Towards Automating Grammar Equivalence Checking

EPFL-REPORT-206921

Ravichandhran Madhavan
EPFL
ravi.kandhadai@epfl.ch

Mikaël Mayer
EPFL
mikael.mayer@epfl.ch

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Viktor Kuncak
EPFL
viktor.kuncak@epfl.ch

Abstract

We consider from practical perspective the (generally undecidable) problem of checking equivalence of context-free grammars. We present both techniques for proving equivalence, as well as techniques for finding counter-examples that establish non-equivalence. Among the key building blocks of our approach is a novel algorithm for efficiently enumerating and sampling words and parse trees from arbitrary context-free grammars; the algorithm supports polynomial time random access to words belonging to the grammar. Furthermore, we propose an algorithm for proving equivalence of context-free grammars that is complete for LL grammars, yet can be invoked on any context-free grammar, including ambiguous grammars.

Our techniques successfully find discrepancies between different syntax specifications of several real-world languages, and is capable of detecting fine-grained incremental modifications performed on grammars. Our evaluation shows that our tool improves significantly on the existing available state of the art tools. In addition, we used these algorithms to develop an online tutoring system for grammars that we then used in an undergraduate course on computer language processing. On questions involving grammar constructions, our system was able to automatically evaluate the correctness of 95% equivalence questions: it disproved 74% of cases and proved 21% of them. This opens up the possibility of using our tool in massive open online courses to introduce grammars to large populations of students.

1. Introduction

Context-free grammars are pervasively used in verification and compilation, both for building input parsers and as foundation of algorithms for model checking, program analysis, and testing. They also play an important pedagogical role in introducing fundamentals of formal language theory, and are an integral part of undergraduate computer science education. Despite their importance, and despite decades of theoretical advances, practical tools that can check semantic properties of grammars are still scarce, expect for specific tasks such as parsing.

$$S \rightarrow S + S \mid S * S \mid ID \qquad \begin{array}{l} S \rightarrow ID\ E \\ E \rightarrow +S \mid \ *S \mid \epsilon \end{array}$$

Figure 1. Grammars recognizing simple arithmetic expressions. An example proven equivalent by our tool.

$$\begin{array}{l} S \rightarrow A \Rightarrow S \mid Int \\ A \rightarrow Int, A \mid Int \end{array} \qquad \begin{array}{l} S \rightarrow Int\ G \\ G \rightarrow \Rightarrow Int\ G \mid , Int\ A \mid \epsilon \\ A \rightarrow , Int\ A \mid \ \Rightarrow \ Int\ G \end{array}$$

Figure 2. Grammars defining well-formed functions signatures over Int. An example proven equivalent by our tool.

In this paper, we develop practical techniques for checking equivalence of context-free grammars. Our techniques can find counter-examples that disprove equivalence, and can prove that two context-free grammars are equivalent, much like a software model checker. Our approaches are motivated by two applications: (a) comparing real-world grammars, such as those used in production compilers, (b) automating tutoring and evaluation of context-free grammars. These applications are interesting and challenging for a number of reasons.

Much of the front-ends of modern compilers and interpreters are automatically or manually derived from grammar-based descriptions of programming languages, more so with integrated language support for domain specific languages. When two compilers or other language tools are built according to two different reference grammars, knowing how they differ in the programs they support is essential. Our experiments show that two grammars for the same language almost always differ, even if they aim to im-

(a) $\begin{array}{l} S \rightarrow A \Rightarrow Int \mid Int \\ A \rightarrow S, Int \mid Int \end{array}$ (b) $\begin{array}{l} S \rightarrow A \Rightarrow S \mid Int \\ A \rightarrow S, S \mid Int \end{array}$

Figure 3. Grammars subtly different from the grammars shown in Fig. 2. The grammar on the left does not accept "$Int \Rightarrow Int \Rightarrow Int$".

plement the same standard. For instance, we found using our tool that two high quality standard Java grammars (namely, the Java grammar [1] used by Antlr v4 parser generator [1], and the Java language specification [2]) disagree on more than 50% of words that are randomly sampled from them.

It is worrying to know this, since we tend to expect seamless portability across compilers. While many differences could be filtered out by type checkers and other back-end phases of compliers, some may still percolate to the user level. In fact, we find numerous differences in the grammars for dynamic languages such as Javascript as well (see section 5 for more details).

Besides detecting incompatibility, comparing real-world grammars can help identify portions of the grammars that are overly permissive. For instance, the string "`enum ID implements char { ID }`" is generated by Antlr Java grammar but is rightly rejected by the Oracle's Java language specification. The counter-examples help in identifying places where the grammars, and hence their parsers can be optimized.

Furthermore, often grammars are rewritten extensively to make them acceptable by parser generators, which is laborious and error prone. Parser generators have become increasingly permissibly over the years to mitigate this problem. However, there still remains considerable overhead in this process, and there is a general need for tools that pinpoint subtle changes in the modified versions (documented in works such as [33]). It is almost always impossible to spot differences between large real-world grammars through manual scanning, because the grammars typically appear similar, and even use the same name for many non-terminals. A challenge this paper addresses is developing techniques that scales to real-world grammars, which have hundreds of non-terminals and productions.

An equally compelling application of grammar comparison arises from the importance of context-free grammar in computer science education. Assignments involving context-free grammars are harder to grade and provide feedback, arguably even more than programming assignments, because of their succinctness; which makes them difficult to comprehend, and also allows for a greater variation in the possible solutions. The complexity is further aggravated when the solutions are required to belong to subclasses like LL(1). For example, Figures 1 and 2 show two pairs of grammars that are equivalent. The grammars shown on the right are LL(1) grammars, and are reference solutions. The grammars shown on the left are intuitive solutions that a student comes up with initially. Proving equivalence of these pairs of grammars is challenging because they do not have any similarity in their structure, but recognize the same language. On the other hand, Figure 3 shows two grammars (written by students) that subtly differ from the grammars of Fig. 2. The smallest counter-example for the grammar shown in Fig. 3(a) is the string "$Int \Rightarrow Int \Rightarrow Int$". We invite the readers to identify a counter-example that differentiates the grammar of Fig. 3(b) from those of Fig. 2.

In our experience, a practical system that can prove that a student's solution is correct and provide a counter-example if it is not can greatly aid tutoring of context-free grammars. The state of the art for giving feedback on programming assignments is to use test cases (though there has been recent work on generating repair based feedback [29]). We bring the same fundamentals to context-free grammar education. Furthermore, we exploit the large, yet under-utilized, theoretical research on decision procedures for equivalence of context-free grammars to develop a practical algorithm that can prove the correctness of solutions provided by the students.

Overview and Contributions. At the core of our system is a fast approach for enumerating words and parse trees of an arbitrary context-free grammar, which supports exhaustive enumeration as well as random sampling of parse trees and words. These features are supported by an efficient *polynomial time* random access operation that constructs a unique parse tree for any given natural number index. We construct a scalable counter-example detection algorithm by integrating our enumerators with a state of the art parsing technique [25].

We develop and implement an algorithm for proving equivalence by extending decision procedures for subclasses of deterministic context-free grammars to arbitrary (possibly ambiguous) context-free grammars, while preserving soundness. We make the algorithm practical by performing numerous optimizations, and use concrete examples to guide the proof exploration. We are not aware of any existing system that supports both proving as well as disproving of equivalence of context-free grammars. The following are our main contributions:

- We present an enumerator for generating parse trees of arbitrary context-free grammars that supports the following operations: 1) a *polynomial time* random access operation $lookup(i, l)$ that given an index i returns the unique parse tree generating a word of length l, corresponding to the index, and 2) $sample(n, l)$ that generates n uniformly random samples from the parse trees of the grammar generating words of length l (Section 2).
- We propose an algorithm for discovering counter-examples for equivalence of context-free grammars using the proposed enumerators.

[1] `github.com/antlr/grammars-v4/blob/master/java/Java.g4`

- We integrate and extend the algorithms of [15], [24], and [12], for proving equivalence of LL context-free grammars to arbitrary context-free grammars. Our extensions are sound but incomplete. We show using experiments that the algorithm is effective on many grammars that are outside the classes with known decision procedures.
- We implement and evaluate an online tutoring system for context-free grammars . Our system is able to decide the veracity of 95% of the submissions, detecting counter-examples in 74% of the submissions, and proving correctness of 21% of the submissions.
- We evaluate the counter-example detection algorithm on 10 real-world grammars describing the syntax of 5 mainstream programming languages. The algorithm discovers deep, fine-grained errors, by finding counter-examples with an average length of 35, detecting almost 3 times more errors than a state of the art approach.

2. Enumeration of Parse Trees and Words

A key ingredient of our approach for finding counter-examples is enumeration of words and parse trees belonging to a context-free grammar. Enumeration is also used in optimizing and improving the scope of our grammar equivalence proof engine. We model our enumerators as functions from natural numbers to objects that are enumerated (which are parse trees or words), as opposed to viewing them as iterators for a sequence of objects as is typical in programming language theory. The enumerators we propose are *bijective* functions from natural numbers to parse trees in which the image and pre-image of any given value is efficiently computable in *polynomial time* (formalized in Theorem 1). The functions are partial if the set that is enumerated is finite. Using bijective functions to construct enumerators has many advantages, for example, it immediately provides a way of sampling elements from the given set. It also ensures that there is no duplication during enumeration. Additionally, the algorithm we present here can be configured to enumerate parse trees that generate words having a desired length.

Notations. A context-free grammar is a quadruple $(\mathcal{N}, \Sigma, P, S)$, where $\mathcal{N}$ is a set of non-terminals, Σ is a set of terminals, $P \subseteq \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$ is a finite set of productions and $S \in \mathcal{N}$ is the start symbol. Let $\mathcal{T}$ denote the set of parse trees belonging to a grammar. We refer to sequences of terminals and non-terminals belonging to $(\mathcal{N} \cup \Sigma)^*$ as *sentential forms* of the grammar. If a sentential form has only terminals, we refer to it as a *word*, and also sometimes as a *string*. We adopt the usual convention of using greek alphabets α, β to represent sentential forms and upper-case latin characters to represent non-terminals. We use lower-case latin characters a, b, c etc. to represent terminals and w, x, y etc. to denote words. We introduce more notations as they are needed.

2.1 Constructing Random Access Enumerators

We use $\mathsf{Enum}[\alpha] : \mathbb{N} \to \mathcal{T}^*$ to denote an enumerator for a sentential form α of the input grammar. The enumerators are *partial functions* from natural numbers to tuples of parse trees of the grammar, one rooted at every symbol in the sentential form. For brevity, we refer to the tuple as parse trees of sentential forms. We define $\mathsf{Enum}[\alpha]$ recursively following the structure of the grammar as explained in the sequel.

