Extracting and Binarizing probabilistic linear context-free rewriting systems

Bachelorarbeit
(revised version)

Lehrstuhl Grundlagen der Programmierung, Institut für Theoretische Informatik, Fakultät Informatik, Technische Universität Dresden

2015-07-12

Sebastian J. Mielke
Matrikelnr.: 3849072

Verantwortlicher Hochschullehrer Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer Dipl.-Inf. Tobias Denkinger

Aufgabenstellung

Regelextraktion und Binarisierung

Unter *Regelextraktion* versteht man das Gewinnen von Regeln (einer Grammatik) aus, üblicherweise von Linguisten erstellten, Baumkorpora. Von jedem Baum eines Baumkorpus lässt sich eine Ableitung in einer Grammatik ablesen, die Regeln der entsprechenden Ableitungen aller Bäume des Baumkorpus werden dann zu einer Grammatik vereint. Die Regelwahrscheinlichkeit geht aus der Anzahl der Regelvorkommen im Baumkorpus hervor. Siehe dazu [KS09; MS08].

Eine Regel ist binär, wenn nur maximal zwei Nichtterminale auf der rechten Regelseite vorkommen. Unter *Binarisierung* versteht man die Überführung einer Grammatik in eine Grammatik mit binären Regeln, die die gleiche (gewichtete) Sprache erzeugt. Siehe dazu [GS09; Góm+09].

Linear context-free rewriting systems

Natürliche Sprachen weisen Merkmale auf, die von kontextfreien Grammatiken nicht darstellbar sind, z.B. kann ein Teilsatz eine Lücke haben, in die vom Kontext abhängiger Inhalt eingefügt wird. Um allerdings die hohe Parsingkomplexität bei kontextsensitiven Grammatiken (nämlich PSPACE-complete) zu vermeiden, betrachtet man Formalismen, die diese Lücken zwar darstellen können, aber dennoch polynomielles Parsing zulassen. Man fasst solche Formalismen unter dem Begriff *mildly context-sensitive formalisms* zusammen. Dazu gehören z.B. head grammars, tree adjoining grammars, combinatory categorial grammars, linear indexed grammars, multiple context-free grammars, und minimalist grammars. *Linear context-free rewriting systems* (kurz: LCFRS) wurden von Vijay-Shanker, Weir, and Joshi [VWJ87] eingeführt, um mildly context-sensitive formalisms einheitlich darzustellen. Es hat sich herausgestellt, dass alle oben genannten (und noch einige weitere) Formalismen bzw. deren Frontsprachen eine kleinere oder die gleiche Sprachklasse wie LCFRS erzeugen [Sek+91; VWJ86; WJ88; Vij87; Mic01a; Mic01b]. Das Parsing von LCFRS ist daher von besonderer Bedeutung für die Verarbeitung natürlicher Sprache [Eva11].

Vanda und Vanda-Studio[1]

Vanda ist ein am Lehrstuhl für Grundlagen der Programmierung entwickeltes System zur Verarbeitung natürlicher Sprachen. Es besteht aus einer Bibliothek von allgemein spezifizierten Datenstrukturen und Algorithmen und einer Sammlung von spezialisierten Programmen für verschiedene Teilaufgaben in der Verarbeitung natürlicher Sprache.

Die graphische Oberfläche *Vanda Studio* bietet eine Nutzerschnittstelle, mit der solche Teilaufgaben zu einem Workflow kombiniert und auf verschiedene Datensätze (z.B. Textkorpora und Grammatiken) angewendet werden können. Neben den Programmen aus Vanda stehen dabei weitere, nicht am Lehrstuhl entwickelte, Programme zur Verfügung.

Aufgaben

Der Student soll probabilistische LCFRS formal einführen und einen Algorithmus zur Extraktion von Regeln aus einem Korpus von Konstituentenbäumen, deren Blattknoten eine zusätzliche Ordnung haben, angeben. Ebenso soll er einen Algorithmus zur Binarisierung von extrahierten probabilistischen LCFRS angeben; dabei sollen Heuristiken entwickelt werden, die den Fan-out der entstehenden probabilistischen LCFRS möglichst klein halten. Es soll gezeigt werden, dass der Extraktionsalgorithmus dem Yield des Korpus, aus dem die probabilistische LCFRS extrahiert wurde, eine Likelihood größer 0 zuordnet. Außerdem soll der Student zeigen, dass der Binarisierungsalgorithmus die generierte gewichtete Sprache erhält.

Die beiden oben genannten Algorithmen sollen in Vanda implementiert und in Vanda-Studio zugänglich gemacht werden. Die Baumkorpora werden dafür aus den Formaten des TIGER-[2] und des NeGra-Korpus[3] eingelesen. Der Student soll die Verteilung von Rang und Fan-out nach der Regelextraktion, nach der naiven Binarisierung (ohne die Heuristiken), und nach der Binarisierung mit den Heuristiken vergleichen.

Form

Die Arbeit muss den üblichen Standards wie folgt genügen: Die Arbeit muss in sich abgeschlossen sein und alle nötigen Definitionen und Referenzen enthalten. Die Struktur der Arbeit muss klar erkennbar sein, und der Leser soll gut durch die Arbeit geführt werden. Die Darstellung aller Begriffe und Verfahren soll mathematisch formal fundiert sein. Für jeden wichtigen Begriff sollen Beispiele angegeben werden, ebenso für die Abläufe der

[1] http://www.inf.tu-dresden.de/index.php?node_id=2550
[2] http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger.html
[3] http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/negra-corpus.html

beschriebenen Verfahren. Wo es angemessen ist, sollen Illustrationen die Darstellung vervollständigen. Schließlich sollen alle Lemmata und Sätze möglichst lückenlos bewiesen werden. Die Beweise sollen leicht nachvollziehbar dokumentiert sein.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst habe.

Dresden, 2015-07-12 Unterschrift

Contents

1 Introduction and intuition of LCFRS

1.1 Outline of the thesis

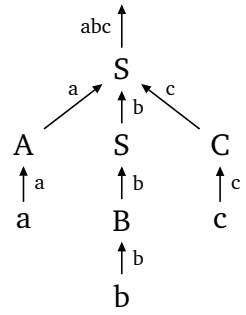
This thesis concerns itself with two tasks, the *extraction* and the *binarization* of *probablistic linear context-free rewriting systems* (short: PLCFRS or probabilistic LCFRS).

After understanding the intuitive idea behind this grammar formalism in chapter 1, we will formally define it in chapter 2. In chapter 3 we will see how a PLCFRS can be extracted from an annotated corpus of natural language by reading off rules from nodes and combining them into the extracted grammar, such that their probability depends on how often they were read off in the corpus (see [KS09; MS08]). Chapter 4 will be about the idea of binarization where we try to break down rules with more than two non-terminal symbols on the right-hand side into smaller rules by fusing two non-terminal symbols into a new one, thereby generating a new rule (see [GS09; Góm+09]). We will prove that this process does not change the semantics of the PLCFRS. After defining this simple fusion process we will introduce a framework that allows us to plan and execute such fusions to binarize rules and whole grammars. Using this framework we will define both a naive approach that generates inferior rules and an approach that generates optimal rules (see [Góm+09]).

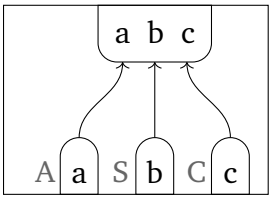
A possible implementation of these processes is sketched in chapter 5, the results of applying the processes defined in earlier chapters to real natural language corpora using this implementation are presented in chapter 6. Finally we will draw a short conclusion in chapter 7.

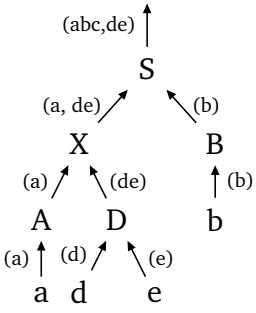
1.2 Intuition of LCFRS

Linear context-free rewriting systems (introduced by Vijay-Shanker, Weir, and Joshi [VWJ87]) are rewriting systems where no information is lost or copied. The generated languages and the data the rewriting process happens on can take various forms, e.g. string tuples, trees and graphs. In this thesis we will focus on LCFRS over string tuples, since this is the instance that is usually used when extracting a formal language from an annotated corpus of natural language [KS09; Eva11].

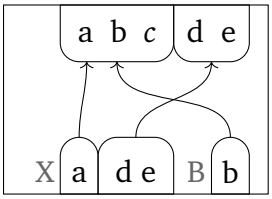


This is an example for a parse tree of a context free grammar. We can imagine that each node has a *yield* (here shown as the labels of the edges going to the parents), the terminal symbols that are created by it and its children. The root node S receives multiple yields from its children, its own yield is the concatenation of all children's yields in order (see this process on the right).





In LCFRS nodes do not just yield strings, but string tuples. Here nodes can recombine the components of the yields of their children however they like (copying or deleting is forbidden, though) and yield a string tuple themselves. The generated tuple can even contain new terminal symbols (like the *c* in our example, which is inserted into the first component of the result tuple).



LCFRS are an example of *mildly context-sensitive formalisms*, some other examples of such formalisms are head grammars, tree adjoining grammars, combinatory categorial grammars, linear indexed grammars, multiple context-free grammars, and minimalist grammars - but all these grammars (and some more) can actually be represented by LCFRS [Sek+91; VWJ86; WJ88; Vij87; Mic01a; Mic01b].

2 Definitions

2.1 Preliminaries

Basic definitions

Let $\mathbb{N}$ be the set of positive natural numbers (without 0) and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

We will denote the set of all numbers up to some $n \in \mathbb{N}$ by $[n] = \{1, \dots, n\}$.

For some totally ordered set S with $a, b \in S$ let $[a; b]$ denote the closed interval and $(a; b]$ the left-open interval. Every finite subset $S' \subseteq S$ can be expressed as the union of maximal intervals in the order of their smallest elements: $S' = [s^1; \overline{s^1}] \cup \dots \cup [\underline{s^m}; \overline{s^m}]$ such that for all $i \in [m]$: $s^i \leq \overline{s^i}$ and for all $i \in [m-1]$ there is a $s \in S$ such that $\overline{s^i} < s < s^{i+1}$. We can obtain the number of these intervals with a function: $\text{fanout}(S') = m$ (we explicitly stress that $\text{fanout}(\emptyset) = 0$).

For tuples of the form $(x_1, \dots, x_n)$ we define *projection functions* $\pi_i^{(n)}$ for all $i \in [n]$, such that $\pi_i^{(n)}(x_1, \dots, x_n) = x_i$. Since the superscript $^{(n)}$ can be easily inferred from context, we leave it out in the notation.

Let ε be the empty string, such that for every string s: $s\varepsilon = s = \varepsilon s$.

For a function $f : S \to \mathbb{R}_{\geq 0}$ the *support* of f is defined by $\text{supp}(f) = \{s \mid s \in S,\ f(s) > 0\}$.

For some set S with $s \in S$ and $\mu \in \mathbb{R}_{\geq 0}$ the expression $\mu.s$ is a function $f : S \to \mathbb{R}_{\geq 0}$ defined by:

$$f(s') = \begin{cases} \mu & \text{if } s = s' \\ 0 & \text{otherwise} \end{cases}$$

The sum of unary functions is defined $(f + g)(x) = f(x) + g(x)$, the multiplication with a scalar $(c \cdot f)(x) = c \cdot f(x)$.

Special sets

An *alphabet* is a finite non-empty set. We call the elements of an alphabet *symbols*.

Let S be any non-empty set. An *S-sorted set* is a tuple $M = (A, sort)$, where A is some set and $sort\colon A \to S$ assigns a *sort* to each $a \in A$. We will let $a \in M$ mean $a \in A$.

Let M be such an S-sorted set and $s \in S$. Then $M_s = \{m \mid m \in M,\ sort(m) = s\}$.

An *S-ranked set M* is an $(S^* \times S)$-sorted set. We define the rank of an element of such a set by $\mathrm{rank}(a) = |\pi_1(sort(a))|$.

Trees

Unranked trees

We define the set U_Σ of all *unranked trees* over some alphabet Σ recursively:

$$U_\Sigma = \{r(\xi_1, \ldots, \xi_k) \mid r \in \Sigma,\ k \geq 0,\ \forall i \in [k]\colon \xi_i \in U_\Sigma\}$$

The number of children k will also be called the *rank* of a node. We call nodes with $k > 0$ *inner nodes*, nodes with $k = 0$ are *leaves*.

The positions of a tree $\xi = r(\xi_1, \ldots, \xi_k)$ are defined as follows:

$$\mathrm{pos}\colon U_\Sigma \to \mathcal{P}(\mathbb{N}^*)$$
$$\mathrm{pos}(\xi) = \{\varepsilon\} \cup \{i\rho \mid i \in [k],\ \rho \in \mathrm{pos}(\xi_i)\} \subset \mathbb{N}^*$$

The relation $< \subseteq \mathrm{pos}(\xi) \times \mathrm{pos}(\xi)$ is defined as:

$$i\rho < j\rho' \Leftrightarrow i < j \vee (i = j \wedge \rho < \rho') \quad \text{where } i, j \in \mathbb{N},\ \rho, \rho' \in \mathrm{pos}(\xi)$$

The label of a tree $\xi = r(\xi_1, \ldots, \xi_k)$ at some position ρ is defined (and denoted):

$$\xi(\rho) = \begin{cases} r & \text{if } \rho = \varepsilon \\ \xi_i(\rho') & \text{otherwise } \rho = i\rho' \text{ for some } i \in [k], \rho' \in \mathrm{pos}(\xi_i) \end{cases}$$

The subtree of a tree $\xi = r(\xi_1, \ldots, \xi_k)$ at some position ρ is defined (and denoted):

$$\xi|_\rho = \begin{cases} \xi & \text{if } \rho = \varepsilon \\ \xi_i|_{\rho'} & \text{otherwise } \rho = i\rho' \text{ for some } i \in [k], \rho' \in \mathrm{pos}(\xi_i) \end{cases}$$

We can also obtain the *rank* of a tree $\xi = r(\xi_1, \ldots, \xi_k)$:

$$\mathrm{rk}(\xi) = k$$

Ranked trees

A *ranked alphabet* (Σ, rk) is a tuple consisting of an alphabet Σ and a mapping $rk\colon \Sigma \to \mathbb{N}_0$.

Ranked trees are defined over a ranked alphabet (Σ, rk), the definition for the set T_Σ of all *ranked trees* over (Σ, rk) is:

$$T_\Sigma = \{r(\xi_1, \dots, \xi_k) \mid r \in \Sigma\, , rk(r) = k,\ \forall i \in [k]\colon \xi_i \in T_\Sigma\}$$

Positions, labels and subtrees of ranked trees are defined just like they are for unranked trees, the rank can be defined by:

$$\mathrm{rk}(r(\xi_1, \dots, \xi_k)) = rk(r)$$

Composition functions

To make them visually distinguishable from other types of tuples we will write tuples of strings of elements of some set S like $(s_1, \dots, s_k) \in (S^*)^k$ for some $k \in \mathbb{N}$ as $\langle s_1, \dots, s_k \rangle$.

Let Σ be an alphabet and $X = \{x_{i,j} \mid i, j \in \mathbb{N}\}$ be a set of variables that stand for strings. We can define an $\mathbb{N}$-ranked set F of *composition functions* over string tuples, where for any $k \in \mathbb{N}_0$ and $s, s_1, \dots, s_k \in \mathbb{N}$:

$$F_{(s_1 \cdots s_k, s)} \subseteq \{f \mid f : (\Sigma^*)^{s_1} \times \cdots \times (\Sigma^*)^{s_k} \to (\Sigma^*)^s\}$$

such that every $f \in F_{(s_1 \cdots s_k, s)}$ is defined by

$$f(\langle x_{1,1}, \dots, x_{1,s_1} \rangle, \dots, \langle x_{k,1}, \dots, x_{k,s_k} \rangle) = \langle y_1, \dots, y_s \rangle$$

where for all $i \in [s]\colon y_i \in (\Sigma \cup X_f)^*$ and $X_f = \{x_{i,j} \mid i \in [k],\ j \in [s_i]\} \subset X$ and f is linear and non-deleting.

A composition function f is *linear*, if every element of X_f occurs at most once in $y_1 \cdots y_k$. It is *non-deleting*, if every element of X_f occurs at least once in $y_1 \cdots y_k$.

Since the arguments of a function f can always be implied from y_i for $i \in [s]$, we can uniquely specify the function above with these strings of variables and terminals alone:

$$f = \langle y_1, \dots, y_s \rangle$$

2.2 Defining LCFRS (on string tuples)

Definition 2.1 (LCFRS)**.** Let Σ be an alphabet. A *linear context-free rewriting system* (short: LCFRS) is a tuple (N, Σ, S, R) where

- N is a finite $\mathbb{N}$-sorted set (of non-terminal symbols, short: NTs)
- $S \subseteq N_1$ (start non-terminals)
- R is a finite $\mathbb{N}$-ranked set (of *rules*) where for every $(s_1 \cdots s_n, s) \in \mathbb{N}^* \times \mathbb{N}$ we have $R_{(s_1 \cdots s_n, s)} \subseteq N_s \times F_{(s_1 \cdots s_n, s)} \times (N_{s_1} \times \cdots \times N_{s_n})$. We will write such a tuple $r \in R$ with $r = (A, f, (B_1, \ldots, B_n))$ as $r = A \to f(B_1, \ldots, B_n)$.

□

We will call the sort of an $A \in N$ its *fanout*. Sometimes we will refer to the fanout of a rule r, meaning the fanout of the non-terminal $\pi_1(r)$.

