Analyzing Ambiguity of Context-Free Grammars

Claus Brabrand[a], Robert Giegerich[b], Anders Møller[1c]

[a] *IT University of Copenhagen*
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark
[b] *Center of Biotechnology, Bielefeld University*
Postfach 100131, 33501 Bielefeld, Germany
[c] *Department of Computer Science, Aarhus University*
Aabogade 34, 8200 Aarhus N, Denmark

Abstract

It has been known since 1962 that the ambiguity problem for context-free grammars is undecidable. Ambiguity in context-free grammars is a recurring problem in language design and parser generation, as well as in applications where grammars are used as models of real-world physical structures.

We observe that there is a simple linguistic characterization of the grammar ambiguity problem, and we show how to exploit this by presenting an ambiguity analysis framework based on conservative language approximations. As a concrete example, we propose a technique based on local regular approximations and grammar unfoldings. We evaluate the analysis using grammars that occur in RNA analysis in bioinformatics, and we demonstrate that it is sufficiently precise and efficient to be practically useful.

1. Introduction

When using context-free grammars to describe formal languages, one has to be aware of potential ambiguity in the grammars, that is, the situation where a string may be parsed in multiple ways, leading to different parse trees. We propose a technique for detecting ambiguities in a given grammar. As the problem is in general undecidable, which was shown by Cantor [6], Floyd [14], and Chomsky and Schützenberger [8], we resort to conservative approximation. This means that our analysis for some grammars is able to guarantee that they are unambiguous, whereas for others it cannot give certain answers.

In bioinformatics, context-free grammars in various guises have important applications, for example in sequence comparison, motif search, and RNA secondary structure analysis [12, 18]. Recently, ambiguity has gained attention in this field, as several important algorithms (such as the Viterbi algorithm on stochastic CFGs) have been shown to deliver incorrect results in the presence of ambiguity [16, 11]. The ambiguity problem arises in biosequence analysis from

[1] Corresponding author. Email address: amoeller@cs.au.dk.

the necessity to check a static property of the dynamic programming algorithms employed – the question whether or not an element of the search space may be evaluated more than once. If so, probabilistic scoring schemes yield incorrect results, and enumeration of near-optimal solutions drowns in redundancy. It may seem surprising that the static analysis of this program property can be approached as a question of language ambiguity on the formal language level. We will explain this situation in some detail in Section 8.

Before we start presenting our method, we state two requirements on a practical ambiguity checker that result from the biosequence analysis domain and must be kept in mind in what follows: First, the grammars to be checked are actually abstractions from richer programming concepts. They may look strange from a formal language design point of view – for example, they may contain "redundant" nonterminal symbols generating the same language. However, different nonterminals model different physical structures with different semantics that are essential for subsequent algorithmic processing. Hence, the grammar must be checked as is and cannot be simplified by, for instance, coalescing such nonterminal symbols. Of course, we may (and will) apply ambiguity-preserving transformations to the grammar. Second, the domain experts are typically molecular biologists with little programming expertise and no training in formal language theory. Hence, when ambiguity is discovered, it must be reported in a way that is meaningful to this category of users.

Besides the applications to biosequence analysis, our motivation behind the work we present here has been analyzing reversibility of transformations between XML and non-XML data, which can be reduced to the grammar ambiguity problem [4]. That work involves scannerless parsing, that is, where lexical descriptions are not separated from the grammars.

Related Work

Our approach is related to building an LR(k) parse table for the given grammar and checking for conflicts. The LR(k) condition has since its discovery by Knuth in 1965 been known as a powerful test for unambiguity [21]. An example of an even larger class of unambiguous grammars is LR-Regular [10]. Nevertheless, not even LR-Regular is sufficient for a considerable class of grammars involving palindromic structures, which often occur in bioinformatics. Another crucial limitation with ambiguity detection tools based on LR(k) or related techniques is the poor quality of the error messages they produce. Any user of Yacc or a similar tool will recognize the difficulty in finding the true cause of a reported conflict.

An increasing number of parser generators, for example, Bison [33], SDF [35], and Elkhound [24], support general context-free grammars rather than unambiguous subclasses, such as LL(k), LR(k), or LALR(k). Such tools usually handle ambiguities by dynamically (that is, during parsing) disambiguating or merging the resulting parse trees [34, 5]. In contrast, our approach is to *statically* analyze the grammars for potential ambiguities. Also, we aim for a *conservative* algorithm, unlike many existing ambiguity detection techniques, for example those by Gorn [19] and Cheung and Uzgalis [7].

In the approach by Schmitz [31], a grammar G is turned into a potentially infinite structure called a position graph, Γ_G, which is then searched for multiple derivations of the same sentential form. By factoring Γ_G with a finitary equivalence relation, the graph search problem results in a terminating analysis. As examples, the LR(k) and LR-Regular conditions may both be adapted to equivalence relations in this framework. The approach by Kuich [23] constructs systems of finite pushdown automata from grammars. If these automata (or ambiguity-preserving transformations thereof) can be shown to be quasi-deterministic, then the grammar is guaranteed to be unambiguous. This method is closely related to the LR(0) variant of Schmitz. As our approach, the analyses of Schmitz and Kuich are sound but incomplete. Their analyses work by searching finite structures for multiple derivations. Our approach is conceptually different in that it transforms the structural problem of ambiguity to a finite collection of linguistic problems. Our experiments (Section 9) demonstrate the incomparable expressiveness of our approach compared to that of Schmitz.

Contributions

Despite decades of work on parsing techniques, which in many cases involve the problem of grammar ambiguity, we have been unable to find tools that are applicable to grammars in the areas mentioned above. This paper contributes with the following results:

- We observe that there is a simple linguistic characterization of grammar ambiguity. This allows us to shift from reasoning about grammar derivations to reasoning about purely linguistic properties, such as, language inclusion and approximations. This results in a framework, called *ACLA* (Ambiguity Checking with Language Approximations).

- We show how Mohri and Nederhof's regular approximation technique for context-free grammars [25] can be adapted in a local manner using the ACLA framework to detect many common sources of ambiguity, including ones that involve palindromic structures. Also, a simple grammar unfolding transformation can be used to improve the precision of the approximation. The flexibility of the framework is additionally substantiated by presenting other approximation techniques that can be combined with the regular approximation to improve analysis performance.

- We demonstrate that ACLA can handle real-world grammars of varying complexity taken from the bioinformatics literature on RNA analysis, acquitting the unambiguous grammars and pinpointing the sources of ambiguity – with shortest possible examples as witnesses – in grammars that are in fact ambiguous. In addition, we investigate the analysis precision and performance on a range of other grammars found in the literature. The implementation is available from our web site.

We here work with plain, context-free grammars. Modifying our approach to work with parsing techniques that involve interoperability with a lexical analyzer, precedence/associativity declarations, or other disambiguation mechanisms is left to future work. (Some progress in this direction is discussed in the context of our implementation in Section 7.)

Overview

We begin in Section 2 by giving a characterization of grammar ambiguity that allows us to reason about the *language* of the nonterminals in the grammar rather than the *structure* of the grammar. In particular, we reformulate the ambiguity problem in terms of language intersection and overlap operations. Based on this characterization, we then in Section 3 formulate a general framework for conservatively approximating the ambiguity problem. In Section 4 we show how regular approximations can be used to obtain a particular decidable approximation, Section 5 shows how precision can be improved using grammar unfolding transformations, and examples of other approximation strategies are presented in Section 6. Our implementation is described in Section 7. Section 8 discusses applications in the area of biosequence analysis where context-free grammars are used to describe RNA structures, and Section 9 summarizes a number of experiments that test the precision and performance of the analysis. The appendices contain proofs of the propositions.

2. A Characterization of Grammar Ambiguity

We begin by briefly recapitulating the basic terminology about context-free grammars.

Definition 1 (Context-free grammar and ambiguity). *A* context-free grammar (CFG) *G is defined by $G = (\mathcal{N}, \Sigma, s, \pi)$ where $\mathcal{N}$ is a finite set of* nonterminals*, Σ is a finite set of* alphabet symbols *(or* terminals*), $s \in \mathcal{N}$ is the* start nonterminal*, and $\pi : \mathcal{N} \to \mathcal{P}(E^*)$ is the finite* production function *where $E = \Sigma \cup \mathcal{N}$. We write $\alpha n \omega \Rightarrow \alpha \theta \omega$ when $\theta \in \pi(n)$ and $\alpha, \omega \in E^*$, and $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$. We assume that every nonterminal $n \in \mathcal{N}$ is reachable from s and derives some string: $\exists x, y, z \in \Sigma^* : s \Rightarrow^* xnz \Rightarrow^* xyz$. The* language *of a sentential form $\alpha \in E^*$ is $\mathcal{L}_G(\alpha) = \{x \in \Sigma^* \mid \alpha \Rightarrow^* x\}$, and the language of G is $\mathcal{L}(G) = \mathcal{L}_G(s)$.*

Assume that $x \in \mathcal{L}(G)$, that is, $s = \phi_0 \Rightarrow \phi_1 \Rightarrow \cdots \Rightarrow \phi_n = x$. Such a derivation sequence gives rise to a derivation tree*, which is a finite tree where each node is labeled with a symbol from E: the root is labeled s, and the sequence of labels of children of a node with label e are in $\pi(e)$. G is* ambiguous *if there exists a string x in $\mathcal{L}(G)$ with multiple derivation trees, and we then say that x is* ambiguous relative to *G.*

We now introduce the properties *vertical* and *horizontal unambiguity* and show that they together characterize grammar unambiguity.

