Investigations into Categorial Grammar: Symmetric Pregroup Grammar and Displacement Calculus

Gijs Wijnholds

Under the supervision of Prof. dr. Michael Moortgat

July 29, 2011

Contents

1 Acknowledgements 4

2 Introduction 5

I Symmetric Pregroup Grammar: extending pregroups with duality and interaction 7

3 Introduction 8

4 The Lambek-Grishin Calculus 9

5 Pregroup$^+$ 10

6 Distributivity 12

7 Adding the $\oplus$ operator to FVect 14

8 Conclusion 16

II Displacement Calculus and Generative Capacity 17

9 Introduction 18
9.1 Formal Language Theory Beyond the Context-free Boundary 18

10 Exploring the domain of Mild Context-Sensitivity 19
10.1 TAGs and the introduction of Mild Context-Sensitivity 19
10.2 The Categorial Approach 19
10.3 The Rewriting Approach 20
10.4 The Convergence of Categorial and Generative Grammar . . . 22

11 Displacement Calculus 24
11.1 Introduction 24
11.2 Displacement Calculus 24
11.3 The generative capacity of first-order Displacement Calculus . 27
11.3.1 The Copy and Double Copy Languages 27
11.3.2 Counting and Crossing Dependencies 31

12 Well-nested simple Range Concatenation Grammar 35
12.1 Well-nested simple Range Concatenation Grammar 35
12.2 A lexicalized normal form for (simple) Range Concatenation Grammar 38

13 The equivalence of first-order Displacement Grammar and well-nested simple Range Concatenation Grammar 50

14 Conclusion and further directions 62

A Copy Language and Crossing Dependencies 67

1 Acknowledgements

I want to thank Michael Moortgat for introducing me to Categorial Grammar and other grammar frameworks, and for helping me work on ideas throughout the last six months. Furthermore, I would like to thank Rosalie Iemhoff for teaching me the basics of computational complexity.

2 Introduction

When we speak about computational linguistics in the context of the field of Artificial Intelligence, we can divide the research field of computational linguistics into three main subdivisions:

- **Mathematical Linguistics** is concerned with the mathematical foundations of natural language: in what way can we ascribe mathematical structure to language? Given a language formalism, what are the structures that it describes? And what is it's complexity with respect to time and space?
- **Cognitive Modeling** tries to give approximations of the way natural language processing occurs in the human mind.
- **Engineering** focuses on the development of technologies that facilitate efficient parsing of natural language.

This thesis is mainly concerned with mathematical linguistics: we focus on several grammatical frameworks and explore their *generative capacity*, i.e. what (formal) languages are recognized and what structures are assigned to sentences? Even within mathematical linguistics, there are numerous approaches to formal grammar, and we will restrict ourselves to *Generative Grammar* and *Categorial Grammar*. The approach of Generative Grammar is characterized by it's predictive behavior: it consists of a set of rules that determine what structures are grammatical and what structures are not. Perhaps the best known example is *Context Free Grammar*. Categorial Grammar on the other hand, is based on a type system with which we assign types to words, together with a logic acting on types with which one can derive grammatical sentences through a proof system, hence sometimes the expression *type-logical grammar* is used. Another approach we should mention is *Tree Adjoining Grammar (TAG)* because since its introduction, it has been well-studied and there are several interesting connections between *TAG* and Generative as well as Categorial Grammar.

As for the contents of this thesis, in the background of Categorial Grammar and more specifically of classical Lambek Calculus and the Lambek-Grishin Calculus, I have done several investigations into the research field of Categorial Grammar. Most of the work originates from questions about generative capacity, in particular we look at formalisms that go beyond the expressive power of Context Free Grammar. Because I have investigated two different

approaches to formal language, I've decided to split this thesis into two independent parts. Although both parts have the same approach, there is a difference in perspective between the two:

- The first (and shortest) part reflects on Pregroup Grammar and a symmetric extension in the light of a distributional formalism introduced by Coecke et al. in [7]. It is inspired by Category Theory and abstract algebra, and shows a gentle combination of a compositional and a distributional perspective on computational linguistics, united through the theory of Cartesian Closed Categories and Weakly Distributive Categories.

- The second part is an investigation into Displacement Calculus [25] and it's generative capacity, and takes the perspective of the Categorial Grammar framework versus the Generative Grammar framework. We try to argue a convergence between the two frameworks through an equivalence between Displacment Grammar and Multiple Context-Free Grammar.