Let $A \in N$. We define the *A-fragment of a set of rules R* by

$$R_A = \{r \mid r \in R,\ \pi_1(r) = A\}.$$

Example 2.2. Let $\Sigma = \{a, b, c\}$, $N = \{S^1, X^2, A^1, B^1, C^1\}$ be an $\mathbb{N}$-sorted set with the superscripts denoting the sort of the element and $S = \{S^1\} \subseteq N_1$. Defining a set R as follows we obtain an LCFRS (N, Σ, S, R):

$$\begin{aligned} R = \{\ & S^1 \to \langle x_{1,1} x_{2,1} x_{1,2} \rangle (X^2, B^1), \\ & X^2 \to \langle x_{1,1}, x_{2,1} \rangle (A^1, C^1), \\ & A^1 \to \langle a \rangle (), \\ & B^1 \to \langle b \rangle (), \\ & C^1 \to \langle c \rangle () \\ \} & \end{aligned}$$

□

2.3 Derivations

Definition 2.3 (derivations of an LCFRS)**.** Let $G = (N, \Sigma, S, R)$ be an LCFRS. Let $A \in N$. The *set of derivations of G starting in A*, denoted by $D_G(A)$, is recursively defined as follows:

$$D_G(A) = \{r(d_1, \ldots, d_k) \mid r = A \to f(B_1, \ldots, B_k) \in R, \forall i \in [k]\colon d_i \in D_G(B_i)\} \subseteq T_R$$

The set of derivations of G, denoted by D_G, is defined as

$$D_G = \bigcup_{A \in S} D_G(A).$$

□

A derivation is thus modeled to be a ranked tree of rules[1].

Definition 2.4 (semantics of a derivation)**.** Let $G = (N, \Sigma, S, R)$ be an LCFRS.
The string tuple $\langle s_1, \ldots, s_l \rangle \in (\Sigma^*)^l$ generated by a derivation $d = r(d_1, \ldots, d_k) \in D_G(\pi_1(r))$ with $l = sort(\pi_1(r))$ is obtained by applying the composition functions of each rule in the derivations nodes to the strings generated by their children. We define the semantics of a derivation:

$$[\![\cdot]\!] : T_R \to (\Sigma^*)^*$$
$$[\![r(d_1, \ldots, d_k)]\!] = f([\![d_1]\!], \ldots, [\![d_k]\!])$$

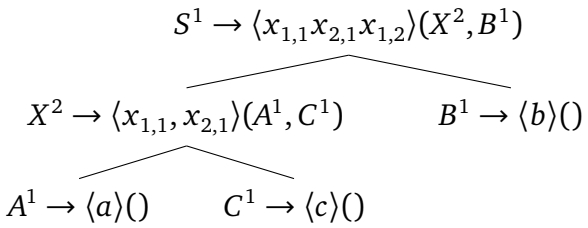
where $f = \pi_2(r)$. □

Definition 2.5 (language of an LCFRS)**.** The language of G, denoted by $L(G)$, is

$$L(G) = \{[\![d]\!] \mid d \in D_G\}$$

□

Example 2.6. It is easy to see that the LCFRS constructed in Example 2.2 has exactly one derivation d:



We can evaluate the semantics of this derivation:

$$\begin{aligned}
[\![d]\!] &= \langle x_{1,1} x_{2,1} x_{1,2} \rangle \Big(\Big[\!\Big[\begin{array}{c} X^2 \to \langle x_{1,1}, x_{2,1} \rangle (A^1, C^1) \\ \diagup \quad \diagdown \\ A^1 \to \langle a \rangle () \qquad C^1 \to \langle c \rangle () \end{array} \Big]\!\Big], \; [\![B^1 \to \langle b \rangle ()]\!] \Big) \\
&= \langle x_{1,1} x_{2,1} x_{1,2} \rangle \Big(\langle x_{1,1}, x_{2,1} \rangle \big([\![A^1 \to \langle a \rangle ()]\!], [\![C^1 \to \langle c \rangle ()]\!] \big), \; \langle b \rangle \Big) \\
&= \langle x_{1,1} x_{2,1} x_{1,2} \rangle \Big(\langle x_{1,1}, x_{2,1} \rangle \big(\langle a \rangle, \langle c \rangle \big), \; \langle b \rangle \Big) \\
&= \langle x_{1,1} x_{2,1} x_{1,2} \rangle \big(\langle a, c \rangle, \langle b \rangle \big) \\
&= \langle abc \rangle
\end{aligned}$$

□

[1] for context free grammars these are known as *abstract syntax trees*, we build these trees of rules in the same way

2.4 Probabilities

Definition 2.7 (PLCFRS)**.** A *probabilistic LCFRS* (short: PLCFRS) is a tuple (G, p) of an LCFRS $G = (N, \Sigma, S, R)$ as defined above and a probability assignment $p : R \to (0; 1]$ for which $\sum_{r \in R_A} p(r) = 1$ for all $A \in N$. □

If G is a PLCFRS, we will still write D_G to actually mean $D_{\pi_1(G)}$.

The following definitions are based on the usage of the probability semiring $(\mathbb{R}_{\geq 0}, +, \cdot, 0, 1)$.

Definition 2.8 (probability of a derivation)**.** Let (G, p) be a PLCFRS.
Let $\widetilde{p} : D_G \to (0; 1]$ assign probabilities to derivations using p:

$$\widetilde{p}(r(d_1, \ldots, d_k)) = p(r) \cdot \widetilde{p}(d_1) \cdot \ldots \cdot \widetilde{p}(d_k)$$

We will write $p(d)$ instead of $\widetilde{p}(d)$, since the argument d can only be either a rule or a tree of rules. □

A sentence can be generated by no derivation, one derivation or many derivations.

Definition 2.9 (probability of a sentence / semantics of a PLCFRS)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS.
We define the probability of a sentence:

$$[\![G]\!] : \Sigma^* \to \mathbb{R}_{\geq 0}$$
$$[\![G]\!](w) = \sum_{d \in D_G : [\![d]\!] = w} p(d)$$

□

The fact that the probability of a sentence is defined as the sum of the probabilities of its possible derivations implies that a sentence that cannot be generated with a given LCFRS has a probability of 0.

3 Extraction from a treebank

3.1 Treebanks

The treebanks from which we want to extract a PLCFRS (like NEGRA and TIGER) contain representations of natural language sentences in formats like XML (unranked trees) or the export format[1] (which too can be interpreted as describing unranked trees). We can interpret these representations as unranked trees with an additional order (where each tree with its order represents a sentence in a natural language and its parse), which we will define as follows:

Definition 3.1 (positions of leaves of ξ)**.** Let ξ be a tree.
Then the set $\{\rho_1, \ldots, \rho_l\} \subseteq \text{pos}(\xi)$ is the set of *positions of leaves of* ξ, defined such that for all $i \in [l]$:

$$\text{rk}(\xi|_{\rho_i}) = 0 \text{ and there is no } \rho \in \text{pos}(\xi) \text{ with } \rho \notin \{\rho_1, \ldots, \rho_k\} \wedge \text{rk}(\xi|_{\rho}) = 0$$

□

Definition 3.2 (tree with leaf order)**.** Let $\xi \in U_\Sigma$ be a tree and $\preceq$ be a total order over positions of leaves of ξ.
Then the tuple $(\xi, \preceq)$ is a *tree with leaf order* over the alphabet Σ. The set of all such trees is denoted by $\overrightarrow{U_\Sigma}$. □

It is important to note that in the input treebanks like NEGRA and TIGER the leaves of a tree ξ have no siblings, i.e. all positions of leaves end with 1. The functions we define in this chapter will make use of this property. This is why they will not work on trees where this is not the case.

The additional leaf order $\preceq$ describes the order the words of the natural language have in the sentence that ξ represents.

Definition 3.3 (corpus)**.** A *corpus over some set* S is a function $C : S \to \mathbb{R}_{\geq 0}$ that assigns each element of S a frequency inside the corpus. □

Definition 3.4 (treebank)**.** A *treebank* is a corpus over $\overrightarrow{U_\Sigma}$ for some alphabet Σ. □

[1] `http://www.coli.uni-sb.de/~thorsten/publications/Brants-CLAUS98.ps.gz`

The alphabet Σ over which the trees are defined contains both the part-of-speech tags of the natural language sentence with which the inner nodes of trees are labeled (they will eventually become the non-terminals of the extracted LCFRS) as well as the words of the natural language themselves with which the leaves are labeled (which will become the terminal alphabet of the LCFRS).

We define the treebank as a function in order to allow duplicates[2] (which is no longer the case in its support).

Since each tree with leaf order represents a natural language sentence, we can read off the generated string:

Definition 3.5 (yield of $(\xi, \preceq)$)**.** Given a pair $(\xi, \preceq)$ as defined above, let $\{\rho_1, \ldots, \rho_l\} \subseteq \text{pos}(\xi)$ be the positions of leaves of ξ such that $\rho_1 \preceq \cdots \preceq \rho_l$.
The *yield* of ξ is defined with $\text{yield}_{\preceq} : U_\Sigma \to \Sigma^*$:

$$\text{yield}_{\preceq}(\xi) = \xi(\rho_1) \cdots \xi(\rho_l)$$

□

Definition 3.6 (sentence corpus)**.** A *sentence corpus* is a corpus over Σ^* (sentences) for some alphabet Σ. □

Definition 3.7 (yield of a treebank)**.** Let C be a treebank over some alphabet Σ. The *yield of this treebank* is a sentence corpus $\widetilde{C}$ over the alphabet Σ:

$$\widetilde{C}(w) = \sum_{\substack{(\xi,\preceq)\in\text{supp}(C):\\ \text{yield}_{\preceq}(\xi)=w}} C(\xi, \preceq)$$

□

Definition 3.8 (likelihood)**.** Let S be a set, $p : S \to [0,1]$ be a probability distribution for this set and $C : S \to \mathbb{R}_{\geq 0}$ a corpus over this set. The *likelihood* is a function $\mathcal{L} : (S \to [0,1]) \times (S \to \mathbb{R}_{\geq 0}) \to [0,1]$ defined by:

$$\mathcal{L}(p, C) = \prod_{s\in\text{supp}(C)} p(s)^{C(s)}$$

□

Definition 3.9 (likelihood of a sentence corpus with some PLCFRS)**.** Let C be a sentence corpus and G be a PLCFRS. We define the likelihood of the sentence corpus C with the PLCFRS G as $\mathcal{L}([\![G]\!], C)$. □

[2] also, this way different parts of a tree corpus could be given different weights

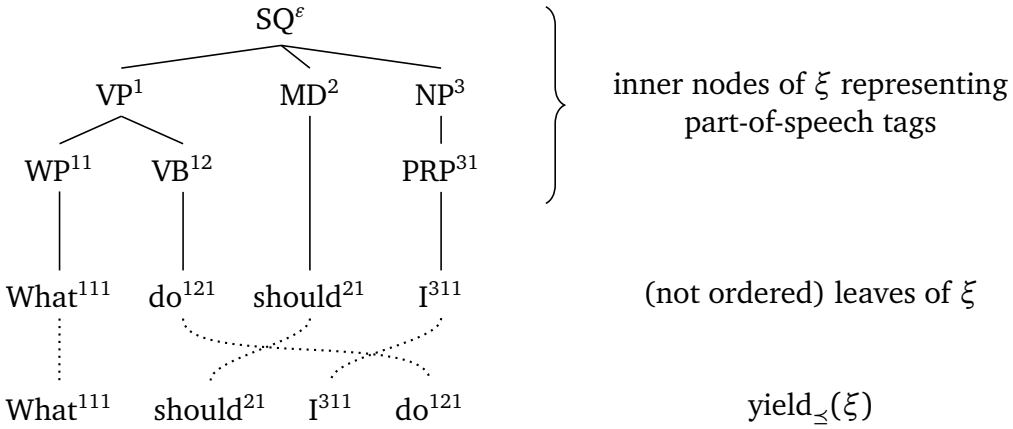


Figure 3.1: An example tree with leaf order, superscripts indicate the position of a node

Example 3.10. Figure 3.1 shows an example tree with leaf order $(\xi, \preceq)$ (taken from Evang [Eva11] and slightly simplified).

The tree $\xi = \mathrm{SQ}\big(\mathrm{VP}\big(\mathrm{WP}(\mathrm{What}), \mathrm{VB}(\mathrm{do})\big), \mathrm{MD}(\mathrm{should}), \mathrm{NP}\big(\mathrm{PRP}(\mathrm{I})\big)\big)$ and the ordering $\preceq$ is defined such that $111 \preceq 21 \preceq 311 \preceq 121$.

□

3.2 Transforming input trees into rules

To extract rules from a treebank we apply a series of transformations to this sequence of trees.

Calculate spans

Each node of a derivation eventually yields a string tuple containing some words in the target string. For every position ρ of such a derivation we can say which words of the target string will be generated by the part of the derivation beginning at ρ.

Definition 3.11 (spans of a position)**.** Let $(\xi, \preceq)$ be a tree with leaf order.
For all positions $\rho \in \mathrm{pos}(\xi)$ define the *spans* of the node (yielded positions in target string):

$$\mathrm{spans}_{(\xi,\preceq)} \colon \mathrm{pos}(\xi) \to \mathcal{P}\big([\,|\mathrm{yield}_{\preceq}(\xi)|\,]\big)$$

$$\mathrm{spans}_{(\xi,\preceq)}(\rho) = \begin{cases} \big\{\,|\{\rho' \mid \rho' \in \mathrm{pos}(\xi),\ \mathrm{rk}(\xi|_{\rho'}) = 0,\ \rho' \preceq \rho\}|\,\big\} & \text{if } \mathrm{rk}(\xi|_{\rho}) = 0 \\ \bigcup_{i \in [\mathrm{rk}(\xi|_{\rho})]} \mathrm{spans}_{(\xi,\preceq)}(\rho i) & \text{otherwise} \end{cases}$$

□

To explain the very specific codomain of $\text{spans}_{(\xi,\preceq)}$ we will make some observations.

Observation 3.12. Let $(\xi, \preceq) \in \overrightarrow{U_\Sigma}$ be a tree with leaf order.

Then $\text{spans}_{(\xi,\preceq)}(\varepsilon) \supseteq \text{spans}_{(\xi,\preceq)}(\rho)$ for every $\rho \in \text{pos}(\xi)$. ■

Proof. From the definition it follows directly that for all $i \in [\text{rk}(\xi|_\rho)]$: $\text{spans}_{(\xi,\preceq)}(\rho) \supseteq \text{spans}_{(\xi,\preceq)}(\rho i)$.

Because $\supseteq$ is transitive and reflexive, $\text{spans}_{(\xi,\preceq)}(\varepsilon) \supseteq \text{spans}_{(\xi,\preceq)}(\rho')$ holds for all $\rho' \in \text{pos}(\xi)$. ■

Observation 3.13. Let $(\xi, \preceq) \in \overrightarrow{U_\Sigma}$ be a tree with leaf order.

Then for any $\rho \in \text{pos}(\xi)$: $|\text{spans}_{(\xi,\preceq)}(\rho)| \leq l$ where l is the number of leaves of ξ. ■

Lemma 3.14. $\text{spans}_{(\xi,\preceq)}(\varepsilon) = [l]$ where $l = |\text{yield}_\preceq(\xi)|$.

Proof.

$$
\begin{aligned}
& |\text{yield}_\preceq(\xi)| = l \\
\Leftrightarrow\ & \xi \text{ has } l \text{ leaves} && \text{(by Def. 3.5)} \\
\Leftrightarrow\ & \rho_1 \cdots \rho_l \text{ are positions of leaves of } \xi \text{ such that } \rho_1 \preceq \cdots \preceq \rho_l \\
& && \text{(by order } \preceq \text{ being total on positions of leaves, see Def. 3.2)} \\
\Rightarrow\ & \rho_1 \cdots \rho_l \text{ are positions of leaves of } \xi \text{ such that } \rho_1 \preceq \cdots \preceq \rho_l \colon \\
& \quad \forall i \in [l] \colon \text{spans}_{(\xi,\preceq)}(\rho_i) = \{|\{\rho' \mid \rho' \in \{\rho_{i'} \mid i' \leq i\}\}|\} && \text{(by Def. 3.11)} \\
\Rightarrow\ & \rho_1 \cdots \rho_l \text{ are positions of leaves of } \xi \text{ such that } \rho_1 \preceq \cdots \preceq \rho_l \colon \\
& \quad \forall i \in [l] \colon \text{spans}_{(\xi,\preceq)}(\rho_i) = \{i\} \\
\Rightarrow\ & \forall i \in [l] \colon \text{spans}_{(\xi,\preceq)}(\varepsilon) \supseteq \{i\} && \text{(by obs. 3.12)} \\
\Leftrightarrow\ & \text{spans}_{(\xi,\preceq)}(\varepsilon) \supseteq [l] \\
\Leftrightarrow\ & \text{spans}_{(\xi,\preceq)}(\varepsilon) = [l] && \text{(by obs. 3.13)}
\end{aligned}
$$

■

Observation 3.15. Let $(\xi, \preceq) \in \overrightarrow{U_\Sigma}$ be a tree with leaf order.

From Lemma 3.14 and Observation 3.12 follows that for all $\rho \in \text{pos}(\xi)$: $\text{spans}_{(\xi,\preceq)}(\rho) \subseteq [l]$ where $l = |\text{yield}_\preceq(\xi)|$.

Thus the codomain of $\text{spans}_{(\xi,\preceq)}$ given in Definition 3.11 is indeed correct. In fact, Lemma 3.14 even shows that $\text{spans}_{(\xi,\preceq)}$ is surjective. ■

Add fanouts and annotate with spans

As explained above, the labels of the inner nodes of the trees will become non-terminals of an LCFRS, but since non-terminals are kept in a sorted set, we have to make sure that each non-terminal will only be associated with one sort. This is not necessarily the case for the inner nodes in the current trees with leaf order, therefore we add the future fanouts (which are the ranks of NTs) to the existing labels.

For this we can make use of the newly defined spans, interpreted as the union of maximal intervals in the order of their smallest elements, since the number of such intervals will be exactly the fanout to associate with this label.

We will also annotate each node with these intervals themselves. This will make working with subtrees a lot easier, because we no longer need to use the order of a tree with leaf order to get the spans of a position but can instead use the annotated spans immediately.

