100 random samples from the learned language $\hat{L}$, and recall on 100 random samples from L_*. 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.

likely because the equivalence oracle is unable to find the counter-examples needed to rule out incorrect generalizations. Indeed, for any given run of the algorithm, at most two calls to the equivalence oracle resulted in counter-examples. In contrast, the checks generated by GLADE serve to avoid overgeneralizing by focusing on the new strings introduced by the candidates.

Additionally, the running time of the algorithm is long, despite being polynomial. We may naïvely expect L-Star to diverge if L_* is non-regular, since its canonical acceptor then has infinitely many states, and the running time of L-Star is linear in the number of states in the canonical acceptor. However, we rule out this explanation for the long running time—the L-Star algorithm issues an equivalence query for each state in the canonical acceptor, but we observe that it never issues more than two equivalence queries. In other words, even learning a *three state* automata is not scalable. Instead, we believe that the slow running time is due to the large alphabet and length of the training examples that it is given. The L-Star algorithm makes queries for pairs of indices in the counter-examples and characters in the alphabet, which can add up to a large number of queries even for simple cases.

Comparison to RPNI. We used the RPNI algorithm [39] to learn a regular language $\hat{R}$ approximating C given both a set of positive examples E_{in} as well as a set of negative examples E_{in}^-. As negative examples, we sample 50 random strings that are not in L_*. As seen from Figure 5 (a), RPNI performs poorly for all grammars, and furthermore does not scale to these problems. We attribute the poor performance of RPNI to its sensitivity to the given inputs. The RPNI algorithm enjoys an "asymptotic" learning guarantee—for any enumeration of all strings $\alpha_1, \alpha_2, \ldots \in \Sigma^*$, its output eventually stabilizes to the correct language. However, RPNI is very sensitive to the given examples:

- **Positive examples:** It cannot generalize beyond the transitions exercised by the seed inputs $E_{\text{in}} \subseteq L_*$; for example, if a valid character never occurs in any seed input, the language $\hat{R}$ rejects any string containing this character.
- **Negative examples:** It is likewise sensitive to having the right kind of negative example to reject incorrect generalizations.

In contrast, GLADE overcomes the first difficulty by flexibly incorporating a number of varying phases of generalization steps tailored to learning grammars found in practice, and overcomes the second difficulty by generating checks to rule out incorrect generalizations. Finally, as can be seen from Figure 5 (b), RPNI is even less scalable than L-Star on our grammars.

9. Evaluation on Fuzz Testing

We use GLADE to fuzz programs. Given 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 inputs the synthesized grammar $\hat{C}$ and the seed inputs E_{in}, and randomly generates new inputs $\alpha \in \mathcal{L}(\hat{C})$; we give details in Section 9.1. These inputs can then be used to test the program.

GLADE is able to generate inputs that are both diverse (since the synthesized grammar $\hat{C}$ has high recall with respect to L_*) and typically valid (since $\hat{C}$ has high precision with respect to L_*. In contrast, existing blackbox fuzzers, which are unaware of the program input language L_* (unless manually specified), typically generate either invalid inputs (e.g., generating `aa>hi</a>` from the seed input α_{XML} in Figure 1) or inputs that do not test interesting new behaviors compared to the seed inputs (e.g., `<a>ai</a>`). In contrast, GLADE is able to discover inputs with new structure (e.g., `<a><a>hi</a></a>`).

For many settings (e.g., differential testing [58]), invalid inputs are useless, but in some settings (e.g., bug finding [35]), error-handling code triggered by invalid inputs should also be tested. In these settings, GLADE generates a mixture of grammatical and non-grammatical inputs (see Section 9.2). Finally, as shown in Section 9.3, GLADE consistently outperforms two baseline fuzzers (described in Section 9.2), especially on the task of generating valid inputs.

9.1 Grammar-Based Fuzzer

To fuzz a program, GLADE first synthesizes a context-free grammar $\hat{C}$ approximating the target language L_* of valid program inputs. Then, GLADE uses our grammar-based fuzzer (described below),

which randomly generates new inputs from the synthesized grammar $\hat{C}$ and the seed inputs E_{in}. Finally, GLADE tests the program using these generated inputs.