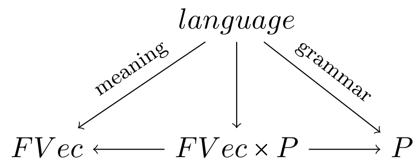
Part I

Symmetric Pregroup Grammar: extending pregroups with duality and interaction

In 2010, Coecke, Sadrzadeh and Clark [7] introduced a mathematical framework that integrates a compositional and a distributional model of meaning. They do so by pairing pregroup grammars (which are closely related to the Lambek Calculus) with vector spaces, which are both instantiations of *compact closed categories*. Linguistic analysis, however, reveals that the pregroup category is not expressive enough to account for phenomena beyond context-freeness. We will look at the Lambek-Grishin Calculus (LG), a symmetric extension of the Lambek Calculus which has as its basis two residual families of connectives and can be extended with distributivity postulates. In the light of LG, we will extend the pregroup category by mapping the types in LG to a bimonoidal extension of pregroup grammar, which we will refer to as pregroup^{+}. When we do not allow distributivity postulates, it turns out that the pregroup^{+} category naturally collapses into what effectively is a normal pregroup. However, adding distributivity postulates requires that the two operations differ, and that they can only communicate through these postulates. We will furthermore show that the resulting pregroup^{+} category instantiates a *weakly distributive category* in accordance with [6], and we will show that the $FVec$ category as well, can be extended to a *weakly distributive category*.

3 Introduction

[7] have introduced an approach that integrates a compositional grammatical model, pregroup grammar, with a distributional model of word meaning using vector spaces. Because both pregroup grammar and vector space models are instantiations of so-called *compact closed categories*, they paired the two structures to handle grammaticality in a symbolic way and word/sentence meaning in a distributional fashion, of which the latter has proven to be quite useful in practical applications. Their approach can be depicted in a simple diagram:



Informally, the idea is that tuples in the $FVec \times P$ category are assigned to words. Each morphism in the P category corresponds to a morphism in the $FVec$ category so that derivation steps in a pregroup can be carried out synchronously with computations on the vector spaces of the sequence of words. In this way, we have a compositional component that allows us to determine whether or not a sequence of words is a grammatical sentence, while the vector space model calculates the meaning from the word level to the sentence level synchronously.

Several computational grammar formalisms have been proposed in order to capture linguistic phenomena beyond context-freeness such as crossing dependencies, etc. The Lambek-Grishin Calculus, introduced by [20], seems to be able to capture these phenomena and [16] has elaborated well on the Lambek Calculus and it's algebraic counterpart, which he called pregroups. A straightforward idea, in the light of the work of Coecke et al. is to study possible extensions of the pregroup formalism that resemble the extension of classical Lambek Calculus to a version that contains a *dual residuated triple* and the so-called *Grishin interactions*, where the latter seem to permit the system to exhibit a powerful mechanism for handling crossing dependencies and other.

4 The Lambek-Grishin Calculus

We define the Lambek-Grishin Calculus as a 7-tuple $(Cat, \otimes, /, \backslash, \oplus, \oslash, \obslash)$ with the following axiomatization:

$$\frac{}{A \to A}\ Ax. \qquad \frac{A \to B \quad B \to C}{A \to C}\ Trans.$$

$$A \to C/B \overset{Res\ /}{\Longleftrightarrow} A \otimes B \to C \overset{Res\ \backslash}{\Longleftrightarrow} B \to A\backslash C \qquad B \obslash C \to A \overset{Res\ \obslash}{\Longleftrightarrow} C \to B \oplus A \overset{Res\ \oslash}{\Longleftrightarrow} C \oslash A \to B$$

Note that introducing the dual residuated triple $(\oplus, \oslash, \obslash)$ doesn't increase weak generative capacity according to [2]. The interesting part involves letting the $\otimes$ and $\oplus$ families interact through *distributivity laws*, which are given here in the form that in [21], is referred to as *(dist)*:

$$\frac{A \otimes B \to C \oplus D}{C \obslash A \to D/B}\ D_1 \qquad \frac{A \otimes B \to C \oplus D}{B \oslash D \to A\backslash C}\ D_2$$

$$\frac{A \otimes B \to C \oplus D}{C \obslash B \to A\backslash D}\ D_3 \qquad \frac{A \otimes B \to C \oplus D}{A \oslash D \to C/B}\ D_4$$