Definition 3.16 (add fanouts, annotate with spans)**.** Let Σ be an alphabet.
We can define a transformation $\text{annotate}_{\preceq}\colon U_\Sigma \times \mathbb{N}^* \to U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$ as follows:

$$\begin{aligned}&\text{annotate}_{\preceq}(\xi,\rho)\\ &\quad= \big((\xi(\rho),m),\text{spans}_{(\xi,\preceq)}(\rho)\big)\big(\text{annotate}_{\preceq}(\xi,\rho 1),\dots,\text{annotate}_{\preceq}(\xi,\rho\,\text{rk}(\xi|_\rho))\big)\end{aligned}$$

where $m = \text{fanout}(\text{spans}_{(\xi,\preceq)}(\rho))$.
We can see that $\text{annotate}_{\preceq}$ is partial, since it is only defined for $\rho \in \text{pos}(\xi) \subset \mathbb{N}^*$. Instead of $\text{annotate}_{\preceq}(\xi,\varepsilon)$ we will just write $\text{annotate}_{\preceq}(\xi)$ to obtain an *annotated tree*. □

Example 3.17. We apply this transformation to the tree from Example 3.10 and we recall that $\preceq$ was defined such that $111 \preceq 21 \preceq 311 \preceq 121$.

First let us try to evaluate $\text{spans}_{(\xi,\preceq)}(\varepsilon)$:

$$\begin{aligned}&\text{spans}_{(\xi,\preceq)}(\varepsilon)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \text{spans}_{(\xi,\preceq)}(2) \cup \text{spans}_{(\xi,\preceq)}(3)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \text{spans}_{(\xi,\preceq)}(21) \cup \text{spans}_{(\xi,\preceq)}(3)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \big\{\,|\{\rho \mid \rho \in \text{pos}(\xi),\ \text{rk}(\xi|_\rho) = 0,\ \rho \preceq 21\}|\,\big\} \cup \text{spans}_{(\xi,\preceq)}(3)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \big\{\,|\{111, 21\}|\,\big\} \cup \text{spans}_{(\xi,\preceq)}(3)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \text{spans}_{(\xi,\preceq)}(3)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \text{spans}_{(\xi,\preceq)}(31)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \text{spans}_{(\xi,\preceq)}(311)\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \big\{\,|\{\rho \mid \rho \in \text{pos}(\xi),\ \text{rk}(\xi|_\rho) = 0,\ \rho \preceq 311\}|\,\big\}\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \big\{\,|\{111, 21, 311\}|\,\big\}\\ &= \text{spans}_{(\xi,\preceq)}(1) \cup \{2\} \cup \{3\}\end{aligned}$$

$$
\begin{aligned}
&= \big(\text{spans}_{(\xi,\preceq)}(11) \cup \text{spans}_{(\xi,\preceq)}(12)\big) \cup \{2\} \cup \{3\} \\
&= \big(\text{spans}_{(\xi,\preceq)}(111) \cup \text{spans}_{(\xi,\preceq)}(121)\big) \cup \{2\} \cup \{3\} \\
&= \Big(\big\{\,|\{\rho \mid \rho \in \text{pos}(\xi),\ \text{rk}(\xi|_\rho) = 0,\ \rho \preceq 111\}|\,\big\} \\
&\qquad \cup \big\{\,|\{\rho \mid \rho \in \text{pos}(\xi),\ \text{rk}(\xi|_\rho) = 0,\ \rho \preceq 121\}|\,\big\}\Big) \cup \{2\} \cup \{3\} \\
&= \Big(\big\{\,|\{111\}|\,\big\} \cup \big\{\,|\{111, 21, 311, 121\}|\,\big\}\Big) \cup \{2\} \cup \{3\} \\
&= \big(\{1\} \cup \{4\}\big) \cup \{2\} \cup \{3\} \\
&= [4]
\end{aligned}
$$

As expected, $\text{spans}_{(\xi,\preceq)}(\varepsilon) = [4]$ holds. We can now use these spans to compute the annotated tree:

$$
\begin{aligned}
&\text{annotate}_\preceq(\xi, \varepsilon) \\
&= \text{annotate}_\preceq\Bigg(\quad \text{SQ}\big(\text{VP}(\text{WP}(\text{What}),\ \text{VB}(\text{do})),\ \text{MD}(\text{should}),\ \text{NP}(\text{PRP}(\text{I}))\big)\quad ,\quad \varepsilon\quad\Bigg) \\
&= \underset{\text{annotate}_\preceq(\xi,1)\quad \text{annotate}_\preceq(\xi,2)\quad \text{annotate}_\preceq(\xi,3)}{\big(\big(\text{SQ}, \text{fanout}(\text{spans}_{(\xi,\preceq)}(\varepsilon))\big), \text{spans}_{(\xi,\preceq)}(\varepsilon)\big)} \\
&= \underset{\text{annotate}_\preceq(\xi,1)\quad \text{annotate}_\preceq(\xi,2)\quad \text{annotate}_\preceq(\xi,3)}{\big(\big(\text{SQ}, \text{fanout}([4])\big), [4]\big)} \\
&= \underset{\text{annotate}_\preceq(\xi,1)\quad \text{annotate}_\preceq(\xi,2)\quad \text{annotate}_\preceq(\xi,3)}{\big((\text{SQ}, 1), [4]\big)}
\end{aligned}
$$

$$
\begin{aligned}
&\big((\mathrm{SQ},1),[4]\big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{annotate}_{\preceq}(\xi,2)\quad \mathrm{annotate}_{\preceq}(\xi,3)\\
=\quad & \mathrm{annotate}_{\preceq}\left(\begin{array}{c}\mathrm{SQ}\\ \mathrm{VP}\quad \mathrm{MD}\quad \mathrm{NP}\\ \mathrm{WP}\quad \mathrm{VB}\quad \mathrm{should}\quad \mathrm{PRP}\\ \mathrm{What}\quad \mathrm{do}\qquad\qquad \mathrm{I}\end{array}\right),\ 1
\end{aligned}
$$

$$
\begin{aligned}
&\big((\mathrm{SQ},1),[4]\big)\\
&\big(\big(\mathrm{VP},\mathrm{fanout}(\mathrm{spans}_{(\xi,\preceq)}(1))\big),\mathrm{spans}_{(\xi,\preceq)}(1)\big)\quad \mathrm{annotate}_{\preceq}(\xi,2)\quad \mathrm{annotate}_{\preceq}(\xi,3)\\
=\quad & \mathrm{annotate}_{\preceq}(\xi,11)\quad \mathrm{annotate}_{\preceq}(\xi,12)
\end{aligned}
$$

$$
\begin{aligned}
&\big((\mathrm{SQ},1),[4]\big)\\
&\big(\big(\mathrm{VP},\mathrm{fanout}(\{1,4\})\big),\{1,4\}\big)\qquad \mathrm{annotate}_{\preceq}(\xi,2)\quad \mathrm{annotate}_{\preceq}(\xi,3)\\
=\quad & \mathrm{annotate}_{\preceq}(\xi,11)\quad \mathrm{annotate}_{\preceq}(\xi,12)
\end{aligned}
$$

$$
\begin{aligned}
&\big((\mathrm{SQ},1),[4]\big)\\
&\big((\mathrm{VP},2),\{1,4\}\big)\qquad\qquad \mathrm{annotate}_{\preceq}(\xi,2)\quad \mathrm{annotate}_{\preceq}(\xi,3)\\
=\quad & \mathrm{annotate}_{\preceq}(\xi,11)\quad \mathrm{annotate}_{\preceq}(\xi,12)
\end{aligned}
$$

$$
=\cdots
$$

$$
\begin{aligned}
&\big((\mathrm{SQ},1),[4]\big)\\
&\big((\mathrm{VP},2),\{1,4\}\big)\qquad \big((\mathrm{MD},1),\{2\}\big)\quad \big((\mathrm{NP},1),\{3\}\big)\\
=\quad & \big((\mathrm{WP},1),\{1\}\big)\quad \big((\mathrm{VB},1),\{4\}\big)\quad \big((\mathrm{should},1),\{2\}\big)\quad \big((\mathrm{PRP},1),\{3\}\big)\\
&\big((\mathrm{What},1),\{1\}\big)\quad \big((\mathrm{do},1),\{4\}\big)\qquad\qquad\qquad\qquad \big((\mathrm{I},1),\{3\}\big)
\end{aligned}
$$

□

Definition 3.18 (yield of an annotated tree)**.** Let Σ be an alphabet. Given an annotated tree $\xi \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$, let $\{\rho_{1,1},\dots,\rho_{1,l_1},\dots,\rho_{m,1},\dots,\rho_{m,l_m}\} \subseteq \mathrm{pos}(\xi)$ be the set of positions of leaves of ξ for some $m, l_1, \dots, l_m \in \mathbb{N}$ such that the following holds:

$$\widetilde{\pi_2}(\xi(\rho_{i,1})) < \cdots < \widetilde{\pi_2}(\xi(\rho_{i,l_i})) \text{ for all } i \in [m]$$
$$\text{and for all } i \in [m-1] \text{ there is an } s \in \mathbb{N}\colon \widetilde{\pi_2}(\xi(\rho_{i,l_i})) < s < \widetilde{\pi_2}(\xi(\rho_{(i+1),1}))$$

where $\widetilde{\pi_2}((t,m),\{i\}) = i$.
Then the *yield of the annotated tree* ξ is defined by:

$$\widetilde{\mathrm{yield}}\colon U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})} \to (\Sigma^*)^*$$
$$\widetilde{\mathrm{yield}}(\xi) = \big(\pi_1(\pi_1(\xi(\rho_{1,1})))\cdots\pi_1(\pi_1(\xi(\rho_{1,l_1}))),\ \dots\ ,$$
$$\pi_1(\pi_1(\xi(\rho_{m,1})))\cdots\pi_1(\pi_1(\xi(\rho_{m,l_m})))\big)$$

□

Lemma 3.19. Let $(\xi, \preceq)$ be a tree with leaf order.

Then $\widetilde{\mathrm{yield}}\big(\mathrm{annotate}_{\preceq}(\xi)\big) = \mathrm{yield}_{\preceq}(\xi)$.

Proof. Given a tree with leaf order $(\xi, \preceq)$ we can interpret the leaves of $\mathrm{annotate}_{\preceq}(\xi)$ as being ordered by the single element in their second component (the annotated spans).

$\widetilde{\mathrm{yield}}$ will return the sequence of leaves in this order (it is just a simple sequence, since $\mathrm{spans}_{(\xi,\preceq)}(\varepsilon) = [l]$ (see Lemma 3.14), where l is the number of leaves of ξ).

It is easy to see that this order implied by the spans and $\preceq$ describe exactly the same order (since we defined the spans using $\preceq$).

We can thus conclude that the sequence generated by $\mathrm{yield}_{\preceq}(\xi)$ is equal to the one generated by $\widetilde{\mathrm{yield}}\big(\mathrm{annotate}_{\preceq}(\xi)\big)$. ■

Read off rules

First we define a function puzzle, that, given a totally ordered set of elements X and a sequence of intervals $S_1 \cdots S_k \in (\mathcal{P}(X))^*$ (whose pairwise intersections must be empty and whose union must be equal to X), tries to arrange the intervals such that they are in the order of their smallest elements and returns the composition function that represents exactly this rearrangement.

Definition 3.20 (interval puzzler)**.** Let E be some ordered set and F the set of all composition functions.
The function puzzle: $\mathcal{P}(E) \times \mathcal{P}(E)^* \to F$ is defined by:

$$\begin{aligned}&\text{puzzle}(E', (E_1, \dots, E_k))\\ &\quad = \langle \text{puzzle}'([s^1, \overline{s^1}], (E_1, \dots, E_k)), \dots, \text{puzzle}'([s^m, \overline{s^m}], (E_1, \dots, E_k)) \rangle\end{aligned}$$

with

$$\begin{aligned}&\text{puzzle}'([s, \bar{s}],\ ([s_1^1, \overline{s_1^1}] \cup \dots \cup [s_1^{m_1}, \overline{s_1^{m_1}}], \dots, [s_k^1, \overline{s_k^1}] \cup \dots \cup [s_k^{m_k}, \overline{s_k^{m_k}}]))\\ &= \begin{cases} x_{i,j} & \text{if } [s, \bar{s}] = [s_i^j, \overline{s_i^j}] \text{ for some } i \in [k], j \in [m_i] \\ x_{i,j}\ \text{puzzle}'([\tilde{s}, \bar{s}], (E_1, \dots, E_k)) & \text{otherwise with } s_i^j = s \text{ and } \tilde{s} \text{ being the smallest} \\ & \qquad s' \in E \text{ with } s' > \overline{s_i^j} \text{ for some } i \in [k], j \in [m_i] \end{cases}\end{aligned}$$

where $[s^1, \overline{s^1}] \cup \dots \cup [s^m, \overline{s^m}]$ is the union of maximal intervals in the order of their smallest elements that is equal to E' and $[s_i^1, \overline{s_i^1}] \cup \dots \cup [s_i^{m_i}, \overline{s_i^{m_i}}]$ is the union of maximal intervals in the order of their smallest elements that is equal to E_i for all $i \in [k]$. □

Definition 3.21 (rule readoff at a node)**.** We define a partial function readoff: $((\Sigma \times \mathbb{N}) \times \mathcal{P}(\mathbb{N})) \times (((\Sigma \times \mathbb{N}) \times \mathcal{P}(\mathbb{N})) \times \mathbb{N}_0)^* \to \Sigma \times F \times \Sigma^*$ [a] that generates a rule from a root label and a sequence of child labels coupled with their rank:

$$\text{readoff}\Big(((t,1),S), \big(((w,1),S),0\big)\Big) = (t,1) \to \langle w \rangle()$$

$$\begin{aligned}&\text{readoff}\Big(((t,m),S), \big(\big(((t_1,m_1),S_1),k_1\big), \dots, \big(((t_k,m_k),S_k),k_k\big)\big)\Big)\\ &\quad = (t,m) \to f\big((t_1,m_1), \dots, (t_k,m_k)\big) \quad \text{with } f = \text{puzzle}\Big(S, (S_1, \dots, S_k)\Big)\end{aligned}$$

□

[a] This set is defined like the set of rules on an LCFRS and in fact the elements of this set will be the rules of the extracted LCFRS.

Example 3.22. Continuing with the annotated tree $\xi' = \text{annotate}_{\preceq}(\xi)$ from Example 3.17 we will now try to read off rules at the positions 31 and 1:

$$\begin{aligned}&\text{readoff from } \xi' \text{ at position } 31\text{:}\\ &\quad \text{readoff}\Big(\xi(31), \big(\big(\xi(311), \text{rk}(\xi|_{311})\big), \dots, \big(\xi(31\,\text{rk}(\xi|_{31})), \text{rk}(\xi|_{31\,\text{rk}(\xi|_{31})})\big)\big)\Big)\\ &= \text{readoff}\Big(((\text{PRP},1),\{3\}), \big(((\text{I},1),\{3\}),0\big)\Big)\\ &= (\text{PRP},1) \to \langle \text{I} \rangle()\end{aligned}$$

readoff from ξ' at position 1:

$$\begin{aligned}
&\text{readoff}\Big(\xi(1), \big(\big(\xi(11), \text{rk}(\xi|_{11})\big), \dots, \big(\xi(1\,\text{rk}(\xi|_1)), \text{rk}(\xi|_{1\,\text{rk}(\xi|_1)})\big)\big)\Big)\\
&= \text{readoff}\Big(\big((\text{VP}, 2), \{1, 4\}\big), \big(\big(((\text{WP}, 1), \{1\}), 1\big), \big(((\text{VB}, 1), \{4\}), 1\big)\big)\Big)\\
&= (\text{VP}, 2) \to \big(\text{puzzle}\big(\{1, 4\}, (\{1\}, \{4\})\big)\big)\big((\text{WP}, 1), (\text{VB}, 1)\big)\\
&= (\text{VP}, 2) \to \langle \text{puzzle}'\big(\{1\}, (\{1\}, \{4\})\big), \text{puzzle}'\big(\{4\}, (\{1\}, \{4\})\big)\rangle\big((\text{WP}, 1), (\text{VB}, 1)\big)\\
&= (\text{VP}, 2) \to \langle x_{1,1}, x_{2,1}\rangle\big((\text{WP}, 1), (\text{VB}, 1)\big)
\end{aligned}$$

□

Definition 3.23 (lifting readoff onto trees)**.** Let Σ be an alphabet and $\xi \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$. F is the set of composition functions.
We define the function $\widehat{\text{readoff}}\colon U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})} \to (\Sigma \times F \times \Sigma^* \to \mathbb{R}_{\geq 0})$ by:

$$\widehat{\text{readoff}}(\xi) = \sum_{\substack{\rho\in\text{pos}(\xi):\\ \text{rk}(\xi|_\rho)>0}} 1.\text{readoff}\Big(\xi(\rho), \big(\big(\xi(\rho 1), \text{rk}(\xi|_{\rho 1})\big), \dots, \big(\xi(\rho\,\text{rk}(\xi|_\rho)), \text{rk}(\xi|_{\rho\,\text{rk}(\xi|_\rho)})\big)\big)\Big)$$

□

Definition 3.24 (lifting readoff into the treebank)**.** Let C be a treebank.
We define the function extract: $(\overrightarrow{U_\Sigma} \to \mathbb{R}_{\geq 0}) \to (\Sigma \times F \times \Sigma^* \to \mathbb{R}_{\geq 0})$ by:

$$\text{extract}(C) = \sum_{(\xi,\preceq)\in\text{supp}(C)} C(\xi, \preceq) \cdot \widehat{\text{readoff}}\big(\text{annotate}_\preceq(\xi)\big)$$

□

So readoff transforms a node of a tree into a rule and $\widehat{\text{readoff}}$ transforms a tree into a function. Much like the treebank definition this function assigns each possible rule the value of how often it was read off from this tree. The function extract applies this idea to a whole treebank.

