Tracking Down the Origins of Ambiguity in Context-Free Grammars

H.J.S. Basten

Centrum Wiskunde & Informatica
P.O. Box 94079
NL-1090 GB Amsterdam, The Netherlands

Abstract. Context-free grammars are widely used but still hindered by ambiguity. This stresses the need for detailed detection methods that point out the sources of ambiguity in a grammar. In this paper we show how the approximative Noncanonical Unambiguity Test by Schmitz can be extended to conservatively identify production rules that do not contribute to the ambiguity of a grammar. We prove the correctness of our approach and consider its practical applicability.

1 Introduction

Context-free grammars (CFGs) are widely used in various fields, like for instance programming language development, natural language processing, or bioinformatics. They are suitable for the definition of a wide range of languages, but their possible ambiguity can hinder their use. Designed ambiguities are not uncommon, but accidentally introduced ambiguities are unwanted. Ambiguities are very hard to detect by hand, so automated ambiguity checkers are welcome tools.

Despite the fact the CFG ambiguity problem is undecidable in general [5,7,6], various detection schemes exist. They can roughly be divided into two categories: exhaustive methods and approximative ones. Methods in the first category exhaustively search the usually infinite set of derivations of a grammar, while the latter ones apply approximation to limit their search space. This enables them to always terminate, but at the expense of potentially incorrect reports. Exhaustive methods do produce precise reports, but only if they find ambiguity before they are halted, because they obviously cannot be run forever.

Because of the undecidability it is impossible to always terminate with a correct and detailed report. The challenge is to develop a method that gives the most precise answer in the time available. In this paper we propose to combine exhaustive and approximative methods as a step towards this goal. We show how to extend the Regular Unambiguity Test and Noncanonical Unambiguity Test [11] to improve the precision of their approximation and that of their ambiguity reports. The extension enables the detection of production rules that do not contribute to the ambiguity of a grammar. These are already helpful reports for the grammar developer, but can also be used to narrow the search space of other detection methods. In an earlier study [3] we witnessed significant reductions in the run-time of exhaustive methods due to our grammar filtering.

1.1 Related Work

The original Noncanonical Unambiguity Test by Schmitz is an approximative test for the unambiguity of a grammar. The approximation it applies is always conservative, so it can only find a grammar to be *unambiguous* or *potentially ambiguous*. Its answers always concern the grammar as a whole, but the reports of a prototype implementation [12] by the author also contain clues about the production rules involved in the potential ambiguity. However, these are very abstract and hard to understand. The extensions that we present do result in precise reports, while remaining conservative.

Another approximative ambiguity detection scheme is the "Ambiguity Checking with Language Approximation" framework [4] by Brabrand, Giegerich and Møller. The framework makes use of a characterization of ambiguity into horizontal and vertical ambiguity to test whether a certain production rule can derive ambiguous strings. The difference with our approach is that we test whether a production rule is vital for the existence of parse trees of ambiguous strings.

1.2 Overview

We start with background information about grammars and languages in Section 2. Then we repeat the definition of the Regular Unambiguity (RU) Test in Section 3. In Section 4 we explain how the RU Test can be extended to identify sets of parse trees of unambiguous strings. From these parse trees we can identify harmless production rules as explained in Section 5. Section 6 explains the Noncanonical Unambiguity (NU) Test, an improvement over the RU Test, and also shows how it improves the effect of our parse tree and production rule filtering. In Section 7 we describe how our approach can be used iteratively to increase its precision. Finally, Section 9 contains the conclusion.

We prove our results in an accompanying technical report [2].

2 Preliminaries

This section gives a quick overview of the theory of grammars and languages, and introduces the notational convention used throughout this document. For more background information we refer to [9,14].

2.1 Context-Free Grammars

A context-free grammar G is a 4-tuple (N, T, P, S) consisting of:

- N, a finite set of *nonterminals*,
- T, a finite set of *terminals* (the alphabet),
- P, a finite subset of $N \times (N \cup T)^*$, called the *production rules*,
- S, the *start symbol*, an element from N.

We use V to denote the set $N \cup T$, and V' for $V \cup \{\varepsilon\}$. The following characters are used to represent different symbols and strings: $a, b, c, \ldots$ represent terminals,

$A, B, C, \ldots$ represent nonterminals, X, Y, Z represent either nonterminals or terminals, $\alpha, \beta, \gamma, \ldots$ represent strings in V^*, $u, v, w, \ldots$ represent strings in T^*, ε represents the empty string.

A production (A, α) in P is written as $A \to \alpha$. We use the function $\mathsf{pid}: P \to \mathbb{N}$ to relate each production to a unique identifier. An *item* [10] indicates a position in the right hand side of a production using a dot. Items are written like $A \to \alpha \bullet \beta$.

The relation $\Longrightarrow$ denotes direct derivation, or derivation in one step. Given the string $\alpha B \gamma$ and a production rule $B \to \beta$, we can write $\alpha B \gamma \Longrightarrow \alpha \beta \gamma$ (read $\alpha B \gamma$ directly derives $\alpha \beta \gamma$). The symbol $\Longrightarrow^*$ means "derives in zero or more steps". A sequence of derivation steps is simply called a *derivation.* Strings in V^* are called *sentential forms.* We call the set of sentential forms that can be derived from S of a grammar G, the *sentential language* of G, denoted $\mathcal{S}(G)$. A sentential form in T^* is called a *sentence.* The set of all sentences that can be derived from S of a grammar G is called the *language* of G, denoted $\mathcal{L}(G)$.

We assume every nonterminal A is *reachable* from S, that is $\exists \alpha A \beta \in \mathcal{S}(G)$. We also assume every nonterminal is *productive*, meaning $\exists u : A \Longrightarrow^* u$.

The *parse tree* of a sentential form α describes how α is derived from S, but disregards the order of the derivation steps. To represent parse trees we use bracketed strings (See Section 2.3). A grammar G is ambiguous iff there is at least one string in $\mathcal{L}(G)$ for which multiple parse trees exist.