Our grammar-based fuzzer is based on standard techniques [26]. 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 20. A single modification is performed as follows:

- Randomly select an internal node N of the parse tree for α.
- Write $\alpha = \alpha_1\alpha_2\alpha_3$ where α_2 is represented by the subtree with root N.
- Let A be the nonterminal labeling N, sample $\alpha' \sim \mathcal{L}(C, A)$, and return $\alpha_1\alpha'\alpha_3$.

As discussed above, our grammar-based fuzzer largely samples valid inputs, so it has difficulty executing error-handling code in programs. Therefore, GLADE can also use our combined fuzzer (described in Section 9.2) to randomly generate new inputs.

9.2 Other Fuzzers

We describe the two baseline fuzzers that we compare to. We also describe our combined fuzzer, which GLADE uses to generate a mixture of valid and invalid inputs.

Afl-fuzz. This fuzzer, which is a production fuzzer developed at Google [59], 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.). Additionally, it instruments the program to obtain branch coverage for each execution, which it uses to identify when an input α causes the program to execute new behavior. 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 this blackbox 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 20. 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.

Combined fuzzer. With probability $\frac{1}{3}$, this fuzzer runs the grammar-based fuzzer, and with probability $\frac{1}{3}$, it runs the naïve fuzzer. With the remaining $\frac{1}{3}$ probability, it performs n_1 grammar-based modifications followed by n_2 naïve modifications, where n_1 and n_2 are (independently) chosen uniformly between 0 and 20. Therefore, the combined fuzzer samples valid inputs like the grammar-based fuzzer as well as invalid inputs that execute error-handling code like the naïve fuzzer.

9.3 Evaluation

We compare GLADE to our naïve fuzzer and to alf-fuzz.

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 of 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 by the interpreter, but inputs that are syntactically correct but possibly have runtime errors are accepted.

Samples. To fuzz a program, we use between 1 and 21 seed inputs $E_{\text{in}} \subseteq L_*$ that capture interesting semantics of the target language L_*. These seed inputs were obtained either from documentation or from small test suites that came with the program. For each fuzzer and program pair, we generate 50,000 samples $E_{\text{all}} \subseteq \Sigma^*$ based on the seed inputs E_{in} for that program.[6] We also compute the subset of valid inputs $E_{\text{valid}} = E_{\text{all}} \cap L_*$ (where L_* is the language of valid program inputs). Additionally, each program comes with a test suite E_{test} that we extract (excluding programmatic tests).

Metrics. The precision is the fraction of inputs in E_{all} that are in the target language L_* of valid program inputs (i.e., $\frac{|E_{\text{valid}}|}{|E_{\text{all}}|}$). The coverage of a set of inputs $E \subseteq \Sigma^*$ is $\frac{\#(\text{lines covered by some input in } E)}{\#(\text{lines coverable})}$, which we compute using gcov. To better compare performance, we also compute the *incremental coverage*, which is the percentage of code covered beyond those covered by the seed inputs E_{in}, thus measuring the ability to execute new code paths. More precisely, for a set of inputs $E \subseteq \Sigma^*$, the incremental coverage is $\frac{\#(\text{lines covered by } E \text{ but not covered by } E_{\text{in}})}{\#(\text{lines coverable but not covered by } E_{\text{in}})}$). Furthermore, we typically normalize the incremental coverage by a baseline E_{baseline} (usually the incremental coverage of the samples from the naïve fuzzer) to obtain the *normalized incremental coverage* $\frac{\text{incremental coverage of } E}{\text{incremental coverage of } E_{\text{baseline}}}$.

Results. In Figure 6, we report the coverages of:

- The empty example, which is the simplest valid input (typically the empty string).
- The seed inputs E_{in}.
- The test suite E_{test} that comes with the program.
- All samples E_{all} generated using the naïve fuzzer, afl-fuzz, the grammar-based fuzzer, and the combined fuzzer (the highest of which is bolded).
- Valid samples E_{valid} generated using the naïve fuzzer, afl-fuzz, and the grammar-based fuzzer (the highest of which is bolded).[7] The coverage on valid inputs is a proxy for measuring coverage along deeper code paths.

In Figure 7, we compare the incremental coverages of the various fuzzers on (a) all, and (b) valid inputs. In (c), we compare the coverage of our grammar-based fuzzer on the XML parser with both a manual grammar and a grammar learned using RPNI. Additionally, we show (d) the precision, (e) a comparison to the incremental coverage on the test set, and (f) how coverage varies with the number of samples.