For a terminal a belonging to a grammar, $\mathsf{Enum}[a]$ is defined as $\{0 \to \mathit{leaf}(a)\}$. That is, the enumerator for a terminal a maps the first index to a parse tree with a single leaf node containing a and is undefined for every other index. We now describe an enumerator for a non-terminal. Consider for a moment the non-terminal S of the grammar shown in Fig. 4. The parse trees rooted at S is constructed out of the parse trees that belong to the non-terminal A and the sentential form BA. Assume that we have enumerators defined for A and BA, namely $\mathsf{Enum}[A]$ and $\mathsf{Enum}[BA]$ that are functions from natural numbers to parse trees (a pair of them in the case of BA). Our algorithm constructs an enumerator for S compositionally using the enumerators for A and BA.

Recall that we consider enumerators as bijective functions from natural numbers. So, given an index i we need to define a unique parse tree of S corresponding to i (provided i is within the number of parse trees rooted at S). To associate a parse tree of S to an index i, we first need to identify a right-hand-side α of S and select a parse tree t of the right-hand-side. To determine a parse tree t of the right-hand-side α, it suffices to determine the index of t in the enumerator for α. Hence, we define a function $\mathsf{Choose}[N] : \mathbb{N} \to ((\mathcal{N} \cup \Sigma)^* \times \mathbb{N})$ for every non-terminal N, that takes an index and returns a right-hand-side of N, and an index for accessing an element of the right-hand-side. We define the enumerator for a non-terminal as: $\mathsf{Enum}[N](i) = node(N, \mathsf{Enum}[\alpha](j))$, where $(\alpha, j) = \mathsf{Choose}[N](i)$. That is, as a node labelled N and having the tuple $\mathsf{Enum}[\alpha](j)$ as children.

In the simplest case, if N has n right-hand-sides $\alpha_0, \alpha_2, \cdots, \alpha_{n-1}$, the choose function $\mathsf{Choose}[N](i)$ could be defined as $(\alpha_{i\%n}, \lfloor i/n \rfloor)$. This definition, besides being simple, also ensures a fair usage of the right-hand-sides of N by mapping successive indices to different right-hand-sides, which ensures that any sequence of enumeration of the words belonging to a non-terminal alternates over the right-hand-sides of the non-terminal. However, this definition is well defined only when every right-hand-side of N has unbounded number of parse trees. For instance, consider the non-terminal A shown in Fig. 4. It has two right-hand-sides a and aS of which a has only a single parse tree. Defining $\mathsf{Choose}[A]$ as $(\alpha_{i\%2}, \lfloor i/2 \rfloor)$ is incorrect as, for example, $\mathsf{Enum}[A](2)$ maps to $\mathsf{Enum}[a](1)$, which is not defined. Therefore, we extend the above function so that it takes into

$$
\begin{array}{lcl}
S & \rightarrow & A \mid BA \\
A & \rightarrow & a \mid aS \\
B & \rightarrow & \mathsf{b}
\end{array}
$$

$\forall t \in \{a, b\}$. $\mathsf{Enum}[t](i) = \mathit{leaf}(t)$ if $i = 0$
$\forall N \in \{S, A, B\}$. $\mathsf{Enum}[N](i) = node(S, \mathsf{Enum}[\alpha](j)),$ where $(\alpha, j) = \mathsf{Choose}[S](i)$
$\mathsf{Enum}[BA](i) = (\mathsf{Enum}[B](j), \mathsf{Enum}[A](k))$ where $(j, k) = \pi(i, \infty, \infty)$
$\mathsf{Enum}[aS](i) = (\mathsf{Enum}[a](j), \mathsf{Enum}[S](k))$ where $(j, k) = \pi(i, 1, \infty)$

Figure 4. An example grammar and illustrations of the Enum functions for the symbols of the grammar. Choose and π are defined in Fig. 6 and Appendix A, respectively.

$$
\begin{aligned}
&\#t(\alpha) = \prod_{i=0}^{n-1} \#t(M_i), \text{ where } \alpha = M_0 \cdots M_{n-1}, n > 1 \\
&\#t(N) = \sum_{i=0}^{n-1} \#t(\alpha_i), \text{ where } N \rightarrow \alpha_0 \mid \cdots \mid \alpha_{n-1} \\
&\#t(a) = 1, \text{ where } a \in \Sigma
\end{aligned}
$$

Figure 5. Equations for computing the number of parse trees of sentential forms. $\#t$ is initialized to ∞ for every non-terminal M.

$$
\begin{aligned}
&\mathsf{Choose}[N](i) = (\alpha_j, b_k + \lfloor (i - i_k)/(n-k) \rfloor), \\
&\text{where } N \rightarrow \alpha_0 \mid \cdots \mid \alpha_{n-1} \text{ s.t.} \\
&\qquad\qquad \forall 1 \le m < n.\ \#t(\alpha_{m-1}) \le \#t(\alpha_m), \\
&\quad j = k + (i - i_k)\%(n-k), \\
&\quad k \text{ is such that } i_k \le i < i_{k+1}, \\
&\quad b_0 = 0, \text{ and } \forall 1 \le m < n.\ b_m = \#t(\alpha_{m-1}), \\
&\quad \forall 0 \le m < n.\ i_m = b_m(n - m + 1) + \sum_{i=0}^{m-1} b_i
\end{aligned}
$$

Figure 6. Choose function for a non-terminal N.

account the number of the parse trees belonging to the right-hand-sides, which is denoted using $\#t(\alpha)$.

It is fairly straightforward to compute the number of parse trees of non-terminals and right-hand-sides in a grammar. For completeness, we show a formal definition in Fig. 5. We define $\#t$ as the greatest fix-point of the equations shown in Fig. 5, which can be computed iteratively starting with an initial value of ∞ for $\#t$. As shown in the equations, the number of (tuples of) parse trees of a sentential form is the product of the number of parse trees of the symbols in the sentential form. The number of parse trees of a non-terminal is the sum of the number of parse trees of its right-hand-sides, and the number of the parse trees of a terminal is one. Note that if the grammar has cycles, $\#t$ could be infinite in some cases.

Fig.6 defines a Choose function, explained below, that can handle right-hand-sides with a finite number of parse trees. The definition guarantees that whenever Choose returns a pair (α, i), i is less than $\#t(\alpha)$, which ensures that $\mathsf{Enum}[\alpha]$ is defined for i. In Fig.6, the right-hand-sides of the non-terminal N, $\alpha_0, \cdots, \alpha_{n-1}$, are sorted in ascending order of the number of parse trees belonging to them. The index i_m is the smallest index (of $\mathsf{Enum}[N]$) at which the m^{th} right-hand-side α_m becomes undefined, which is determined using the number of parse trees of each right-hand-side as shown. Given an index i, $\mathsf{Choose}[N](i)$ first determines the right-hand-sides that need to be skipped i.e, whose enumerators are not defined for the index i, by finding a k such that $i_k \le i < i_{k+1}$. It then chooses a right-hand-side (namely α_j) from the remaining $n - k$ right-hand-sides whose enumerators are defined for the index i, and computes the index to enumerate from the chosen right-hand-side.

Note that the Choose function degenerates to the simple definition $(\alpha_{i\%n}, \lfloor i/n \rfloor)$ presented earlier when $\#t$ is unbounded for every right-hand-side of N. The function also preserves fairness by mapping successive indices to different right-hand-sides of the non-terminals. For instance, in the case of the non-terminal A shown in Fig. 4, the Choose function maps index 0 to $(a, 0)$, index 1 to $(aS, 0)$, but index 2 is mapped to $(aS, 1)$ as a has only one parse tree, i.e, $\#t(a) = 1$.

We now describe the enumerator for a sentential form α with more than one symbol. Let $\alpha = M_1 M_2 \cdots M_m$. The tuples of parse trees belonging to the sentential form is the cartesian product of the parse trees of $M_1, \cdots, M_m$. However, eagerly computing the cartesian product is impossible for most realistic grammars because it is either unbounded or untractably large. Nevertheless, we are interested only in accessing a tuple at a given index i. Hence, it suffices to determine for every symbol M_j, the parse tree t_j that is used to construct the tuple at index i. The tree t_j can be determined if we know its index in $\mathsf{Enum}[M_j]$. Therefore, it suffices to define a bijective function $\pi : \mathbb{N} \rightarrow \mathbb{N}^m$ that maps a natural number (the index of $\mathsf{Enum}[\alpha]$) to a point in an m-dimensional space of natural numbers. The j^{th} component of $\pi(i)$ is the index of $\mathsf{Enum}[M_j]$ that corresponds to the j^{th} parse tree of the tuple. In other words, $\mathsf{Enum}[M_1 \cdots M_m](i)$ could be defined as $(\mathsf{Enum}[M_1](i_1), \cdots, \mathsf{Enum}[M_m](i_m))$, where i_j is the j^{th} component of $\pi(i)$.

When m is two, the function π reduces to an *inverse pairing function* that is a bijection from natural numbers to pairs of natural numbers. Our algorithm uses only an inverse pairing function as we normalize the right-hand-sides

`Prgm`	$\rightarrow$	`import QName ; ClassDef`
`QName`	$\rightarrow$	`ID	ID . QName`
`ClassDef`	$\rightarrow$	`class { Body }`

Figure 7. A grammar snippet illustrating the need to bound the length of the generated words during enumeration.

of the productions in the grammar to have at most two symbols. We use the well known *Cantor's inverse pairing function* [26]. But, this function assumes that the two dimension space is unbounded in both directions, and hence cannot be employed directly when the number of parse trees generated by the symbols in the sentential form are bounded. We extend the inverse paring functions to two dimensional spaces that are bounded in one or both the directions. The extended functions take three arguments, the index that is to be mapped, and the sizes of the x and y dimensions (or infinity if they are unbounded). We present a formal definition of the functions in Appendix A. Using the extended Cantor's pairing function π we define the enumerator for a sentential form with two symbols as: $\mathsf{Enum}[M_1M_2](i) = (\mathsf{Enum}[M_1](i_1), \mathsf{Enum}[M_2](i_2))$, where $(i_1, i_2) = \pi(i, \#t(M_1), \#t(M_2))$.