Definition 2 (Vertical and horizontal unambiguity). *Given a grammar G, two sentential forms $\alpha, \alpha' \in E^*$ are* vertically unambiguous*, written $\Vdash_G \alpha; \alpha'$, iff*

$$\mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\alpha') = \emptyset$$

A grammar G is vertically unambiguous*, written $\Vdash G$, iff*

$$\forall n \in \mathcal{N}, \alpha, \alpha' \in \pi(n) \text{ where } \alpha \neq \alpha' : \Vdash_G \alpha; \alpha'$$

Two sentential forms $\alpha, \alpha' \in E^*$ are horizontally unambiguous*, written $\models_G \alpha; \alpha'$, iff*

$$\mathcal{L}_G(\alpha) \between \mathcal{L}_G(\alpha') = \emptyset$$

where $\between$ is the language overlap *operator defined by*

$$X \between Y = \{\, xay \mid x, y \in \Sigma^* \ \wedge\ a \in \Sigma^+ \ \wedge\ x, xa \in X \ \wedge\ y, ay \in Y \}$$

A grammar G is horizontally unambiguous*, written $\models G$, iff*

$$\forall n \in \mathcal{N}, \alpha\alpha' \in \pi(n) \text{ where } \alpha, \alpha' \in E^* : \models_G \alpha; \alpha'$$

We write $\Vdash\!\models G$ when both $\Vdash G$ and $\models G$ are satisfied.

Intuitively, vertical unambiguity means that, during parsing of a string, there is never a choice between two different productions of a nonterminal. The overlap $X \between Y$ is the set of strings in XY that can be split non-uniquely in an X part and a Y part. For example, if $X = \{\mathtt{x}, \mathtt{xa}\}$ and $Y = \{\mathtt{y}, \mathtt{ay}\}$ then $X \between Y = \{\mathtt{xay}\}$. Horizontal unambiguity then means that, when parsing a string according to a production, there is never any choice of how to split the string into substrings corresponding to the entities in the production.

Proposition 3 (Characterization of ambiguity).

$$\Vdash\!\models G \quad \Leftrightarrow \quad G \text{ is unambiguous}$$

Proof. Any ambiguity must result from a choice between two productions of some nonterminal or from a choice of how to split a string according to a single production. A detailed proof is given in Appendix A.

This proposition essentially means that we have transformed the problem of context-free grammar ambiguity from a *grammatical* property to a *linguistic* property dealing solely with the languages of the nonterminals in the grammar rather than with derivation trees. As we shall see in the next section, this characterization can be exploited to obtain a good conservative approximation for the problem without violating the two requirements described in Section 1.

Note that this linguistic characterization of grammar ambiguity should not be confused with the notion of *inherently ambiguous languages* [20]. (A language is inherently ambiguous if all its grammars are ambiguous.)

We now give examples of vertical and horizontal ambiguities.

Example 4 (Vertical ambiguity).

```
Z   :   A "y"
        ↕
    |   "x" B
A   :   "x" "a"
B   :   "a" "y"
```

The string `xay` can be parsed in two ways by choosing either the first or the second production of `Z`, that is, `xay` $\in \mathcal{L}_G($`A "y"`$) \cap \mathcal{L}_G($`"x" B`$)$. The name *vertical* ambiguity comes from the fact that productions are often written on separate lines as in this example.

Example 5 (Horizontal ambiguity).

```
Z   :   "x" A ↔ B
A   :   "a"
    |   ε
B   :   "a" "y"
    |   "y"
```

Also here, the string `xay` can be parsed in two ways, by parsing the `a` either in `"x" A` (using the first production of `A` and the second of `B`) or in `B` (using the second production of `A` and the first of `B`). More formally, `xay` $\in \mathcal{L}_G($`"x" A`$) \between \mathcal{L}_G($`B`$)$. Here, the ambiguity is at a split-point between entities on the right-hand side of a particular production, hence the name *horizontal* ambiguity.

3. A Framework for Conservative Approximation

The characterization of ambiguity presented above can be used as a foundation for a framework, *ACLA* (Ambiguity Checking with Language Approximations), for obtaining decidable, conservative approximations of the ambiguity problem. When the analysis says "`the grammar is unambiguous!`", we know that this is indeed the case. The key to this technique is that the linguistic characterization allows us to reason about languages of nonterminals rather than derivation trees.

Definition 6 (Grammar approximation). *A* grammar approximation *relative to a CFG G is a function $\mathcal{A}_G : E^* \to \mathcal{P}(\Sigma^*)$ where $\mathcal{L}_G(\alpha) \subseteq \mathcal{A}_G(\alpha)$ for every $\alpha \in E^*$.*

In other words, $\mathcal{A}_G$ provides for every sentential form α an upper approximation of the language of α.

Definition 7 (Approximated vertical and horizontal unambiguity). *Given a grammar approximation $\mathcal{A}_G$, two sentential forms $\alpha, \alpha' \in E^*$ are* vertically unambiguous relative to $\mathcal{A}_G$, *written $\Vdash_{\mathcal{A}_G} \alpha; \alpha'$, iff*

$$\mathcal{A}_G(\alpha) \cap \mathcal{A}_G(\alpha') = \emptyset$$

A grammar G is vertically unambiguous relative to $\mathcal{A}_G$*, written* $\Vdash_{\mathcal{A}_G} G$*, iff*

$$\forall n \in \mathcal{N}, \alpha, \alpha' \in \pi(n) \text{ where } \alpha \neq \alpha' : \Vdash_{\mathcal{A}_G} \alpha; \alpha'$$

Two sentential forms $\alpha, \alpha' \in E^*$ are horizontally unambiguous relative to $\mathcal{A}_G$*, written* $\models_{\mathcal{A}_G} \alpha; \alpha'$*, iff*

$$\mathcal{A}_G(\alpha) \between \mathcal{A}_G(\alpha') = \emptyset$$

A grammar G is horizontally unambiguous relative to $\mathcal{A}_G$*, written* $\models_{\mathcal{A}_G} G$*, iff*

$$\forall n \in \mathcal{N}, \alpha\alpha' \in \pi(n) \text{ where } \alpha, \alpha' \in E^* : \models_{\mathcal{A}_G} \alpha; \alpha'$$

We write $\Vdash\!\models_{\mathcal{A}_G} G$ when both $\Vdash_{\mathcal{A}_G} G$ and $\models_{\mathcal{A}_G} G$ are satisfied.

Note that Definition 7 gives an algorithm for deciding $\Vdash\!\models_{\mathcal{A}_G} G$, assuming that algorithms for deciding $\Vdash_{\mathcal{A}_G} \alpha; \alpha'$ and $\models_{\mathcal{A}_G} \alpha; \alpha'$ are provided. The next proposition opens the floor for a variety of approaches to sound ambiguity checking that differ in the way they construct approximations.

Proposition 8 (Approximation soundness).

$$\Vdash\!\models_{\mathcal{A}_G} G \quad \Rightarrow \quad G \text{ is unambiguous}$$

Proof. The result follows straightforwardly from Definitions 2, 6, and 7 and Proposition 3. For details, see Appendix B.

Definition 9 (Approximation strategy). *An* approximation strategy $\mathcal{A}$ *is a function that returns a grammar approximation $\mathcal{A}_G$ given a CFG G. We say that $\mathcal{A}$ is* decidable *if $\Vdash\!\models_{\mathcal{A}_G} G$ is decidable with the grammar G as input.*

As an example of a decidable but not very useful approximation strategy, the one which returns the constant Σ^* approximation corresponds to the trivial analysis that reports that every grammar may be (vertically and horizontally) ambiguous at all possible locations. In the other end of the spectrum, the approximation strategy which for every grammar G returns $\mathcal{L}_G(\alpha)$ for each α has full precision but is undecidable (since it involves checking language disjointness for context-free grammars).

Note that two different approximations, $\mathcal{A}_G$ and $\mathcal{A}'_G$, may be combined: the function $\mathcal{A}''_G$ defined by $\mathcal{A}''_G(\alpha) = \mathcal{A}_G(\alpha) \cap \mathcal{A}'_G(\alpha)$ is a grammar approximation that subsumes both $\mathcal{A}_G$ and $\mathcal{A}'_G$. Such a pointwise combination may lead to higher precision than running the two analyses independently as one of the approximations might be good in one part of the grammar, and the other in a different part. Nonetheless, approximations can also be combined in another way that is more useful in practice:

Proposition 10 (Combining approximations). *Given a collection of grammar approximations $\mathcal{A}^1_G, \ldots, \mathcal{A}^k_G$, if*

$$\forall n \in \mathcal{N}, \alpha, \alpha' \in \pi(n) \text{ where } \alpha \neq \alpha' : \exists j \in \{1, \ldots, k\} : \Vdash_{\mathcal{A}^j_G} \alpha; \alpha'$$

then $\Vdash G$.

If

$$\forall n \in \mathcal{N}, \alpha\alpha' \in \pi(n) \text{ where } \alpha, \alpha' \in E^* : \exists j \in \{1, \ldots, k\} : \models_{\mathcal{A}^j_G} \alpha; \alpha'$$

then $\models G$.

Proof. This is a trivial modification of the proof of Proposition 8. Intuitively, to argue that a given grammar is unambiguous, it suffices to check vertical and horizontal unambiguity of certain pairs of sentential forms, and these checks may be performed using different strategies.

Recall from Section 1 that we stated as a requirement that for the grammars of interest, we do not allow grammars to be simplified, even if they contain nonterminals that have the same language. The use of approximations and transformations that we now present does not violate that requirement: The ACLA framework reduces the ambiguity problem for the original grammar to a finite collection of linguistic vertical and horizontal ambiguity problems – and each of these can safely be attacked with techniques that involve approximations and transformations.