2.2 Bracketed Grammars

From a grammar $G = (N, T, P, S)$ a *bracketed grammar* G_b can be constructed, by adding unique terminals to the beginning and end of every production rule [8]. The bracketed grammar G_b is defined as the 4-tuple (N, T_b, P_b, S), where:

- $T_b = T \cup T_\langle \cup T_\rangle$,
- $T_\langle = \{ \langle_i \mid \exists p \in P : i = \mathsf{pid}(p)\}$,
- $T_\rangle = \{ \rangle_i \mid \exists p \in P : i = \mathsf{pid}(p)\}$,
- $P_b = \{A \to \langle_i \alpha \rangle_i \mid A \to \alpha \in P, i = \mathsf{pid}(A \to \alpha)\}$.

V_b is defined as $T_b \cup N$, and V_b' as $V_b \cup \{\varepsilon\}$. We use $a_b, b_b, \ldots$ and X_b, Y_b, Z_b to represent symbols in respectively T_b and V_b. Similarly, $u_b, v_b, \ldots$ and $\alpha_b, \beta_b, \ldots$ represent strings in respectively T_b^* and V_b^*, The relation $\Longrightarrow_b$ denotes direct derivation using productions in P_b. The homomorphism h from V_b^* to V^* maps each string in $\mathcal{S}(G_b)$ to $\mathcal{S}(G)$. It is defined by $h(\langle_i) = \varepsilon$, $h(\rangle_i) = \varepsilon$, and $h(X) = X$.

2.3 Parse Trees

$\mathcal{L}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{L}(G)$. $\mathcal{S}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{S}(G)$. We divide it into two disjoint sets:

Definition 1. *The set of parse trees of ambiguous strings of* G *is* $\mathcal{P}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{S}(G_b), \exists \beta_b \in \mathcal{S}(G_b) : \alpha_b \neq \beta_b, h(\alpha_b) = h(\beta_b)\}$. *The set of parse trees of unambiguous strings of* G *is* $\mathcal{P}^u(G) = \mathcal{S}(G_b) \setminus \mathcal{P}^a(G)$.

Example 1. Below is an example grammar (1) together with its bracketed version (2). The string aaa has two parse trees, $\langle_1\langle_2\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\rangle_2\rangle_1$, and is therefore ambiguous.

$$1 : S \rightarrow A, \qquad 2 : A \rightarrow AA, \qquad 3 : A \rightarrow a \tag{1}$$

$$1 : S \rightarrow \langle_1 A\rangle_1,\ 2 : A \rightarrow \langle_2 AA\rangle_2,\ 3 : A \rightarrow \langle_3 a\rangle_3 \tag{2}$$

We call the set of the smallest possible ambiguous sentential forms of G the *ambiguous core* of G. These are the ambiguous sentential forms that cannot be derived from other sentential forms that are already ambiguous. Their parse trees are the smallest indicators of the ambiguities in G.

Definition 2. *The set of parse trees of the* ambiguous core *of a grammar G is* $\mathcal{C}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{P}^a(G),\ \neg\exists\beta_b \in \mathcal{P}^a(G) : \beta_b \Longrightarrow_b \alpha_b\}$

From $\mathcal{C}^a(G)$ we can obtain $\mathcal{P}^a(G)$ by adding all sentential forms reachable with $\Longrightarrow_b$. And since $\mathcal{C}^a(G) \subseteq \mathcal{P}^a(G)$ we get the following Lemma:

Lemma 1. *A grammar G is ambiguous iff $\mathcal{C}^a(G)$ is non-empty.*

Similar to $\mathcal{P}^u(G)$, we define the complement of $\mathcal{C}^a(G)$ as $\mathcal{C}^u(G) = \mathcal{S}(G_b)\backslash\mathcal{C}^a(G)$, for which holds that $\mathcal{P}^u(G) \subseteq \mathcal{C}^u(G)$.

Example 2. The two parse trees $\langle_1\langle_2\langle_2 AA\rangle_2 A\rangle_2\rangle_1$ and $\langle_1\langle_2 A\langle_2 AA\rangle_2\rangle_2\rangle_1$, of the ambiguous sentential form AAA, are in the ambiguous core of Grammar (1).

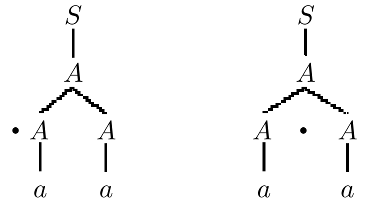
2.4 Positions

A *position* in a sentential form is an element in $V_b^* \times V_b^*$. The position (α_b, β_b) is written as $\alpha_b{\bullet}\beta_b$. $\mathsf{pos}(G_b)$ is the set of all positions in strings of $\mathcal{S}(G_b)$, defined as $\{\alpha_b{\bullet}\beta_b \mid \alpha_b\beta_b \in \mathcal{S}(G_b)\}$.

Every position in $\mathsf{pos}(G_b)$ is a position in a parse tree, and corresponds to an item of G. The item of a position can be identified by the closest enclosing $\langle_i$ and $\rangle_i$ pair around the dot, considering balancing. For positions with the dot at the beginning or the end we introduce two special items ${\bullet}S$ and $S{\bullet}$.

We use the function item to map a position to its item. It is defined by $\mathsf{item}(\gamma_b{\bullet}\delta_b) = A \rightarrow \alpha'{\bullet}\beta'$ iff $\gamma_b{\bullet}\delta_b = \eta_b\ \langle_i\, \alpha_b{\bullet}\beta_b\, \rangle_i\, \theta_b$, $A \rightarrow \langle_i\alpha'\beta'\rangle_i \in P_b$, $\alpha' \Longrightarrow_b^* \alpha_b$ and $\beta' \Longrightarrow_b^* \beta_b$, $\mathsf{item}({\bullet}\alpha_b) = {\bullet}S$, and $\mathsf{item}(\alpha_b{\bullet}) = S{\bullet}$. Another function items returns the set of items used at all positions in a parse tree. It is defined as $\mathsf{items}(\alpha_b) = \{A \rightarrow \alpha{\bullet}\beta \mid \exists\gamma_b{\bullet}\delta_b : \gamma_b\delta_b = \alpha_b,\ A \rightarrow \alpha{\bullet}\beta = \mathsf{item}(\gamma_b{\bullet}\delta_b)\}$.

Example 3. The following shows the parse tree representations of the positions $\langle_1\langle_2{\bullet}\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3{\bullet}\langle_3 a\rangle_3\rangle_2\rangle_1$. We see that the first position is at item $A \rightarrow {\bullet}AA$ and the second is at $A \rightarrow A{\bullet}A$.