Termination of Random Access. We later in section 2.3 we present a bound on the running time of the algorithm, but now we briefly discuss termination. If N is a recursive non-terminal e.g, if it has a production of the form $N \rightarrow \alpha N \beta$, the enumerator for N may recursively invoke itself, either directly or through other enumerators. However, for every index other than 0 and 1, the recursive invocations will always be passed a strictly smaller index. This follows from the definition of the Choose and the inverse pairing functions used by our algorithm. (In the case of the inverse pairing function, if $\pi(i) = (j, k)$, j and k are strictly smaller than i for all $i > 1$). For indices 0 and 1 the recursive invocations may happen with the same index. However, this will not result in non-termination if the following properties are ensured: (a) for every non-terminal, the right-hand-side chosen for index 0 is the first production in the shortest derivation starting from the non-terminal and ending at a word. (b) There are no unproductive non-terminals (which are non-terminals that do not generate any word) in the input grammar.

From Parse Trees to Words. We obtain enumerators for words using the enumerators for parse trees by mapping the enumerated parse trees to words. However, when the input grammar is ambiguous, the resulting enumerators are no longer bijective mappings from indices to words. The number of indices that map to a word is equal to the number of parse trees of the word.

2.2 Enumerating Fixed Length Words

The enumeration algorithm we have described so far is agnostic to the lengths of the enumerated words. As a consequence, the algorithm may generate undesirably long words, and in fact may also favour the enumeration of long words over shorter ones. Fig. 7 shows a snippet from the Java grammar that results in this behavior.

$$\begin{array}{rcl} S & \rightarrow & a \mid BS \\ B & \rightarrow & \mathrm{b} \end{array} \quad \text{(a)}$$

$$\text{(b)} \quad \begin{array}{rcl} S_3 & \rightarrow & B_1S_2 \mid B_2S_1 \\ S_2 & \rightarrow & B_1S_1 \\ S_1 & \rightarrow & \mathrm{a} \\ B_1 & \rightarrow & \mathrm{b} \end{array}$$

Figure 8. (a) An example grammar. (b) the result of restricting the grammar shown in (a) to words of size 3.

In the Fig. 7, the productions of the non-terminal Body are not shown for brevity. It generates all syntactically correct bodies allowed for a class in a Java program. Consider the enumeration of the words (or parse trees) belonging to the non-terminal Prgm starting from index 1. A fair enumeration strategy, such as ours, will try to generate almost equal number of words from the non-terminals QName and ClassDef. However, the lengths of the words generated for the same index differ significantly between the non-terminals. For instance, the word generated from the non-terminal QName at an index i has length $i + 1$. On the other hand, the lengths of the words generated from the non-terminal ClassDef grow slowly relative to their indices, since it has many right-hand-sides, and each right-hand-side is in turn composed of non-terminals having many alternatives. In essence, the words generated for the non-terminal Prgm will have long import declarations followed by very short class definitions.

Moreover, this also results in reduced coverage of rules since the enumeration heavily reuses productions of QName, but fails to explore many alternatives reachable through ClassDef. We address this issue by extending the enumeration algorithm so that it generates only parse trees of words having a specified length, We accomplish this by transforming the input grammar in such way that it produces only strings that are of the required length, and use the transformed grammar in enumeration. The idea behind the transformation is quite standard e.g, works such as [14] and [19] that develop theoretical algorithms for random sampling of unambiguous grammars also resort to a similar approach. However, what is unique to our algorithm is using the transformation to construct bijective enumerators while guaranteeing random access property for all words of the specified length.

Fig. 8 illustrates this transformation on an example, which is explained in detail below. For explanatory purposes, assume that the input grammar is in *Chomsky's Normal Form* (CNF) [16] which ensures that every right-hand-side of the grammar is either a terminal or has two non-terminals.

Fig. 9 formally defines the transformation. For every non-terminal N of the input grammar, the transformation creates a non-terminal N_l that generates only those words of N that have a length l. The productions of N_l are obtained by transforming the productions of N. For every production of

$$
\begin{aligned}
[\![N]\!]_l &= [\![N \to \alpha_1]\!]_l \cup \cdots \cup [\![N \to \alpha_n]\!]_l, \\
&\quad \text{where } N \to \alpha_1, \mid \cdots \mid \alpha_n \\
[\![N \to a]\!]_l &= \begin{cases} \{N_l \to a\} & \text{if } l = 1 \\ \emptyset & \text{otherwise} \end{cases} \\
[\![N \to AB]\!]_l &= \bigcup_{i=1}^{l-1} \left(\{N_l \to A_i B_{l-i}\} \cup [\![A]\!]_i \cup [\![B]\!]_{l-i}\right)
\end{aligned}
$$

Figure 9. Transforming non-terminals and productions of a grammar to a new set of non-terminals and productions that generate only words of length l.

the form $N \to a$, where a is a terminal, the transformation creates a production $N_l \to a$ if $l = 1$. For every production of the form $N \to AB$ that has two non-terminals on the right-hand-side, the transformation considers every possible way in which a word of size l can be split between the two non-terminals, and creates a production of the form $N_l \to A_i B_{l-i}$ for each possible split $(i, l-i)$. Additionally, the transformation recursively produces rules for the non-terminals A_i and B_{l-i}. The transformed grammar may have unproductive non-terminals and rules that do not generate any word (like the non-terminal B_2 and rule $S_3 \to B_2 S_1$ of Fig. 9(b)), and hence may have to be simplified. Observe that the transformer grammar is acyclic and generates only a finite number of parse trees.

The grammar produced by this transformation has size $O(n \cdot l^2)$, where n is the size of the input grammar. For efficiency reasons, we constructs the productions of the transformed grammar on demand, when it is required during the enumeration of a parse tree.

Sampling Parse Trees and Words. Having constructed enumerators with the above characteristics, it is straightforward to sample parse trees and words of a non-terminal N having a given length l. We uniformly randomly sample numbers in the interval $[0, \#t(N) - 1]$, and lookup the parse tree or word at the sampled index using the enumerators. Since we have a bijection from numbers in the range $[0, \#t(N) - 1]$ to parse trees of N, this approach guarantees a uniform random sampling of parse trees. However, sampling of words is guaranteed to be uniform only if the grammar is unambiguous. In general, the probability of choosing a word w of length l in a sample of size s is equal to $\frac{t \times s}{\#t(N)}$, where t is the number of parse trees of the word w.

2.3 Running Time of Random Access

We now derive an upper bound on the time taken to access a parse tree at an index i, when the lengths of the words generated are restricted to l. Assume that the number of parse trees ($\#t$) generated by every non-terminal and right-hand-side is precomputed. (We will discuss the complexity of this shortly). Since the Enum function is recursive, we compute an upper bound for the function by deriving a bound on the number of recursive invocations of Enum, and the time spent between two successive invocations of Enum on the non-terminals in the grammar.

Recall that for a non-terminal N, $\mathsf{Enum}[N](i)$ first uses the Choose function to select a right-hand-side of N and then uses the inverse paring functions π to recurse into the non-terminals in the chosen right-hand-side. Let r be the largest number of right-hand-sides of a non-terminal in the input grammar. The number of right-hand-sides of a non-terminal in the grammar specialized to length l (using the transformation described earlier) is bounded by $r \cdot l$. The running time of the function $\mathsf{Choose}[N]$, defined in Fig. 6, is bounded by $O(r \cdot l \cdot |i|^2)$, where $|i|$ is the size of the index in terms of number of bits. The running time of the π function, defined in Appendix A, is bounded by $O(|i|^2)$ as it only performs a sequence of elementary arithmetic and integer square root operations on the index. Hence, the time spent between two recursive invocations of Enum on the non-terminals in the grammar is $O(r \cdot l \cdot |i|^2)$.

Observe that the time taken by Choose and inverse pairing functions are independent of the number of parse trees ($\#t$), even though some values of $\#t$ are used by the functions. This is because, the functions use only those values of $\#t$ that are smaller than the index i in the arithmetic operations, and only need to know for any α whether $\#t(\alpha)$ is greater than i, which can be computed in time $O(i)$ given that $\#t$ has been precomputed.

To bound the number of recursive invocations of Enum, we require that the input grammar is expressed in Chomsky's Normal Form (CNF). For a grammar in CNF, the number of edges in any parse tree of a word of length l is $2l - 1$ [16], which is also equal to the number of recursive invocations of Enum on the non-terminals in the grammar. Therefore, we have the following theorem.

Theorem 1. *Let G be a grammar in Chomsky's Normal Form. Let r denote the largest number of right-hand-sides of a non-terminal. The time taken to access a parse tree generating a word of length l at an index i of a non-terminal belonging to the grammar is upper bounded by $O(r \cdot |i|^2 \cdot l^2)$, provided the number of the parse trees generated by each non-terminal and right-hand-side is precomputed.*

Notice that the time taken for random access is polynomial in the size of the input grammar, the number of bits in the index ($|i|$), and the size of the generated word l. The complexity can be further reduced to $O(|i|^2 \cdot l \cdot \log l \cdot \log r)$ using efficient data structures. We now briefly discuss the complexity of computing the number of parse trees ($\#t$). For a grammar in CNF, the number of parse trees that generate a word of length l is $O(r^{2^l})$ in the worst case. (For unambiguous grammars, it is $O(c^l)$, where c is the number of terminals.) Thus, computing the number of parse trees could, in principle, be expensive. However, in practice, the number of parse trees, in spite of being large, is efficiently computable.

Prior theoretical works on uniform random sampling (such as [19]) for context-free grammars assume that the input grammar is a constant, and that the arithmetic operations take constant time. (Our approach matches the best known running time $O(l \log l)$ under these assumptions). But, this assumption is quite restrictive in the real-world. For example, the Java 7 grammar has 430 non-terminals and 2447 rules when normalized to CNF, and the number of parse trees increases rapidly with the length of the generated word. In fact, for length 50, it is a 84 digit number (in base 10). Using numbers as big as these in computation introduces significant overhead which cannot be discounted. Our enumerators offer quite some flexibility in sampling by supporting random access. For example, we can sample only from a restricted range instead of using the entire space of parse trees. Since we ensure a fair usage of rules while mapping rules to indices, restricting the sizes of indices still provides a good coverage of rules. In fact, our implementation exposes a parameter for limiting the range of the sample space, which we found useful in practice.

3. Counter-Example Detection

We apply the enumerators described in previous sections to find counter-examples for equivalence of two context-free grammars. We sample words (of length within a predefined range) from one grammar and check if they are accepted by the other and vice versa. Bounding the length of words greatly aids in reducing the parsing overhead especially while using generic parsers.

In section 5, we present detailed results about the efficiency and accuracy of counter-example detection. The results show that the implementation is able to enumerate and parse millions of words within a few minutes on large real-world grammars. In the sequel, we present an overview of the parsers used by the tool.