3.3 Extracting a PLCFRS from the transformed trees

Definition 3.25 (extracting a PLCFRS from a treebank)**.** Let C be a treebank. We can define the following sets:

$$R = \{r \text{ with sort } (m_1 \cdots m_k, m) \mid r = (t, m) \to f((t_1, m_1), \ldots, (t_k, m_k)) \in \text{supp}(\text{extract}(C))\}$$
$$N = \{(t, m) \text{ with sort } m \mid (t, m) \to f(B_1, \ldots, B_k) \in R\}$$
$$\Sigma = \{f() \mid A \to f() \in R\}$$
$$S = \{\pi_1(r) \mid r(\xi_1, \ldots, \xi_k) = \text{annotate}_{\preceq}(\xi),\ (\xi, \preceq) \in \text{supp}(C)\}$$

Then $G = (N, \Sigma, S, R)$ is the LCFRS extracted from the treebank C.

The probability of a rule $r = (A \to f(B_1, \ldots, B_k)) \in R$ is defined by:

$$p(r) = \frac{\text{extract}(C)(r)}{\sum_{r' \in R_A} \text{extract}(C)(r')}$$

(G, p) is the *PLCFRS extracted from the treebank* C. □

Observation 3.26. For all $r \in R$: $p(r) \in (0; 1]$. ■

Observation 3.27. For all $A \in N$:

$$\sum_{r \in R_A} p(r) = 1$$

■

Proof.

$$\begin{aligned}
&\sum_{r \in R_A} p(r) \\
&= \sum_{r \in R_A} \frac{\text{extract}(C)(r)}{\sum_{r' \in R_A} \text{extract}(C)(r')} && \text{(by Def. 3.25)} \\
&= \frac{\sum_{r \in R_A} \text{extract}(C)(r)}{\sum_{r' \in R_A} \text{extract}(C)(r')} \\
&= 1
\end{aligned}$$

■

To prove that the extracted PLCFRS G does in fact assign the yield of the corpus C (from which it was extracted) a likelihood $\mathcal{L}_G(\widetilde{C}) > 0$, we need to introduce another readoff-based transformation.

Definition 3.28 (transforming an annotated tree into a derivation)**.** Let Σ be an alphabet and $\xi \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$ an annotated tree. For readability, let $\zeta_i = (\xi(i), \mathrm{rk}(\xi|_i))$.
Then we can construct a new tree $\widetilde{\mathrm{readoff}}(\xi) \in T_{\Sigma\times F\times\Sigma^*}$ as follows.

$$\widetilde{\mathrm{readoff}}(\xi) = \Big(\mathrm{readoff}\big(\xi(\varepsilon), (\zeta_1, \dots, \zeta_{rk(\xi)})\big)\Big)\Big(\widetilde{\mathrm{readoff}}(\xi|_{i_1}), \dots, \widetilde{\mathrm{readoff}}(\xi|_{i_{k'}})\Big)$$

$$\text{where } \{i_1, \dots, i_{k'}\} = \{i \mid i \in [\mathrm{rk}(\xi)], \mathrm{rk}(\xi|_i) > 0\} \text{ with } i_1 < \dots < i_{k'}$$

□

Lemma 3.29. Let Σ be an alphabet and $\xi \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$ be an annotated tree.

Then $\widetilde{\mathrm{yield}}(\xi) = [\![\widetilde{\mathrm{readoff}}(\xi)]\!]$.

Proof. We prove by induction.

Base case: Let $\xi = ((t,1),S)\big(((w,1),S)\big) \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$.

$$\begin{aligned}
&\Big[\!\Big[\widetilde{\mathrm{readoff}}\big(((t,1),S)\big(((w,1),S)\big)\big)\Big]\!\Big] \\
&= \Big[\!\Big[\mathrm{readoff}\big(\big((t,1),S\big),\big(((w,1),S),0\big)\big)\big)\Big]\!\Big] && \text{(by Def. 3.28)} \\
&= [\![(t,1) \to \langle w\rangle()]\!] && \text{(by Def. 3.21)} \\
&= w && \text{(by Def. 2.4)} \\
&= \widetilde{\mathrm{yield}}\big(((t,1),S)\big(((w,1),S)\big)\big) && \text{(by Def. 3.18)}
\end{aligned}$$

Induction hypothesis: Let $\xi \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$. Then $\widetilde{\mathrm{yield}}(\xi) = [\![\widetilde{\mathrm{readoff}}(\xi)]\!]$.

Induction step: Let for all $i \in [k]$: $\xi_i \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$ for some k with $\bigcap_{i\in[k]} \pi_2(\xi_i(\varepsilon)) = \emptyset$.

Let $\xi = ((t,m),S)\big(\xi_1, \dots, \xi_k\big) \in U_{(\Sigma\times\mathbb{N})\times\mathcal{P}(\mathbb{N})}$ for some $t \in \Sigma$, $S = \bigcup_{i\in[k]} \pi_2(\xi_i(\varepsilon))$ and $m = \mathrm{fanout}(S)$.

Let us try to construct $\widetilde{\mathrm{yield}}(\xi) = (w_1, \dots, w_m)$. We know that for all $i \in [k]$ the sequence w_i is made up of the labels of all leaves ρ of ξ for which $\pi_2(\xi(\rho))$ is a subset of the i-th interval of S (representing S as the union of maximal intervals in the order of their smallest elements). As a matter of fact, even the order of these leaves can be taken from the idea of trying to puzzle them into the i-th interval of S.

This description heavily implies that we can use a composition function to obtain $\widetilde{\mathrm{yield}}(\xi)$ from the $\widetilde{\mathrm{yield}}$s of the children of ξ. This function is $f = \mathrm{puzzle}\big(S, \big(\pi_2(\xi_1(\varepsilon), \dots, \pi_2(\xi_k(\varepsilon))\big)\big)$, so we can now say:

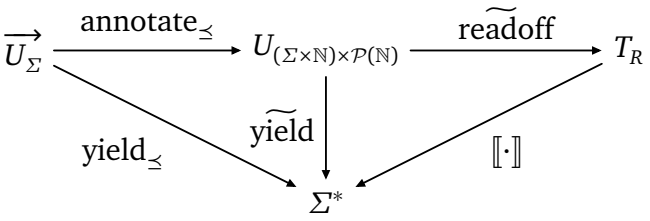


Figure 3.2: Sketch of the proof for Lemma 3.30, the left half is shown by Lemma 3.19, the right half by Lemma 3.29

$$
\begin{aligned}
\widetilde{\text{yield}}(\xi) &= f\left(\widetilde{\text{yield}}(\xi_1), \dots, \widetilde{\text{yield}}(\xi_k)\right) && \text{(by intuition of } \widetilde{\text{yield}} \text{ as described above)} \\
&= f\left([\![\widetilde{\text{readoff}}(\xi_1)]\!], \dots, [\![\widetilde{\text{readoff}}(\xi_k)]\!]\right) && \text{(by induction hypothesis)} \\
&= \left[\!\!\left[(t,m) \to f\left(\pi_1(\xi_1(\varepsilon)), \dots, \pi_1(\xi_k(\varepsilon))\right)\left(\widetilde{\text{readoff}}(\xi_1), \dots, \widetilde{\text{readoff}}(\xi_k)\right)\right]\!\!\right] && \text{(by Def. 2.4)} \\
&= \left[\!\!\left[\text{readoff}\left(((t,m),S),\left((\pi_1(\xi_1),\text{rk}(\xi_1)), \dots, (\pi_1(\xi_k),\text{rk}(\xi_k))\right)\right)\right.\right. \\
&\qquad \left.\left.\left(\widetilde{\text{readoff}}(\xi_1), \dots, \widetilde{\text{readoff}}(\xi_k)\right)\right]\!\!\right] && \text{(by Def. 3.21)} \\
&= [\![\widetilde{\text{readoff}}(\xi)]\!] && \text{(by Def. 3.28)}
\end{aligned}
$$

Note how we use the fact that the composition function f we defined to construct $\widetilde{\text{yield}}(\xi)$ is defined precisely like the composition function for the new rule in readoff. ■

Lemma 3.30. Let C be a treebank, $\widetilde{C}$ the yield of this treebank and G the PLCFRS extracted from C.

Then $[\![G]\!](w) > 0$ for any $w \in \text{supp}(\widetilde{C})$.

Proof. We know that there is a $(\xi, \preceq) \in C$, such that $w = \text{yield}_\preceq(\xi)$ (follows from the definitions of support and $\widetilde{C}$).

From Lemma 3.19 and Lemma 3.29 follows that we can transform this tree into a derivation d with $[\![d]\!] = w$ (see Fig. 3.2).

This derivation $d \in D_G$, because all rules used for it were read off the same way when we extracted G.

From that follows that $[\![G]\!](w) > 0$. ■

Theorem 3.31. Let C be a treebank, $\widetilde{C}$ the yield of this treebank and G the PLCFRS extracted from C.

Then $\mathcal{L}([\![G]\!], \widetilde{C}) > 0$.

Proof.

$$
\begin{aligned}
& \mathcal{L}([\![G]\!], \widetilde{C}) \\
= & \prod_{w \in \mathrm{supp}(\widetilde{C})} [\![G]\!](w)^{\widetilde{C}(w)} && \text{(by Def. 3.9)} \\
> & \, 0 && \text{(by Lem. 3.30 and the definition of supp)}
\end{aligned}
$$

■

4 Binarization

Binarization is a process in which a PLCFRS is transformed into a new PLCFRS that contains only rules with a rank that does not exceed 2, while generating the same weighted language (weak equivalence[1]).

Grammars are often binarized *rule-by-rule*, meaning that we binarize each rule individually (thus splitting it into multiple smaller rules) and are able to create the new grammar from all these newly binarized rules.

4.1 Fusion of NTs

In the rule-by-rule binarization that we will perform here this is accomplished by *fusion* of NTs (called reduction of a rule by Gómez-Rodríguez et al. [Góm+09]): given a rule $r = A \to f(B_1, \ldots, B_\alpha, \ldots, B_\beta, \ldots, B_k)$ we introduce a new NT C and a new rule $C \to f_{fusion}(B_\alpha, B_\beta)$; in the original rule these two NTs are replaced by the new NT C.

[1] The derivations, however, are changed, so we have no strong equivalence.

Definition 4.1 (NT fusion inside a rule)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS.
Let $r_{old} = A \to f(B_1, \dots, B_k) \in R$ with m being the sort/fanout of A, m_i being the sort/fanout of B_i for all $i \in [k]$ and f being defined by:

$$f = \langle x_{\varphi(1,1)} \cdots x_{\varphi(1,l_1)}, \dots, x_{\varphi(m,1)} \cdots x_{\varphi(m,l_m)} \rangle \text{ with some function } \varphi : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

Let $\alpha, \beta \in [k], \alpha < \beta$ be the positions of the NTs to be fused on the right-hand side of r. Introduce a fresh non-terminal $C \notin N$ with the sort/fanout m_C that is indirectly defined below.

Let $u_1, \dots, u_{m_C}$ be the maximal substrings of all components of f (in shorthand form as above) in order of their occurrence, such that for all $i \in [m_C]$: $u_i = y_{i,1} \cdots y_{i,l_i}$ and then for all $j \in [l_i]$: $y_{i,j} = x_{a,b}$ with $a \in \{\alpha, \beta\}$ and $b \in [m_a]$. Then we can specify a composition function:

$$f_{fusion} = \langle \psi(y_{1,1}) \cdots \psi(y_{1,l_1}), \dots, \psi(y_{m_C,1}) \cdots \psi(y_{m_C,l_{m_C}}) \rangle \text{ with } \psi(x_{a,b}) = \begin{cases} x_{1,b} & \text{if } a = \alpha \\ x_{2,b} & \text{if } a = \beta \end{cases}$$

We define these tuples of variables:

$$\text{for all } i \in [k] \backslash \{\alpha, \beta\}: \; t_i = \begin{cases} \langle x_{i+1,1}, \dots, x_{i+1,m_i} \rangle & \text{if } i < \alpha \\ \langle x_{i,1}, \dots, x_{i,m_i} \rangle & \text{if } i > \alpha \wedge i < \beta \\ \langle x_{i-1,1}, \dots, x_{i-1,m_i} \rangle & \text{if } i > \beta \end{cases}$$

$$\text{for } i \in \{\alpha, \beta\}: t'_i = \langle t'_{i,1}, \dots, t'_{i,m_i} \rangle \text{ and for } j \in [m_i]: t'_{i,j} = \begin{cases} x_{1,\lambda} & \exists \lambda \in \mathbb{N}: y_{\lambda,1} = x_{i,j} \\ \varepsilon & \text{otherwise} \end{cases}$$

We can define another composition function by passing these tuples as arguments to the old composition function:

$$f_{rem} = f(t_1, \dots, t_{\alpha-1}, t'_\alpha, t_{\alpha+1}, \dots, t_{\beta-1}, t'_\beta, t_{\beta+1}, \dots, t_k)$$

We denote the entire fusing process by:

$$\text{rulefuse}(r_{old}, \alpha, \beta) = (r_{fusion}, r_{rem}) \quad \text{with}$$

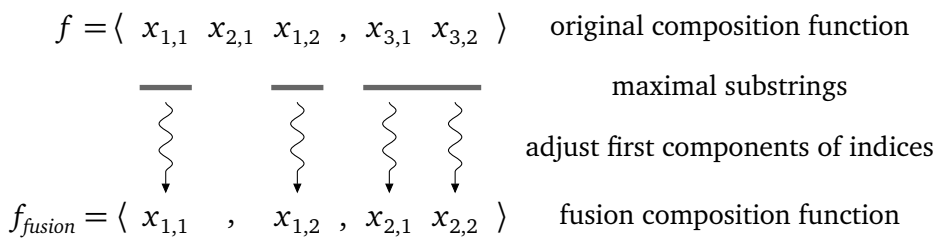
$$r_{fusion} = C \to f_{fusion}(B_\alpha, B_\beta)$$
$$r_{rem} = A \to f_{rem}(C, B_1, \dots, B_{\alpha-1}, B_{\alpha+1}, \dots, B_{\beta-1}, B_{\beta+1}, \dots, B_k)$$

We will call r_{fusion} the fusion rule and r_{rem} the remainder rule. □

Claim 4.2. Let r be some rule with $k = \text{rank}(r)$, $f = \pi_2(r)$ and for some $\alpha, \beta \in [k]$ with $\alpha < \beta$: $(r_{fusion}, r_{rem}) = \text{rulefuse}(r, \alpha, \beta)$, $f_{rem} = \pi_2(r_{rem})$ and $f_{fusion} = \pi_2(r_{fusion})$. Then for correspondingly chosen string tuples $t_1, \dots, t_k$ the following holds:

$$f(t_1, \dots, t_k) = f_{rem}(f_{fusion}(t_\alpha, t_\beta), t_1, \dots, t_{\alpha-1}, t_{\alpha+1}, \dots, t_{\beta-1}, t_{\beta+1}, \dots, t_k)$$

We will show this on the following example.



$t'_1 = \langle x_{1,1}, x_{1,2}\rangle$ $x_{1,1}$ appears as the first component of the *first* maximal substring and $x_{1,2}$ is the first component of the *second* maximal substring, so the new second components of indices are *1* and *2*, respectively

$t_2 = \langle x_{2,1}\rangle$ the original variables belonging to B^1 with the first component of their indices shifted accordingly, so just ±0 here (since $2 > 1 \wedge 2 < 3$)

$t'_3 = \langle x_{1,3}, \varepsilon\rangle$ $x_{3,1}$ appears as the first component of the *third* maximal substring (this is where the 3 in the index comes from), but $x_{3,2}$ does not, so the second component stays empty

$$\begin{aligned} f(t'_1, t_2, t'_3) &= \langle x_{1,1}x_{2,1}x_{1,2}, x_{3,1}x_{3,2}\rangle\big(\langle x_{1,1}, x_{1,2}\rangle, \langle x_{2,1}\rangle, \langle x_{1,3}, \varepsilon\rangle\big) \\ &= \langle\ x_{1,1}\ x_{2,1}\ x_{1,2}\ ,\ x_{1,3}\ \varepsilon\ \rangle \\ &= \langle x_{1,1}x_{2,1}x_{1,2}, x_{1,3}\rangle \\ &= f_{rem} \end{aligned}$$

Figure 4.1: Finding the new composition function in rulefuse for Example 4.3

Example 4.3. Let us consider the rule $r = S^2 \to \langle x_{1,1}x_{2,1}x_{1,2}, x_{3,1}x_{3,2}\rangle(A^2, B^1, C^2)$ (the superscripts of the NTs coincide with their fanouts).

We can now fuse A^2 and C^2 in this rule (generating the fresh NT Z^3 in the process):

$$\begin{aligned} &\text{rulefuse}(r, 1, 3) = \big(Z^3 \to f_{fusion}(A^2, C^2),\ S^2 \to f_{rem}(Z^3, B^1)\big) \\ &f_{fusion} = \langle x_{1,1}, x_{1,2}, x_{2,1}x_{2,2}\rangle \\ &f_{rem} = \langle x_{1,1}x_{2,1}x_{1,2}, x_{1,3}\rangle \end{aligned}$$

where we obtain f_{fusion} and f_{rem} by applying the definition, visualized in Figure 4.1.