The function $\mathsf{proditems}$ maps a production rule to the set of all its items. It is defined as $\mathsf{proditems}(A \to \alpha) = \{A \to \beta\bullet\gamma \mid \beta\gamma = \alpha\}$. If a production rule is used to construct a parse tree, then all its items occur at one or more positions in the tree.

Lemma 2. $\forall \alpha_b\langle_i\beta_b\rangle_i\gamma_b \in \mathcal{S}(G_b) : \exists A\to\delta \in P$: $\mathsf{pid}(A\to\delta)=i$, $\mathsf{proditems}(A\to\delta) \subseteq \mathsf{items}(\alpha_b\langle_i\beta_b\rangle_i\gamma_b)$.

2.5 Automata

An *automaton* A is a 5-tuple (Q, Σ, R, Q_s, Q_f) where Q is the set of *states*, Σ is the input alphabet, R in $Q \times \Sigma \times Q$ is the set of *rules* or *transitions*, $Q_s \subseteq Q$ is the set of *start states*, and $Q_f \subseteq Q$ is the set of *final states*. A transition (q_0, a, q_1) is written as $q_0 \overset{a}{\longmapsto} q_1$. The language of an automaton is the set of strings read on all paths from a start state to an end state. Formally, $\mathcal{L}(A) = \{\alpha \mid \exists q_s \in Q_s,\, q_f \in Q_f : q_s \overset{\alpha}{\longmapsto}{}^* q_f\}$.

3 Regular Unambiguity Test

This section introduces the Regular Unambiguity (RU) Test [11] by Schmitz. The RU Test is an approximative test for the existence of two parse trees for the same string, allowing only false positives.

3.1 Position Automaton

The basis of the Regular Unambiguity Test is a *position automaton*, which describes all strings in $\mathcal{S}(G_b)$. The states of this automaton are the positions in $\mathsf{pos}(G_b)$. The transitions are labeled with elements from V_b.

Definition 3. *The position automaton*[1] $\Gamma(G)$ *of a grammar* G *is the tuple* (Q, V_b, R, Q_s, Q_f), *where*

- $Q = \mathsf{pos}(G_b)$,
- $R = \{\alpha_b\bullet X_b\beta_b \overset{X_b}{\longmapsto} \alpha_b X_b\bullet\beta_b \mid \alpha_b X_b\beta_b \in \mathcal{S}(G_b)\}$,
- $Q_s = \{\bullet\alpha_b \mid \alpha_b \in \mathcal{S}(G_b)\}$,
- $Q_f = \{\alpha_b\bullet \mid \alpha_b \in \mathcal{S}(G_b)\}$.

There are three types of transitions: *derives* with labels in $T_\langle$, *reduces* with labels in $T_\rangle$, and *shifts* of terminals and nonterminals in V. The symbols read on a path through $\Gamma(G)$ describe a parse tree of G. Thus, $\mathcal{L}(\Gamma(G)) = \mathcal{S}(G_b)$.

$\Gamma(G)$ contains a unique subgraph for each string in $\mathcal{S}(G_b)$. The string read by a subgraph can be identified by the positions on the nodes of the subgraph. Every position dictates the prefix read up until its node, and the postfix required to reach the end state of its subgraph. Therefore, every path that corresponds to a string in $\mathcal{L}(\Gamma(G))$ must pass all positions of that string.

[1] We modified the original definition of the position automaton to be able to explain our extensions more clearly. This does not essentially change the RU Test and NU Test however, since their only requirement on $\Gamma(G)$ is that it defines $\mathcal{S}(G_b)$.

Lemma 3. $\forall \alpha_b, \beta_b : \alpha_b \bullet \beta_b \in Q \Leftrightarrow \alpha_b\beta_b \in \mathcal{L}(\Gamma(G))$.

A grammar G is ambiguous iff two paths exist through $\Gamma(G)$ that describe different parse trees in $\mathcal{P}^a(G)$ — strings in $\mathcal{S}(G_b)$ — of the same string in $\mathcal{S}(G)$. We call such two paths an *ambiguous path pair.*

Example 4. The following shows the first part of the position automaton of the grammar from Example 1. It shows paths for parse trees S, $\langle_1 A\rangle_1$ and $\langle_1\langle_3 a\rangle_3\rangle_1$.

$$\begin{tikzcd}
{} \arrow[d] & \\
\boxed{\bullet S} \arrow[r, "S"] & \boxed{S\bullet}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[d] & & & \\
\boxed{\bullet\langle_1 A\rangle_1} \arrow[r, "\langle_1"] & \boxed{\langle_1\bullet A\rangle_1} \arrow[r, "A"] & \boxed{\langle_1 A\bullet\rangle_1} \arrow[r, "\rangle_1"] & \boxed{\langle_1 A\rangle_1\bullet}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[d] & & & & & \\
\boxed{\bullet\langle_1\langle_3 a\rangle_3\rangle_1} \arrow[r, "\langle_1"] & \boxed{\langle_1\bullet\langle_3 a\rangle_3\rangle_1} \arrow[r, "\langle_3"] & \boxed{\langle_1\langle_3\bullet a\rangle_3\rangle_1} \arrow[r, "a"] & \boxed{\langle_1\langle_3 a\bullet\rangle_3\rangle_1} \arrow[r, "\rangle_3"] & \boxed{\langle_1\langle_3 a\rangle_3\bullet\rangle_1} \arrow[r, "\rangle_1"] & \boxed{\langle_1\langle_3 a\rangle_3\rangle_1\bullet}
\end{tikzcd}$$

3.2 Approximated Position Automaton

If G has an infinite number of parse trees, the position automaton is also of infinite size. Checking it for ambiguous path pairs would take forever. Therefore the position automaton is approximated using equivalence relations on the positions. The approximated position automaton has equivalence classes of positions for its states. For every transition between two positions in the original automaton a new transition with the same label then exists between the equivalence classes that the positions are in. If an equivalence relation is used that yields a finite set of equivalence classes, the approximated automaton can be checked for ambiguous path pairs in finite time.