Parsing. We use a suite of parsers consisting of CYK parser [16], Antlr v4 parser [1] (in compiler and interpreter modes), and LL(1) [3] parser, and employ them selectively depending on the context. For instance, for testing large programming language grammars for equivalence, we compile the grammars to parsers (at runtime) using Antlr v4, which uses adaptive LL(*) parsing algorithm [25], and use the parsers to check if the generated words are accepted by the grammar. The CYK parsing algorithm we implement in our tool, is a top-down, non-recursive algorithm that memoizes the parsing information computed for the substrings of the word being parsed (using a trie data structure), and reuses the information on encountering the same substring again, during a parse of the same or another word. Though the top-down evaluation introduces some overheads compared to the conventional dynamic programming approach, it improves the performance of the CYK parser by orders of magnitude when used in batch mode to parse a collection of words using the same grammar.

(a) $\begin{array}{lcl} S & \rightarrow & aT \\ T & \rightarrow & aTb \mid b \end{array}$ (b) $\begin{array}{lcl} P & \rightarrow & aR \\ R & \rightarrow & abb \mid aRb \mid b \end{array}$

Figure 10. GNF grammars for the language $a^n b^n$.

We mostly rely on the optimized CYK parser for checking the correctness of students' solutions. We find that quite often the solutions provided by students are convoluted, and are tricky to parse using specialized parsers. For instance, for a grammar with productions $S \rightarrow a \mid B$ and $B \rightarrow aaBb \mid aB \mid \epsilon$, the performance of the Antlr v4 parser degenerates severely with the length of word that is parsed.

4. Proving Equivalence

Our approach for proving equivalence is based on the algorithms proposed in [15] and extended by the works of [24] and [12]. This family of algorithms is attractive because it works directly on context-free grammars without requiring conversions to other representation like push-down automatons. Moreover, they come with strong completeness guarantees. [15] introduces a decision procedure for checking equivalence of *simple deterministic grammars*, which are LL(1) grammars in Griebach Normal Form (GNF). [24] extends the algorithm of [15] to LL(k) grammars in GNF, and [12] extends [15] in another direction, which is to decide equivalence of deterministic GNF grammars when one of them is LL(1).

Our approach extends the work of [24] by incorporating several aspects of [12]. The resulting algorithm is applicable to arbitrary context-free grammars, but at the same time is complete for LL grammars (our implementation is complete only for LL(2) grammars since we limit the lookahead for efficiency reasons). Furthermore, we perform several extension to the algorithm that improves its precision and also performance in practice. In particular, we extend the approach to handle inclusion relations, which provides a alternative way of establishing equivalence when the equivalence query is not directly provable. We also introduce transformations that use concrete examples to dynamically refine the queries during the course of the algorithm. Our experiments show that the algorithm succeeds in 82% of the cases that passed all test cases, proving queries involving ambiguous grammars (see section 5).

The algorithm. We use the grammars shown in Fig. 10 for the language $a^n b^n$ as a running example. Observe that the grammar shown on the right is ambiguous – it has two parse trees for $aabb$. We formalize the verification algorithm as a proof system that uses the inference rules shown in Fig. 11. We later discuss an extension to the algorithm that augments the rules with a fixed *lookahead* distance k. Fig. 12 illustrates the algorithm on our running example. In the sequel, we make the following assumptions: (a) the input grammars do not have any epsilon productions, (b) the grammars have the same set of terminals, and (c) the non-terminals belonging to grammars are unique.

Derivatives. We express our algorithm using the notion of a *derivative* of a sentential form which is defined as follows. A derivative $d : \Sigma^* \times (\mathcal{N} \cup \Sigma)^* \to 2^{(\mathcal{N}\cup\Sigma)^*}$ is a function that given a word w and a sentential form α, computes the sentential forms β that remain immediately after deriving w from α. For a sentential form α, w will be derivable from α (if it is derivable) in exactly $|w|$ steps if the grammar is in GNF, since every production of the grammar starts with an alphabet. For keeping the definition simple, we consider going over a terminal symbol in the sentential form as a step in the derivation. That is, if $A \to a$ and $B \to b$, we say AaB derives aab in 3 steps and denote it as $AaB \Rightarrow^3 aab$. We define the derivative as $d(w, \alpha) = \{\beta \mid \alpha \Rightarrow^n w\beta\}$, where n is the length of the word w. This notion can be extended to grammars that are not in GNF as well. We lift the derivative operation to a set of sentential forms as: $\hat{d}(w, \boldsymbol{\alpha}) = \bigcup_{\alpha \in \boldsymbol{\alpha}} d(w, \alpha)$.

Inference Rules. We consider two types of relations between sets of sentential forms: equivalence ($\equiv$) and inclusion ($\subseteq$). A relation $\boldsymbol{\alpha} \equiv \boldsymbol{\beta}$ (or $\boldsymbol{\alpha} \subseteq \boldsymbol{\beta}$) holds if the set of words generated by the sentential forms in $\boldsymbol{\alpha}$ i.e, $\bigcup_{\alpha \in \boldsymbol{\alpha}} L(\alpha)$ is equal to (or included in) the set of word generated by the sentential forms in $\boldsymbol{\beta}$ i.e, $\bigcup_{\beta \in \boldsymbol{\beta}} L(\beta)$. Though we are only interested in proving equivalence of sentential forms, our algorithm sometimes uses inclusion relations in the intermediate steps to establish equivalence. As a consequence, the approach can also be used to prove inclusion of grammars. But, the rules do not guarantee completeness. The rules shown Fig. 11 use judgements of the form $\mathcal{C} \vdash \boldsymbol{\alpha} \subseteq \boldsymbol{\beta}$, where $\mathcal{C}$ is a set of relations which can be assumed to hold when deciding the truth of $\boldsymbol{\alpha} \subseteq \boldsymbol{\beta}$. Every inference rule shown Fig. 11 provides a set of judgements, given by the antecedents, which when established guarantees that the consequent holds. In other words, the antecedents provide a sufficient condition for the consequent. (Sometimes they are also necessary conditions.)

Consider the illustration shown in Fig. 12. Our goal is to establish that the start symbols of the two grammars are equivalent under an empty context, i.e, $\emptyset \vdash [\mathtt{S} \equiv \mathtt{P}]$. We prove this by finding a derivation for the judgement using the inference rules. In Fig. 12, the relations that are added to the context are marked with †. At any step in the derivation, we can assume that every relation that is marked in the preceding steps leading to the current step hold.

BRANCH Rule. Initially, we apply the BRANCH rule to $[\mathtt{S} \equiv \mathtt{P}]$. The rule asserts that a relation $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$ holds in a context $\mathcal{C}$ if for every alphabet a, the derivatives of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ with respect to a are related under the same operation. The correctness of this part is obvious: if two sentential forms are equivalent, the sentential forms that remain after deriving the first character ought to be equivalent. Additionally, the BRANCH rule allows the relation $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$ to be considered as valid when proving the antecedents. This is because, the BRANCH rule also incorporates inductive reasoning. To prove $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$, the rule hypothesises that the relation holds for all words with length smaller than k, and attempts to establish that the relation holds for words of length k. It suffices for the antecedents to hold for all words of length less than k since we peel off the first character from the words generated by $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ by computing the derivative. Therefore, during the proof of the antecedents if the relation $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$ is encountered again then we know that it needs to hold only for words of length less than k, which holds by hypothesis.

An equivalent contra-positive argument is that, if the relation $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$ has a counter-example then the antecedents will have a strictly smaller counter-example. However, when $\boldsymbol{\alpha}$ op $\boldsymbol{\beta}$ is encountered during the proof of the antecedents it need not be explored any further because it would not lead to the smallest counter-example. [12] refers to this property, wherein the counter-examples of the newly created relations (antecedents) are strictly smaller than the counter-examples of the input relation (consequent) when they exist, as *monotonicity*. In our system, the only other rule that is monotonic is the SPLIT rule.

Applying the BRANCH rule to $[\mathtt{S} \equiv \mathtt{P}]$ produces the relation $[\mathtt{T} \equiv \mathtt{R}]$ for the alphabet a since T and R are derivatives of S and P w.r.t a, and produces the empty relation $[\emptyset \equiv \emptyset]$ for alphabet b. The empty relation trivially holds, as asserted by rule *Empty*, and hence is not shown.

Equivalence to Inclusion. The INCLUSION rule reduces equivalence relations to pairs of inclusion relations, (e.g. see relation 3 in Fig. 12). The DIST rule simplifies the inclusion relations by distributing the inclusion operation over the left-hand-sides, as illustrated on the relation 7. These rules ensure that every relation generated during the algorithm is normalized to the form $\{\alpha\} \equiv \{\beta\}$, or $\{\alpha\} \subseteq \boldsymbol{\beta}$.

The TESTCASES rule applies to a relation of the form $\{\alpha\} \subseteq \boldsymbol{\beta}$. It samples a predefined set of words from α and searches for a strict subset of $\boldsymbol{\beta}$ that accepts all the samples. On finding such a subset S, it construct a stronger relation $\{\alpha\} \subseteq S$ that implies the input relation. For instance, the rule reduces the relation 6: $[\mathtt{Tb} \subseteq \mathtt{Rb} \cup \mathtt{bb}]$ to $[\mathtt{Tb} \subseteq \mathtt{Rb}]$ using a set of examples. This rule uses an enumerator to sample words from sentential forms and a parser to check if the sample words are accepted by the sentential forms. In our implementation, we use a CYK parser extended for parsing sentential forms to check if the sample words are accepted by the sentential forms.