On this example we can now show that Claim 4.2 holds (a_1, a_2, b_1, c_1, c_2 stand for some strings):

$$\begin{aligned} &f_{rem}(f_{fusion}(\langle a_1, a_2\rangle, \langle c_1, c_2\rangle), \langle b_1\rangle) \\ &= f_{rem}(\langle a_1, a_2, c_1c_2\rangle, \langle b_1\rangle) \\ &= \langle a_1b_1a_2, c_1c_2\rangle \\ &= f(\langle a_1, a_2\rangle, \langle b_1\rangle, \langle c_1, c_2\rangle) \end{aligned}$$

□

Definition 4.4 (NT fusion inside a PLCFRS)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS.
Let $r_{old} = A \to f(B_1, \dots, B_k) \in R$ and $\alpha, \beta \in [k]$ with $\alpha < \beta$.
We define

$$\text{plcfrsfuse}(G, r_{old}, \alpha, \beta) = (((N \cup \{\pi_1(r_{fusion})\},\ \Sigma,\ S,\ (R \backslash \{r_{old}\}) \cup \{r_{fusion}, r_{rem}\}),\ p')\ ,\ r_{rem})$$

where $(r_{fusion}, r_{rem}) = \text{rulefuse}(r_{old}, \alpha, \beta)$ and p' is the new probability assignment working on the new rules r_{fusion} and r_{rem} defined by:

$$p'(r) = \begin{cases} 1 & \text{if } r = r_{fusion} \\ p(r_{old}) & \text{if } r = r_{rem} \\ p(r) & \text{otherwise} \end{cases}$$

□

So $\text{plcfrsfuse}(G, r_{old}, \alpha, \beta)$ is a tuple consisting of a new PLCFRS (where the rule r_{old} is split into r_{fusion} and r_{rem}) and the rule r_{rem} (which we might continue to fuse in later on).

We want to show that this fusion will not change the semantics of the PLCFRS. To do so we define a function that transforms a derivation of the original PLCFRS into a derivation in the new PLCFRS.

Definition 4.5 (derivation transformer ϕ)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS and $r_{old} \in R$ any rule with $\text{rank}(r) > 2$ of it. $(G', r_{rem}) = \text{plcfrsfuse}(G, r_{old}, \alpha, \beta) = (((N', \Sigma, S, R'), p'), r_{rem})$ is the PLCFRS resulting from fusing in r_{old} to obtain $r_{fusion}, r_{rem} \in R'$ with $f_{rem} = \pi_2(r_{rem})$ and $f_{fusion} = \pi_2(r_{fusion})$.
Then $\phi : D_G \to D_{G'}$ transforms a derivation of the original PLCFRS into a derivation of the new PLCFRS.

$$\phi(r(d_1, \dots, d_k)) = \begin{cases} r_{rem}(r_{fusion}(\phi(d_\alpha), \phi(d_\beta)), \phi(d_1), \dots, \phi(d_{\alpha-1}), \phi(d_{\alpha+1}), & \text{if } r = r_{old} \\ \quad \dots, \phi(d_{\beta-1}), \phi(d_{\beta+1}), \dots, \phi(d_k)) & \\ r(\phi(d_1), \dots, \phi(d_k)) & \text{otherwise} \end{cases}$$

□

Lemma 4.6. ϕ is a bijection.

Proof. Since we have a construction for ϕ itself, we only need to show the fact that an inverse function ϕ^{-1} exists.

The construction of ϕ^{-1} is not difficult, we simply reverse the construction given above. This is possible, because we can guarantee that r_{rem} and r_{fusion} can only appear in the form given above - this is a logical consequence of the way these two rules are constructed: the fresh NT from the fusion only appears on a right-hand side as the first NT of r_{rem} and only on the left-hand side of r_{fusion} (since we made sure it was a fresh NT), so in any derivation, r_{fusion} must always be the first child of r_{rem}. ■

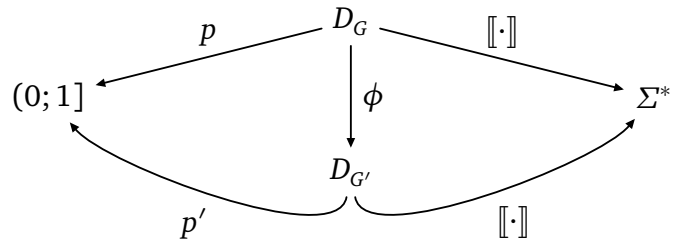


Figure 4.2: Sketch of the proof for Theorem 4.9, the left half is shown in Lemma 4.7, the right half in Lemma 4.8

Lemma 4.7. Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS and $r_{old} \in R$ any rule of it with $\text{rank}(r) > 2$.

Let $(G', r_{rem}) = \text{plcfrsfuse}(G, r_{old}, \alpha, \beta)$ be the PLCFRS resulting from applying a fusion in r_{old} for some $\alpha, \beta \in [\text{rank}(r)]$ with $\alpha < \beta$. Let $p' = \pi_2(G')$.

Then for any $d \in D_G$: $p(d) = p'(\phi(d))$.

Proof. Prove that $p(d) = p'(\phi(d))$ by induction.

Base case: Let $d = r()$ for some $r = A \to f() \in R$. Since $\text{rank}(r_{old}) > 2$, it follows that $r \neq r_{old}$ and thus $r \in R'$, so:

$$\begin{aligned}
p(r()) &= p(r) && \text{(by Def. 2.8)}\\
&= p'(r) && \text{(by Def. 4.4)}\\
&= p'(r()) && \text{(by Def. 2.8)}\\
&= p'(\phi(r())) && \text{(by Def. 4.5)}
\end{aligned}$$

Induction hypothesis: Let $d \in D_G$, then $p(d) = p'(\phi(d))$.

Induction step: Let $d = r(d_1, \ldots, d_k)$ for some $r = A \to f(\pi_1(d_1(\varepsilon)), \ldots, \pi_1(d_k(\varepsilon))) \in R$.

If $r = r_{old}$:

$$\begin{aligned}
&p(r_{old}(d_1, \ldots, d_k))\\
&= p(r_{old}) \cdot \prod_{i \in [k]} p(d_i) && \text{(by Def. 2.8)}\\
&= p(r_{old}) \cdot p(d_\alpha) \cdot p(d_\beta) \cdot \prod_{i \in [k], \alpha \neq i \neq \beta} p(d_i) && \text{(by commutativity of } \cdot)\\
&= p(r_{old}) \cdot 1 \cdot p(d_\alpha) \cdot p(d_\beta) \cdot \prod_{i \in [k], \alpha \neq i \neq \beta} p(d_i) && \text{(by 1 being the neutral element of } \cdot)\\
&= p'(r_{rem}) \cdot p'(r_{fusion}) \cdot p(d_\alpha) \cdot p(d_\beta) \cdot \prod_{i \in [k], \alpha \neq i \neq \beta} p(d_i) && \text{(by Def. 4.4)}
\end{aligned}$$

$$= p'(r_{rem}) \cdot p'(r_{fusion}) \cdot p'(\phi(d_\alpha)) \cdot p'(\phi(d_\beta)) \cdot \prod_{i \in [k], \alpha \neq i \neq \beta} p'(\phi(d_i))$$

(by induction hypothesis)

$$= p'(r_{rem}) \cdot p'(r_{fusion}(\phi(d_\alpha), \phi(d_\beta))) \cdot \prod_{i \in [k], \alpha \neq i \neq \beta} p'(\phi(d_i))$$

(by Def. 2.8, associativity of $\cdot$)

$$= p'(r_{rem}(r_{fusion}(\phi(d_\alpha), \phi(d_\beta)), \phi(d_1), \dots, \phi(d_{\alpha-1}), \phi(d_{\alpha+1}), \dots, \phi(d_{\beta-1}), \phi(d_{\beta+1}), \dots, \phi(d_k))) \quad \text{(by Def. 2.8)}$$

$$= p'(\phi(r_{old}(d_1, \dots, d_k))) \quad \text{(by Def. 4.5)}$$

Otherwise (again, since $r \neq r_{old}$, it follows that $r \in R'$):

$$\begin{aligned} p(r(d_1, \dots, d_k)) &= p(r) \cdot p(d_1) \cdot \dots \cdot p(d_k) && \text{(by Def. 2.8)} \\ &= p'(r) \cdot p(d_1) \cdot \dots \cdot p(d_k) && \text{(by Def. 4.4)} \\ &= p'(r) \cdot p'(\phi(d_1)) \cdot \dots \cdot p'(\phi(d_k)) && \text{(by induction hypothesis)} \\ &= p'(r(\phi(d_1), \dots, \phi(d_k))) && \text{(by Def. 2.8)} \\ &= p'(\phi(r(d_1, \dots, d_k))) && \text{(by Def. 4.5)} \end{aligned}$$

■

Lemma 4.8. Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS and $r_{old} \in R$ any rule of it with $\text{rank}(r) > 2$.

Let $(G', r_{rem}) = \text{plcfrsfuse}(G, r_{old}, \alpha, \beta)$ be the PLCFRS resulting from applying a fusion in r_{old} for some $\alpha, \beta \in [\text{rank}(r)]$ with $\alpha < \beta$.

Then for any $d \in D_G$: $[\![d]\!] = [\![\phi(d)]\!]$.

Proof. Prove that $[\![d]\!] = [\![\phi(d)]\!]$ by induction.

Base case: Let $d = r()$ for some $r = A \to f() \in R$. Since the $\text{rank}(r_{old}) > 2$, it follows that $r \neq r_{old}$ and thus $r \in R'$, so:

$$[\![r()]\!] = [\![\phi(r())]\!] \quad \text{(by Def. 4.5)}$$

Induction hypothesis: Let $d \in D_G$, then $[\![d]\!] = [\![\phi(d)]\!]$.

Induction step: Let $d = r(d_1, \dots, d_k)$ for some $r = A \to f(\pi_1(d_1(\varepsilon)), \dots, \pi_1(d_k(\varepsilon))) \in R$.

If $r = r_{old}$:

$$\begin{aligned} & [\![r_{old}(d_1, \dots, d_k)]\!] \\ &= f([\![d_1]\!], \dots, [\![d_k]\!]) && \text{(by Def. 2.4)} \\ &= f([\![\phi(d_1)]\!], \dots, [\![\phi(d_k)]\!]) && \text{(by induction hypothesis)} \end{aligned}$$

$$
\begin{aligned}
&= f_{rem}(f_{fusion}([\![\phi(d_\alpha)]\!], [\![\phi(d_\beta)]\!]), [\![\phi(d_1)]\!], \ldots, [\![\phi(d_{\alpha-1})]\!], [\![\phi(d_{\alpha+1})]\!], \\
&\quad \ldots, [\![\phi(d_{\beta-1})]\!], [\![\phi(d_{\beta+1})]\!], \ldots, [\![\phi(d_k)]\!]) && \text{(by Claim 4.2)} \\
&= f_{rem}([\![r_{fusion}(\phi(d_\alpha), \phi(d_\beta))]\!], [\![\phi(d_1)]\!], \ldots, [\![\phi(d_{\alpha-1})]\!], [\![\phi(d_{\alpha+1})]\!], \\
&\quad \ldots, [\![\phi(d_{\beta-1})]\!], [\![\phi(d_{\beta+1})]\!], \ldots, [\![\phi(d_k)]\!]) && \text{(by Def. 2.4)} \\
&= [\![r_{rem}(r_{fusion}(\phi(d_\alpha), \phi(d_\beta)), \phi(d_1), \ldots, \phi(d_{\alpha-1}), \phi(d_{\alpha+1}), \ldots, \\
&\quad \phi(d_{\beta-1}), \phi(d_{\beta+1}), \ldots, \phi(d_k))]\!] && \text{(by Def. 2.4)} \\
&= [\![\phi(r_{old}(d_1, \ldots, d_k))]\!] && \text{(by Def. 4.5)}
\end{aligned}
$$

Otherwise (again, since $r \neq r_{old}$, it follows that $r \in R'$):

$$
\begin{aligned}
[\![r(d_1, \ldots, d_k)]\!] &= f([\![d_1]\!], \ldots, [\![d_k]\!]) && \text{(by Def. 2.4)} \\
&= f([\![\phi(d_1)]\!], \ldots, [\![\phi(d_k)]\!]) && \text{(by induction hypothesis)} \\
&= [\![r(\phi(d_1), \ldots, \phi(d_k))]\!] && \text{(by Def. 2.4)} \\
&= [\![\phi(r(d_1, \ldots, d_k))]\!] && \text{(by Def. 4.5)}
\end{aligned}
$$

■

Theorem 4.9. Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS and $r_{old} \in R$ any rule of it with $\text{rank}(r) > 2$.

Let $(G', r_{rem}) = \text{plcfrsfuse}(G, r_{old}, \alpha, \beta)$ be the PLCFRS resulting from applying a fusion in r_{old} for some $\alpha, \beta \in [\text{rank}(r)]$ with $\alpha < \beta$. Let $p' = \pi_2(G')$.

Then $[\![G]\!] = [\![G']\!]$.

Proof. Since the claim $[\![G]\!] = [\![G']\!]$ is equivalent to the claim that for all $w \in \Sigma^*$: $[\![G]\!](w) = [\![G']\!](w)$ this equality is the one we will prove:

$$
\begin{aligned}
&[\![G]\!](w) \\
&= \sum_{d \in D_G : [\![d]\!] = w} p(d) && \text{(by Def. 2.9)} \\
&= \sum_{d \in D_G : [\![d]\!] = w} p'(\phi(d)) && \text{(by Lem. 4.7)} \\
&= \sum_{d \in D_G : [\![\phi(d)]\!] = w} p'(\phi(d)) && \text{(by Lem. 4.8)} \\
&= \sum_{\phi(d) \in D_{G'} : [\![\phi(d)]\!] = w} p'(\phi(d)) && \text{(by Lem. 4.6)} \\
&= [\![G']\!](w) && \text{(by Def. 2.9)}
\end{aligned}
$$

■

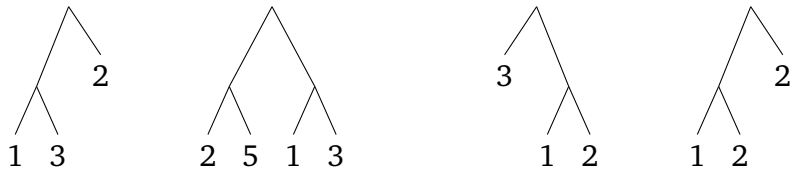


Figure 4.3: $((1,3),2)$, $((2,5),(1,3))$, $(3,(1,2))$ and $((1,2),2)$ visualized as trees

4.2 Complete binarization of a rule

We can fuse two NTs of an rule, thereby reducing its rank by 1. But, of course, we also want to binarize rules with rank > 3, therefore we will have to define a sequence of such fusions to describe a *complete binarization* of a rule. This binarization blueprint will be a binary tree (i.e. nested 2-tuples) defined as follows.

Definition 4.10 (binarization blueprint)**.** A *binarization blueprint* describes a complete binarization for some rule r with $k = \text{rank}(r) > 0$. The set of all binarization blueprints for some $k \in \mathbb{N}$ is defined by:

$$\Theta(k) = [k] \cup \{(\theta_1, \theta_2) \mid \theta_1, \theta_2 \in \Theta(k),\ \text{ind}(\theta_1) \cap \text{ind}(\theta_2) = \emptyset,\ \theta_1 < \theta_2\}$$

with $\Theta = \bigcup_{k \in \mathbb{N}} \Theta(k)$, the set of indices contained in a blueprint $\theta \in \Theta$ being defined by

$$\text{ind}(\theta) = \begin{cases} \{i\} & \theta = i \in \mathbb{N} \\ \text{ind}(\theta_1) \cup \text{ind}(\theta_2) & \theta = (\theta_1, \theta_2) \in \Theta \times \Theta \end{cases}$$

and the order $<$ is imposed on all $\theta \notin \mathbb{N}$ by:

$$\theta_1 < \theta_2 \text{ if } \theta_1 \in \Theta \times \Theta, \theta_2 \in \mathbb{N} \text{ and } (\theta_1^1, \theta_1^2) < (\theta_2^1, \theta_2^2) \text{ if } \theta_1^1 < \theta_2^1 \vee (\theta_1^1 = \theta_2^1 \wedge \theta_1^2 < \theta_2^2)$$

A binarization blueprint $\theta \in \Theta(k)$ for some $k \in \mathbb{N}$ is *total in* $\Theta(k)$ if $\text{ind}(\theta) = [k]$, otherwise it is *partial*. □

Example 4.11. $((1,3),2)$ is a total binarization blueprint in $\Theta(3)$ and $((2,5),(1,3))$ is a partial blueprint in $\Theta(k)$ for some $k > 5$ while $(3,(1,2))$ and $((1,2),2)$ are no binarization blueprints at all. These examples are visualized as trees in Figure 4.3. □

The natural numbers at the leaves of these blueprints shall be indices for the NTs on the right-hand side of a rule.

However, the problem with binarizing a rule fusion by fusion is that the positions of the NT occurrences inside a rule's right-hand side (and thus the indices, our only way to address them) do not stay the same when fusing NTs in the rule.

If we for example fuse the first and second NT of some rule $A \to f(B_1, B_2, B_3)$ to create a new NT C we get the new remainder rule $A \to f_{rem}(C, B_3)$. The original third NT is now the second and any information contained in a binarization blueprint about this third NT can no longer be associated with B_3. This is why we want consistent indices for the right-hand sides of our rules.