All inputs. Figure 7 (a) shows fuzzer performance normalized by that of the naïve fuzzer (black dotted line). Here, GLADE (black) uses the combined fuzzer, which consistently outperforms the naïve fuzzer (substantially so for flex and Python). This result shows that our synthesized grammar can be used to consistently improve the quality of an otherwise grammar-unaware blackbox fuzzer.

GLADE also outperforms afl-fuzz, except for the XML parser and bison. In these cases, we believe afl-fuzz is able to search

[6] For afl-fuzz, we generate $\frac{n}{k}$ samples for each input, where $k = |E_{\text{in}}|$.

[7] We omit the combined fuzzer since the grammar-based fuzzer performs better for valid inputs.

Program	Time (min)	Lines of Code			Empty	Seeds	Tests	All Samples				Valid Samples			Precision		
		program	input	test				afl	naïve	gram	comb	afl	naïve	gram	afl	naïve	gram
sed	0.25	2K	3	1486	9.65	20.2	37.6	41.9	43.2	38.6	**44.0**	34.8	**36.6**	33.9	5.99	26.3	74.4
flex	1.83	6K	15	1018	32.5	37.4	45.5	47.4	47.2	**52.4**	52.2	40.8	42.5	**46.3**	4.40	19.6	64.0
grep	0.26	12K	4	110	10.0	15.5	19.2	28.6	28.5	28.5	**29.7**	25.8	27.1	**29.0**	16.6	81.3	96.3
bison	4.91	13K	14	295	27.4	30.4	32.5	**38.8**	37.1	35.6	38.2	31.0	31.0	**31.8**	2.48	2.85	80.9
xml	2.30	123K	7	1137	1.94	2.59	5.69	**3.73**	3.37	3.01	3.44	2.62	2.64	**2.74**	3.13	9.28	84.9
ruby	229	120K	80	224K	18.9	19.0	22.2	19.5	20.9	20.3	**21.0**	19.1	19.5	**19.8**	23.8	15.4	32.9
python	269	128K	267	156K	10.4	12.9	15.8	13.4	13.5	**14.1**	13.9	13.0	13.2	**14.0**	23.0	2.52	30.6
javascript	113	156K	118	590K	10.5	10.8	14.1	–	12.7	12.3	**12.8**	–	11.4	**11.6**	–	4.23	39.1

Figure 6. We show lines of program code, lines of seed inputs E_{in}, lines in the test suite, and running time of our grammar synthesis algorithm. All coverages are reported as a percentage of the code that is covered by the samples (computed using gcov). We show the coverages for the empty input, the seed inputs E_{in} given to the fuzzers, and the tests E_{test} that come with each program. Then, we show the coverage on all inputs achieved by our naïve fuzzer ("naïve"), afl-fuzz ("afl"), our grammar-based fuzzer ("gram"), and our combined fuzzer ("comb"); the highest coverage achieved for each program is bolded. Similarly, we show the coverage when restricted to valid samples E_{valid}, with the highest fuzzer coverage bolded. Finally, we show the precision (reported as a percentage) of afl-fuzz, our naïve fuzzer, and our grammar-based fuzzer.