INDUCT Rule. The INDUCT rule asserts that all relations implied by the context hold. The implication check only uses syntactic equality of the sentential forms. In particular, for equality relations $\boldsymbol{\alpha} \equiv \boldsymbol{\beta}$, we check if the context contains the same relation or $\boldsymbol{\beta} \equiv \boldsymbol{\alpha}$. For inclusion relations of the form $\boldsymbol{\alpha} \subseteq \boldsymbol{\beta}$, we check if the context contains an equivalence relation between $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ or an inclusion relation of the form

$$\textsc{Branch}\quad \frac{\forall a \in \Sigma.\ \mathcal{C} \cup \{\boldsymbol{\alpha}\ \mathtt{op}\ \boldsymbol{\beta}\} \vdash \hat{d}(a,\boldsymbol{\alpha})\ \mathtt{op}\ \hat{d}(a,\boldsymbol{\beta})}{\mathcal{C} \vdash \boldsymbol{\alpha}\ \mathtt{op}\ \boldsymbol{\beta}}$$

$$\textsc{Inclusion}\quad \frac{\mathcal{C} \vdash \boldsymbol{\alpha} \subseteq \boldsymbol{\beta} \qquad \mathcal{C} \vdash \boldsymbol{\beta} \subseteq \boldsymbol{\alpha}}{\mathcal{C} \vdash \boldsymbol{\alpha} \equiv \boldsymbol{\beta}}$$

$$\textsc{Dist}\quad \frac{\forall 1 \le i \le m.\ \mathcal{C} \vdash \{\alpha_i\} \subseteq \boldsymbol{\beta}}{\mathcal{C} \vdash \bigcup_{i=1}^{m} \alpha_i \subseteq \boldsymbol{\beta}}$$

$$\textsc{Induct}\quad \frac{rel \in \mathcal{C} \qquad rel \Rightarrow \boldsymbol{\alpha}\ \mathtt{op}\ \boldsymbol{\beta}}{\mathcal{C} \vdash \boldsymbol{\alpha}\ \mathtt{op}\ \boldsymbol{\beta}}$$

$$\textsc{Split}\quad \frac{\begin{array}{c} ||A|| = x \qquad |\gamma| > 1 \qquad \Psi = \bigcup_{i=1}^{m} \psi_i\beta_i \qquad \hat{d}(x,\Psi) = \bigcup_{i=1}^{m} \boldsymbol{\rho}_i\beta_i \qquad \forall 0 \le i \le m.\ |\beta_i| > 0 \\ \mathcal{C}' = \mathcal{C} \cup \{\{A\gamma\}\ \mathtt{op}\ \Psi\} \qquad \mathcal{C}' \vdash \{\gamma\}\ \mathtt{op}\ \hat{d}(x,\Psi) \qquad \forall 0 \le i \le m.\ \mathcal{C}' \vdash A\boldsymbol{\rho}_i\ \mathtt{op}\ \{\psi_i\} \end{array}}{\mathcal{C} \vdash \{A\gamma\}\ \mathtt{op}\ \Psi}$$

$$\textsc{Testcases}\quad \frac{S \subset \boldsymbol{\beta} \qquad sample(n,\alpha) \subseteq \bigcup_{\beta \in S} L(\beta) \qquad \mathcal{C} \vdash \{\alpha\} \subseteq S}{\mathcal{C} \vdash \{\alpha\} \subseteq \boldsymbol{\beta}}$$

$$\textsc{Empty1}\quad \vdash \emptyset \equiv \emptyset \qquad \textsc{Empty2}\quad \vdash \emptyset \subseteq \boldsymbol{\beta}$$

$$\textsc{Epsilon}\quad \frac{\mathcal{C} \vdash \boldsymbol{\alpha}\ \mathtt{op}\ \boldsymbol{\beta}}{\mathcal{C} \vdash (\boldsymbol{\alpha} \cup \{\epsilon\})\ \mathtt{op}\ (\boldsymbol{\beta} \cup \{\epsilon\})}$$

Figure 11. Basic inference rules of the verification algorithm. In the figure, $\mathtt{op} \in \{\equiv, \subseteq\}$, $||A||$ is the shortest word derivable from A, and $rel_1 \Rightarrow rel_2$ is a syntactic implication check that holds if rel_1 is stronger than rel_2.

$${}^{1}[\mathtt{S} \equiv \mathtt{P}]^{\dagger} \xrightarrow{\textsc{Branch}} {}^{2}[\mathtt{T} \equiv \mathtt{R}]^{\dagger} \xrightarrow{\textsc{Branch}} {}^{3}[\mathtt{Tb} \equiv \mathtt{Rb} \cup \mathtt{bb}] \wedge {}^{4}[\mathtt{b} \equiv \mathtt{b}]$$
$${}^{4}[\mathtt{b} \equiv \mathtt{b}] \xrightarrow{\textsc{Branch}} {}^{5}[\epsilon \equiv \epsilon] \xrightarrow{\textsc{Epsilon}} [\emptyset \equiv \emptyset] \xrightarrow{\textsc{Empty}} \text{proved}$$
$${}^{3}[\mathtt{Tb} \equiv \mathtt{Rb} \cup \mathtt{bb}] \xrightarrow{\textsc{Inclusion}} {}^{6}[\mathtt{Tb} \subseteq \mathtt{Rb} \cup \mathtt{bb}] \wedge {}^{7}[\mathtt{Rb} \cup \mathtt{bb} \subseteq \mathtt{Tb}]$$
$${}^{6}[\mathtt{Tb} \subseteq \mathtt{Rb} \cup \mathtt{bb}] \xrightarrow{\textsc{Testcases}} {}^{8}[\mathtt{Tb} \subseteq \mathtt{Rb}]^{\dagger} \xrightarrow{\textsc{Split}} {}^{9}[\mathtt{b} \subseteq \mathtt{b}] \wedge {}^{10}[\mathtt{T} \subseteq \mathtt{R}] \xrightarrow{\textsc{Induct}} {}^{11}[\mathtt{b} \subseteq \mathtt{b}] \xrightarrow{*} \text{proved}$$
$${}^{7}[\mathtt{Rb} \cup \mathtt{bb} \subseteq \mathtt{Tb}] \xrightarrow{\textsc{Dist}} {}^{12}[\mathtt{Rb} \subseteq \mathtt{Tb}] \wedge {}^{13}[\mathtt{bb} \subseteq \mathtt{Tb}]$$
$${}^{13}[\mathtt{bb} \subseteq \mathtt{Tb}]^{\dagger} \xrightarrow{\textsc{Branch}} {}^{14}[\mathtt{b} \subseteq \mathtt{b}] \wedge {}^{15}[\emptyset \subseteq \mathtt{Tbb}] \xrightarrow{*} \text{proved}$$
$${}^{12}[\mathtt{Rb} \subseteq \mathtt{Tb}]^{\dagger} \xrightarrow{\textsc{Split}} {}^{16}[\mathtt{b} \subseteq \mathtt{b}] \wedge {}^{17}[\mathtt{R} \subseteq \mathtt{T}] \xrightarrow{\textsc{Induct}} {}^{18}[\mathtt{b} \subseteq \mathtt{b}] \xrightarrow{*} \text{proved}$$

Figure 12. Illustration of application of the rules on the running example. A star (*) denotes application of one or more rules. Curly braces around singleton sets are omitted.

$\boldsymbol{\alpha} \subseteq S$, where S has fewer sentential forms than $\boldsymbol{\beta}$. For instance, in the derivation shown in Fig. 12, the relations 10 and 17 are implied by the relation 2: $[\mathtt{T} \equiv \mathtt{R}]$, added to the context in step 2 during the application of BRANCH rule, and hence are considered valid.

SPLIT Rule. The main purpose of the SPLIT Rule is to prevent the sentential forms in the relations from becoming excessively long. The key idea behind the rule is to *split* the sentential forms that are compared (say $[\mathtt{A}\gamma] \equiv [\beta\delta]$) into smaller chunks that are piece-wise equivalent e.g. as $[\mathtt{A}\rho \equiv \beta]$, and $[\gamma \equiv \rho\delta]$ (where ρ is a sentential form derived from β), while preserving completeness under some restrictions. It identifies the point to split by deriving the shortest word of A from the other side of the relation.

We apply this rule only to a relation whose left-hand-side is a singleton (since all relations will be reduced to this form). Let r_1 be the relation $\{A\gamma\}\ \mathtt{op}\ \Psi$ (with non-empty γ). Let x be the shortest word that can be derived from A, denoted $||A||$. The SPLIT rule requires that every sentential form in Ψ can be split into $\psi_i\beta_i$ such that ψ_i can derive x, and β_i is non-empty. (However, this requirement can be relaxed as described shortly). This implies that the derivative of Ψ w.r.t the word x will preserve the suffix β_i. That is, $\hat{d}(x,\Psi)$ will be of the form $\bigcup_i \boldsymbol{\rho}_i\beta_i$ (where $\boldsymbol{\rho}_i$ is a set of sentential forms).

Under the above conditions, the rule asserts that if $\gamma\ \mathtt{op}\ \hat{d}(x,\Psi)$, and, for all i, $A\rho_i\ \mathtt{op}\ \psi_i$ holds, then so does r_1. Furthermore, the rule allows assuming r_1 while proving the antecedents. The requirement that all β_is are non-empty ensures the monotonicity of the rule. If this requirement does not hold, the rule is still applicable but we cannot add r_1 to the context.

The soundness of this assertion is easy to establish. For all i, $A\boldsymbol{\rho}_i\ \mathtt{op}\ \{\psi_i\}$ implies $A\boldsymbol{\rho}_i\beta_i\ \mathtt{op}\ \{\psi_i\beta_i\}$ (since we are concatenating the left- and the right-hand-sides with the same sentential form). This entails that $\bigcup_i A\boldsymbol{\rho}_i\beta_i\ \mathtt{op}\ \bigcup_i \psi_i\beta_i$. We are also given that γ and $\bigcup_i \boldsymbol{\rho}_i\beta_i$ (which is $\hat{d}(x,\Psi)$) are related by op, where $\mathtt{op} \in \{\equiv, \subseteq\}$. Substituting $\bigcup_i \boldsymbol{\rho}_i\beta_i$ with γ yields $A\gamma\ \mathtt{op}\ \bigcup_i \psi_i\beta_i$, which is the relation r_1. Hence, the antecedents imply the consequent. However, the converse does not necessarily hold. It holds (at least for equivalence) only when the grammars satisfy the *suffix property*: $\alpha\beta \equiv \gamma\beta \Rightarrow \alpha \equiv \gamma$, and are *strict* deterministic [12].

In the illustration shown in Fig. 12, the split rule is applied on relations 8 and 12. Consider the relation 8: $[\texttt{Tb} \subseteq \texttt{Rb}]$. The shortest word derivable from T is b (see Fig. 14). Since, $d(b, Rb) = b$, we can deduce that ψ_1 is R, β_1 is b, and $\rho_1 = \epsilon$ (which is the sentential form that remains after deriving b from R). The new relations created by the SPLIT rule are $\gamma \ \texttt{op} \ d(b, Rb)$, and $A\rho_1 \ \texttt{op} \ \psi_1$, which correspond to $[\texttt{b} \subseteq \texttt{b}]$ and $[\texttt{T} \subseteq \texttt{R}]$. Note that without the application of the SPLIT rule, the relation $[\texttt{Tb} \subseteq \texttt{Rb}]$ will gradually grow with the application of BRANCH rule and lead to non-termination.