4. Regular Approximation

One approach for obtaining decidability is to consider *regular* approximations, that is, ones where $\mathcal{A}_G(\alpha)$ is a regular language for each α: the family of regular languages is closed under both intersection and overlap, and emptiness on regular languages is decidable. Also, shortest examples can easily be extracted from nonempty regular languages. As a concrete approximation strategy we propose using Mohri and Nederhof's algorithm for constructing regular approximations of context-free grammars [25].

We will not repeat their algorithm in detail, but some important properties are worth mentioning. Given a CFG G, the approximation results in another CFG G' which is strongly regular (and hence its language is regular), $\mathcal{L}(G) \subseteq \mathcal{L}(G')$, and G' is at most twice the size of G. Whenever $n \Rightarrow^* \alpha n \omega$ and $n \Rightarrow^* \theta$ in G for some $\alpha, \omega, \theta \in E^*$ and $n \in \mathcal{N}$, the grammar G' has the property that $n \Rightarrow^* \alpha^m \theta \omega^k$ for any m, k. Intuitively, G' keeps track of the order that alphabet symbols may appear in, but it loses track of the fact that α and ω must appear in balance.

Definition 11 (Mohri-Nederhof approximation strategy). *Let MN be the approximation strategy that given a CFG* $G = (\mathcal{N}, \Sigma, s, \pi)$ *returns the grammar approximation* MN_G *defined by* $MN_G(\alpha) = \mathcal{L}(G_\alpha)$ *where* G_α *is the Mohri-Nederhof approximation of the grammar*

$$(\mathcal{N} \cup \{s_\alpha\}, \Sigma, s_\alpha, \pi[s_\alpha \mapsto \{\alpha\}])$$

for some $s_\alpha \notin \mathcal{N}$. *(The notation* $\pi[s_\alpha \mapsto \{\alpha\}]$ *denotes the function* π *extended to map* s_α *to* $\{\alpha\}$.*)*

In other words, whenever we need to compute $\mathcal{A}_G(\alpha)$ for some $\alpha \in E^*$, we apply Mohri and Nederhof's approximation algorithm to the grammar G modified to derive α as the first step.

Example 12 (Palindromes). A classical example of an unambiguous grammar that is not LR(k) (nor LR-Regular) is the following, called Pal, whose language consists of all palindromes over the alphabet $\{\mathtt{a}, \mathtt{b}\}$:

```
P  :  "a" P "a"  |  "b" P "b"  |  "a"  |  "b"  |  ε
```

Running our analysis implementation on this grammar immediately gives the result "`the grammar is unambiguous!`". It computes MN_{Pal} for each of the five right-hand sides of productions and all their prefixes and suffixes and then performs the checks described in Definition 7. As an example, MN_{Pal}(`"a" P "a"`) is the regular language $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{a}$, and MN_{Pal}(`"b" P "b"`) is $\mathtt{b}(\mathtt{a}+\mathtt{b})^*\mathtt{b}$. Since these two languages are disjoint, there is no vertical ambiguity between the first two productions.

A variant of the grammar above is the following language, AntiPal, which our analysis also verifies to be unambiguous:

```
R  :  "a" R "b"  |  "b" R "a"  |  "a" |  "b"  |  ε
```

As we shall see in Section 8, this grammar is closely related to grammars occurring naturally in biosequence analysis.

Example 13 (Ambiguous expressions). To demonstrate the capabilities of producing useful warning messages, let us run the analysis on the following tiny ambiguous grammar representing simple arithmetical expressions:

```
Exp[plus]   :   Exp "+" Exp
   [mult]   |   Exp "*" Exp
   [var]    |   "x"
```

(Notice that we allow productions to be labeled.) The analysis output is:

```
*** vertical ambiguity: Exp[plus] <--> Exp[mult]
    ambiguous string: "x*x+x"
*** horizontal ambiguity: Exp[plus]: Exp <--> "+" Exp
    ambiguous string: "x+x+x"
    matched as "x" <--> "+x+x" or "x+x" <--> "+x"
*** horizontal ambiguity: Exp[plus]: Exp "+" <--> Exp
    ambiguous string: "x+x+x"
    matched as "x+" <--> "x+x" or "x+x+" <--> "x"
*** horizontal ambiguity: Exp[mult]: Exp <--> "*" Exp
    ambiguous string: "x*x*x"
    matched as "x" <--> "*x*x" or "x*x" <--> "*x"
*** horizontal ambiguity: Exp[mult]: Exp "*" <--> Exp
    ambiguous string: "x*x*x"
    matched as "x*" <--> "x*x" or "x*x*" <--> "x"
the grammar is ambiguous!
```

Each source of ambiguity is clearly identified, even with example strings that have been verified to be non-spurious (of course, it is easy to check with a CFG parser whether a concrete string is ambiguous or not). Obviously, these messages are more useful to a non-expert than, for example, the shift/reduce conflicts and reduce/reduce conflicts being reported by LR(k) parser generators.

Complexity

The theoretical worst-case time complexity of the ACLA framework of course depends on the choice of approximation strategies. For a given approximation strategy, let $t_V(G)$ denote the maximum time for performing a single vertical unambiguity check, $\Vdash_{\mathcal{A}_G} \alpha;\alpha'$ for any $\alpha, \alpha' \in \pi(n)$ where $n \in \mathcal{N}$. Similarly, $t_H(G)$ bounds the time for performing a single horizontal unambiguity check, $\models_{\mathcal{A}_G} \alpha;\alpha'$ for any $\alpha\alpha' \in \pi(n)$ where $n \in \mathcal{N}$. ACLA decides $\Vdash\!\models_{\mathcal{A}_G} G$ by performing

$$\sum_{n\in\mathcal{N}} |\pi(n)|\big(|\pi(n)|-1\big)/2$$

vertical unambiguity checks and

$$\sum_{n\in\mathcal{N}} \sum_{\alpha\in\pi(n)\setminus\{\epsilon\}} \big(|\alpha|-1\big)$$

horizontal unambiguity checks. Thus, if we let $|G|$ denote the size of the grammar G (for example, measured as the number of symbols required to write down G), the time complexity of ACLA is

$$\mathcal{O}\big(t_V(G)|G|^2 + t_H(G)|G|\big)$$

For our implementation of the *MN* approximation strategy, both t_V and t_H are exponential in $|G|$: The grammar transformation itself at most doubles the size of the grammar, however the subsequent conversion from strongly regular grammars to finite-state automata may theoretically lead to an exponential blow-up. In our experiments, however, this does not appear to be problematic (see Section 9).

5. Improving Precision with Grammar Unfolding

Precision of the regular approximation can be improved by a simple grammar unfolding technique. A related approach has been used by Nederhof [28] but not in the context of analyzing ambiguity.

Example 14 (Unambiguous expressions). Although the grammar in Example 13 is ambiguous, its language is not inherently ambiguous. By introducing parentheses and fixing precedence and associativity of the operators, '+' and '*', an unambiguous grammar, called Exp, can easily be constructed:

```
Exp[plus]      :   Exp "+" Term
   [term]      |   Term
Term[mult]     :   Term "*" Factor
    [factor]   |   Factor
Factor[var]    :   "x"
      [par]    |   "(" Exp ")"
```

While even the LR(k) technique is perfectly capable of acquitting this grammar as unambiguous, our analysis, as presented so far, reports the following spurious errors:

```
*** potential vertical ambiguity: Exp[plus] <--> Exp[term]
*** potential horizontal ambiguity at Exp[plus]: Exp <--> "+" Term
*** potential horizontal ambiguity at Exp[plus]: Exp "+" <--> Term
*** potential horizontal ambiguity at Term[mult]: Term <--> "*" Factor
*** potential horizontal ambiguity at Term[mult]: Term "*" <--> Factor
```

Note that, in contrast to the output being generated in Example 13, there are no example strings and the word "`potential`" has been added. For example, $MN_{\mathrm{Exp}}(\texttt{Exp "+" Term}) \cap MN_{\mathrm{Exp}}(\texttt{Term})$ is nonempty, but any example string in this intersection turns out to be unambiguous (our implementation tests just a single string in the intersection). This means that the analysis, as presented in the previous sections, cannot say with certainty that this grammar is ambiguous, nor that it is unambiguous.

By investigating the potential vertical ambiguity reported above, we see that a string $x = \alpha\texttt{+}\omega \in \mathcal{L}_{\mathrm{Exp}}(\texttt{Exp})$ must have unmatched parentheses in α if and only if x can be derived from `Exp[term]` and not from `Exp[plus]`. Hence, if we can distinguish between operators that appear within parentheses from ones that appear outside, we can eliminate this kind of spurious error.

Fortunately, the ambiguity characterization provided by Proposition 3 allows us to employ language-preserving transformations and conservative approximations on the grammar without risking violations of the requirements set up in Section 1. A simple language-preserving grammar transformation is *unfolding* recursive nonterminals, and, as explained in the following, this happens to be enough to eliminate all five spurious errors in the example above.

More generally, for *balanced grammars* [22, 3] where parentheses are balanced in each production, we can regain some precision that is lost by the Mohri-Nederhof approximation by *unfolding* the grammar to distinguish between inside and outside parentheses.