The distributivity laws can be applied to give proofs for more complicated sequents such as $np \otimes (((i/np) \oslash ((np\backslash s) \obslash i)) \otimes np) \to s$ because we are now allowed to rotate and swap the ordering of the types such that all input and output types are contracted. In this particular sequent, the outer np formulas are moved into the inner formula by means of distribution, where they can contract with the (i/np) and $(np\backslash s)$ formulae, respectively. See Figure 1 for the full proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(i/np) \to (i/np)}\ Ax.}{(i/np) \otimes np \to i}\ Res\ / \qquad \dfrac{\dfrac{}{(np\backslash s) \obslash i \to (np\backslash s) \obslash i}\ Ax.}{i \to (np\backslash s) \oplus ((np\backslash s) \obslash i)}\ Res\ \obslash}{(i/np) \otimes np \to (np\backslash s) \oplus ((np\backslash s) \obslash i)}\ Trans.}{(i/np) \oslash ((np\backslash s) \obslash i) \to (np\backslash s)/np}\ D_4}{((i/np) \oslash ((np\backslash s) \obslash i)) \otimes np \to np\backslash s}\ Res\ /}{np \otimes (((i/np) \oslash ((np\backslash s) \obslash i)) \otimes np) \to s}\ Res\ \backslash$$

Figure 1: A LG proof for $np \otimes (((i/np) \oslash ((np\backslash s) \obslash i)) \otimes np) \to s$

5 Pregroup$^+$

Now that we have defined the Lambek-Grishin Calculus, we can begin our exploration into the algebraic approach. First, let's recall the definition of a pregroup by [3]:

Definition 1. *A pregroup is an ordered monoid* $(C, \leq, \cdot, 1, \cdot^l, \cdot^r)$ *such that:*

- $p \cdot p^r \leq 1 \leq p^r \cdot p$
- $p^l \cdot p \leq 1 \leq p \cdot p^l$
- 1 *acts as a multiplicative unit*

and a^l *and* a^r *are called respectively left and right* ***adjoints*** *of* a.

Then, on a given pregroup P we can define a pregroup grammar by a set of words Σ, a distinguished goal type s and a typing function δ that assigns types (elements of the pregroup on which the grammar is defined) to words in Σ. It can be shown that grammaticality in a pregroup can be checked by only making use of the contraction rules, by checking whether a concatenation of types $a_1 \cdot a_2 ... a_n$ is less than or equal to s (when we let $1a = a1 \leq a$).

We can map (Nonassociative) Lambek Calculus into a pregroup format using the following homomorphism $[\cdot]$:

$$
\begin{array}{ll}
[A] = a & [A \otimes B] = [A] \cdot [B] \\
[A/B] = [A] \cdot [B]^l & [A \backslash B] = [A]^r \cdot [B]
\end{array}
$$

A first idea for extension can be to add a new operator and two new adjoints, and by giving a similar type translation for the $\oplus$ family of **LG**, obtain a + family within the pregroup structure. Doing so, we obtain the following structure, which I will call a pregroup$^+$:

Definition 2. *A pregroup*$^+$ *is a partially ordered bimonoid* $(C, \leq, \cdot, +, 1, 0, \cdot^l, \cdot^r, \cdot^u, \cdot^d)$ *such that:*

- $(C, \leq, \cdot, 1, \cdot^l, \cdot^r)$ *is a pregroup*
- $p + p^u \leq 0 \leq p^u + p$
- $p^d + p \leq 0 \leq p + p^d$
- 0 *acts as a comultiplicative unit*

and a^u and a^d are called respectively up and down ***adjoints*** *of a.*

On a given pregroup$^+$ P^+, we can construct a pregroup$^+$ grammar in the same way as we did for pregroups. We now can map the $\oplus$ family into pregroup$^+$ types by extending our $[\cdot]$ translation:

$$[A \oplus B] = [A] + [B] \quad [A \oslash B] = [A] + [B]^u \quad [A \otimes B] = [A]^d + [B]$$

Conceptually, we can graph a matrix of each element and it's adjoints, giving an intuition of how we look at elements and adjoints:

$$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & p^{lu} = p^{ul} & p^u & p^{ru} = p^{ur} & \cdots \\ \cdots & p^l & p & p^r & \cdots \\ \cdots & p^{ld} = p^{dl} & p^d & p^{rd} = p^{dr} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

As [3] states, in a pregroup, it is required that inverses collapse, i.e. $p^{lr} = p = p^{rl}$. For the up and down inverse we have the same requirements. The above matrix shows how we can view pregroup elements on a line, and how up and down inverses create a 2-dimensional element matrix. So the inverses of an element show what path has been traversed through this matrix, e.g. p^{lurd} denotes a step to the left, then up, then right and finally, a step down, *returning at the origin.* In this way, we may conclude that $p^{lurd} = p$.