Definition 4.12 (fusing with consistent indices)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS. We define some helper functions to associate the NTs on the right-hand side of a rule with *consistent indices* (natural numbers):

$$\text{indexWith}(A \to f(B_1, \dots, B_k), (i_1, \dots, i_k)) = A \to f((B_1, i_1), \dots, (B_k, i_k))$$
$$\text{deIndex}(A \to f((B_1, i_1), \dots, (B_k, i_k))) = (A \to f(B_1, \dots, B_k), (i_1, \dots, i_k))$$
$$\text{firstI}(A \to f((B_1, i_1), \dots, (B_k, i_k))) = i_1$$

Now we can make plcfrsfuse work on these consistent indices:

$$\widetilde{\text{plcfrsfuse}}(G, \widetilde{r}, \widetilde{\alpha}, \widetilde{\beta}) = (G', \widetilde{r}') \text{ where}$$
$$(r, (i_1, \dots, i_k)) = \text{deIndex}(\widetilde{r})$$
$$(G', r') = \text{plcfrsfuse}(G, r, \alpha, \beta) \text{ where } \alpha, \beta \in [k] \text{ such that } i_\alpha = \widetilde{\alpha} \text{ and } i_\beta = \widetilde{\beta}$$
$$\widetilde{r}' = \text{indexWith}(r', (\max\{i_1, \dots, i_k\} + 1, i_1, \dots, i_{\alpha-1}, i_{\alpha+1}, \dots, i_{\beta-1}, i_{\beta+1}, \dots, i_k))$$

□

Definition 4.13 (complete binarization of a rule)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS. The *complete binarization of a rule* $r \in R$ *with* $k = \text{rank}(r) > 0$ *given a total binarization blueprint* $\theta = (\theta^a, \theta^b) \in \Theta(k)$ is defined as the new resulting PLCFRS $\text{binarizeBy}(G, r, \theta)$:

$$\text{binarizeBy}(G, r, \theta) = \begin{cases} G & \text{if } \theta \in [k] \times [k] \text{ or } \theta \in [k] \\ \pi_1(\text{tfuse}(G, \widetilde{r}, \theta^a)) & \text{if } \theta = (\theta^a, \theta^b) \in \Theta(k) \times [k] \\ \pi_1(\text{tfuse}(G', \widetilde{r}', \theta^b)) & \text{if } \theta = (\theta^a, \theta^b) \in \Theta(k) \times \Theta(k) \\ & \quad \text{where } (G', \widetilde{r}') \\ & \quad\quad = \text{tfuse}(G, \widetilde{r}, \theta^a) \\ & \\ & \quad \text{with } \widetilde{r} = \text{indexWith}(r, (1, \dots, k)) \end{cases}$$

$$\text{tfuse}(G, \widetilde{r}, \theta) = \begin{cases} \widetilde{\text{plcfrsfuse}}(G, \widetilde{r}, \widetilde{\alpha}, \widetilde{\beta}) & \text{if } \theta = (\widetilde{\alpha}, \widetilde{\beta}) \in [k] \times [k] \\ \widetilde{\text{plcfrsfuse}}(G', \widetilde{r}', \text{firstI}(\widetilde{r}'), \widetilde{\alpha}) & \text{if } \theta = (\theta', \widetilde{\alpha}) \in \Theta(k) \times [k] \\ \quad \text{where } (G', \widetilde{r}') = \text{tfuse}(G, \widetilde{r}, \theta') & \\ \widetilde{\text{plcfrsfuse}}(G'', \widetilde{r}'', \text{firstI}(\widetilde{r}''), \text{firstI}(\widetilde{r}')) & \text{if } \theta = (\theta', \theta'') \in \Theta(k) \times \Theta(k) \\ \quad \text{where } (G'', \widetilde{r}'') = \text{tfuse}(G, \widetilde{r}', \theta'') & \\ \quad \text{and } (G', \widetilde{r}') = \text{tfuse}(G, \widetilde{r}, \theta') & \end{cases}$$

□

4.3 Naive binarization

Given a PLCFRS $G = ((N, \Sigma, S, R), p)$ naive binarization is not very difficult once we have the concept of complete rule binarizations as explained above. For each rule $r \in R$ with $\text{rank}(r) > 2$ we just always fuse the last two NT occurrences on the right-hand side of r.

Definition 4.14 (naive binarization)**.** For every LCFRS (N, Σ, S, R) the *naive binarization blueprint* $\theta(k) \in \Theta(k)$ is defined for each rule $r \in R$ with $\text{rank}(r) = k$ by:

$$\breve{\theta}(k) = \breve{\theta}(k, 1) \text{ with } \breve{\theta}(k, i) = \begin{cases} 1 & \text{if } k = i \\ (\breve{\theta}(k, i+1), i) & \text{otherwise} \end{cases}$$

□

Example 4.15. The naive binarization of a rule with rank 1 is just $\breve{\theta}(1) = 1$ and for rank 2 we obtain $\breve{\theta}(2) = (1, 2)$. These blueprints are obviously useless, since rules with a rank lower than 3 do not need to be binarized, but they can still be constructed, since binarizeBy is defined to return the given PLCFRS itself when encountering such an useless blueprint.

For rules of rank 3 the naive blueprint is $\breve{\theta}(3) = ((2, 3), 1)$, for rank 4 we have $\breve{\theta}(4) = (((3, 4), 2), 1)$ and so on. □

4.4 Simulating binarization blueprints

We have now established the framework for binarizing a rule according to blueprints and have even specified the naive binarization by giving a simple blueprint we can always use.

However, we of course want to construct better blueprints that result in a binarization producing rules of a lower fanout.

The search for such blueprints will rate a constructed blueprint θ by calculating the fanouts of rules θ would create in binarizeBy. It will do so by inspecting the composition function of the rule it was made for.

To define this fanout calculation, we first need a way to locate variables belonging to specific NT occurrences of a rule in its composition function.

Definition 4.16 (variable occurrences in composition functions)**.** Given a rule r with $\text{rank}(r) = k$ and $\pi_2(r) = f$, we can describe f as:

$$f = \langle x_{\varphi(1,1)} \cdots x_{\varphi(1,l_1)}, \ \ldots, \ x_{\varphi(m,1)} \cdots x_{\varphi(m,l_m)} \rangle \text{ with some function } \varphi : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

We can specify all occurrences of variables belonging to one NT occurrence $n \in [k]$ on the right-hand side of r by:

$$\text{varpositions}(r, n) = \big(\{\rho \mid \exists i \in \mathbb{N} \colon x_{\varphi(1,\rho)} = x_{n,i}\}, \ldots, \{\rho \mid \exists i \in \mathbb{N} \colon x_{\varphi(m,\rho)} = x_{n,i}\}\big)$$

□

Definition 4.17 (simulating binarization blueprints)**.** Given a rule r with $\text{rank}(r) = k$ we can *simulate a binarization blueprint* $\theta \in \Theta(k)$:

$$\text{simulate}(r, \theta) = \begin{cases} \text{varpositions}(r, \theta) & \text{if } \theta \in [k] \\ \text{merge}(\text{simulate}(r, \theta_1), \text{simulate}(r, \theta_2)) & \text{if } \theta = (\theta_1, \theta_2) \in \Theta \times \Theta \end{cases}$$

where $\text{merge}((P_1^a, \ldots, P_m^a), (P_1^b, \ldots, P_m^b)) = (P_1^a \cup P_1^b, \ldots, P_m^a \cup P_m^b)$. □

Example 4.18. Recall rule $r = S^2 \to \langle x_{1,1} x_{2,1} x_{1,2}, x_{3,1} x_{3,2} \rangle (A^2, B^1, C^2)$ from example 4.3.

The binarization blueprint we followed in the previous example (without knowing it was one) was $((1,3),2)$. The blueprint suggested by the naive binarization would have been $((2,3),1)$. Let us simulate this blueprint. We calculate $\text{simulate}(r, ((2,3),1))$ iteratively:

$$\begin{aligned}
&\text{simulate}(r,1) = \text{varpositions}(r,1) = (\{1,3\}, \emptyset) \\
&\text{simulate}(r,2) = \text{varpositions}(r,2) = (\{2\}, \emptyset) \\
&\text{simulate}(r,3) = \text{varpositions}(r,3) = (\emptyset, \{1,2\}) \\
&\text{simulate}(r,(2,3)) = \text{merge}(\text{simulate}(r,2), \text{simulate}(r,3)) \\
&\qquad = \text{merge}\big((\{2\}, \emptyset), (\emptyset, \{1,2\})\big) = (\{2\}, \{1,2\}) \\
&\text{simulate}(r,((2,3),1)) = \text{merge}(\text{simulate}(r,(2,3)), \text{simulate}(r,1)) \\
&\qquad = \text{merge}\big((\{2\}, \{1,2\}), (\{1,3\}, \emptyset)\big) = ([3], [2])
\end{aligned}$$

Note that because the indices of θ span the whole right-hand side of r (thus θ is total), the set of positions $\text{simulate}(r, ((2,3),1))$ reaches over the whole composition function. □

Our reason for defining this simulation process is that we can now easily study the fanout of rules the binarization process would create from a binarization blueprint.

Definition 4.19 (fanout of a tuple of sets)**.** Similar to the definition of fanout: $\mathcal{P}(S) \to \mathbb{N}$ for some ordered set S we can define:

$$\widehat{\text{fanout}}\colon (\mathcal{P}(S))^* \to \mathbb{N}$$
$$\widehat{\text{fanout}}(S_1, \ldots, S_m) = \text{fanout}(S_1) + \cdots + \text{fanout}(S_m)$$

□

We can use this to check a binarization blueprint θ of a rule $r \in R$ for some PLCFRS $G = ((N, \Sigma, S, R), p)$ for the maximal fanout of the new rules being created in binarizeBy(G, r, θ).

Definition 4.20 (maximal fanout of a binarization blueprint)**.** Given a rule r and a binarization blueprint $\theta \in \Theta(\text{rank}(r))$ we define:

$$\text{maxfanout}(r, \theta) = \begin{cases} 0 & \text{if } \theta \in \mathbb{N} \\ \max\Big\{\widehat{\text{fanout}}(\text{simulate}(r, \theta)), & \text{if } \theta = (\theta_1, \theta_2) \in \Theta \times \Theta \\ \qquad \text{maxfanout}(r, \theta_1), \text{maxfanout}(r, \theta_2)\Big\} & \end{cases}$$

□

Example 4.21. We continue the previous example and consider the blueprint $\theta = ((2,3),1)$.

$$\begin{aligned}
&\text{maxfanout}(r, ((2,3),1)) \\
&= \max\Big\{\widehat{\text{fanout}}(\text{simulate}(r, ((2,3),1))), \text{maxfanout}(r, (2,3)), \text{maxfanout}(r, 1)\Big\} \\
&= \max\Big\{\widehat{\text{fanout}}([3],[2]), \text{maxfanout}(r, (2,3)), 0\Big\} \\
&= \max\Big\{2, \text{maxfanout}(r, (2,3)), 0\Big\} \\
&= \max\Big\{2, \max\Big\{\widehat{\text{fanout}}(\text{simulate}(r, (2,3))), \text{maxfanout}(r, 2), \text{maxfanout}(r, 3)\Big\}, 0\Big\} \\
&= \max\Big\{2, \max\Big\{\widehat{\text{fanout}}([1;2],[2]), 0, 0\Big\}\Big\} \\
&= \max\Big\{2, \max\Big\{2, 0, 0\Big\}, 0\Big\} \\
&= 2
\end{aligned}$$

□

4.5 Optimal binarization

It is easy to see that we could list all possible binarization blueprints $\Theta(k)$ of a rule r with $\text{rank}(r) = k$; the set is finite[2], even if the number of elements increases rapidly with k. We can however limit ourselves to only constructing specifically the subset of binarization blueprints $\Theta^m(k)$ that contains all these $\theta \in \Theta(k)$ for which $\text{maxfanout}(r, \theta) \leq m$ by introducing a constraint based on $\widehat{\text{fanout}}$ into Definition 4.10:

[2] because $\text{ind}(\theta)$ for a blueprint θ grows with each new addition to θ, but can never exceed $[k]$

Definition 4.22 (bounded binarization blueprint)**.** Let $m \in \mathbb{N}$ and r be a rule with $k = \text{rank}(r)$.
The we can define the *set of all binarization blueprints for the rule r bounded by m* as a subset of $\Theta(k)$:

$$\Theta^m(r) = [k] \cup \Big\{(\theta_1, \theta_2) \mid \theta_1, \theta_2 \in \Theta^m(r),\ \text{ind}(\theta_1) \cap \text{ind}(\theta_2) = \emptyset,\ \theta_1 < \theta_2,\ \widehat{\text{fanout}}\Big(\text{merge}\Big(\text{simulate}(r, \theta_1), \text{simulate}(r, \theta_2)\Big)\Big) \leq m\Big\}$$

with ind and $<$ being defined like in 4.10. □

Using this construction idea we no longer have to create all blueprints, but only the ones that do not (yet) yield a rule with a fanout larger than our target m.

We can now use this set to get the set of all *optimal*[3] binarization blueprints:

Definition 4.23 (optimal binarization blueprints)**.** Let $((N, \Sigma, S, R), p)$ be a PLCFRS. We can define the optimal binarization blueprint of any rule $r \in R$ starting the search at some $m \in \mathbb{N}$:

$$\widehat{\theta}(r, m) \begin{cases} \widehat{\theta}(r, m+1) & \text{if } \Theta = \emptyset \\ \theta & \text{otherwise choose any } \theta \in \Theta \\ \\ \text{with } \Theta = \{\theta \mid \theta \in \Theta^m(r),\ \text{ind}(\theta) = [\text{rank}(r)]\} \end{cases}$$

We will just write $\widehat{\theta}(r)$ instead of $\widehat{\theta}(r, \pi_2(sort(r)))$. □

We can give this lower bound because the final remainder rule of the binarization (the one that is described by the root of a blueprint) will always have the same original NT on the the left-hand side and its fanout is $\pi_2(sort(r))$, so we know that no binarization could achieve a lower maximum fanout.

4.6 Complete binarization of a PLCFRS

Definition 4.24 (complete binarization of a PLCFRS)**.** Let $G = ((N, \Sigma, S, R), p)$ be a PLCFRS. Given a function $\widetilde{\theta} : R \to \Theta$ we can define the *complete binarization of this PLCFRS*:

$$\text{binarizeUsing}(G, \widetilde{\theta}) = \begin{cases} G & \text{if } R' = \emptyset \\ \text{binarizeUsing}\Big(\text{binarizeBy}\big(G, r, \widetilde{\theta}(r)\big), \widetilde{\theta}\Big) & \text{otherwise choose any } r \in R' \end{cases}$$

with $R' = \{r \mid r \in R,\ \text{rank}(r) > 2\}$. □

[3] I will use the term optimal to refer to binarizations that have the lowest possible maximal fanout in all resulting rules, other metrics are also imaginable and could be targeted with the framework of binarization blueprints.

Thus the naive complete binarization of a PLCFRS G is binarizeBy$(G, \widetilde{\theta})$ where $\widetilde{\theta}(r) = \check{\theta}(\text{rank}(r))$.

The optimal complete binarization is binarizeBy$(G, \widehat{\theta})$.

5 Implementation

5.1 Representing a PLCFRS

5.1.1 Rules

The PLCFRS we considered so far all worked on strings. Since working with strings tends to be needlessly expensive in memory, we replace all strings (node labels) of our treebank with natural numbers, and create a dictionary that will allow us to find the strings corresponding to numbers again. Once we thus *intified* our input trees, the sets N, Σ and S of the resulting PLCFRS of course only consist of these natural numbers and for every rule $r = A \to f(B_1, \ldots, B_k)$ we now know that $A, B_1, \ldots, B_k \in \mathbb{N}$. We realize that to store a PLCFRS $((N, \Sigma, S, R), p)$ we really only need a set of rules R (with their composition functions, which imply the alphabet Σ), the set of start non-terminals S (since R and S imply the set of all non-terminals N) and the probability assignment p. This thought leads to the first type definition for our implementation, the definition of an (unweighted) rule.

```
type Rule = ((Int, [Int]), [[NTT]])
```

The first component is a tuple consisting of the left-hand side NT and the NTs on the right-hand side (here represented as a list). The second component of the tuple defines the composition function used in the rule: The outer list represents the components of the tuple this function returns (its length is the fanout of the NT on the left-hand side), the inner list represents the concatenation of new terminal symbols and variables (like the $\langle y_1, \ldots, y_s \rangle$ tuple used to define a composition function), represented in the data structure `NTT`:

```
data NTT = NT Int | T Int
```

This NTTs Ts are indeed the new terminal symbols, the NTs however are the variables that we used to define a composition function, we can use a linear index here, since the number of variables that stand for strings in the input is fixed and known.

Since p assigns each rule a probability, it makes sense to store weighted rules as tuples of unweighted rules and probabilities, so an example weighted rule might look like this:

```
(((1, [2,3]), [[NT 3, NT 1], [NT 2]]), 0.4711) :: (Rule, Double)
```

A whole PLCFRS thus has the following type (initial NTs, rules, NT/T intification dictionaries):

```
type PLCFRS = ([Int], [(Rule, Double)], (Array Int String, Array Int String))
```

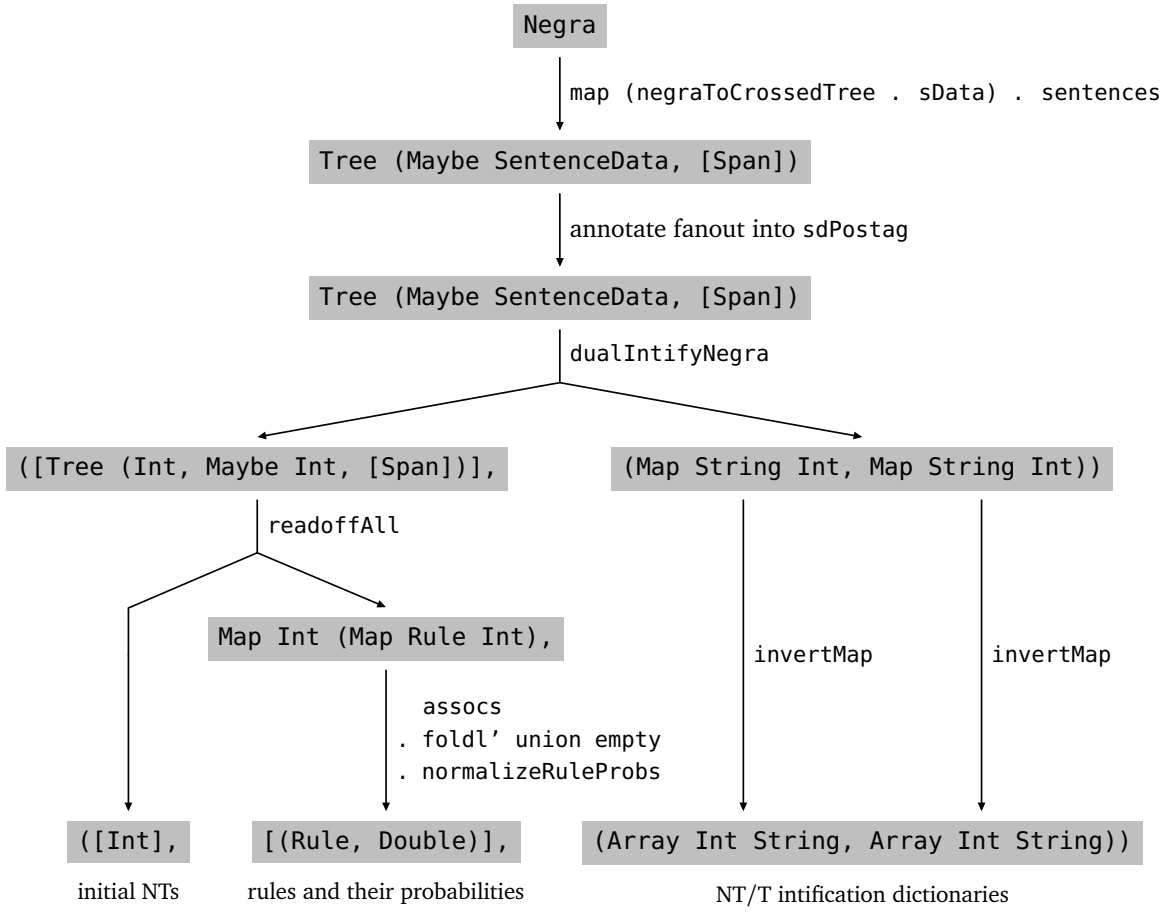


Figure 5.1: Data flow and type signatures during the extraction process

5.1.2 IRTGs

We can also express an LCFRS G as a *interpreted regular tree grammar* (IRTG, Koller and Kuhlmann [KK11]), which consists of a regular tree grammar that generates derivations and a homomorphism that assigns each node of the derivation a function that can then be evaluated in an algebra. Our design is not so different, the generated derivations are the set of derivations D_G, the homomorphism would apply π_2 to each node so that the resulting tree of composition functions can then be evaluated in the algebra (so this process is exactly like our definition of the semantics of a derivation).