Definition 15 (Balanced grammar[1]). *A grammar $G = (\mathcal{N}, T, s, \pi)$ is* balanced *if (1) the terminal alphabet has a decomposition $T = \Sigma \cup \Gamma \cup \widehat{\Gamma}$ where Γ is a set of left parentheses and $\widehat{\Gamma}$ a complementary set of right parentheses, and (2) all productions in $\pi(n)$ where $n \in \mathcal{N}$ are of the form α or $\alpha\gamma\phi\widehat{\gamma}\omega$, where $\alpha, \phi, \omega \in (\Sigma \cup \mathcal{N})^*, \gamma \in \Gamma, \widehat{\gamma} \in \widehat{\Gamma}$ (that is, γ is the complementary parenthesis of $\widehat{\gamma}$).*

Definition 16 (Unfolded balanced grammar). *Unfolding a balanced grammar $G = (\mathcal{N}, \Sigma \cup \Gamma \cup \widehat{\Gamma}, s, \pi)$ produces the grammar $\widetilde{G} = (\mathcal{N} \cup \underline{\mathcal{N}}, \Sigma \cup \Gamma \cup \widehat{\Gamma} \cup \underline{\Sigma} \cup \underline{\Gamma} \cup \underline{\widehat{\Gamma}}, s, \widetilde{\pi})$ where $\underline{\mathcal{N}}$, $\underline{\Sigma}$, $\underline{\Gamma}$, and $\underline{\widehat{\Gamma}}$ are copies of $\mathcal{N}$, Σ, Γ, and $\widehat{\Gamma}$, respectively, with all symbols underlined, and*

$$\widetilde{\pi}(n) = \begin{cases} \alpha & \text{if } \pi(n) = \alpha \\ \alpha\gamma\underline{\phi}\widehat{\gamma}\omega & \text{if } \pi(n) = \alpha\gamma\phi\widehat{\gamma}\omega \text{ where } \gamma \in \Gamma \text{ and } \widehat{\gamma} \in \widehat{\Gamma} \text{ match} \end{cases}$$

[1] This is a syntactic variation of the definition by Berstel and Boasson [3]. It is also a generalization of the definition by Knuth [22], allowing multiple pairs of parenthesis symbols.

$$\widetilde{\pi}(\underline{n}) = \begin{cases} \underline{\alpha} & \text{if } \pi(n) = \alpha \\ \underline{\alpha\gamma\phi\widehat{\gamma}\omega} & \text{if } \pi(n) = \alpha\gamma\phi\widehat{\gamma}\omega \text{ where } \gamma \in \Gamma \text{ and } \widehat{\gamma} \in \widehat{\Gamma} \text{ match} \end{cases}$$

where the notation $\underline{\theta}$ means θ with all symbols underlined.

Clearly, there is a bijection between the derivations of $\mathcal{L}(G)$ and $\mathcal{L}(\widetilde{G})$ by adding/removing underscores: the languages are identical, except that in all strings in $\mathcal{L}(\widetilde{G})$ a symbol is underlined if and only if it is enclosed by parentheses. The same applies to the derivation trees. Thus, when checking vertical unambiguity of two productions in G it is sound to check the corresponding productions in $\widetilde{G}$ instead, and similarly for horizontal unambiguity. As the following example shows, this may improve precision of the analysis.

Example 17 (Unambiguous expressions, unfolded). The grammar Exp from Example 14 is balanced with $\Gamma = \{$`(`$\}$ and $\widehat{\Gamma} = \{$`)`$\}$. Unfolding yields this grammar, named $\widetilde{\text{Exp}}$:

```
Exp    : Exp "+" Term                Exp    : Exp "+" Term
       | Term                               | Term
Term   : Term "*" Factor             Term   : Term "*" Factor
       | Factor                             | Factor
Factor : "x"                         Factor : "x"
       | "(" Exp ")"                        | "(" Exp ")"
```

A string parsed in the original grammar is parsed in exactly the same way in the new unfolded grammar, except that everything enclosed by parentheses is now underlined. For example, the string `x+(x+(x)+x)+x` from the original grammar corresponds to `x+(x+(x)+x)+x` with the resulting grammar. Now, every string in $MN_{\widetilde{\text{Exp}}}$(`Exp "+" Term`) contains the symbol '`+`' whereas no strings in $MN_{\widetilde{\text{Exp}}}$(`Term`) have this property (all '`+`' symbols are here underlined), so the potential vertical ambiguity warning is eliminated, and similarly for the other warnings.

This unfolding transformation can be generalized straightforwardly to *multiple* levels of unfolding. In general, unfolding n levels increases the grammar size by a factor of n and allows symbols in a derived string to be distinguished if enclosed by a different number of parentheses, up to depth n. The grammars G7, G8, and Voss-Light introduced in Section 8 need two unfoldings in order to be rightfully acquitted as unambiguous, and Voss needs three. These grammars all share the property of having many different nonterminals producing the same pairs of balanced parentheses.

Example 18 (A non-balanced grammar). Any grammar is balanced if we set, for example, $\Gamma = \widehat{\Gamma} = \emptyset$ but that precludes unfolding to have any effect on the regular approximation. Some grammars exhibit parenthesis structures without being balanced with any suitable choice of Γ and $\widehat{\Gamma}$. An example is the following:

```
S : A A
A : "x" A "x" | y
```

Our present technique is not able to detect that there is no horizontal ambiguity in the first production. The grammar is LR(0), however, so this example along with Example 12 establishes the incomparability of our approach and LR(k). Still, our ambiguity characterization allows further grammar transformations to be applied beyond the unfolding technique described above; we leave it to future work to discover other practically useful variations of unfolding. In this tiny example, a simple solution would be to apply a transformation that underlines all symbols enclosed by `x`'s. An alternative approach would be to combine our technique with, for example, LR(k). Of course, one can just run both techniques and see if one of them is able to verify unambiguity of the given grammar, but there may be ways to combine them more locally (that is, for individual checks of vertical/horizontal unambiguity) to gain more precision.

6. Other Approximation Strategies

To illustrate the flexibility of the ACLA framework, we present three examples of other approximation techniques that lead to substantial performance improvements without losing precision. The approximation strategies presented in this section all have the property that they involve sets of nonterminals or terminals that can be computed efficiently for a given grammar, and, from these sets, they define language approximations that allow many vertical and horizontal unambiguity checks to be performed efficiently. We use the technique from Proposition 10 to combine the entire collection of strategies.

The EmptyString Approximation Strategy

First, we add a simple approximation strategy, *EmptyString*, that focuses on derivation of the empty string for vertical ambiguity checks:

$$\mathit{EmptyString}_G(\alpha) = \begin{cases} \Sigma^0 & \text{if } \alpha = \epsilon \\ \Sigma^+ & \text{if } \alpha \not\Rightarrow^* \epsilon \\ \Sigma^* & \text{otherwise} \end{cases}$$

Productions matching the first case derive the language $\Sigma^0 = \{\epsilon\}$, which is disjoint from the language of a production matching the second case. The first is trivial to check, and the second can be checked efficiently with a well-known algorithm [20]. This approximation effectively checks disjointness of pairs of productions where one derives only the empty string and the other never does. The third case in the definition of $\mathit{EmptyString}_G$ corresponds to this analysis not being able to verify disjointness involving the given production.

Example 19 (*EmptyString*). This check quickly verifies that there is no vertical ambiguity between, for example, the following two productions in the palindrome grammar from Example 12:

```
P   :   "a" P "a"
P   :   ϵ
```

It computes $\mathit{EmptyString}_{\mathrm{Pal}}(\texttt{"a" P "a"}) = \Sigma^+$ and $\mathit{EmptyString}_{\mathrm{Pal}}(\epsilon) = \Sigma^0$, and these sets are obviously disjoint.

The MayMust Approximation Strategy

The approximation strategy *MayMust* considers terminal symbols that *may* and *must* occur in derivable strings. As a preliminary step, we define the function $\mathit{MAY}_G(\alpha) \subseteq \Sigma$ for $\alpha \in E^*$ by the smallest solution to the following equations:

$$\mathit{MAY}_G(\alpha_1 \cdots \alpha_m) = \bigcup_{i=1\ldots m} \mathit{MAY}_G(\alpha_i)$$

$$\mathit{MAY}_G(\sigma) = \{\sigma\} \text{ where } \sigma \in \Sigma$$

$$\mathit{MAY}_G(n) = \bigcup_{\alpha \in \pi(n)} \mathit{MAY}_G(\alpha) \text{ where } n \in \mathcal{N}$$

In other words, $\mathit{MAY}_G(\alpha)$ is simply the set of alphabet symbols that occur in *some* string derivable from α. Likewise, $\mathit{MUST}_G(\alpha) \subseteq \Sigma$ is the set of alphabet symbols that occur in *every* string derivable from α, which is smallest solution to the following equations:

$$\mathit{MUST}_G(\alpha_1 \cdots \alpha_m) = \bigcup_{i=1\ldots m} \mathit{MUST}_G(\alpha_i)$$

$$\mathit{MUST}_G(\sigma) = \{\sigma\} \text{ where } \sigma \in \Sigma$$

$$\mathit{MUST}_G(n) = \bigcap_{\alpha \in \pi(n)} \mathit{MUST}_G(\alpha) \text{ where } n \in \mathcal{N}$$

We now define *MayMust* by

$$\mathit{MayMust}_G(\alpha) = \\ \{\sigma_1 \cdots \sigma_m \in \big(\mathit{MAY}_G(\alpha)\big)^* \mid \forall \rho \in \mathit{MUST}_G(\alpha) : \exists i \in \{1, \ldots, m\} : \sigma_i = \rho\}$$

Both $\mathit{MAY}_G(\alpha)$ and $\mathit{MUST}_G(\alpha)$ can be precomputed for every α appearing in G in linear time in the size of G, and $\mathit{MayMust}_G(\alpha)$ can be represented efficiently by the pair $(\mathit{MAY}_G(\alpha), \mathit{MUST}_G(\alpha))$. This approximation can verify vertical unambiguity of two productions, α and α', when every string derivable from α contains a terminal symbol that never occurs in strings derivable from α', or vice versa.