Definition 4. *Given an equivalence relation $\equiv$ on positions, the approximated position automaton $\Gamma_{\equiv}(G)$ of the automaton $\Gamma(G) = (Q, V_b, R, Q_s, Q_f)$, is the tuple $(Q_{\equiv}, V_b', R_{\equiv}, \{q_s\}, \{q_f\})$ where*

- $Q_{\equiv} = Q/{\equiv} \cup \{q_s, q_f\}$, *where $Q/{\equiv}$ is the set of non-empty equivalence classes over* $\mathsf{pos}(G_b)$ *modulo $\equiv$, defined as $\{[\alpha_b \bullet \beta_b]_{\equiv} \mid \alpha_b \bullet \beta_b \in Q\}$,*
- $R_{\equiv} = \{[q_0]_{\equiv} \overset{X_b}{\longmapsto} [q_1]_{\equiv} \mid q_0 \overset{X_b}{\longmapsto} q_1 \in R\} \cup \{q_s \overset{\varepsilon}{\longmapsto} [q]_{\equiv} \mid q \in Q_s\} \cup \{[q]_{\equiv} \overset{\varepsilon}{\longmapsto} q_f \mid q \in Q_f\}$,
- *q_s and q_f are respectively the start and final state.*

The paths through $\Gamma_{\equiv}(G)$ describe an overapproximation of the set of parse trees of G, thus $\mathcal{L}(\Gamma(G)) \subseteq \mathcal{L}(\Gamma_{\equiv}(G))$. So if no ambiguous path pair exists in $\Gamma_{\equiv}(G)$, grammar G is unambiguous. But if there is an ambiguous path pair, it is unknown if its paths describe real parse trees of G or approximated ones. In this case we say G is *potentially ambiguous.*

The item_0 Equivalence Relation. Checking for ambiguous paths in finite time also requires an equivalence relation with which $\Gamma_{\equiv}(G)$ can be built in finite

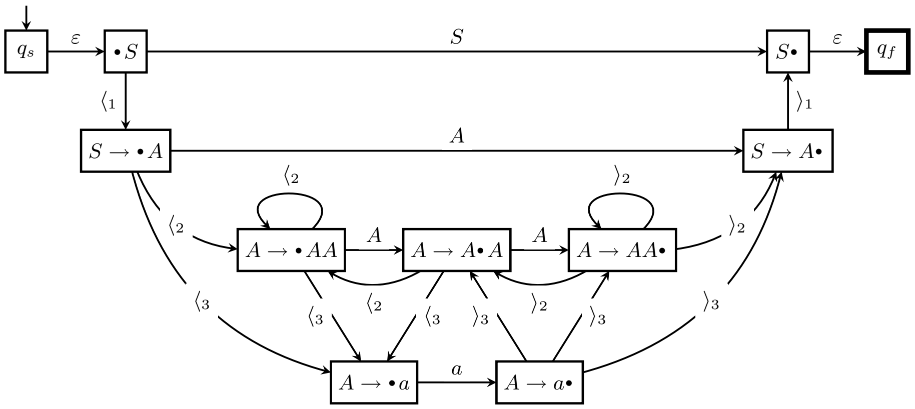


Fig. 1. The item_0 position automaton of the grammar of Example 1

time. A relation like that should enable the construction of the equivalence classes without enumerating all positions in $\mathsf{pos}(G_b)$. A simple but useful equivalence relation with this property is the item_0 relation [11]. Two positions are equal modulo item_0 if they are both at the same item.

Definition 5. $\alpha_b\bullet\beta_b \; \mathsf{item}_0 \; \gamma_b\bullet\delta_b$ *iff* $\mathsf{item}(\alpha_b\bullet\beta_b) = \mathsf{item}(\gamma_b\bullet\delta_b)$.

Intuitively the item_0 position automaton $\Gamma_{\mathsf{item}_0}(G)$ of a grammar resembles that grammar's LR(0) parse automaton [10]. The nodes are the LR(0) items of the grammar and the X and $\rangle$ edges correspond to the shift and reduce actions in the LR(0) automaton. The $\langle$ edges do not have LR(0) counterparts. Every item with the dot at the beginning of a production of S is a start node, and every item with the dot at the end of a production of S is an end node.

The difference between an LR(0) automaton and an item_0 position automaton is in the reductions. $\Gamma_{\mathsf{item}_0}(G)$ has reduction edges to every item that has the dot after the reduced nonterminal, while an LR(0) automaton jumps to a different state depending on the symbol that is at the top of the parse stack. As a result, a certain path through $\Gamma_{\mathsf{item}_0}(G)$ with a $\langle_i$ transition from $A \to \alpha\bullet B\gamma$ does not necessarily need to have a matching $\rangle_i$ transition to $A \to \alpha B\bullet\gamma$.

Example 5. Figure 1 shows the item_0 position automaton of the grammar of Example 1. Strings $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$ and $\langle_1\langle_3 a\rangle_3\rangle_1$ form an ambiguous path pair.

The item_0 relation can be combined with the look_k relation to get position automata that resemble LR(k) automata. This results in the item_k relation, which groups positions if they are equal modulo both item_0 and look_k. Two positions are equal modulo look_k if their first k terminal symbols after the dot are identical.

Definition 6. $\alpha_b\bullet\beta_b \; \mathsf{look}_k \; \gamma_b\bullet\delta_b$ *iff* $(\exists u, v, w : h(\beta_b) = uv,\ h(\delta_b) = uw,\ |u| = k)$ $\vee$ $(h(\beta_b) = h(\delta_b) \wedge |h(\beta_b)| < k)$.

The RU Test becomes more precise with increasing k values, because then $\Gamma_{\mathsf{item}_k}(G)$ better approximates $\mathcal{S}(G)$.

3.3 Position Pair Automaton

The existence of ambiguous path pairs in a position automaton can be checked with a *position pair automaton*, in which every state is a pair of states from the position automaton. Transitions between pairs are described using the *mutual accessibility relation* ma.