5.2 Extraction

The extraction process works with many functions, transforming a corpus into trees and trees into rules. The data flow is sketched in figure 5.1.

This leads us to the type signature of the function that encapsulates this process:

```
extractPLCFRSFromNegra :: Negra -> PLCFRS
```

5.2.1 Cleaning and preparing input trees

Using the machine translation system *Vanda*[1] to parse the TIGER and NEGRA treebanks results in trees similar to the ξs we defined - the nodes of these trees are labeled with `SentenceData`:

```
data SentenceData
  = SentenceWord
    { sdWord :: String
    , sdPostag :: String
    , sdMorphtag :: String
    , sdEdge :: Edge
    , sdSecEdges :: [Edge]
    , sdComment :: Maybe String
    }
  | SentenceNode
    { sdNum :: Int
    , sdPostag :: String
    , sdMorphtag :: String
    , sdEdge :: Edge
    , sdSecEdges :: [Edge]
    , sdComment :: Maybe String
    }
```

Listing 5.1: `SentenceData` as defined in Vanda.Corpus.Negra

However, the leaves of these trees (`SentenceWord`, inner nodes are `SentenceNode`) contain two labels, one for an inner node (`sdPostag`) and one for the actual leaf (`sdWord`), these would be unrolled into two separate nodes of rank 1 and 0 in our ξs:

Furthermore, they do not define an order $\preceq$ like we did, but in the tree representation `negraToCrossedTree` gives us each node of the tree is already annotated with its spans[2], so we do not need the order.

This span-annotation was one of the two things we had to do manually using the function annotate when formally describing the extraction process, the other thing this function did is something that we still have left to do here: we have to add their fanouts to all part-of-speech tags so that later we are able to distinguish between NTs of the same name with different fanouts.

[1] `http://www.inf.tu-dresden.de/index.php?node_id=2550`

[2] The type `[Span]` stands for `[(Int, Int)]` and contains the representation of the spans as the union of maximal intervals sorted by their smallest element that we used in our definitions.

Finally we already perform the intification described above and replace the natural language words (becoming our terminal symbols) and the part-of-speech tags (we just enriched these with their fanouts, so they can become proper NTs).

Now that we have properly prepared trees, we can begin the actual readoff procedure:

5.2.2 Reading off rules

Now we can apply the readoff transformation to the trees.

```
readoffAll
  :: [(Tree (Int, Maybe Int, [Span]))]
  -> (Set Int, M.Map Int (M.Map Rule Int))
readoffNode
  :: (Tree (Int, Maybe Int, [Span]))
  -> Rule
solveSpanPuzzle :: [Span] -> Span -> [Int]
```

Listing 5.2: Types of functions used for reading off rules as defined in Vanda.Corpus.Negra

`readoffAll` folds over all trees and returns a set of all NTs at the roots of these trees (which will become the initial NTs) and a map of all rules and their counts (still categorized by the NT on the left-hand side to make obtaining the probabilities from the counts easier).

`readoffNode` reads off a single rule from a single node (we pass it the whole tree for the sake of simplicity). Like our definition of readoff in chapter 3 it relies on a puzzle function, which works exactly the same way, it just has flipped arguments and returns linear indices.

The rules obtained by each `readoffNode` call are inserted into the rule map in `readoffAll` with a count of 1 or if they already are present in the map, this entry's count is increased by 1.

Once we have this map of rules and counts, we can easily calculate a flat list of rules and their probabilities.

5.3 Binarization

5.3.1 ProtoRules

To avoid ending up with many more new NTs than we need (because it is quite likely that the same new rule and with it the same new NT on its left-hand side is created many times when binarizing many rules like these read off from a natural language tree corpus) the (de)duplication of equivalent NTs (and with it equivalent rules) is something that has to be addressed by giving new NTs proper names:

Instead of immediately saving the new NTs as Integers with the other intified NTs, each NT will have to be a representation of the fusion rule (or rather its semantics) for which it was created, so that if this rule and this NT is created once more, the created NT would be the same (implying with it the same rule). Now we only have to check the generated rules for duplicates and remove them to end up with a lot less NTs and rules (see the chapter 6, p. 43).

This representation shall be a binary tree where the leaves are labeled with NTs of the original non-binarized LCFRS and an inner node is labeled with the composition functions of the fusion rule resulting from fusing the two NT it has as children. Their structure is thus similar to that of the binarization blueprints we defined.

Using these `NTRepTrees` we can define `ProtoNTs` [3] and with these `ProtoRules` that are very similar to the normal `Rules` we defined above.

```
data NTRepTree = NTRepLeaf Int -- original NTs at the leaves
               | NTRepInner [[NTT]] NTRepTree NTRepTree
type ProtoNT = (NTRepTree, Int) -- unintified ProtoNTs carry their fanout
type ProtoRule = ((ProtoNT, [ProtoNT]), [[NTT]])
```

Listing 5.3: `ProtoRules` in Vanda.Grammar.XRS.LCFRS.Binarize

It is easy to see that we can always transform a NT (represented as an `Int`) into a `ProtoNT` and thus a `Rule` into a `ProtoRule`.

5.3.2 NT fusion

The NT fusion works with a function of a similar type signature to the one we defined in chapter 4 that also extracts maximal substrings in the composition function to create a new composition function for the new fusion rule and rearrange holes in the remaining composition function to obtain the composition function of the remainder rule.

```
fuseInRule
  :: (ProtoRule, Double) -- rule
  -> (Int, Int) -- indices of the NTs that are to be fused
  -> ((ProtoRule, Double), (ProtoRule, Double)) -- fusion / remainder rule
```

Listing 5.4: Type signature of the fusion function in Vanda.Grammar.XRS.LCFRS.Binarize

5.3.3 Binarizing a rule

For binarizing a rule we have introduced two strategies: a naive binarization and an optimal one. Both share the same type signature, requiring a dictionary to look up the fanouts of NTs and the `Rule` that is to be binarized and returning a list of new `ProtoRules`.

[3] the normal intified NTs do not carry their fanout individually, but can be looked up just like their string origin, but these unintified `ProtoNTs` need to carry it themselves

```
binarizeNaively, binarizeByAdjacency
  :: Array Int Int
  -> (Rule, Double)
  -> [(ProtoRule, Double)]
```

Listing 5.5: Type signatures of the functions that compute the naive and optimal binarizations of a rule in Vanda.Grammar.XRS.LCFRS.Binarize

While `binarizeNaively` can easily be implemented without consistent indices and binarization blueprints, `binarizeByAdjacency` (which is our optimal binarization) makes use of these concepts.

It is interesting to note that the two steps of constructing a binarization blueprint (using its simulation) and computing the actual binarization with the blueprint are more intertwined in `binarizeByAdjacency`: the constructed blueprints do not just contain NTs, they also contain a function `binarizer` that can compute the actual binarization the blueprint represents. This means that constructing the trees involves a function composition of the children's `binarizers` and obtaining a fully computed binarization from a binarization blueprint only requires applying the root node's `binarizer` to the original rule.

```
data NTTree = NTTreeLeaf Int -- an old NT: no action required here
            | NTTreeInner
              (   [(ProtoIndexedRule, Double)] -- binarizer
              -> ([(ProtoIndexedRule, Double)], Int) )
              (Tree Int) -- containing all NTs like a bin. blueprint
type CandidateEndpoints = (NTTree, Endpoints)
```

Listing 5.6: Binarization blueprints in `binarizeByAdjacency`

5.3.4 Binarizing a PLCFRS

Since we already represent a PLCFRS as a set of rules with probabilities we can just compute a complete binarization for each rule, properly reassigning probabilities to the resulting rules. We will never need to update the set of initial NTs, but we will intify the new `ProtoNTs`, so the intification dictionaries will change with the list of rules. This leads us to the definition for the PLCFRS-binarizing function `binarizeUsing` that given a rule binarizer (either `binarizeNaively` or `binarizeByAdjacency`) can transform a PLCFRS into a binarized one:

```
binarizeUsing
  :: (Array Int Int -> (Rule, Double) -> [(ProtoRule, Double)])
  -> PLCFRS
  -> PLCFRS
```

Listing 5.7: PLCFRS binarization in Vanda.Grammar.XRS.LCFRS.Binarize

6 Evaluation

We extract a PLCFRS from the German TIGER[1] and NEGRA[2] treebanks.

6.1 Extraction and naive binarization

We binarize this PLCFRS using the naive binarization described above.

From the diagrams in Fig. 6.1 we can see the effect the NT deduplication efforts described in the implementation chapter have on the number of new fusion rules. If we do not filter out duplicates, we should theoretically end up with 118883 fusion rules in TIGER and 61363 fusion rules in NEGRA. If we check for duplicates, however, we end up with only 18457 (TIGER) and 11454 (NEGRA) rules, so only 15% and 19% of the rules theoretically expected are really generated.

Fig. 6.2 shows the fanouts before and after naive binarization. We can see the fanouts increase for all rules with a rank greater than 1, but it is interesting to note that the highest occurring fanout (24 in TIGER, 40 in NEGRA) stays the same, even the number of rules with this fanout (1 in TIGER, 2 in NEGRA) does not change.

6.2 Optimal binarization

Here the computational complexity of the optimal binarization algorithm described above makes the evaluation difficult, binarizing all rules takes a very long time. Therefore we only binarize rules of a rank below 7 from our extracted PLCFRS (these still make up roughly 98% of both TIGER and NEGRA). Of course, the hypothetical PLCFRS that would contain just this subset of rules does not generate the same language at all, but we assume that this subset is representative enough for the original full rule set. In Fig. 6.3 we plot the fanouts of both naive and optimal binarization next to the original distribution.

We can visualize the difference between the naive and the optimal binarization better by plotting the ratio between the binarized fanouts and the original fanouts for both binarization strategies, Fig. 6.4 shows the result.

[1] http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger.html
[2] http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/negra-corpus.html