Example 20 (*MayMust*). Continuing Example 17, we compute, in particular, $\mathit{MUST}_{\widetilde{\mathrm{Exp}}}(\texttt{Exp "+" Term}) = \{\texttt{+}\}$ and $\mathit{MAY}_{\widetilde{\mathrm{Exp}}}(\texttt{Term}) = \{\texttt{x}, \texttt{*}, \texttt{(}, \texttt{)}, \underline{\texttt{+}}, \underline{\texttt{x}}, \underline{\texttt{*}}, \underline{\texttt{(}}, \underline{\texttt{)}}\}$. From this, we get that $\mathit{MayMust}_{\widetilde{\mathrm{Exp}}}(\texttt{Exp "+" Term})$ and $\mathit{MayMust}_{\widetilde{\mathrm{Exp}}}(\texttt{Term})$ are disjoint, so there is no vertical unambiguity at this point in the grammar $\widetilde{\mathrm{Exp}}$. This example also illustrates that unfolding is effective for other approximation strategies than the one from Section 4.

The FirstLast Approximation Strategy

The two approximation strategies mentioned above focus on vertical ambiguity, whereas the next, *FirstLast*, is effective also for many horizontal checks:

$$\mathit{FirstLast}_G(\alpha) = \big(\mathit{MAY}_G(\alpha)\big)^* \ \cap \ \mathit{FIRST}_G(\alpha)\Sigma^* \ \cap \ \Sigma^*\mathit{LAST}_G(\alpha)$$

We here use the definition of FIRST_G from Aho and Ullman [1] for computing the first-set of $\alpha \in E^*$. The function LAST_G is computed as FIRST_G but with the productions reversed, thus considering the last symbols, rather than the first ones, in derivable strings:

$$\mathit{FIRST}_G(\alpha) = \{\sigma \mid (\sigma \in \Sigma \wedge \exists x \in \Sigma^* : \alpha \Rightarrow^* \sigma x) \ \vee \ (\sigma = \epsilon \wedge \alpha \Rightarrow^* \epsilon)\}$$

$$\mathit{LAST}_G(\alpha) = \{\sigma \mid (\sigma \in \Sigma \wedge \exists x \in \Sigma^* : \alpha \Rightarrow^* x\sigma) \ \vee \ (\sigma = \epsilon \wedge \alpha \Rightarrow^* \epsilon)\}$$

For vertical ambiguity, this approximation effectively verifies disjointness of pairs of productions with disjoint first-sets or disjoint last-sets. For horizontal ambiguity, this approximation exploits the fact that the language overlap $X \between Y$ must be empty if the symbols that occur first in strings from Y do not occur at all in X or conversely, if the symbols that occur last in strings from X do not occur at all in Y.

Example 21 (*FirstLast*). Consider the following grammar, S, which is a fragment of a larger grammar by Schmitz [31]:

```
D[1]   :   F
 [2]   |   "s" "f" F

E[1]   :   "q" "i" "q"

F[1]   :   E
 [2]   |   F "f" E
```

For the vertical ambiguity check of the two production of `D`, we compute, in particular, $\mathit{FIRST}_{\mathrm{S}}(\texttt{F}) = \{\texttt{q}\}$ and $\mathit{FIRST}_{\mathrm{S}}(\texttt{"s" "f" F}) = \{\texttt{s}\}$. Since these sets are disjoint, we have that $\mathit{FirstLast}_{\mathrm{S}}(\texttt{F}) \cap \mathit{FirstLast}_{\mathrm{S}}(\texttt{"s" "f" F}) = \emptyset$, which implies that there is no vertical ambiguity at this point in S.

For the second horizontal ambiguity check for production `D[2]`, we compute, in particular, $\mathit{MAY}_{\mathrm{S}}(\texttt{"s" "f"}) = \{\texttt{s}, \texttt{f}\}$ and $\mathit{FIRST}_{\mathrm{S}}(\texttt{F}) = \{\texttt{q}\}$. Since these sets are disjoint, we have that $\mathit{FirstLast}_{\mathrm{S}}(\texttt{"s" "f"}) \between \mathit{FirstLast}_{\mathrm{S}}(\texttt{F}) = \emptyset$, which implies that there is no horizontal ambiguity at this point in S.

Regarding precision, the *MN* strategy from Section 4 subsumes each of the techniques presented here. However, by generally trying these simple approximation strategies on each vertical or horizontal check before running the more expensive *MN* strategy, we gain a notable performance improvement because fewer expensive automata operations are needed. We measure the effect in Section 9.

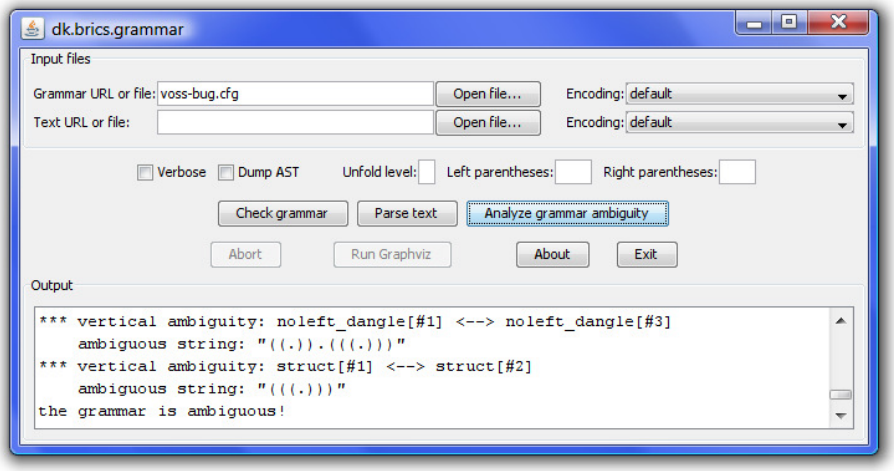


Figure 1: The graphical interface to the ambiguity analyzer implementation.

7. Implementation

An implementation of our ambiguity analysis technique is available at

`http://www.brics.dk/grammar/`

This is an open source Java library (7,400 lines of code) with command-line and graphical interfaces. A screenshot of the running tool is shown in Figure 1.

The tool is designed to be a flexible framework that supports plugging in new approximation strategies. The current built-in strategies are *EmptyString*, *MayMust*, *FirstLast* (from Section 6), and *MN* (from Section 4) – applied in that order according to Proposition 10. The characterization of ambiguity (Proposition 3) leads to a highly parallelizable approach: It reduces the ambiguity problem to a collection of independent vertical/horizontal checks that can be performed separately (although this is currently not exploited in our implementation). Potentially ambiguous example strings (see Example 13) are categorized as potential or definite ambiguities using an Earley-style parser [13]. As a convenient feature, the tool can generate parse trees from the ambiguous example strings and show them graphically via Graphviz.

Grammars can optionally be unfolded as in Section 5. Unfolding here uses a slightly more lenient definition of "balanced grammar" that allows multiple parenthesis pairs in each production. In the current version of the tool, the user specifies the unfold level and the Γ and $\widehat{\Gamma}$ sets, however these parameters could in many cases easily be auto-detected.

The tool also supports terminal sequences to be specified as *regular expressions* within the grammars, as a supplement to the single terminal symbols used in the example grammars shown in the previous sections. This does not affect

the expressive power of the formalism since regular expressions can be transformed into ordinary context-free grammar productions, but regular expressions can often be used for concisely specifying lexical tokens in grammars. Since we treat each substring matched by a regular expressions as an individual parse tree node, we are not concerned with ambiguity that may appear within the regular expressions.

Additionally, the grammar notation allows production constituents to be marked as *ignorable* meaning that the corresponding parse tree fragments are omitted when parsing. This is particularly useful for describing whitespace, comments, etc. in programming language grammars. The ambiguity analyzer handles this feature by simply omitting certain horizontal ambiguity checks. As an experimental feature, the grammar notation also allows the productions of a nonterminal to be *prioritized*, which is a disambiguation mechanism that is useful for expressing, for example, operator precedence. If a nonterminal has two productions with different priorities then a parser will resolve any ambiguity between them by always selecting the one with the highest priority. Similarly, production constituents can be marked as *maximal* or *minimal*, which is a disambiguation mechanism that is useful for expressing operator associativity. These two disambiguation mechanisms are handled by the ambiguity analyzer by omitting certain vertical and horizontal ambiguity checks, respectively.

The implementation builds on two other tools: `dk.brics.automaton` [26] for handling regular expressions and finite-state automata using the Unicode character set as alphabet, and JSA [9] for performing regular approximations of context-free grammars and converting regular grammar components into automata.

8. Application to Biosequence Analysis

The languages of biosequences are trivial from the formal language point of view. The alphabet of DNA is $\Sigma_{\texttt{DNA}} = \{A, C, G, T\}$, of RNA it is $\Sigma_{\texttt{RNA}} = \{A, C, G, U\}$, and for proteins it is a 20 letter amino acid code. In each case, the language of biosequences is Σ^*. A biomolecule can be of any length and any composition. Of course, molecules having a particular function have all kinds of restriction associated, such as minimal or maximal size, species-specific composition, characteristic sequence motifs, or potential for forming a well-defined 3D structure. Biosequence analysis relates two sequences to each other (sequence alignment, similarity search) or one sequence to itself (folding). The latter is our application domain – RNA structure analysis.

RNA is a chain molecule, built from the four *bases* adenine (A), cytosine (C), guanine (G), and uracil (U), connected via a *backbone* of sugar and phosphate. Mathematically, it is a string over $\Sigma_{\texttt{RNA}}$ of moderate length (compared to genomic DNA), ranging from 20 to 10,000 bases.

RNA forms structure by folding back on itself. Certain bases, located at different positions in the backbone, may form hydrogen bonds. Such bonded *base pairs* arise between *complementary* bases $G-C$, $A-U$, and $G-U$. By forming these bonds, the two pairing bases are arranged in a plain, and this in

turn enables them to stack very densely onto adjacent bases also forming pairs. Helical structures arise, which are energetically stable and mechanically rather stiff. They enable RNA to perform its wide variety of functions.