6 Distributivity

The *(dist)* laws give rise to the following characteristic **LG** theorems:

$$\begin{array}{ll} (A \oslash B) \otimes C \to A \oslash (B \otimes C) & C \otimes (B \oslash A) \to (C \otimes B) \oslash A \\ C \otimes (A \oslash B) \to A \oslash (C \otimes B) & (B \oslash A) \otimes C \to (B \otimes C) \oslash A \end{array}$$

Under the mapping $[\cdot]$, we would end up with the following ordering postulates:

$$\begin{array}{ll} (a^d + b) \cdot c \leq a^d + (b \cdot c) & c \cdot (b + a^u) \leq (c \cdot b) + a^u \\ c \cdot (a^d + b) \leq a^d + (c \cdot b) & (b + a^u) \cdot c \leq (b \cdot c) + a^u \end{array}$$

Because we have in a pregroup$^+$ that $a^{du} = a = a^{ud}$, it is easy to see that the four postulates lead to *weakly distributive laws* by the following:

$$\begin{array}{l} (a + b) \cdot c = (a^{ud} + b) \cdot c \leq a^{ud} + (b \cdot c) = a + (b \cdot c) \;\; (d_1) \\ c \cdot (b + a) = c \cdot (b + a^{du}) \leq (c \cdot b) + a^{du} = (c \cdot b) + a \;\; (d_2) \\ c \cdot (a + b) = c \cdot (a^{ud} + b) \leq a^{ud} + (c \cdot b) = a + (c \cdot b) \;\; (d_3) \\ (b + a) \cdot c = (b + a^{du}) \cdot c \leq (b \cdot c) + a^{du} = (b \cdot c) + a \;\; (d_4) \end{array}$$

It may be clear already that if we use the seemingly more 'restricted' distributivity postulates, we have to apply the same trick used to derive the weakly distributive laws in our derivations. Instead of doing this, it is more economic to directly introduce the derived postulates.
Now consider the example from section 2:
$np \otimes (((i/np) \oslash ((np\backslash s) \oslash i)) \otimes np \to s$.
This sequent is derivable in **LG** with distributivity. Translating the types, we obtain:

$[np] = np$
$[s] = s$
$[(((i/np) \oslash ((np\backslash s) \oslash i))]$
$= [i/np] + [((np\backslash s) \oslash i))]^u$
$= (i \cdot np^l) + ([(np\backslash s)]^d + [i])^u$
$= (i \cdot np^l) + ((np^r \cdot s)^d + i)^u$
$= (i \cdot np^l) + (i^u + (np^r \cdot s)^{du})$ $((a \cdot b)^l = b^l \cdot a^l)$
$= (i \cdot np^l) + (i^u + (np^r \cdot s))$ (up and down inverses collapse)

Making use of the weakly distributivity postulates, we obtain the following chain of inequalities:

$$np \cdot ((i \cdot np^l) + (i^u + (np^r \cdot s))) \cdot np \leq$$
$$np \cdot ((i \cdot np^l \cdot np) + i^u + (np^r \cdot s)) \leq (d_4)$$
$$np \cdot (i + i^u + (np^r \cdot s)) \leq \text{(left contraction)}$$
$$np \cdot np^r \cdot s \leq \text{(up contraction)}$$
$$s \text{ (right contraction)}$$

Little intuition is needed to see that distributivity requires two operators to act on types, thus it is, though also conceptually tempting, a necessity to introduce a comultiplication operator. However, in the case we allow the comultiplication and the multiplication operators to collapse but keep the up and down inverses, we can equally well describe the non-distributive case. Furthermore, it can easily be shown that the resulting (non-distributive) monoidal structure falls within context-freeness as well as is the case with pregroup grammars and the Lambek-Grishin Calculus without distributivity (pregroup grammars can be shown to be context-free by giving a nondeterministic pushdown-automaton, a similar automaton can recognize pregroup$^+$ grammars). This might lead one to think there is a strong connection between the Lambek-Grishin Calculus and pregroup$^+$ grammar. Furthermore, Moot in [24] shows that **LG** generates some interesting languages that LTAG generates as well. In appendix A, we give the **LG** as well as the pregroup$^+$ grammars for respectively the copy language and crossing dependencies. With some exercise, we can see that these pregroup$^+$ grammars generate exactly the copy language and crossing dependencies respectively.