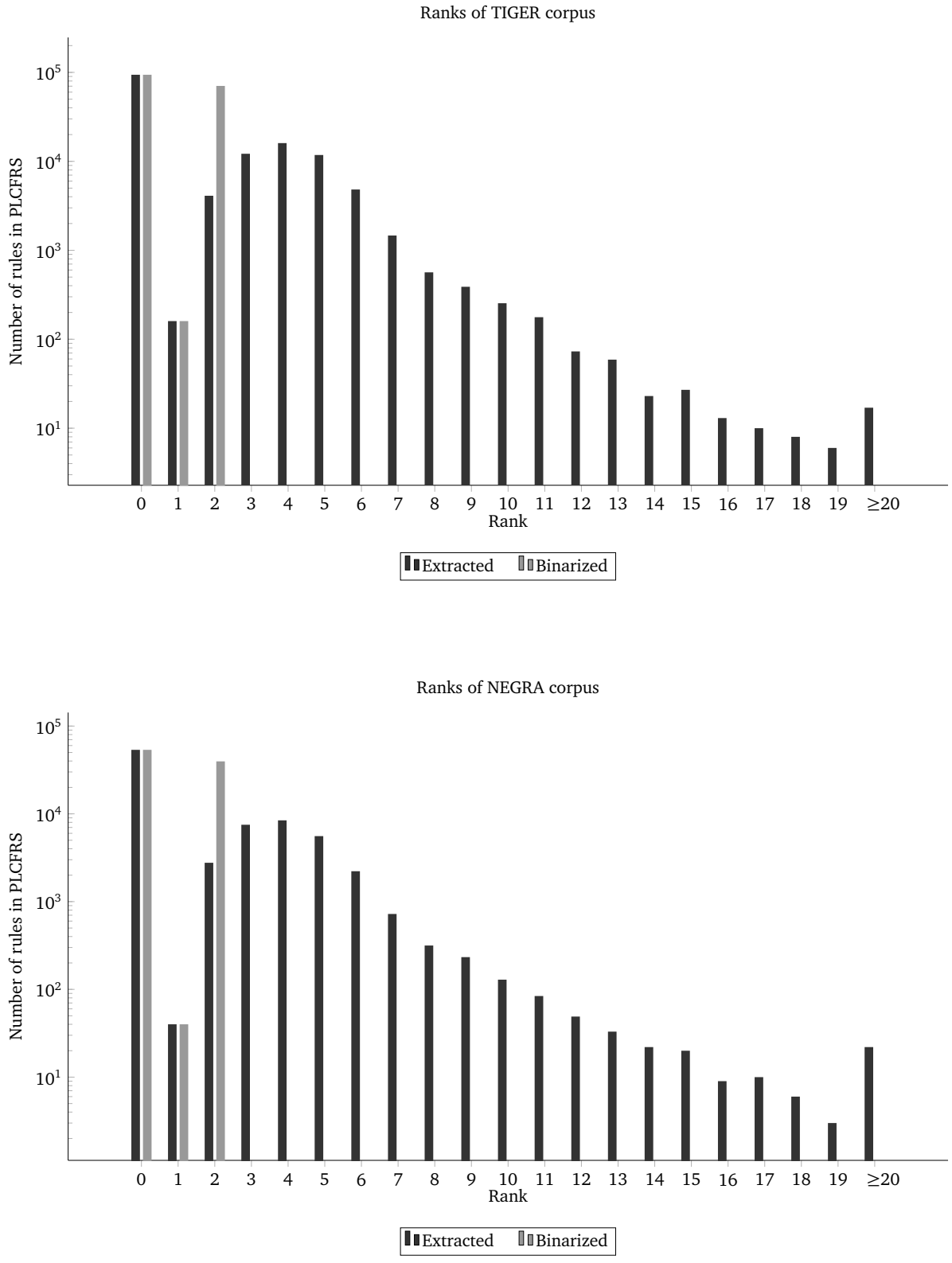


Figure 6.1: Ranks in the corpora before and after naive binarization

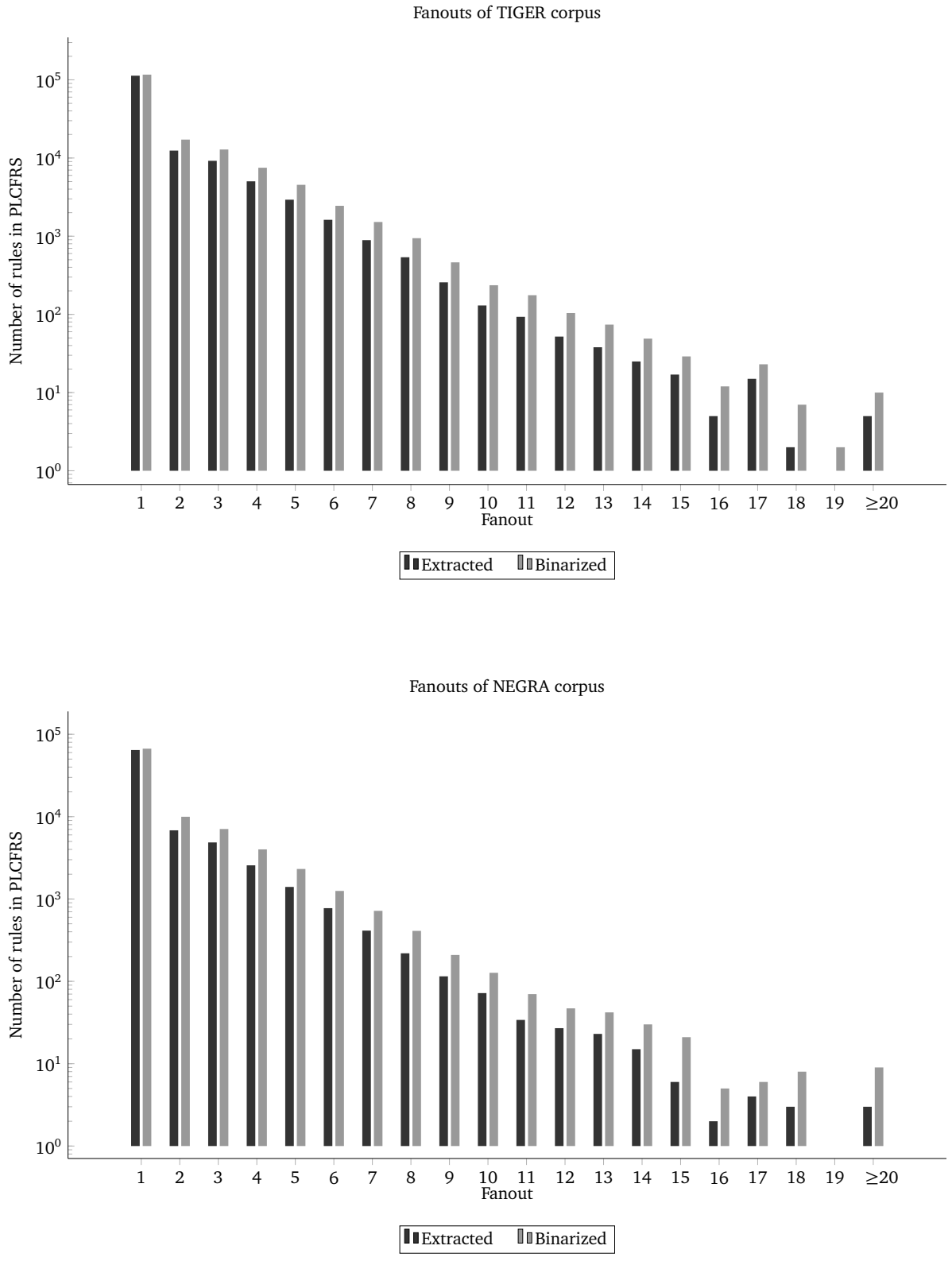
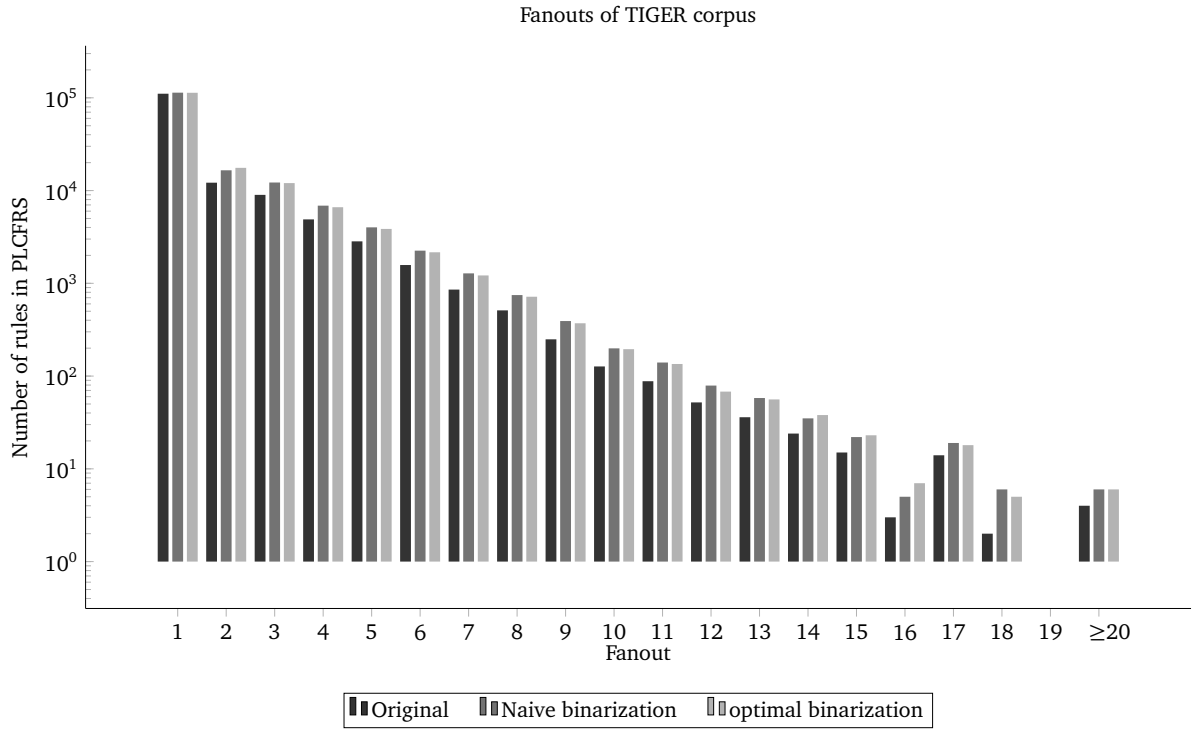


Figure 6.2: Fanouts in the corpora before and after naive binarization



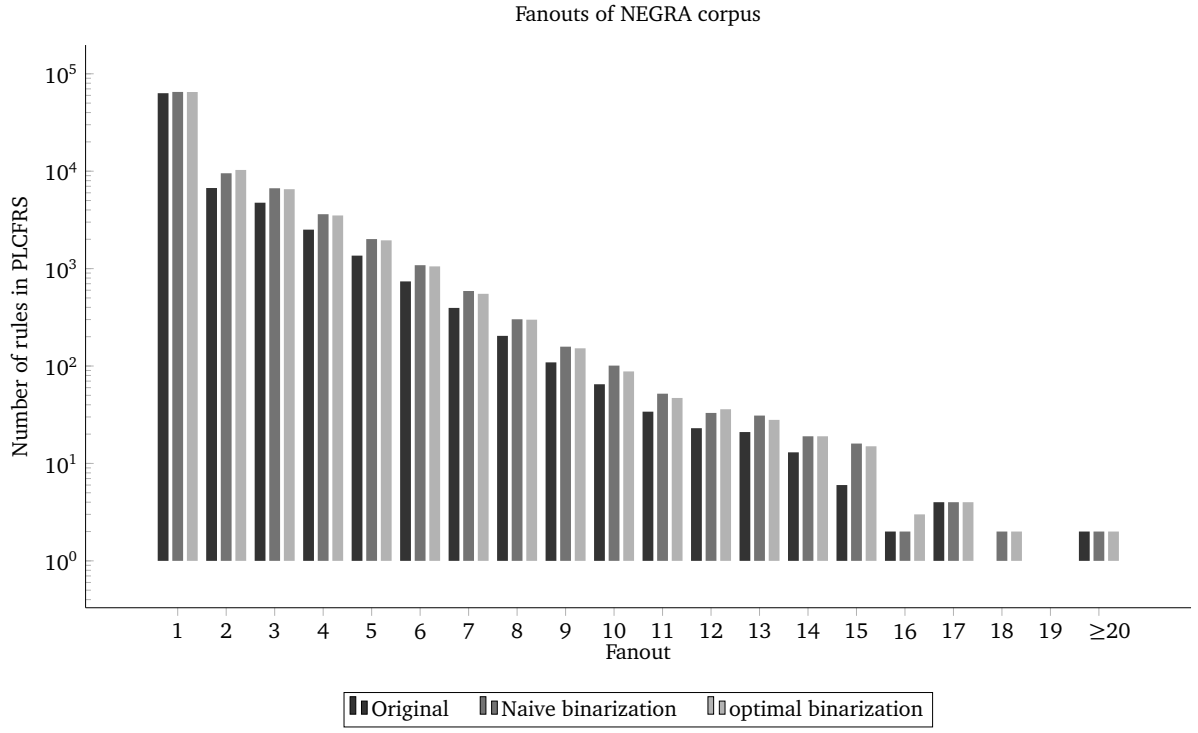
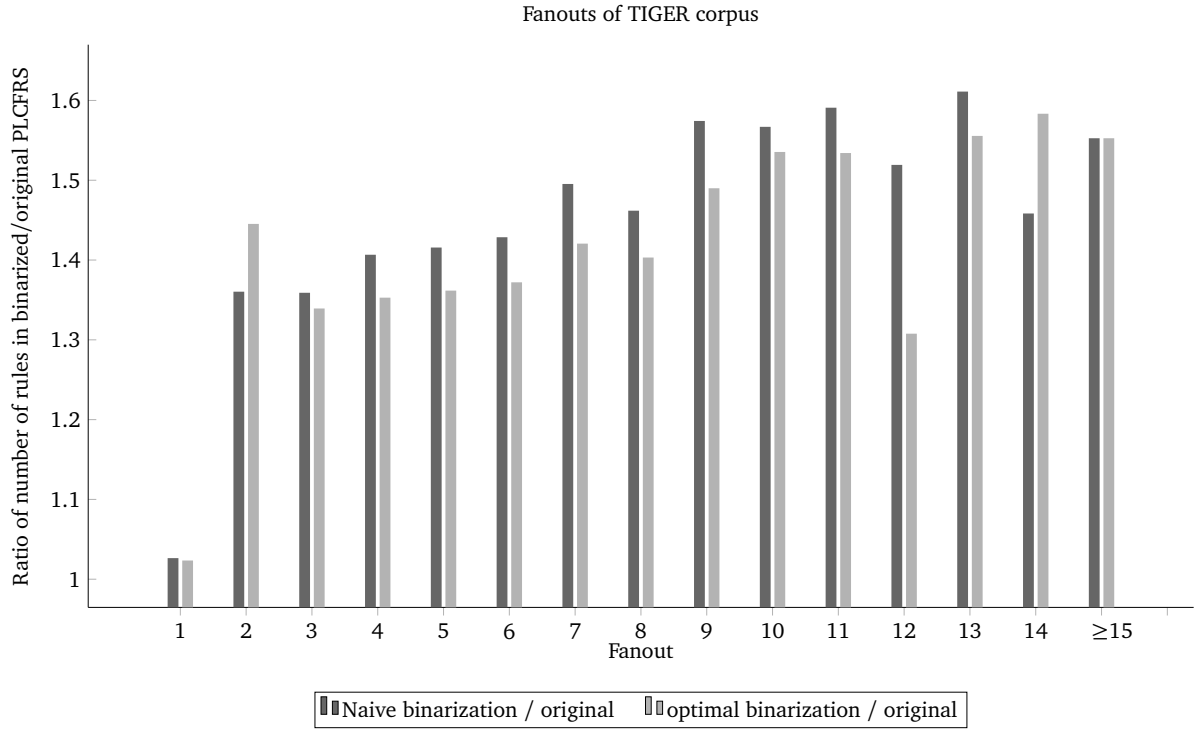


Figure 6.3: Fanouts in the filtered corpora before and after both naive and optimal binarization



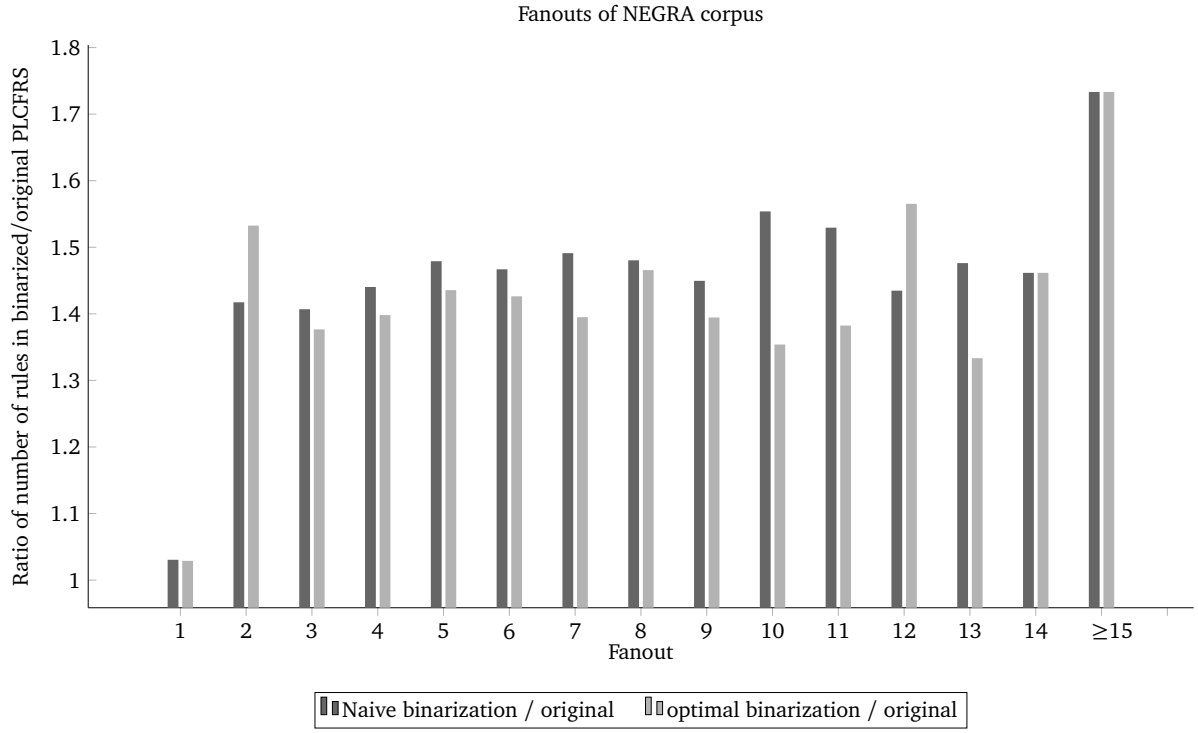


Figure 6.4: Ratio of fanouts in the naively and optimally binarized filtered corpora to the fanouts in the original filtered corpora

7 Conclusion

We have introduced probabilistic LCFRS over string tuples as a formalism that can be used for natural language processing and have shown how they can be extracted from an annotated corpus of natural language.

After defining the primitive fusing operation that our rule-by-rule binarizations rely on, we have constructed a framework of blueprints to be able to plan a complete binarization of rules and whole PLCFRSs. This framework allowed us to express both a naive approach that makes no guarantees about the resulting rules and another approach where we carefully choose the fusions such that the generated rules are optimal with regard to the maximal fanout. The evaluation of our implementation has shown that the extraction and the naive binarization are easily and quickly computable, but computing the optimal binarizations has a very high complexity and thus is not feasible for rules with a very high rank.

Future efforts might be directed to using this framework to find faster algorithms for finding optimal binarizations or acceptable approximations.

Bibliography

[Eva11] Kilian Evang. “Parsing discontinuous constituents in English”. MA thesis. Universität Tübingen, Jan. 5, 2011.

[Góm+09] Carlos Gómez-Rodríguez et al. “Optimal Reduction of Rule Length in Linear Context-free Rewriting Systems”. In: *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. NAACL ’09. Boulder, Colorado: Association for Computational Linguistics, 2009, pp. 539–547. ISBN: 978-1-932432-41-1. URL: http://dl.acm.org/citation.cfm?id=1620754.1620833.

[GS09] Carlos Gómez-Rodríguez and Giorgio Satta. “An Optimal-time Binarization Algorithm for Linear Context-free Rewriting Systems with Fan-out Two”. In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2 - Volume 2*. ACL ’09. Suntec, Singapore: Association for Computational Linguistics, 2009, pp. 985–993. ISBN: 978-1-932432-46-6. URL: http://dl.acm.org/citation.cfm?id=1690219.1690284.

[KK11] Alexander Koller and Marco Kuhlmann. “A Generalized View on Parsing and Translation”. In: *Proceedings of the 12th International Conference on Parsing Technologies*. IWPT ’11. Dublin, Ireland: Association for Computational Linguistics, 2011, pp. 2–13. ISBN: 978-1-932432-04-6. URL: http://dl.acm.org/citation.cfm?id=2206329.2206331.

[KS09] Marco Kuhlmann and Giorgio Satta. “Treebank Grammar Techniques for Non-projective Dependency Parsing”. In: *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. EACL ’09. Athens, Greece: Association for Computational Linguistics, 2009, pp. 478–486. URL: http://dl.acm.org/citation.cfm?id=1609067.1609120.

[Mic01a] Jens Michaelis. “Derivational Minimalism Is Mildly Context–Sensitive”. English. In: *Logical Aspects of Computational Linguistics*. Ed. by Michael Moortgat. Vol. 2014. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 179–198. ISBN: 978-3-540-42251-8. DOI: 10.1007/3-540-45738-0_11.

[Mic01b] Jens Michaelis. “Transforming Linear Context-Free Rewriting Systems into Minimalist Grammars”. English. In: *Logical Aspects of Computational Linguistics*. Ed. by Philippe Groote, Glyn Morrill, and Christian Retoré. Vol. 2099. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 228–244. ISBN: 978-3-540-42273-0. DOI: 10.1007/3-540-48199-0_14.

[MS08] Wolfgang Maier and Anders Søgaard. “Treebanks and mild context-sensitivity”. In: *Proceedings of Formal Grammar*. 2008, p. 61. URL: http://web.stanford.edu/group/cslipublications/cslipublications/FG/2008/maier.pdf.

[Sek+91] Hiroyuki Seki et al. “On multiple context-free grammars”. In: *Theoretical Computer Science* 88.2 (1991), pp. 191–229. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90374-B.

[Vij87] Krishnamurti Vijay-Shanker. “A study of tree adjoining grammars”. PhD thesis. University of Pennsylvania, 1987.

[VWJ86] Krishnamurti Vijay-Shanker, David Jeremy Weir, and Aravind K Joshi. “Tree adjoining and head wrapping”. In: *Proceedings of the 11th coference on Computational linguistics*. Association for Computational Linguistics. 1986, pp. 202–207. DOI: 10.3115/991365.991425.

[VWJ87] Krishnamurti Vijay-Shanker, David Jeremy Weir, and Aravind K. Joshi. “Characterizing Structural Descriptions Produced by Various Grammatical Formalisms”. In: *Proceedings of the 25th Annual Meeting on Association for Computational Linguistics*. ACL ’87. Stanford, California: Association for Computational Linguistics, 1987, pp. 104–111. DOI: 10.3115/981175.981190.

[WJ88] David Jeremy Weir and Arvind K Joshi. “Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems”. In: *Proceedings of the 26th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 1988, pp. 278–285. DOI: 10.3115/982023.982057.