Because of the backbone turning back on itself, RNA structures can be viewed as palindromic languages. Starting from palindromes in the traditional sense (as described in Example 12) we can characterize palindromic languages for RNA structure via five generalizations: (1) a letter does not match to itself but to a complementary base; (2) the two arms of a palindrome may be separated by a non-palindromic string (of length at least 3) called a *loop*; (3) the two arms of the palindrome may hold non-pairing bases called *bulges*; (4) a string may hold zero or more adjacent palindromes separated by unpaired bases; and (5) palindromes can be recursively nested, that is, a loop or a bulge may contain further palindromes. Example 22 shows a grammar modeling base pair complementarity and nesting, as required by generalization 1. It captures only this aspect and is not a valid RNA grammar.

Example 22 (RNA "palindromes" – base pairs only).

```
R   :   "C" R "G"   |   "G" R "C"
    |   "A" R "U"   |   "U" R "A"
    |   "G" R "U"   |   "U" R "G"    |    ε
```

Context-free grammars are used to describe the structures that can be formed by a given RNA sequence. (The grammars G1 through G8, which we describe later, are different ways to achieve this.) All grammars generate the full language $\Sigma^*_{\mathtt{RNA}}$, the different derivations of a given RNA string corresponding to its possible physical structures in different ways.

Figure 2 shows an ambiguous grammar (G1 from Reeder et al. [30]), a short RNA sequence x, and two of its possible structures, presented as parse trees from G1 and as a so-called Vienna (or "dot-bracket") strings. In the grammar, a and $\hat{a}$ denote complementary two bases, which can form a base pair. In a Vienna string, matched parentheses indicate the base pairs, while dots denote unpaired bases. Such strings represent structures uniquely, and are often used by RNA-related software.

The number of possible structures under the rules of base pairing is exponentially related to the length of the molecule. In formal language terms, each string has an exponential number of parse trees. This has been termed the "good" ambiguity in a grammar describing RNA structure.

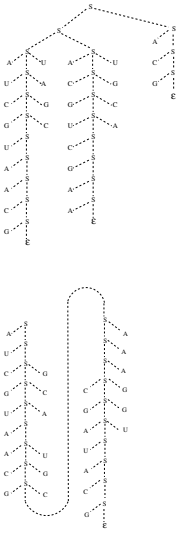
The set of all structures is the search space from which we want to extract the "true" (physical) structure(s). This is achieved by evaluating structures under a variety of scoring schemes, the most common of which is a thermodynamic model. A CYK-style parser [1] constructs the search space and applies dynamic programming along the way to choose one or more, optimal or near-optimal foldings.

The problem at hand arises when different parse trees correspond to the same physical structure. In Figure 3, two parse trees (for sequence x relative to G1) are shown which exhibit the same base pairs and hence denote the same secondary structure.

Grammar G1:
$S \to SS \mid aS \mid Sa \mid aS\hat{a} \mid \epsilon$

x: `AUCGUAACGCGAU`
`ACGUCGAAACGUACG`

(a) Grammar G1 and RNA sequence x



(b) Two parse trees of x relative to G1

Structure found by parse 1:
`((((.....))))((((....))))...`
2 short "hairpins",
followed by 3 unpaired bases

Structure found by parse 2:
`..(((.((((((....)))...))))))`
2 unpaired bases,
followed by a long "hairpin".

(c) Two different secondary structures

Figure 2: Good ambiguity in RNA folding: Alternative parses construct all possible foldings, from which the best ones are chosen under a thermodynamic model.

In this situation, the thermodynamically optimal structure can still be determined, but everything else goes wrong: The number of structures is wrongly counted. Boltzmann statistics over the complete folding space becomes incorrect, because individual states are accounted for a varying number of times. Enumeration of near-optimal structures will produce duplicates, of which there may be an exponential number (in terms of sequence length). Finally, when used with a stochastic scoring scheme, the most likely parse does not find the most likely structure. We say that the grammar exhibits the "bad" kind of ambiguity. It makes no sense to check the grammar for ambiguity *as is*, since "the bad ambiguity hides within the good" (using a phrase from Reeder et al. [30]).

Fortunately, the grammar can be transformed such that the good ambiguity is eliminated, while the bad persists and can now be checked by formal language techniques such as ours. The grammar remains structurally unchanged in the transformation, but is rewritten to no longer generate RNA sequences, but Vienna strings. These strings represent structures uniquely, and if one of them has two different parse trees, then the original grammar has the bad type of ambiguity.

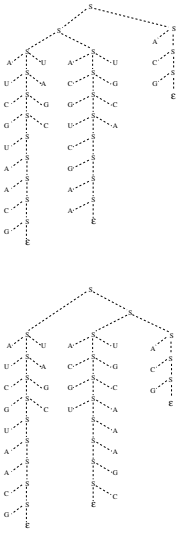
At this point, the reader may wonder why this problem is not solved once and for all time by designing a grammar that adequately models RNA structure, and is proved non-ambiguous by human effort.

Biologist are interested in families of RNA that execute a specific regulatory function in different organisms. Such families of RNA sequences are character-

Grammar G1:
$S \to SS \mid aS \mid Sa \mid aS\hat{a} \mid \epsilon$

x: `AUCGUAACGCGAU`
`ACGUCGAAACGUACG`

(a) Grammar G1 and RNA sequence x



(b) Two parse trees of x relative to G1

Base pairs in both trees:
`((((.....))))((((....))))...`
2 short "hairpins",
followed by 3 unpaired bases

← different arrangement at top

← different productions used for loop in second hairpin

(c) Twice the same secondary structure

Figure 3: Bad ambiguity in RNA folding: Alternative parses for sequence x constructing the same physical structure, confounding various types of analyses.

ized by a common structure. A model grammar for such a family restricts the allowed foldings to variations of the common structure. Bioinformatics tools, which generate such grammars, are *Locomotif* [29] or *Infernal* [27]. The Rfam database [15] currently holds 1,300 stochastic models based on RNA family grammars. The ACLA system has recently been included in a pipeline to prove unambiguity of such models [17].

9. Experiments

We applied our ambiguity checker to several grammars that were obtained by the above transformation from stochastic grammars used in the bioinformatics literature [11, 30, 36]. Grammars G1 and G2 were studied as ambiguous grammars by Dowell and Eddy [11], and our algorithm nicely points out the sources of ambiguity by indicating shortest ambiguous words. Dowell and Eddy [11] introduced G2 as a refinement of G1, to bring it closer to grammars used in practice. Our ambiguity checker detects an extra vertical ambiguity in G2 (see Table 1) and clearly reports it by producing the ambiguous word "`()`" for the productions `P[aPa]` and `P[S]`. Grammars G3 through G8, taken from the same source, are unambiguous grammars, but a proof has been lacking. Our approach demonstrates their unambiguity. Grammars G1 through G8 describe the same language of structures, but they lead to different stochastic models due to their different use of nonterminals and productions. This difference was investigated by Dowell and Eddy [11].

The grammar Voss from Voss et al. [36] has 28 nonterminals and 65 productions. This grammar clearly asks for automatic support (even for experts in formal grammars). We describe this application in two steps. First, we study a grammar, Voss-Light, which demonstrates an essential aspect of the Voss grammar: unpaired bases in bulges and loops (the dots in the transformed grammar) must be treated differently, and they hence are derived from different nonterminal symbols even though they recognize the same language. This takes the grammar Voss-Light (and consequently also Voss) beyond the capacities of, for example, LR(k) parsing, whereas our technique succeeds in verifying unambiguity.

Example 23 (Voss-Light).

```
P  :  "(" P ")"  |  "(" O ")"          // P: closed structure
O  :  L P  |  P R  |  S P S  |  H      // O: open structure
L  :  "." L  |  "."                    // L: left bulge
R  :  "." R  |  "."                    // R: right bulge
S  :  "." S  |  "."                    // S: singlestrand
H  :  "." H  |  "." "." "."            // H: hairpin 3+ loop
```

As the second step, we took the full grammar. Our method succeeded to show unambiguity (using three unfoldings), which implies that the Boltzmann statistics computed according to Voss et al. [36] are indeed correct.

Table 1 summarizes the results of running our ACLA ambiguity analysis, compared with LR(1) and the technique by Schmitz [31, 32], on the example grammars from biosequence analysis. The first column lists the name of the grammar along with a source reference. The second column, #prod, shows the size of each grammar measured as number of productions. The LR(1) column shows the number of shift/reduce and reduce/reduce conflicts using LR(1). The Schmitz column shows the number of potential ambiguities reported by Schmitz's approach (using LR(1) items). The ACLA column shows the result from our analysis. Here, V! and H! mark the number of *definite* vertical and horizontal ambiguities, respectively (for ambiguous grammars). For each tool, *unamb.* means that the tool succeeds in verifying unambiguity. The "real" column shows whether the grammar is ambiguous or not. The last column shows the time (in seconds) for the ACLA analysis, running on a 2.4GHz PC. For the ACLA tests, unfolding has been used in many of the grammars, all using $\Gamma = \{(\}$ and $\widehat{\Gamma} = \{)\}$.

Notice that ACLA reports no false positives, unlike LR(1) and Schmitz's tool. (Many of the grammars are in fact beyond LR-Regular.) For the ambiguous grammars (G1 and G2), our tool even provides example strings, thereby proving that the grammars are indeed ambiguous, whereas the other tools do not provide guarantees in this direction.