Definition 7. *The* regular position pair automaton $\Pi^R_{\equiv}(G)$ *of* $\Gamma_{\equiv}(G)$ *is the tuple* $(Q^2_{\equiv}, V'^2_b, \mathsf{ma}, q^2_s, q^2_f)$*, where* ma *over* $Q^2_{\equiv} \times V'^2_b \times Q^2_{\equiv}$*, denoted by* $\rightrightarrows$ *, is the union of the following subrelations:*

$$\begin{aligned}
\mathsf{maDl} &= \{(q_0, q_1) \xrightarrow{(\langle_i, \varepsilon)} (q_2, q_1) \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{maDr} &= \{(q_0, q_1) \xrightarrow{(\varepsilon, \langle_i)} (q_0, q_3) \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{maS} &= \{(q_0, q_1) \xrightarrow{(X, X)} (q_2, q_3) \mid q_0 \overset{X}{\longmapsto} q_2 \wedge q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{maRl} &= \{(q_0, q_1) \xrightarrow{(\rangle_i, \varepsilon)} (q_2, q_1) \mid q_0 \overset{\rangle_i}{\longmapsto} q_2\},\\
\mathsf{maRr} &= \{(q_0, q_1) \xrightarrow{(\varepsilon, \rangle_i)} (q_0, q_3) \mid q_1 \overset{\rangle_i}{\longmapsto} q_3\}.
\end{aligned}$$

Every path through this automaton from q^2_s to q^2_f describes two paths through $\Gamma_{\equiv}(G)$ that shift the same symbols. The language of $\Pi^R_{\equiv}(G)$ is thus a set of pairs of strings. A path indicates an ambiguous path pair if its two bracketed strings are different, but equal under the homomorphism h. Because $\mathcal{L}(\Gamma_{\equiv}(G))$ is an over-approximation of $\mathcal{S}(G_b)$, $\mathcal{L}(\Pi^R_{\equiv}(G))$ contains at least all ambiguous path pairs through $\Gamma(G)$.

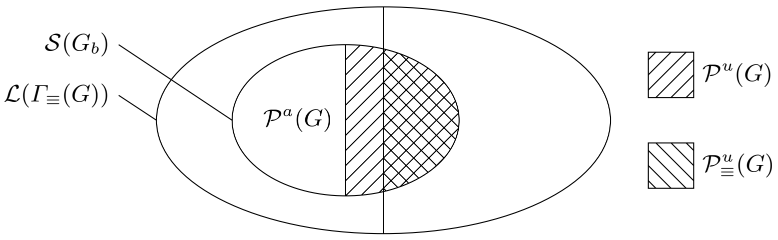
Lemma 4. $\forall \alpha_b, \beta_b \in \mathcal{P}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^R_{\equiv}(G))$.

4 Finding Parse Trees of Unambiguous Strings

The Regular Unambiguity Test described in the previous section can conservatively detect the unambiguity of a given grammar. If it finds no ambiguity we are done, but if it finds potential ambiguity this report is not detailed enough to be useful. In this section we show how the RU Test can be extended to identify parse trees of unambiguous strings. These will form the basis of more detailed ambiguity reports, as we will see in Section 5.

4.1 Unused Positions

From the states of $\Gamma_{\equiv}(G)$ that are not used on ambiguous path pairs, we can identify parse trees of unambiguous strings. For this we use the fact that every bracketed string that represents a parse tree of G must pass all its positions on its path through $\Gamma(G)$ (Lemma 3). Therefore, all positions in states of $\Gamma_{\equiv}(G)$ that are not used by any ambiguous path pair through $\Pi^R_{\equiv}(G)$ are positions in parse trees of unambiguous strings.



$\mathcal{S}(G_b)$
$\mathcal{L}(\Gamma_{\equiv}(G))$
$\mathcal{P}^a(G)$
$\mathcal{P}^u(G)$
$\mathcal{P}^u_{\equiv}(G)$

Fig. 2. Venn diagram showing the relationship between $\mathcal{S}(G_b)$ and $\mathcal{L}(\Gamma_{\equiv}(G))$. The vertical lines divide both sets in two: their parse trees of ambiguous strings (left) and parse trees of unambiguous strings (right).

Definition 8. *The set of states of $\Gamma_{\equiv}(G)$ that are used on ambiguous path pairs through $\Pi^R_{\equiv}(G)$ is $Q^a_{\equiv} =$*

$$\{q_0, q_1 \mid \exists \alpha_b, \beta_b, \alpha'_b, \beta'_b : \alpha_b\beta_b \neq \alpha'_b\beta'_b,\ q^2_s \xrightarrow{(\alpha_b,\alpha'_b)}{}^* (q_0, q_1) \xrightarrow{(\beta_b,\beta'_b)}{}^* q^2_f\}.$$

The set of states not used on ambiguous path pairs is $Q^u_{\equiv} = Q_{\equiv} \setminus Q^a_{\equiv}$.

Definition 9. *The set of parse trees of unambiguous strings of G that are identifiable with $\equiv$, is $\mathcal{P}^u_{\equiv}(G) = \{\alpha_b\beta_b \mid \exists q \in Q^u_{\equiv} : \alpha_b\bullet\beta_b \in q\}$.*

This set is always a subset of $\mathcal{P}^u(G)$, as illustrated by Fig. 2.

Theorem 1. *For all equivalence relations $\equiv$, $\mathcal{P}^u_{\equiv}(G) \subseteq \mathcal{P}^u(G)$.*

The positions in the states in $Q^a_{\equiv}$ and $Q^u_{\equiv}$ thus identify parse trees of respectively potentially ambiguous strings and certainly unambiguous strings. However, iterating over all positions in $\mathsf{pos}(G)$ is infeasible if this set is infinite. The used equivalence relation should therefore allow the direct identification of parse trees from the states of $\Gamma_{\equiv}(G)$.

For instance, a state in $\Gamma_{\mathsf{item}_0}(G)$ represents all parse trees in which a particular item appears. With this information we can identify production rules that only appear in parse trees in $\mathcal{P}^u_{\equiv}(G)$, as we will show in the next section.

4.2 Join Points

Gathering $Q^a_{\equiv}$ is also impossible in practice because it requires the inspection of all paths through $\Gamma_{\equiv}(G)$, of which there can be infinitely many. We therefore need a definition that can be calculated in finite time. For this we use the notion of *join points*. These are the points in $\Pi^R_{\equiv}(G)$ where we see that two different paths through $\Gamma_{\equiv}(G)$ potentially come together in the same state.