Application Strategy and Termination Checks. In order to preserve termination and completeness of the algorithm for LL grammars, we adopt a specific strategy for applying the rules. We use the INCLUSION rule to convert an equivalence relation to inclusion relations only when at least one of the operands of the relation has size greater than one. Such cases will not arise if both the grammars are LL(1) (or LL(k) when the rules are augmented with a lookahead distance of k). We prevent the sentential forms from growing beyond a threshold by applying SPLIT rule whenever the threshold is reached. We prioritize the application of rules EMPTY, INDUCT, and TESTCASES that simplify the relations over the BRANCH rule.

We use a set of filters to identify relations that are false and to terminate the algorithm. An important filter is the *Length* filter, which checks for every equivalence query $\{\alpha\} \ \texttt{op} \ \{\beta\}$, whether the length of the left sentential form α is larger than the length of the shortest word that can be generated by β, and vice versa. If this check fails, one of the sentential forms cannot generate the shortest word of the other and the relation does not hold. (Recall that the input grammar do not have epsilon productions.) We also use other filters especially for inclusion relations to quickly abort the search and report a failure. We elide details for brevity.

The algorithm described above reduces to the algorithm of [15] for LL(1) grammars that are in GNF. Hence, our algorithm is a decision procedure for LL(1) grammars in GNF. However, our algorithm may not terminate for grammars outside this class, since the sentential forms in an inclusion relation can grow arbitrarily long. In our implementation, we abort the algorithm and return failure if the algorithm exhausts the memory resources or exceeds a parametrizable time limit (fixed as 10s in our experiments).

4.1 Incorporating Lookahead

The rules shown in Fig. 11 do not use of any lookahead, which, loosely speaking, means that the inferences made from a relation depend on at most one alphabet, analogous to a parser which looks at only the current input character to make a parsing decision. We now briefly explain the extensions for incorporating a finite amount of lookahead. Our extensions are based on the approach of [24].

We perform two major extensions to the relations and sentential forms: (a) We qualify every relation with a (possibly empty) word x, which restricts the relations to only words having x as a prefix. For instance, $\boldsymbol{\alpha} \equiv_x \boldsymbol{\beta}$ holds iff $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ generate the same set of words having the prefix x. (b) We introduce two special types of sentential forms: *prefix restricted sentential forms* (PRS), and *grouped variables*. A PRS is of the form $[\![x, \alpha]\!]$ where x is a word and α is a sentential form. It allows only those derivations of α that will lead to a word having x as the prefix. A grouped variable is a disjoint union of two or more PRS that have different prefixes. A grouped variable allows all derivations that are possible through its individual members, akin to a union of sentential forms. PRS and grouped variables are formally defined in [24] They can be treated as any other sentential form e.g. can be concatenated with other sentential forms, used in derivative computation and so on.

We extend the definition of a derivative $d(w, \alpha)$ so that it additionally accepts a string x and refines the result of $d(w, \alpha)$ to include only those sentential forms that can derive the string x. That is, $d(w, x, \alpha) = \{\beta \mid \alpha \Rightarrow^n w\beta \Rightarrow^* wx\gamma\}$, where n is the length of the word w. We refer to this parameter x as a lookahead as it not consumed by the derivative but is used to select the sentential forms. We denote using $\hat{d}$ the operation d lifted to a set of sentential forms.

We adapt the BRANCH and SPLIT rules shown in Fig. 11 to extended domain of relations and sentential forms. (Other rules in Fig. 11 do not require any extensions.) We now discuss the extended branch rule. For brevity, we present the extended SPLIT rule in Appendix B.

BRANCHEXT.

$$\frac{x = aw \qquad \forall b \in \Sigma.\ \mathcal{C} \cup \{\boldsymbol{\alpha} \ \texttt{op} \ \boldsymbol{\beta}\} \vdash \hat{d}(a, wb, \boldsymbol{\alpha}) \ \texttt{op}_{wb} \ \hat{d}(a, wb, \boldsymbol{\beta})}{\mathcal{C} \vdash \boldsymbol{\alpha} \ \texttt{op}_x \ \boldsymbol{\beta}}$$

Similar to BRANCH rule, the BRANCHEXT rule removes the first character a (of the words considered by the relation) from the sentential forms in $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. However, unlike the BRANCH rule that compares all the sentential forms left behind after deriving the first character, the BRANCHEXT rule looks ahead at the string wb that follows the character a to choose the sentential forms that have to compared. Note that the derivative operation only returns the sentential forms that can derive the lookahead string wb.

Given a lookahead distance k, and two grammars with start symbols S_1 and S_2, we begin the algorithm with the initial set of relations $S_1 \equiv_{w_{k-1}} S_2$, where w_{k-1} is a word of length $\leq k - 1$. The grammars are equivalent if every relation is proven using the inference rules. Our algorithm reduces to the algorithm of [24] when the input grammars are LL(k) GNF grammars. In our implementation, we fix the lookahead distance as 2. Hence, our implementation is complete for LL(2) grammars in GNF.

Query	# Ctr. Exs.	# Samples	RA time
$c1 \equiv c2$	82	227	1ms
$p1 \equiv p2$	417	1053	0.2ms
$js1 \equiv js2$	75	150	0.8ms
$j1 \equiv j2$	133	240	1.5ms
$v1 \equiv v2$	41	52	2.7ms

Figure 14. Counter-examples found in 1min when comparing grammars of the same programming language. The column *RA time* denotes the average time taken for one random access.

5. Experimental Results

We developed a grammar analysis system based on the algorithms presented in this paper, using the Scala programming language, that runs on a Java Virtual Machine. All the experiments discussed in this section were performed on a server with two 3.5 GHz Intel Xeon processors, having 128GB memory, and running Ubuntu Linux operating system.

5.1 Evaluations with Programming Language Grammars

Fig. 13 present details about the benchmarks used in the evaluation. The column *Lang.* shows the list of programming languages chosen for evaluation. For each language we considered at least two grammars which are denoted using the names shown in column *B*. We hand-picked grammars that cover almost all features of the language. For example, in the case of Javascript and VHDL, the grammars we choose implemented the same standard, namely ECMA standard and VHDL-93. In some cases, the grammars even use identical names for many non-terminals. The column *Size* shows the number of non-terminals and productions in each grammar when expressed in standard BNF form. The column *Source* shows the source of the grammars.

Comparing Real-world Grammars. As an initial experiment, we compared the grammars belonging to the same programming language for equivalence. We ran the counter-example detector for 1 minute on each pair of grammars, fixing the maximum length of the word that is enumerated as 50. Fig. 14 show the results of this experiment. The column *Ctr.Exs* shows the number of counter-examples that were found in 1min, and the column *Samples* shows the number of samples generated during counter-example detection.

Interestingly, as shown by the results, the grammars have large number of differences even when they implement the same standard. In many cases, more than 40% of the sampled words are counter-examples. Manually inspecting a few counter-examples revealed that this is mostly due to rules that are more permissible than they ought to be. For instance, the string "`enum ID implements char { ID }`" is generated $j2$ (Antlr v4 Java Grammar), but is not accepted by $j1$ [2]. Surprisingly, even for Javascript that has a very permissible syntax, the grammars had significant differences. This suggests that identifying such counter-examples may help make the grammars, and also their parsers, more optimal. The column *RA time* shows the average time taken for accessing one word (of length between 1 and 50) uniformly at random. The results show that the operation is quite efficient taking only a few milliseconds across all benchmarks.

Discovering Injected Errors. In this experiment, we evaluate the effectiveness of our tool on grammars that have comparatively fewer, and subtle counter-examples. Since grammars obtained from independent sources are likely to have many differences, in order to obtain pairs of grammars that almost recognize the same language, we resort to automatic, controlled tweaking of our benchmarks. We introduce 3 types of errors as explained below. (Let G_m denote the modified grammar and G the original grammar).

- **Type 1 Errors.** We construct G_m by removing one production of G chosen at random. In this case, $L(G_m) \subseteq L(G)$. The inclusion is proper only if the production that is removed is not redundant.
- **Type 2 Errors.** We create G_m by choosing (at random) one production of G having at least two non-terminals, and removing (at random) one non-terminal from the right-hand-side. In this case, neither $L(G_m)$ nor $L(G)$ has to necessarily include the other.
- **Type 3 Errors.** We construct G_m as follows. We randomly choose one production of the grammar, say P, having at least two non-terminals, and also choose one non-terminal of the right-hand-side, say N. We then create a copy (say N') of the non-terminal N that has every production of N except one (determined at random). We replace N by N' in the production P.

Furthermore, we avoid injecting errors that can be discovered through small counter-examples using the following heuristic. We repeat the random error injection process until the modified grammar agrees with the original grammar on the number of parse trees (the function $\#t$ defined in Fig. 5) generating words of length ≤ 15. This ensures that the minimum counter-example, if it exists, is at least 15 tokens long. We relax this bound to 10 and 7 for C and JavaScript grammars, respectively, since the approach failed to produce errors that satisfy larger bounds within reasonable time limits. We also ensured the same error is not introduced more than once. It is to be noted that the counter-example detection algorithm is not aware of the similarities between the input grammars, neither does it attempt to discover such similarities.

For each benchmark b and error type t, we create 10 defective versions of b each containing one error of type t. In total, we create 300 defective grammars. In each case, we query the tool for the equivalence of the erroneous and the original versions, with a time out of 15 minutes. Fig. 15 shows the results of this experiment. We categorize the re-

Language	B	Size	Source
C 2011	*c1*	(228, 444)	Antlr v4
	c2	(75, 269)	`www.quut.com/c/ANSI-C-grammar-y.html`
Pascal	*p1*	(177, 79)	`ftp://ftp.iecc.com/pub/file/pascal-grammar`
	p2	(148, 244)	Antlr v3
JavaScript	*js1*	(128, 336)	`www-archive.mozilla.org/js/language/grammar14.html`
	js2	(124, 278)	Antlr v4
Java 7	*j1*	(256, 530)	`docs.oracle.com/javase/specs/jls/se7/html/jls-18.html`
	j2	(229, 490)	Antlr v4
VHDL	*v1*	(286, 587)	`tams-www.informatik.uni-hamburg.de/vhdl/vhdl.html`
	v2	(475, 945)	Antlr v4

Figure 13. Benchmarks, their sizes as pairs of number of non-terminals and productions, and their sources. Antlr v4 and Antlr v3 denote the repositories: `github.com/antlr/grammars-v4/` and `www.antlr3.org/grammar/`.

sults based on the type of the error that was injected. For now consider only the sub-columns labelled *ours*.