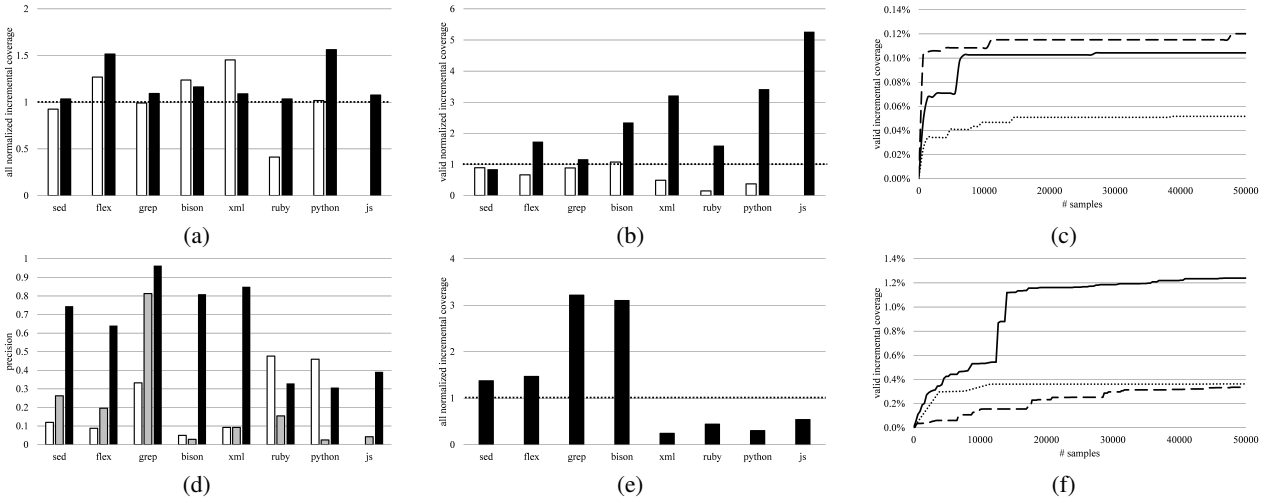


Figure 7. We show the normalized incremental coverage for (a) all samples E_{all} and (b) valid samples E_{valid} for the naïve fuzzer (black dotted line), afl-fuzz (white), and GLADE (black), which uses the grammar-based fuzzer if using valid samples and the combined fuzzer if using all samples. In (c), we compare the coverage of GLADE (solid) to a grammar-based fuzzer using the handwritten XML grammar (dotted) and to a grammar-based fuzzer using a grammar learned using the RPNI algorithm (dashed). In (d), we show the precision of afl-fuzz (white), the naïve fuzzer (grey), and GLADE (white). In (e), we show the normalized incremental coverage of GLADE with the combined fuzzer (black), except normalized by the program test suite (black dotted line). Finally, in (f), we show how the incremental coverage (on valid inputs E_{valid}) varies with the number of samples in E_{in} on the Python parser for afl-fuzz (dotted), naïve (dashed), and GLADE (solid), where GLADE uses the grammar-based fuzzer.

a portion of the input space unavailable to our fuzzers (e.g., the current implementations of GLADE and the naïve fuzzers are restricted to ASCII characters). Because our grammar-based fuzzer consistently improves over our naïve fuzzer, we believe that incorporating grammar-based modifications (based on the grammar synthesized by GLADE) into afl-fuzz would be profitable as well. On the other hand, afl-fuzz performs surprisingly poorly for some of the programs, most likely because afl-fuzz operates at the bit level, whereas our fuzzers are aware that programs have textual inputs and operate at the character level. Note that afl-fuzz results are missing for Javascript since we were unable to set up afl-fuzz on this program.

Finally, in Figure 6, we see that oftentimes, a large amount of code is trivially covered by the empty input, which highlights the vast discrepancy between the ease with which different lines of code get covered. In particular, these numbers suggest that the incremental lines of code covered are the most valuable since they are the hardest to cover.

Valid inputs. In many settings (e.g., differential testing [58]), only valid inputs are useful. We compare the coverage of the various fuzzers on E_{valid} obtained by removing invalid inputs from E_{all}. Additionally, valid inputs give a proxy for the number of lines of code covered that correspond to deeper code paths (e.g., those deeper in or beyond the parser). As can be seen from Figure 7 (b), GLADE (black) is especially effective in this setting, significantly outperforming both the naïve fuzzer (black dotted line) and afl-fuzz (white) except on grep and sed. Since grep and sed have relatively simple input formats, using a grammar-based fuzzer is understandably less effective.

For the six other programs, our grammar-based fuzzer performs between 1.5 and 5 times better than the naïve fuzzer, showing that our grammar-based fuzzer not only generates substantially more valid inputs, but that these generated inputs execute different code paths. In Figure 7 (d), we see that the precision of the grammar-based fuzzer is substantially higher than the precision of the naïve fuzzer. In fact, for bison and Python, the precision of the naïve fuzzer is less than 3%, whereas the grammar-based fuzzer has a precision consistently above 30%. While high precision does not necessarily imply better coverage (e.g., we can just return the inputs), poor precision suggests that the naïve fuzzer is unable to find inputs executing code paths beyond the parser. For example, afl-fuzz has good precision on Ruby and Python, but still performs substantially worse than either GLADE or the naïve fuzzer on these programs.