Definition 10. *The set of join points J in $\Pi^R_{\equiv}(G)$, over $Q^2_{\equiv} \times Q^2_{\equiv}$, is defined as $J = \{((q_0, q_1), (q_2, q_2)) \mid (q_0, q_1) \xrightarrow{(X_b,X'_b)} (q_2, q_2),\ q_0 \neq q_1,\ X_b \in T_{\rangle} \vee X'_b \in T_{\rangle}\}$.*

With J we then define the following alternative to $Q^a_{\equiv}$:

Definition 11. *The set of states in $\Gamma_{\equiv}(G)$ that are used in pairs of $\Pi^R_{\equiv}(G)$ that can reach, or can be reached by, a join point, is $Q^{a\prime}_{\equiv} =$*
$\{q_0, q_1 \mid \exists (p_0, p_1) \in J : q^2_s \rightrightarrows^* (q_0, q_1) \rightrightarrows^* p_0 \vee p_1 \rightrightarrows^* (q_0, q_1) \rightrightarrows^* q^2_f\}$.

This is a safe over-approximation of $Q^a_{\equiv}$, because all ambiguous path pairs through $\Gamma_{\equiv}(G)$ will eventually join in a certain state. It can be calculated by iterating over the edges of $\Pi^R_{\equiv}(G)$ to collect J, and then computing the images of the join points through ma^* and $(\mathsf{ma}^{-1})^*$. Both steps are linear in the number of edges in $\Pi^R_{\equiv}(G)$ (see [14] Chapter 2), which is worst case $\mathcal{O}(|Q_{\equiv}|^4)$.

5 Harmless Production Rules

In this section we show how we can use $Q^a_{\equiv}$ to identify production rules that do not contribute to the ambiguity of G. These are the production rules that can never occur in parse trees of ambiguous strings. We call them *harmless production rules.*

5.1 Finding Harmless Production Rules

A production rule is certainly harmless if it is only used in parse trees in $\mathcal{P}^u_{\equiv}(G)$. We should therefore search for productions that are never used on ambiguous path pairs of $\Pi^R_{\equiv}(G)$ that describe valid parse trees in G. We can find them by looking at the items of the positions in the states of $Q^a_{\equiv}$. If not all items of a production rule are used then the rule cannot be used in a valid string in $\mathcal{P}^a(G)$ (Lemma 2), and we know it is harmless.

Definition 12. *The set of items used on the ambiguous path pairs through $\Pi^R_{\equiv}(G)$ is $I^a_{\equiv} = \{A \to \alpha \bullet \beta \mid \exists q \in Q^a_{\equiv} : \exists \gamma_b \bullet \delta_b \in q : A \to \alpha \bullet \beta = \mathsf{item}(\gamma_b \bullet \delta_b)\}$.*

With it we can identify production rules of which all items are used:

Definition 13. *The set of potentially harmful production rules of G, identifiable from $\Pi^R_{\equiv}(G)$, is $P_{\mathrm{hf}} = \{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq I^a_{\equiv}\}$.*

Because of the approximation it is uncertain whether or not they can really be used to form valid parse trees of ambiguous strings. Nevertheless, all the other productions in P will certainly not appear in parse trees of ambiguous strings.

Definition 14. *The set of harmless production rules of G, identifiable from $\Pi^R_{\equiv}(G)$, is $P_{\mathrm{hl}} = P \setminus P_{\mathrm{hf}}$.*

Theorem 2. *$\forall p \in P_{\mathrm{hl}} : \neg\exists \alpha_b \langle_i \beta_b \rangle_i \gamma_b \in \mathcal{P}^a(G) : i = \mathsf{pid}(p)$.*

Example 6 in Section 7 shows finding P_{hl} for a small grammar.

5.2 Complexity

Finding P_{hf} comes down to building $\Pi^R_{\equiv}(G)$, finding $Q^{a'}_{\equiv}$, and enumerating all positions in all classes in $Q^{a'}_{\equiv}$ to find $I^{a}_{\equiv}$. The number of these classes is finite, but the number of positions might not be. It would therefore be convenient if the definition of the chosen equivalence relation could be used to collect $I^{a}_{\equiv}$ in finitely many steps. With the item_0 relation this is possible, because all the positions in a class are all in the same item.

Constructing $\Pi^R_{\mathsf{item}_0}(G)$ can be done in $\mathcal{O}(|G|^2)$ (see [11]), where $|G|$ is the number of items of G. After that, $Q^{a'}_{\mathsf{item}_0}$ can be gathered in $\mathcal{O}(|G|^4)$, because $|Q_{\mathsf{item}_0}|$ is linear with $|G|$. Since this is the most expensive step, the worst case complexity of finding P_{hf} with item_0 is therefore also $\mathcal{O}(|G|^4)$.

5.3 Grammar Reconstruction

Finding P_{hl} can be very helpful information for the grammar developer. Also, P_{hf} represents a smaller grammar that can be checked again more easily to find the true origins of ambiguity. However, the reachability and productivity properties of this smaller grammar might be violated because of the removed productions in P_{hl}. To restore these properties we have to introduce new terminals and productions, and a new start symbol. We must prevent introducing new ambiguities in this process.

From P_{hf} we can create a new grammar G' by constructing:

1. The set of defined nonterminals of P_{hf}: $N_{\mathrm{def}} = \{A \mid A \to \alpha \in P_{\mathrm{hf}}\}$.
2. The used but undefined nonterminals of P_{hf}:
   $N_{\mathrm{undef}} = \{B \mid A \to \alpha B \beta \in P_{\mathrm{hf}}\} \backslash N_{\mathrm{def}}$.
3. The unproductive nonterminals:
   $N_{\mathrm{unpr}} = \{A \mid A \in N_{\mathrm{def}}, \neg\exists u : A \Longrightarrow^* u$ using only productions in $P_{\mathrm{hf}}\}$.
4. The start symbols of P_{hf}: $S_{\mathrm{hf}} = \{A \mid A \in N_{\mathrm{def}}, \neg\exists B \to \beta A \gamma \in P_{\mathrm{hf}}\}$.
5. New terminal symbols t_A, b_A, e_A for each nonterminal A.
6. New productions to define a new start-symbol S':
   $P_{S'} = \{S' \to b_A A e_A \mid A \in S_{\mathrm{hf}}\}$.
7. Productions to complete the unproductive and undefined nonterminals:
   $P' = P_{\mathrm{hf}} \cup P_{S'} \cup \{A \to t_A \mid A \in N_{\mathrm{undef}} \cup N_{\mathrm{unpr}}\}$.
8. The new set of terminal symbols: $T' = \{a \mid A \to \beta a \gamma \in P'\}$.
9. Finally, the new grammar: $G' = (N_{\mathrm{def}} \cup N_{\mathrm{undef}} \cup \{S'\}, T', P', S')$.