The column *Disproved* shows the number of queries disproved, i.e, the cases where the defective grammar was identified to be not equivalent to the original version. (The maximum possible value for this columns is 10.) The column *Avg.Time/query* shows the average time taken by the tool on queries where it found a counter-example. The column *Avg.Ctr.Size* shows the average length of the counter-example discovered by the tool. The last row of the table summaries the results by showing the total number of queries disproved, average time taken to disprove a query, and the average length of a counter-example.

The results show that the tool was successful in disproving all queries except 3 for *Type 1 Errors*, and 92 out of 100 queries for *Type 2 Errors*, within a few seconds. For *Type 3 Errors*, which are quite subtle, the tool succeeded in finding counter-examples for 73 out of 100 queries taking at most 200s. It timed out after 15 min in the remaining cases. We found that the tool generated millions of words before timing out on a query, across all benchmarks,.

To put these results in perspective, we now present a comparison with a state of the art approach proposed in [4], and used by more recent works such as [7].

Comparisons with cfgAnalyzer. The approach proposed in [4] finds counter-examples for equivalence by constructing a propositional formula that is unsatisfiable iff the input grammars are equivalent upto a bound l, i.e, they accept (or reject) the same set of words of length $\leq l$. The approach uses a SAT solver to obtain a satisfying assignment of the formula, which corresponds to a counter-example for equivalence. We ran their tool cfgAnalyzer on the same set of equivalence queries constructed by automatically injecting errors in our benchmarks as described earlier, with the same time out of 15 minutes. We present the results obtained using their tool in Fig. 15 adjacent to our results, under the sub-column *cfga*. The cfgAnalyzer tool was run in its default mode, wherein the bound l on the length of the words is incremented in unit steps starting from 1 until a counter-example is found. (Other modes of the tool also results in a similar behaviour.)

The results show that our tool out performs cfgAnalyzer by a huge margin on these benchmarks. When aggregated over all benchmarks, our tool disproves 3 times more queries than cfgAnalyzer. Observe that on Java, VHDL and the first Javascript ($js1$) benchmarks, cfgAnalyzer timed out on almost all queries. In general, we found that the performance of cfgAnalyzer degrades with the length of the counter-examples, and with the sizes of the grammars. On the other hand, as highlighted by the results in Fig. 15, our tool discovers large counter-examples within seconds.

5.2 A Tutoring System For Context-Free Grammars

We implemented an online grammar tutoring system available at `grammar.epfl.ch` using our tool. The tutoring system offers three types of exercises: (a) constructing (LL(1) as well as arbitrary) context-free grammars from English descriptions, (b) converting a given context-free grammar to normal forms like CNF and GNF, and (c) writing left most derivations for automatically generated strings belonging to a grammar. The system also supports checking LL(1) property and ambiguity at any point in time when the students are writing their solutions. Moreover, it also has experimental support for generating hints (a feature outside the scope of this paper). Each class of exercise has about 20 problems each (a total of 60 problems) with varying levels of difficulty. It allows a very intuitive syntax for writing grammars, and also supports EBNF form that permits using regular expressions in right-hand-sides of productions.

5.3 Evaluations of the algorithms in the context of a Tutoring System

We used our tutoring system in a 3rd year undergraduate course on computer language processing. We summarize the results of this study in Fig. 16. The column *Queries* shows the total number of *distinct* equivalence queries that the tool was run on. The system refuted 1042 queries by finding counter-examples. (It was configured to enumerate at most 1000 words of length 1 to 11). Among the 353 submissions

Type 1 Errors						
B	**Disproved**		**Avg.Time/query**		**Avg.Ctr.Size**	
	our	cfga	our	cfga	our	cfga
$c1$	10	7	12.7s	396.7s	29.1	10.0
$c2$	10	4	13.8s	325.0s	30.3	10.3
$p1$	10	7	6.8s	127.8s	39.3	15.0
$p2$	10	5	6.8s	329.2s	43.2	16.2
$js1$	10	0	10.9s	-	32.2	-
$js2$	10	9	9.6s	190.9s	31.2	8.1
$j1$	8	0	14.5s	-	41.1	-
$j2$	9	0	14.3s	-	32.1	-
$v1$	10	1	16.9s	810.4s	39.3	15.0
$v2$	10	0	23.4s	-	39.0	-
Type 2 Errors						
$c1$	9	3	13.3s	319.1s	33.8	10.0
$c2$	10	6	9.1s	300.7s	35.6	10.3
$p1$	10	5	6.2s	358.5s	41.3	16.0
$p2$	10	5	7.9s	229.8s	40.0	15.8
$js1$	10	0	12.3s	-	33.8	-
$js2$	7	8	15.3s	52.8s	31.4	7.4
$j1$	7	0	16.3s	-	33.9	-
$j2$	9	0	15.1s	-	38.1	-
$v1$	10	2	16.4s	729.2s	43.7	15.0
$v2$	10	0	58.0s	-	35.8	-
Type 3 Errors						
$c1$	5	4	37.2s	413.6s	17.8	10.3
$c2$	6	5	131.3s	361.2s	30.3	10.0
$p1$	10	3	11.0s	272.5s	34.8	15.0
$p2$	10	5	7.5s	526.8s	34.8	15.8
$js1$	5	0	198.6s	-	28.2	-
$js2$	5	2	34.0s	79.3s	33.2	7.5
$j1$	8	0	25.7s	-	35.4	-
$j2$	6	0	24.8s	-	36.3	-
$v1$	9	0	17.7s	-	38.6	-
$v2$	9	0	54.6s	-	37.3	-
	262	81	28.1s	342.6s	35.0	12.2

Figure 15. Identification of automatically injected errors, using our tool (*our*) and the implementation of [4] (*cfga*).

Queries	Refuted	Proved	Unprvd.	time/query
1395 (100%)	1042 (74.6%)	289 (20.7%)	64 (4.6%)	107ms

Figure 16. Summary of evaluating students' solutions.

for which no counter-example was found, the tool proved the correctness of 289 submissions. For 64 submissions, the tool was neither able to find a counter-example nor was able to prove correctness. In essence, the tool was to able to decide the veracity of 95% of the submissions, and was incomplete on the remaining 5% (in which cases we report that the student's solution is possibly correct). The grammars submitted by students on average had around 3 non-terminals and 6 productions (the maximum was 9 non-terminals and 43 productions). Moreover, at least 370 of the submission were ambiguous. We now present detailed results on the effectiveness of the verification algorithm, which is, to our knowledge, a unique feature of our grammar tutoring system.

Quer.	Proved	Time	LL1	LL2	Amb
353 100%	289 81.9%	410ms	7 2%	56 15.9%	101 28.6%
w/o TESTCASES rule					
353	280	630ms	7	56	94

Figure 17. Evaluation of the verification algorithm on students' solutions.

Evaluation of the Verification Algorithm. Our tutoring system uses the verification algorithm described in section 4 to establish the correctness of the submissions for which no counter-examples are found within the given time limit and sample size. In our evaluation, there are 353 such submissions. The first row of Fig. 17 shows the results of using the algorithm, with all of its features enabled, on the 353 submissions. We used a time out of 10s per query. The system proved almost 82% of the queries taking on average less than half a second per query (as shown by column *Time*). The remaining columns further classify the queries that were verified based on nature of the grammars that are compared.

The column *LL1* shows the number of queries in which the grammars that are compared are LL(1) when normalized to GNF. The algorithm of [15] is applicable only to these cases. The results show that only a meager 2% of the queries belong this category. This is expected since even LL(1) grammars may become non-LL(1) when epsilon productions are eliminated [28] (which is required by the verification algorithm).

The column *LL2* shows the number of queries in which the grammars compared are LL(2) but not LL(1), after conversion to GNF. About 16% of the queries belong this category. This class is interesting because the algorithm of [24] is complete for these cases. (Although the algorithm of [12] is also applicable, it seldom succeeds for these queries since it uses no lookhead). A vast majority (72%) of the queries that are proven involved at least one grammar that is not LL(2). In fact, about 28% of the queries involved ambiguous grammars. (Neither [24] nor [15] is directly applicable to this class of grammars, and [12] is likely to be ineffective.) This indicates that without our extensions a vast majority of the queries may remain unproven. We are not aware of any existing algorithm that can prove equivalence queries involving ambiguous grammars.

We also measure the impact of the TESTCASES inference rule, which uses concrete examples to refine inclusion relations (see section 4). The second row of Fig. 17 show the results of running the verification algorithm without this rule. Overall, the number of queries proven decreases by 9 when this rule is disabled. The impact is mostly on queries involving ambiguous grammars. Moreover, the verifier is slower in this case as shown by the increase in the average time per query. It also timed out on 25 queries after 10s. This is due to the increase in the number and sizes of *relations* created during the verification algorithm. We measured a two fold increase in the average number of sentential forms contained in a relation.

6. Related work

Grammar Analysis Systems. [4] presents a constraint based approach for checking bounded properties of context-free grammars including equivalence and ambiguity. In section 5 we presented a comparison of our counter-example detection algorithm with this work, which shows that our approach does better especially when the counter-examples and grammars are large. [7] presents RACSO an *online judge* for context-free grammars. [7] integrates many strategies for counter-example detection including the approach of [4]. We differ from this work in many aspects. For instance, our enumerators support random access and uniform sampling, scale to large programming language grammars generating millions of strings within seconds. Our system can additionally prove equivalence of grammars. (An empirical comparison with this work was not possible since their interface restricts the sizes of grammars that can be used while creating problems, by requiring that non-terminals have to be upper case characters.)

Decision Procedures for Equivalence. Decision procedures for restricted classes of context-free grammars have been extensively researched: [15], [24], [12], [28], [32], [23], [5], [31]. [15], [5] present decision procedures for *simple grammars*. [24] and [28] show that LL(k) grammars are decidable, and [23] presents a similar result for LL-regular grammars (which properly contain LL(k) grammars). [12] and [32] study equivalence of proper subclasses of deterministic grammars, and [31] shows that the equivalence of arbitrary deterministic grammars is decidable. We are not aware of any practical applications of these algorithms. We extend [24], [12] to a sound but incomplete approach for proving equivalence of arbitrary grammars, and use it to power a grammar tutoring system.

Uniform Sampling of Words. [14] and [19] present algorithms for sampling words from unambiguous grammars uniformly at random (u.a.r). [10] proposes a subexponential time algorithm for sampling words from (possibly ambiguous) grammars, where the probability of generating a word varies from uniform by a factor $1+\epsilon$, $\epsilon \in (0,1)$. [6] presents an algorithm for sampling from a finitely ambiguous grammar in polynomial time.