Comparison to fuzzing with other grammars Figure 7 (c) compares GLADE (solid line) to two other grammar-based fuzzers. First, we used RPNI to learn the target XML language L_* as in Section 8, but taking E_{in} to be the seed inputs for XML fuzzing (as opposed to random samples from the handwritten XML grammar). Then, we use the grammar-based fuzzer with the learned grammar (dotted line). As can be seen, GLADE significantly outperforms this baseline. We tried a similar comparison to the L-Star algorithm, but the L-Star algorithm did not terminate. Second, we show a comparison to a grammar-based fuzzer that uses the manual, handwritten XML grammar (dashed line). As can be seen, the performance of GLADE comes very close to the handwritten XML grammar.

Comparison to test suites. Figure 7 (e) compares the incremental coverage of the combination fuzzer to that of the test suite. All fuzzers outperform the relatively small test suites that came with grep, sed, flex, and bison. The coverage of the larger test suites can be thought of as a proxy for coverable code, showing that there is much code that we may simply be unable to cover. In these programs, GLADE recovers a substantial fraction of the difference between the inputs E_{in} and the test suite E_{test}.

Coverage over time. Figure 7 (f) shows how the incremental coverage on valid inputs E_{valid} 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.

10. Related Work

Language learning. There has been a line of work (also 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 [39]. These algorithms have a number of applications including model checking [19], model-assisted fuzzing [12, 13], verification [54], and specification inference [6]. To the best of our knowledge, our approach is the first to focus on learning common program input languages from positive examples and a membership oracle. Additionally, [29] discusses approaches to learning context-free grammars, including from positive examples and a membership oracle. As they discuss, these algorithms are often either slow [49] or do not generalize well [28].

Reverse engineering message formats. Work on reverse engineering network protocol message formats [8, 31, 32, 57] focuses 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 [31] constructs syntax trees for given inputs. All 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.

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, [46] randomly tests COBOL compilers and [43] generated random inputs to test parsers. An early application of blackbox fuzzing to find bugs in real-world programs was [35], who executed Unix utilies 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, 36, 51]; see [52] for a survey. Finally, afl-fuzz [59] is essentially blackbox but leverages simple instrumentation to guide the search.

Whitebox fuzzing. Most whitebox approaches to fuzzing [4, 23] build on techniques known as *dynamic symbolic execution* [9–11, 21, 47]; 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 [56].

Grammar-based fuzzing. Many blackbox 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 [33, 58]; see [7] for a survey. Such techniques have also been used to fuzz interpreters; for example, [26] 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, [22] fuzzes a just-in-time compiler for Javascript using a handwritten grammar for the Javascript language in conjuction with a technique for solving constraints involving grammars. Additionally, [34] combines exhaustive enumeration of valid inputs with symbolic execution techniques to improve coverage. In [52], 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 [55], 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 such as 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 [24], number [48], and table [25] transformations (and combinations thereof [41, 42]), as well as recursive programs [1, 16] and parsers [30]. Unlike these approaches, our approach exploits an oracle that our algorithm uses to reject invalid candidates.

11. Conclusion

We have presented 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. We have shown that our algorithm addresses shortcomings of existing grammar inference algorithms. Futhermore, we have implemented GLADE, which leverages a grammar synthesized by our algorithm to fuzz programs that take highly structured inputs. We have shown that GLADE consistently increases the incremental coverage compared to two baseline fuzzers. Finally, while we have focused on blackbox fuzzing, our approach can be used in conjunction with any grammar-based fuzzing technique.

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 or-

chestrated 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 *ACM SIGPLAN Notices*, volume 48, pages 443–456. ACM, 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 *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 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, 2014. URL `https://www.gnu.org/software/bison`.

[21] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[22] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

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

[24] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[25] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–328. ACM, 2011.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. *ACM SIGPLAN Notices*, 47(1):373–386, 2012.

[38] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

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

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

[41] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *ACM SIGPLAN Notices*, volume 49, pages 408–418. ACM, 2014.

[42] 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.

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

[44] M. Rinard. Acceptability-oriented computing. *Acm sigplan notices*, 38(12):57–75, 2003.