Surrounding the nonterminals in S_{hf} with unique terminals at step 6 prevents the new rules of S' from being ambiguous with each other. The unique terminals at step 7 make sure we do not create new parse trees for existing strings in $\mathcal{L}(G)$.

6 Noncanonical Unambiguity Test

In this section we explain the Noncanonical Unambiguity (NU) Test [11], which is more precise than the Regular Unambiguity Test. It enables the identification of a larger set of irrelevant parse trees, namely the ones in $\mathcal{C}^u(G)$. From these we can also identify a larger set of harmless production rules and tree patterns.

6.1 Improving the Regular Unambiguity Test

The regular position pair automaton described in Section 3 checks all pairs of paths through a position automaton for ambiguity. However, it also checks some spurious paths that are unnecessary for identifying the ambiguity of a grammar.

These are the path pairs that derive the same unambiguous substring for a certain nonterminal. We can ignore these paths because in this situation there are also two paths in which the nonterminal was shifted instead of derived. For instance, consider paths $\langle_1\langle_2\langle_3 a\rangle_3\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\beta_b\rangle_2\rangle_1$. If they form a pair in $\mathcal{L}(\Pi^R_\equiv(G))$ then the shorter paths $\langle_1\langle_2 A\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2 A\beta_b\rangle_2\rangle_1$ will too (considering $A \to \langle_3 a\rangle_3 \in P_b$). In addition, if the first two paths form an ambiguous path pair, then these latter two will also, because $\langle_3 a\rangle_3$ does not contribute to the ambiguity. In this case we prefer the latter paths because they describe smaller parse trees than the first paths.

6.2 Noncanonical Position Pair Automaton

To avoid common unambiguous substrings we should only allow path pairs to take identical reduce transitions if they do not share the same substring since their last derives. To keep track of this property we add two extra boolean flags c_0 and c_1 to the position pairs. These flags tell for each position in a pair whether or not its path has been in conflict with the other, meaning it has taken different reduce steps as the other path since its last derive. A value of 0 means this has not occurred yet, and we are thus allowed to ignore an identical reduce transition.

All start pairs have both flags set to 0, and every derive step resets the flag of a path to 0. The flag is set to 1 if a path takes a *conflicting* reduce step, which occurs if the other path does not follow this reduce at the same time (for instance $\rangle_2$ in the parse trees $\langle_1\langle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$). We use the predicate confl (called eff by Schmitz) to identify a situation like that.

$$\mathsf{confl}(q,i) = \exists u \in T^*_\langle : q \overset{u}{\longmapsto}{}^* q_f \vee (\exists q' \in Q_\equiv, X \in V \cup T_\rangle : X \neq \rangle_i, q \overset{uX}{\longmapsto}{}^+ q') \quad (3)$$

It tells whether there is another shift or reduce transition other than $\rangle_i$ possible from q, ignoring $\langle$ steps, or if q is at the end of the automaton.

Definition 15. *The* noncanonical *position pair automaton $\Pi^N_\equiv(G)$ of $\Gamma_\equiv(G)$ is the tuple $(Q^p, V_b'^2, \mathsf{nma}, (q_s,0)^2, (q_f,1)^2)$, where $Q^p = (Q_\equiv \times \mathbb{B})^2$, and* nma *over $Q^p \times V_b'^2 \times Q^p$ is the* noncanonical *mutual accessibility relation, defined as the union of the following subrelations:*

$$\begin{aligned}
\mathsf{nmaDl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\langle_i,\varepsilon)} (q_2,q_1)0,c_1 \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{nmaDr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\langle_i)} (q_0,q_3)c_0,0 \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{nmaS} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(X,X)} (q_2,q_3)c_0,c_1 \mid q_0 \overset{X}{\longmapsto} q_2,\ q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{nmaCl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\varepsilon)} (q_2,q_1)1,c_1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ \mathsf{confl}(q_1,i)\},\\
\mathsf{nmaCr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\rangle_i)} (q_0,q_3)c_0,1 \mid q_1 \overset{\rangle_i}{\longmapsto} q_3,\ \mathsf{confl}(q_0,i)\},\\
\mathsf{nmaR} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\rangle_i)} (q_2,q_3)1,1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ q_1 \overset{\rangle_i}{\longmapsto} q_3,\ c_0 \vee c_1\}.
\end{aligned}$$

As with $\Pi^R_{\equiv}(G)$, the language of $\Pi^N_{\equiv}(G)$ describes ambiguous path pairs through $\Gamma_{\equiv}(G)$. The difference is that $\mathcal{L}(\Pi^N_{\equiv}(G))$ does not include path pairs without conflicting reductions. Therefore $\mathcal{L}(\Pi^N_{\equiv}(G)) \subseteq \mathcal{L}(\Pi^R_{\equiv}(G))$. Nevertheless, $\Pi^N_{\equiv}(G)$ does at least describe all the *core* parse trees in $\mathcal{C}^a(G)$:

Theorem 3. $\forall \alpha_b, \beta_b \in \mathcal{C}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^N_{\equiv}(G))$.

The Theorem shows that if G is ambiguous — that is $\mathcal{C}^a(G)$ is non-empty — $\mathcal{L}(\Pi^N_{\equiv}(G))$ is also non-empty. This means that if $\mathcal{L}(\Pi^N_{\equiv}(G))$ is empty, G is unambiguous.

6.3 Effects on Filtering Parse Trees and Production Rules

The new nma relation enables our parse tree identification algorithm of Section 4 to potentially identify a larger set of irrelevant parse trees, namely $\mathcal{C}^u(G)$. These trees might be ambiguous, but this is not a problem because we are interested in finding the trees of the smallest possible sentential forms of G, namely the ones in $\mathcal{C}^a(G)$.