Our approach has a comparable running time for sampling a word u.a.r under the assumptions of [14] and [19], and is not restricted to uniform random sampling. We are not aware of any implementations of these related works.

Enumeration in the Context of Testing. Grammar-based software testing approaches, such as [27], [22], [30], [21], [13], [18], [20], [9], [11] generate strings belonging to grammars describing the structure of the input, and use them to test softwares like refactoring engines and compilers. In contrast to our objective, there the focus is on generating strings from grammars satisfying complex semantic properties, such as data-structure invariants, type correctness etc., that will expose bugs in the software under test. [27], [21] present a specialized algorithm for generating small number of test cases that result in semantically correct strings useful for detecting bugs. [22], [30], and [11] perform stochastic enumeration of strings from probabilistic grammars (where productions are weighted by probabilities). The probabilities are either manually provided or dynamically adjusted during enumeration. A difference compared to our approach is that they do not sample words by restricting their length (which is hard in the presence of semantic properties), but control the frequency with which the productions are used.

[13], [18] explore various criteria for covering the productions of the grammar that can be beneficial in discovering bugs in softwares. [20] and [9] propose approaches for selectively generating strings that will exercise a path in the program under test, using symbolic execution.

[8] and [17] present generic approaches for constructing enumerators for arbitrary structures, by way of enumerator combinators. They allow combining simple enumerators using a set of *combinators* (such as union and product) to produce more complex enumerators. These approaches ([17] in particular) were an inspiration for our enumeration algorithm, which is specialized for grammars, and provides more functionalities like polynomial time random access, and uniform random sampling.

References

[1] Antlr version 4. `http://www.antlr.org/`.

[2] Java 7 language specification. `http://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html`.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[4] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental SAT solver. In *Automata, Languages and Programming, ICALP*, pages 410–422, 2008. . URL `http://dx.doi.org/10.1007/978-3-540-70583-3_34`.

[5] C. Bastien, J. Czyzowicz, W. Fraczak, and W. Rytter. Prime normal form and equivalence of simple grammars. *Theor. Comput. Sci.*, 363(2):124–134, 2006.

[6] A. Bertoni, M. Goldwurm, and M. Santini. Random generation and approximate counting of ambiguously described combinatorial structures. In *STACS 2000*, pages 567–580. 2000.

[7] C. Creus and G. Godoy. Automatic evaluation of context-free grammars (system description). In *Rewriting and Typed Lambda Calculi RTA-TLCA*, pages 139–

148, 2014. . URL `http://dx.doi.org/10.1007/978-3-319-08918-8_10`.

[8] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Foundations of Software Engineering*, pages 185–194, 2007.

[9] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, pages 206–215, 2008.

[10] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. R. Mahaney. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inf. Comput.*, 134(1):59–74, 1997.

[11] H. Guo and Z. Qiu. Automatic grammar-based test generation. In *Testing Software and Systems ICTSS*, pages 17–32, 2013.

[12] M. A. Harrison, I. M. Havel, and A. Yehudai. On equivalence of grammars through transformation trees. *Theor. Comput. Sci.*, 9:173–205, 1979.

[13] M. Hennessy. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Automated Software Engineering*, pages 104–113, 2005.

[14] T. J. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4): 645–655, 1983.

[15] A. J. Korenjak and J. E. Hopcroft. Simple deterministic languages. In *Symposium on Switching and Automata Theory (Swat)*, pages 36–46, 1966.

[16] D. Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997. ISBN 978-0-387-94907-9.

[17] I. Kuraj and V. Kuncak. Scife: Scala framework for efficient enumeration of data structures with invariants. In *Scala Workshop*, pages 45–49, 2014.

[18] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Testing of Communicating Systems, TestCom*, pages 19–38, 2006.

[19] H. G. Mairson. Generating words in a context-free language uniformly at random. *Inf. Process. Lett.*, 49(2):95–99, 1994.

[20] R. Majumdar and R. Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, pages 553–556, 2007.

[21] B. A. Malloy. An interpretation of purdoms algorithm for automatic generation of test cases. In *International Conference on Computer and Information Science*, pages 3–5, 2001.

[22] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[23] A. Nijholt. The equivalence problem for LL- and LR-regular grammars. pages 149–161, 1982.

[24] T. Olshansky and A. Pnueli. A direct algorithm for checking equivalence of LL*(k)* grammars. *Theor. Comput. Sci.*, 4(3): 321–349, 1977.

[25] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In *Object Oriented Programming Systems Languages & Applications, OOPSLA*, pages 579–598, 2014.

[26] S. Pigeon. Pairing function. `http://mathworld.wolfram.com/PairingFunction.html`.

[27] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, pages 366–375, 1972.

[28] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Symposium on Theory of Computing STOC*, pages 165–180, 1969.

[29] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Programming Language Design and Implementation PLDI*, pages 15–26, 2013.

[30] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Domain-Specific Languages DSL*, pages 1–13, 1999.

[31] G. Snizergues. L(a)=l(b)? decidability results from complete formal systems. *Theoretical Computer Science*, 251(12):1 – 166, 2001.

[32] L. G. Valiant. Decision procedures for families of deterministic pushdown automata. Technical report, University of Warwick, Coventry, UK, 1973.

[33] A. Warth, J. R. Douglass, and T. D. Millstein. Packrat parsers can support left recursion. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM*, pages 103–110, 2008.

A. Cantor's Inverse Pairing Functions for Bounded and Unbounded Domains

The basic inverse pairing function π that maps a natural number in one dimensional space to a number in two dimensional space that is unbounded along both directions [26]. $\pi(z) = (x, y)$, where x and y are defined as follows:

$$\begin{aligned} x &= w - y \\ y &= z - t \\ (t, w) &= simple(z) \end{aligned}$$

where, $simple(z) = (t, w)$ is a function defined as follows:

$$\begin{aligned} t &= \frac{w(w+1)}{2} \\ w &= \left\lfloor \frac{\lfloor \sqrt{8z+1} \rfloor - 1}{2} \right\rfloor \end{aligned}$$

We extend the Cantor's inverse pairing function to two dimensional spaces bounded along one or both directions. π takes three arguments: the number z that has to be mapped, and the bounds of the x and y dimensions x_b and y_b (which could be ∞). x_b is the (inclusive) bound on the x-axis i.e, $\forall x. x \leq x_b$, and y_b is the (exclusive) bound on the y-axis i.e,

SPLITEXT

$$\frac{\begin{array}{c} ||A|| = x \qquad \Psi = \bigcup_{i=1}^{m} \alpha_i\delta_i\beta_i \qquad \forall w \in \Theta_{k-1}(\gamma).\ d(x, w, \alpha_i\delta_i\beta_i) = \boldsymbol{\rho}_i^w \delta_i\beta_i \qquad \forall 0 \le i \le m.\ |\delta_i| > 0, |\beta_i| > 0 \\ \mathcal{C}' = \mathcal{C} \cup \{\{A\gamma\}\ \mathtt{op}_z\ \Psi\} \\ \forall w \in \Theta_{k-1}(\gamma).\ \mathcal{C}' \vdash \{\gamma\}\ \mathtt{op}_w\ \hat{d}(x, w, \Psi) \qquad \forall 0 \le i \le m.\ \mathcal{C}' \vdash \left(\bigcup_{w \in \Theta_{k-1}(\gamma)} A[\![w, \boldsymbol{\rho}_i^w \delta_i]\!]\right)\ \mathtt{op}_z\ \{\alpha_i\delta_i\} \end{array}}{\mathcal{C} \vdash \{A\gamma\}\ \mathtt{op}_z\ \Psi}$$

Figure 18. Extended Split Rule.

$\forall y. y < y_b.\ \pi(z, x_b, y_b) = (x, y)$, where

$$\begin{aligned} x &= w - y \\ y &= z - t \\ (t, w) &= \begin{cases} bskip(z) & \text{if } z \ge z_b \\ xskip(z) & \text{if } z_x \le z < z_b \\ yskip(z) & \text{if } z_y \le z < z_b \\ simple(z) & \text{Otherwise} \end{cases} \end{aligned}$$

where, z_x, z_y and z_b are indices at which the bounds along the x or y or both directions are crossed, respectively. The values are defined as follows:

$$\begin{aligned} z_y &= \frac{y_b(y_b + 1)}{2} \\ z_x &= \frac{(x_b + 1)(x_b + 2)}{2} \\ z_b &= \begin{cases} y_b(x_b - y_b + 1) + z_y & \text{if } x_b > y_b - 1 \\ (x_b + 1)(y_b - x_b - 1) + z_x & \text{if } y_b - 1 > x_b \\ z_y & \text{Otherwise} \end{cases} \end{aligned}$$

Definition of $xskip(z)$. Let $xskip(z) = (t, w)$, where t and w are defined as follows:

$$\begin{aligned} t &= \frac{2wx_b - x_b^2 + x_b}{2} \\ w &= \left\lfloor \frac{2z + x_b^2 + x_b}{2(x_b + 1)} \right\rfloor \end{aligned}$$

Definition of $yskip(z)$. Let $yskip(z) = (t, w)$, where t and w are defined as follows:

$$\begin{aligned} t &= \frac{2wy_b - y_b^2 + y_b}{2} \\ w &= \left\lfloor \frac{2z + y_b^2 - y_b}{2y_b} \right\rfloor \end{aligned}$$

Definition of $bskip(z)$. Let $bskip(z) = (t, w)$, where t and w are defined as follows:

$$\begin{aligned} t &= \frac{(2w_b - 1)w - w^2 - s_b + w_b}{2} \\ w &= \left\lfloor \frac{r - \left\lceil \sqrt{r^2 - 8z - 4s_b + 4y_b - 4x_b} \right\rceil}{2} \right\rfloor \\ r &= 2w_b + 1 \\ w_b &= x_b + y_b \\ s_b &= x_b^2 + y_b^2 \end{aligned}$$

Note that all the operations make use of only integer division and integer square root i.e, they compute floor of division, floor or ceil of square roots, which can be efficiently implemented. Moreover, many multipliers and divisors are powers of 2, and hence can be computed by bit shift operations.

B. Extended Split Rule

Fig. 18 shows the extended split rule. We require that the lookahead distance k is greater than or equal to 2. Define $\Theta_{k-1}(\gamma)$ as the set of all words of length $\le k - 1$ derivable from a sentential form γ. That is, $\Theta_{k-1}(\gamma) = \{w \mid w \in \bigcup_{j=1}^{k-1} \Sigma^j, \gamma \Rightarrow^* w\}$. If the grammar is LL(k) then we use the split rule defined in [24].