[1] Dowell and Eddy [11]
[2] Voss et al. [36]

Grammar	#prod	LR(1)	Schmitz	ACLA	real	time
G1[1]	5	24 + 12	14	5V! + 1H!	*amb.*	<0.01s
G2[1]	7	25 + 13	13	6V! + 1H!	*amb.*	<0.01s
G3[1]	8	4 + 0	2	*unamb.*	*unamb.*	<0.01s
G4[1]	6	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
G5[1]	3	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
G6[1]	6	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
G7[1]	13	5 + 0	3	*unamb.*	*unamb.*	0.02s
G8[1]	11	*unamb.*	*unamb.*	*unamb.*	*unamb.*	0.02s
Base pairs (Ex. 22)	7	6 + 0	12	*unamb.*	*unamb.*	<0.01s
Voss-Light (Ex. 23)	14	0 + 3	*unamb.*	*unamb.*	*unamb.*	0.01s
Voss[2]	64	16 + 0	5	*unamb.*	*unamb.*	0.93s
Pal (Ex. 12)	5	6 + 2	8	*unamb.*	*unamb.*	<0.01s
AntiPal (Ex. 12)	5	6 + 2	8	*unamb.*	*unamb.*	<0.01s

Table 1: Grammars from bioinformatics (and Example 12).

Also, the running times are seen to be satisfactory. For the largest grammar, Voss, which takes around one second to analyze, ACLA performs 87 vertical and 103 horizontal ambiguity checks. Of these, 13 and 71 checks, respectively, are handled by the simple approximation strategies described in Section 6 and the remaining ones by the regular approximation strategy from Section 4. In many of the smaller grammars, the simple approximation strategies cover almost all the vertical and horizontal checks. Generally, if we disable the simple approximation strategies, running time increases but, for these grammars, not dramatically.

Although our primary focus is on grammars from bioinformatics, we also tested our tool on grammars from other domains, in most cases interesting fragments of programming languages and some more "artificial" grammars. Table 2 show the results from grammars collected by Schmitz [32]. Here, V? and H? measure the number of *potential* vertical and horizontal ambiguities, respectively. ACLA gives perfect results in 30 of these 35 cases. In two of the remaining five cases (04_11_047 and 91_08_014), ACLA succeeds in proving ambiguity but also reports some potential ambiguities. In one case (sml_fvalbind), which is ambiguous, ACLA reports potential ambiguities but fails to provide concrete examples to prove ambiguity. Finally, in only two cases (S5 and S7), ACLA produces spurious warnings about potential ambiguities although these grammars are in fact unambiguous. Studying these two grammars further reveals that a slightly more aggressive variant of unfolding (as discussed in Section 5) would handle these cases also.

Of 70 grammars collected by Basten [2] (we exclude grammars tested in the previously mentioned experiments), we obtain similarly encouraging results. The size of these grammars ranges from 3 to 16 productions, and ACLA analyzes each grammar in less than 0.04 seconds. 24 of the grammars are unambiguous, and ACLA produces spurious warnings about potential ambiguities in only four of those. Of the remaining 46 ambiguous grammars, ACLA gives perfect results in 27 cases (that is, it here reports only *certain* ambiguities – with example

Grammar	#prod	LR(1)	Schmitz	ACLA	real	time
S1	7	2 + 0	3	1V! + 2H!	*amb.*	<0.01s
S2	3	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
S3	9	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
S4	9	1 + 0	1	1V!	*amb.*	<0.01s
S5	6	0 + 1	*unamb.*	1V?	*unamb.*	<0.01s
S6	5	6 + 2	8	*unamb.*	*unamb.*	<0.01s
S7	3	*unamb.*	*unamb.*	1H?	*unamb.*	<0.01s
01_05_076	11	2 + 0	4	1H!	*amb.*	0.01s
03_01_011	8	1 + 0	1	1H!	*amb.*	<0.01s
03_02_124	6	0 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
03_05_170	11	1 + 0	1	1V! + 2H!	*amb.*	<0.01s
03_09_027	8	2 + 0	*unamb.*	*unamb.*	*unamb.*	0.01s
03_09_081	10	1 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
04_02_041	4	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
04_11_047	27	12 + 3	31	1V! + 1V? + 6H?	*amb.*	0.06s
05_03_092	12	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
05_03_114	7	0 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
05_06_028	8	1 + 1	5	2H!	*amb.*	0.11s
06_10_036	33	0 + 6	1	1V!	*amb.*	0.08s
90_10_042	7	2 + 0	6	*unamb.*	*unamb.*	<0.01s
91_08_014	15	17 + 1	4	2V! + 1V? + 3H?	*amb.*	0.09s
98_05_030	6	1 + 0	*unamb.*	*unamb.*	*unamb.*	0.01s
98_08_215	7	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
ada_is	9	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
ada_calls	17	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
isocpp_qualid	11	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
java_modifiers	48	0 + 49	*unamb.*	*unamb.*	*unamb.*	0.02s
java_names	20	0 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
java_arrays	12	1 + 0	*unamb.*	*unamb.*	*unamb.*	<0.01s
java_casts	9	0 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
pascal_typed	8	0 + 1	*unamb.*	*unamb.*	*unamb.*	<0.01s
pascal_begin	5	*unamb.*	*unamb.*	*unamb.*	*unamb.*	<0.01s
set_exp	35	0 + 8	2	*unamb.*	*unamb.*	0.08s
sml_fvalbind	13	1 + 0	2	1V? + 4H?	*amb.*	0.01s
sml_patterns	5	2 + 0	1	1V!	*amb.*	<0.01s

Table 2: Grammars from Schmitz [32].

strings).

Interestingly, for our entire benchmark suite, a combination of Schmitz's tool and ours eliminates *all* spurious errors, which indicates complementary strengths of these two techniques.

10. Conclusion

We have presented a technique, ACLA (Ambiguity Checking with Language Approximations), for statically analyzing ambiguity of context-free grammars. Based on a linguistic characterization, the technique allows the use of grammar transformations, in particular regular approximation and unfolding, without sacrificing soundness. Moreover, the analysis is often able to pinpoint sources

of ambiguity through concrete examples being automatically generated. The analysis may be used when LR(k) and related techniques are inadequate, for example in biosequence analysis, as our examples show. Our experiments indicate that the precision, the speed, and the quality of ambiguity reports are sufficient to be practically useful.

Acknowledgments. Thanks are due to Sylvain Schmitz for his insightful comments and for providing benchmark grammars.

References

[1] Aho, A. V., Ullman, J. D., 1972. The Theory of Parsing, Translation and Compiling, Volume 1: Parsing. Prentice Hall.

[2] Basten, H., 2009. The usability of ambiguity detection methods for context-free grammars. Electronic Notes in Theoretical Computer Science 238 (5), 35–46.

[3] Berstel, J., Boasson, L., 2002. Formal and Natural Computing. Springer-Verlag, Ch. 1: Balanced Grammars and Their Languages, pp. 3–25.

[4] Brabrand, C., Møller, A., Schwartzbach, M. I., June 2008. Dual syntax for XML languages. Information Systems 33 (4), earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer-Verlag, LNCS vol. 3774.

[5] Brabrand, C., Schwartzbach, M. I., 2007. The metafront system: Safe and extensible parsing and transformation. Science of Computer Programming 68 (1), 2–20.

[6] Cantor, D. G., 1962. On the ambiguity problem of Backus systems. Journal of the ACM 9 (4), 477–479.

[7] Cheung, B. S. N., Uzgalis, R. C., 1995. Ambiguity in context-free grammars. In: Proc. ACM Symposium on Applied Computing, SAC '95.

[8] Chomsky, N., Schützenberger, M.-P., 1963. The algebraic theory of context-free languages. In: Computer Programming and Formal Systems. North-Holland, pp. 118–161.

[9] Christensen, A. S., Møller, A., Schwartzbach, M. I., June 2003. Precise analysis of string expressions. In: Proc. 10th International Static Analysis Symposium, SAS '03. Vol. 2694 of LNCS. Springer-Verlag, pp. 1–18.

[10] Čulik II, K., Cohen, R. S., 1973. LR-regular grammars - an extension of LR(k) grammars. Journal of Computer and System Sciences 7 (1), 66–96.

[11] Dowell, R. D., Eddy, S. R., 2004. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. BMC Bioinformatics 5 (71).

[12] Durbin, R., Eddy, S. R., Krogh, A., Mitchison, G., 1998. Biological Sequence Analysis. Cambridge University Press.

[13] Earley, J., 1970. An efficient context-free parsing algorithm. Communications of the ACM 13 (2), 94–102.

[14] Floyd, R. W., 1962. On ambiguity in phrase structure languages. Communications of the ACM 5 (10), 526.

[15] Gardner, P. P., Daub, J., Tate, J. G., Nawrocki, E. P., Kolbe, D. L., Lindgreen, S., Wilkinson, A. C., Finn, R. D., Griffiths-Jones, S., Eddy, S. R., Bateman, A., Jan 2009. Rfam: updates to the RNA families database. Nucleic Acids Res 37 (Database issue), 136–140.

[16] Giegerich, R., 2000. Explaining and controlling ambiguity in dynamic programming. In: Proc. 11th Annual Symposium on Combinatorial Pattern Matching. Vol. 1848 of LNCS. Springer-Verlag, pp. 46–59.

[17] Giegerich, R., Höner zu Siederdissen, C., 2009. Semantics and ambiguity of stochastic RNA family models, submitted manuscript.

[18] Giegerich, R., Meyer, C., Steffen, P., 2004. A discipline of dynamic programming over sequence data. Science of Computer Programming 51 (3), 215–263.

[19] Gorn, S., 1963. Detection of generative ambiguities in context-free mechanical languages. Journal of the ACM 10 (2), 196–208.

[20] Hopcroft, J. E., Ullman, J. D., 1979. Introduction to Automata Theory, Languages and Computation. Addison-Wesley.