Definition 16. *Given $Q^u_{\equiv}$ from $\Pi^N_{\equiv}(G)$, the set of parse trees not in the ambiguous core of G, identifiable with $\equiv$, is $\mathcal{C}^u_{\equiv}(G) = \{\alpha_b\beta_b \mid \exists q \in Q^u_{\equiv}, \alpha_b \bullet \beta_b \in q\}$.*

Theorem 4. *For all equivalence relations $\equiv$, $\mathcal{C}^u_{\equiv}(G) \subseteq \mathcal{C}^u(G)$.*

The set of harmless production rules that can be identified with $\Pi^N_{\equiv}(G)$ is also potentially larger. It might include rules that can be used in parse trees of ambiguous strings, but not in parse trees in $\mathcal{C}^a(G)$. Therefore they are not vital for the ambiguity of G.

Definition 17. *Given $Q^a_{\equiv}$ and $I^a_{\equiv}$ from $\Pi^N_{\equiv}(G)$, the set of harmless productions of G, identifiable from $\Pi^N_{\equiv}(G)$, is $P'_{\text{hl}} = P \setminus \{A \rightarrow \alpha \mid \mathsf{proditems}(A \rightarrow \alpha) \subseteq I^a_{\equiv}\}$.*

Theorem 5. $\forall p \in P'_{\text{hl}} : \neg\exists \alpha_b \langle_i \beta_b \rangle_i \gamma_b \in \mathcal{C}^a(G) : i = \mathsf{pid}(p)$.

7 Excluding Parse Trees Iteratively

Our approach for the identification of parse trees of unambiguous strings is most useful if applied in an iterative setting. By checking the remainder of the potentially ambiguous parse trees again, there is possibly less interference of the trees during approximation. This could result in less ambiguous path pairs in the position pair automaton. We could then exclude a larger set of parse trees and production rules.

Example 6. The grammar below (4) is unambiguous but needs two iterations of the NU Test with item_0 to detect this. At first, $\Pi^N_{\mathsf{item}_0}(G)$ contains only the ambiguous path pair $\langle_1\langle_4 c\rangle_4\rangle_1$ and $\langle_2\langle_5\langle_6 c\rangle_6\rangle_3\rangle_1$. The first path describes a valid parse tree, but the second does not. From $B \rightarrow \bullet Cb$ it derives to $C \rightarrow \bullet c$, but

Table 1. Excerpt from Results of prototype implementation

Grammar		Harmless rules			Time AMBER		Time CFGANALYZER	
Name	Rules	LR(0)	SLR(1)	LR(1)	Original	Filtered	Original	Filtered
SQL.1	79	65	65	65	28m26s	0.1s	17.6s	1.8s
Pascal.3	176	21	30	144	31.8s	0.0s	9.6s	1.3s
C.2	212	41	44	44	4.5h[1]	4.12s[1]	3.0h	1.1h
Java.1	349	56	70	74	25.0h[2]	22m52s[2]	48.9s	32.4s

[1]for sentences of length 7 (first ambiguity at length 13)
[2]for sentences of length 12 (first ambiguity at length 13)

from $C \to c\bullet$ it reduces to $A \to aC\bullet$. Therefore productions 2, 5 and 3 are only used partially, and they are thus harmless. After removing them and checking the reconstructed grammar again there are no ambiguous path pairs anymore.

$$1 : S \to A,\ \ 2 : S \to B,\ \ 3 : A \to aC,\ \ 4 : A \to c,\ \ 5 : B \to Cb,\ \ 6 : C \to c \quad (4)$$

We can gain even higher precision by choosing a new equivalence relation with each iteration. If with each step $\Gamma_{\equiv}(G)$ better approximates $\mathcal{S}(G_b)$, we might end up with only the parse trees in $\mathcal{P}^u(G)$. Unfortunately, the ambiguity problem is undecidable, and this process does not necessarily have to terminate. There might be an infinite number of equivalence relations that yield a finite number of equivalence classes. Or at some point we might need to resort to equivalence relations that do not yield a finite graph. Therefore, the iteration has to stop at a certain moment, and we can continue with an exhaustive search of the remaining parse trees.

In the end this exhaustive searching is the most practical, because it can point out the exact parse trees of ambiguous strings. A drawback of this approach is its exponential complexity. Nevertheless, excluding sets of parse trees beforehand can reduce its search space significantly, as we see in the next section.

8 Prototype Results

In [3] we tested a prototype implementation of our approach on a collection of programming language grammars. From unambiguous grammars of SQL, Pascal, C and Java, we created 5 ambiguous versions for each language. For each grammar we tested the number of harmless production rules we could find with the NU Test, using different equivalence relations. Columns 3-5 of Table 1 show the results of these tests for a selection of 4 ambiguous grammars. Similar numbers of harmless rules could be found for the other grammars.

Columns 7-9 show the effect that the removal of the harmless productions had on the run-time of the two exhaustive derivation generators AMBER [13] and CFGANALYZER [1]. They mention the time needed to find the first ambiguous derivation of a grammar before and after filtering with LR(1). We see significant reductions in run-time, sometimes orders of magnitude. For the other grammars we witnessed similar effects.

9 Conclusions

We showed how the Regular Unambiguity Test and Noncanonical Unambiguity Test can be extended to conservatively identify parse trees of unambiguous strings. From these trees we can identify production rules that do not contribute to the ambiguity of the grammar. This information is already very useful for a grammar developer, but it can also be used to significantly reduce the search space of other ambiguity detection methods.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Basten, H.J.S.: Tracking down the origins of ambiguity in context-free grammars. Tech. Rep. SEN-1005, CWI, Amsterdam, The Netherlands (2010)
3. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM, New York (2010)
4. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Science of Computer Programming 75(3), 176–191 (2010)
5. Cantor, D.G.: On the ambiguity problem of Backus systems. Journal of the ACM 9(4), 477–479 (1962)
6. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P. (ed.) Computer Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
7. Floyd, R.W.: On ambiguity in phrase structure languages. Communications of the ACM 5(10), 526–534 (1962)
8. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
10. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
11. Schmitz, S.: Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 692–703. Springer, Heidelberg (2007)
12. Schmitz, S.: An experimental ambiguity detection tool. Science of Computer Programming 75(1-2), 71–84 (2010)
13. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), `http://accent.compilertools.net/Amber.html`
14. Sippu, S., Soisalon-Soininen, E.: Parsing theory. Languages and parsing, vol. 1. Springer, New York (1988)