[45] M. C. Rinard. Living in the comfort zone. *ACM SIGPLAN Notices*, 42(10):611–622, 2007.

[46] 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.

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

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

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

[50] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

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

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

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

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

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 4.4, 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) $R' = P\alpha_1\alpha_2^*\alpha_3 Q$, 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

$$\mathcal{L}(R) = \mathcal{L}(P\alpha Q) \subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha_3 Q) = \mathcal{L}(R').$$

- **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 $Q' =$ "$)^*\alpha_3 Q$", so $R' = P'\alpha_2 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha_1\alpha'\alpha_3\delta &\in \mathcal{L}(P\alpha_1\alpha'\alpha_3 Q) \\ &\subseteq \mathcal{L}(P\alpha_1(\alpha')^*\alpha_3 Q) \\ &= \mathcal{L}(P'\alpha' Q'), \end{aligned}$$

  where the first inclusion follows by applying (1) to the context (γ, δ) with $\alpha_1\alpha'\alpha_3 \in \Sigma^*$. 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 $R' = P'\alpha_3 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha_1\alpha_2\alpha'\delta &\in \mathcal{L}(P\alpha_1\alpha_2\alpha' Q) \\ &\subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha' Q) \\ &= \mathcal{L}(P'\alpha' Q'), \end{aligned}$$

  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 $R' = P'\alpha_2 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned} \gamma'\alpha'\delta' = \gamma\alpha'\alpha_2\delta &\in \mathcal{L}(P\alpha'\alpha_2 Q) \\ &= \mathcal{L}(P(\alpha' + \alpha_2)^* Q) \\ &= \mathcal{L}(P(\alpha' + \alpha_2) Q) \\ &= \mathcal{L}(P'\alpha' Q'), \end{aligned}$$

  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 completeness results discussed in Section 4.2.

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 4.2 (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., $R = \bar{\epsilon} + \texttt{a}$).

As described in Section 4.1, the first modification can be addressed by using multiple inputs (see Section 6.2), 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 $\tilde{C}_{\text{regex}}$ 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 $R' \in \mathcal{L}(\tilde{C}_{\text{regex}})$ such that $\mathcal{L}(R) = \mathcal{L}(R')$ (which we express as $R \equiv R'$).

First, we prove the result for $\mathcal{C}^{\epsilon}_{\text{regex}}$, which is identical to $\mathcal{C}_{\text{regex}}$ except that we allow $\beta = \epsilon$. Let $R \in \mathcal{L}(\tilde{C}_{\text{regex}})$. Suppose that either $R = S_1 + S_2$, $R = S_1 S_2$, 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 $Y_{i,j} = Z_1 Z_2$. Then

$$X_i = Y_{i,1}...Y_{i,j}...Y_{i,k_i} = Y_{i,1}...Z_1 Z_2...Y_{i,k_i}$$

  which is again in the form (3).

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

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

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 $R = S_1 + S_2$, $R = S_1 S_2$, 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. □

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

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

in $\mathcal{C}_{\text{regex}}$ as described in Section 4.2. 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

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

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 $\mu T^{(i)}_{\text{rep}} \nu \Rightarrow \mu\beta(T^{(j)}_{\text{alt}})^* T^{(k)}_{\text{rep}} \nu$: We add productions $A_i \rightarrow \beta A'_i A_k$ and $A'_i \rightarrow \epsilon \mid A'_i A_j$.
- Step $\mu T^{(i)}_{\text{alt}} \nu \Rightarrow \mu(T^{(j)}_{\text{rep}} + T^{(k)}_{\text{alt}})\nu$: We add production $A_i \rightarrow A_j \mid A_k$.

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

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

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

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

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

$$\begin{aligned}
(\eta_1 = T^{(1)}_{\text{rep}}) &\Rightarrow ... \\
&\Rightarrow (\eta_i = P T^{(i)}_{\text{rep}} Q) \\
&\Rightarrow (\eta'_{i+1} = P\beta'(T^{(j')}_{\text{alt}})^* T^{(k')}_{\text{rep}} Q) \\
&\Rightarrow ... \\
&\Rightarrow \eta'_{n'} = PR'Q
\end{aligned}$$

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 $\mathcal{L}(\hat{C}') = \mathcal{L}(\hat{R}') = \mathcal{L}(PR'Q)$. □