[21] Knuth, D. E., 1965. On the translation of languages from left to right. Information and Control 8, 607–639.

[22] Knuth, D. E., 1967. A characterization of parenthesis languages. Information and Control 11, 269–289.

[23] Kuich, W., 1970. Systems of pushdown acceptors and context-free grammars. Elektronische Informationsverarbeitung und Kybernetik 6 (2), 95–114.

[24] McPeak, S., Necula, G. C., 2004. Elkhound: A fast, practical GLR parser generator. In: Proc. 13th International Conference on Compiler Construction, CC '04.

[25] Mohri, M., Nederhof, M.-J., 2001. Robustness in Language and Speech Technology. Kluwer Academic Publishers, Ch. 9: Regular Approximation of Context-Free Grammars through Transformation.

[26] Møller, A., 2008. dk.brics.automaton – finite-state automata and regular expressions for Java. `http://www.brics.dk/automaton/`.

[27] Nawrocki, E. P., Kolbe, D. L., Eddy, S. R., May 2009. Infernal 1.0: inference of RNA alignments. Bioinformatics 25 (10), 1335–1337.

[28] Nederhof, M.-J., 2000. Practical experiments with regular approximation of context-free languages. Computational Linguistics 26 (1), 17–44.

[29] Reeder, J., Reeder, J., Giegerich, R., 2007. Locomotif: From graphical motif description to RNA motif search. Bioinformatics 23 (13), i392–400.

[30] Reeder, J., Steffen, P., Giegerich, R., 2005. Effective ambiguity checking in biosequence analysis. BMC Bioinformatics 6 (153).

[31] Schmitz, S., 2007. Conservative ambiguity detection in context-free grammars. In: Proc. 34th International Colloquium on Automata, Languages and Programming, ICALP '07.

[32] Schmitz, S., 2009. An experimental ambiguity detection tool. Science of Computer Programming.

[33] Scott, E., Johnstone, A., Hussein, S. S., 2000. Tomita style generalised parsers. Tech. Rep. CSD-TR-00-A, Royal Holloway, University of London.

[34] van den Brand, M., Scheerder, J., Vinju, J. J., Visser, E., 2002. Disambiguation filters for scannerless generalized LR parsers. In: Proc. 11th International Conference on Compiler Construction, CC '02.

[35] Visser, E., 1997. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam.

[36] Voss, B., Giegerich, R., Rehmsmeier, M., 2006. Complete probabilistic analysis of RNA shapes. BMC Biology 4 (5).

A. Proof of Proposition 3

To show

$$\|\models G \quad \Leftrightarrow \quad G \textit{ is unambiguous}$$

for a CFG $G = (\mathcal{N}, \Sigma, s, \pi)$ (see Definitions 1 and 2) we consider each direction in turn.

We prove $\|\models G \Rightarrow G$ *is unambiguous* by contrapositively establishing that if G is ambiguous then $\|\not\models G$, that is, $\|\not\models G \vee \not\models G$. Assume that G is ambiguous, which means that there are two different derivation trees, T and T', for some string $\omega \in \mathcal{L}(G)$. We proceed by induction in the maximum height h of T and T' (where the height of a derivation tree is the maximum number of edges from the root to a leaf).

Base case, $h = 0$: The result holds vacuously; a derivation tree of height zero consists of a single node labeled with the start nonterminal, and at most one such derivation tree can exist.

Inductive case: We assume the property holds for all derivation trees of maximum height $h - 1$ and show that it also holds for height h. Recall that every tree node is labeled by a nonterminal or a terminal from $E = \Sigma \cup \mathcal{N}$. Let α and α' be the sequences of labels of the children of the roots of T and T', respectively. There are two cases depending on whether or not the top-most productions in the trees are the same:

(1) $\alpha \neq \alpha'$: In this case, the trees differ due to the different initial productions α and α' that ultimately produce the same string ω, so $\omega \in \mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\alpha')$ and hence $\|\not\models G$.

(2) $\alpha = \alpha'$: Let $\alpha_1, \ldots, \alpha_n \in E$ be the sequence defined by $\alpha = \alpha_1 \cdots \alpha_n$. Let ω_i and ω'_i be the substrings of ω that are derived by the i'th child of the root node in T and T', respectively, as depicted in Figure 4. We again split into two cases:

(2a) $\exists i : \omega_i \neq \omega'_i$: Let $k = min\{i \mid \omega_i \neq \omega'_i\}$ and, depending on this k, let $\omega_L = \omega_0 \cdots \omega_k$, $\omega_R = \omega_{k+1} \cdots \omega_n$, $\omega'_L = \omega'_0 \cdots \omega'_k$, and $\omega'_R = \omega'_{k+1} \cdots \omega'_n$.

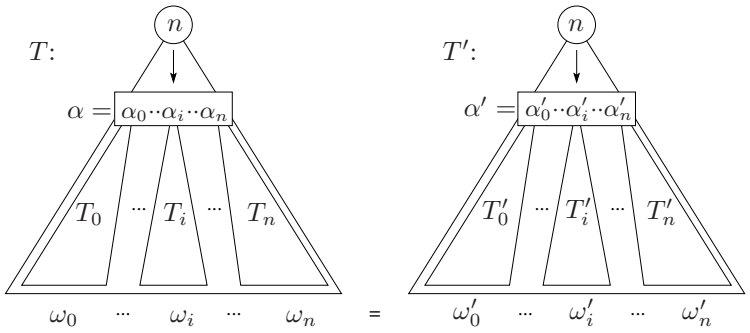


Figure 4: Two derivation trees, T and T', both deriving ω from n.

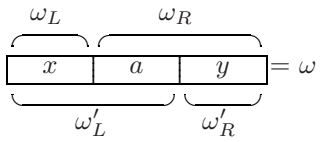


Figure 5: Overlap: $\omega \in \mathcal{L}_G(\alpha_0..\alpha_k) \between \mathcal{L}_G(\alpha_{k+1}..\alpha_n)$.

Since k was chosen as the minimum we must have that $\omega_L \neq \omega'_L$ and since $\omega_L\omega_R = \omega = \omega'_L\omega'_R$, the strings must be organized according to Figure 5 (assuming without loss of generality that $|\omega_L| < |\omega'_L|$) where $a \in \Sigma^+$ is given by $\omega = \omega_L a \omega'_R$ and $x = \omega_L$, $y = \omega'_R$. We thus have a language overlap $\omega = xay \in X \between Y$ for the two languages $X = \mathcal{L}_G(\alpha_0 \cdots \alpha_k)$ and $Y = \mathcal{L}_G(\alpha_{k+1} \cdots \alpha_n)$ and hence $\not\models G$.

(2b) $\forall i : \omega_i = \omega'_i$: In this case, the difference between the two trees T and T' must be further down. Pick any i such that the subtrees T_i and T'_i (where T_i is the subtree corresponding to α_i) are different (such an i must exist since T and T' were different in the first place). The induction hypothesis applied to these smaller trees, using α_i as start nonterminal and ambiguously deriving ω_i, gives us $\not\models G$.

We now consider the other direction: $\models G \Leftarrow G$ *is unambiguous*. This is shown by contrapositively establishing that if $\not\models G \vee \not\models G$ then G is ambiguous. We split into two cases; one for each kind of ambiguity:

(1) $\not\models G$: By definition, there exist $n \in \mathcal{N}$ and $\alpha, \alpha' \in \pi(n)$ where $\alpha \neq \alpha'$ such that $\mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\alpha') \neq \emptyset$. Let $y \in \Sigma^*$ be an element of this intersection, that is, $\alpha \Rightarrow^* y$ and $\alpha' \Rightarrow^* y$. We have assumed that every nonterminal is derivable from s and derives a nonempty set of string. This means that we can construct two different derivation trees starting at s, corresponding to the following derivations of the string xyz for some $x, z \in \Sigma^*$:

$$s \Rightarrow^* x\ n\ z \Rightarrow x\ \alpha\ z \Rightarrow^* x\ y\ z$$
$$s \Rightarrow^* x\ n\ z \Rightarrow x\ \alpha'\ z \Rightarrow^* x\ y\ z$$

(2) $\not\models G$: By definition, there exist $n \in \mathcal{N}$ and $\alpha\alpha' \in \pi(n)$ such that $x, xa \in \mathcal{L}_G(\alpha)$ and $y, ay \in \mathcal{L}_G(\alpha')$ for some $x, y \in \Sigma^*$, $a \in \Sigma^+$. Similar to the case above, we can make two different derivation trees corresponding to the following derivations of the string $uxayv$ for some $u, v \in \Sigma^*$:

$$s \Rightarrow^* u\ n\ v \Rightarrow u\ \alpha\ \alpha'\ v \Rightarrow^* u\ x\ \alpha'\ v \Rightarrow^* u\ x\ a\ y\ v$$
$$s \Rightarrow^* u\ n\ v \Rightarrow u\ \alpha\ \alpha'\ v \Rightarrow^* u\ x\ a\ \alpha'\ v \Rightarrow^* u\ x\ a\ y\ v$$

B. Proof of Proposition 8

It suffices to show that $\models_{\mathcal{A}_G} G \Rightarrow \models G$ since the result then follows from Proposition 3. However, this is immediate from Definitions 2, 6, and 7: Since $\mathcal{A}_G$ is a conservative approximation, we have for all $\alpha, \beta \in E^*$,

$$\begin{array}{lcl} \mathcal{A}_G(\alpha) \cap \mathcal{A}_G(\beta) = \emptyset & \Rightarrow & \mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\beta) = \emptyset \\ \mathcal{A}_G(\alpha) \between \mathcal{A}_G(\beta) = \emptyset & \Rightarrow & \mathcal{L}_G(\alpha) \between \mathcal{L}_G(\beta) = \emptyset \end{array}$$