7 Adding the $\oplus$ operator to FVect

Since we are interested in the **LG** with distributivity (beyond context-freeness) we will need our pregroup$^+$ structure to be a bimonoid. If we want to interpret this in the compositional distributional framework, we want to add a $\oplus$ operator into the **FVec** category. There are numerous possibilities here, but a problem arises: in the model of [7], tensored vectors are entangled ones, with their weights coming from a corpus-determined weighting matrix. If we would distribute the $\otimes$ and $\oplus$ operators, we would have to look up our weights every time we apply a distributivity law. So when we have a vector living in the vector space $C \otimes (A \oplus B)$, thus having weights combining C with $A \oplus B$, we would need to obtain a new weight for $C \otimes B$ in order to distribute to a vector living in the vector space $A \oplus (C \otimes B)$. One way to avoid this problem is simply stating that $\oplus = \otimes$. In this case the FVect category might be too degenerate for linguistic purposes. Another option we have is to treat $\oplus$ as *disjoint sum*, and thus composing two objects while retaining their individual information. For disjoint sum, we have the following: If $\{\overrightarrow{a_1}, .., \overrightarrow{a_n}\}$ is a base of A and $\{\overrightarrow{b_1}, .., \overrightarrow{b_n}\}$ is a base of B, then $\{(\overrightarrow{a_1}, \overrightarrow{0}), .., (\overrightarrow{a_n}, \overrightarrow{0}), (\overrightarrow{0}, \overrightarrow{b_1}), .., (\overrightarrow{0}, \overrightarrow{b_n})\}$ is a base of $A \oplus B$.
Then, for vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ respectively in A and B, we have:
$\overrightarrow{a} = \sum_i C_i \overrightarrow{a_i}$ with $\overrightarrow{a_i}$ a base vector
$\overrightarrow{b} = \sum_j C_j \overrightarrow{b_j}$ with $\overrightarrow{b_j}$ a base vector
$\overrightarrow{a} \oplus \overrightarrow{b} = \sum_i C_i(\overrightarrow{a_i}, \overrightarrow{0}) + \sum_j C_j(\overrightarrow{0}, \overrightarrow{b_j})$

To overcome the problem of the tensored vectors, we want to make sure that we can apply all distributivity postulates first, then look up the appropriate weights, and then apply contractions. For this, we need to establish the following theorem:

Theorem 1. *For a Pregroup$^+$ grammar with distributivity, each chain of inequalities $\alpha_1 \leq \alpha_2 ... \leq \alpha_n$ can be rewritten into an equivalent form $\beta_1 \leq \beta_2 ... \leq \beta_m$ for which the following holds:*

- $\alpha_1 = \beta_1$
- $\alpha_n = \beta_m$
- *For some $i \geq 0 : \beta_1 \leq ... \leq \beta_i$ are applications of distributivity laws and $\beta_i \leq ... \leq \beta_m$ are applications of contraction rules*

PROOF Let $\alpha_1 \leq \alpha_2 ... \leq \alpha_n$ be a chain of inequalities. We call α_1 the *initial type sequence* and α_n the *final type sequence*. The idea is to shift

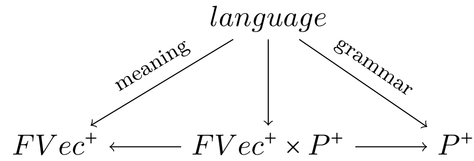
all applications of distributivity postulates to the left in order to obtain a sequence of distributions and a sequence of contractions. Set $\beta_1 = \alpha_1$ and $\beta_n = \alpha_m$. In the trivial cases that $\alpha_1 \leq \alpha_2 ... \leq \alpha_n$ consists of distributions only or contractions only, we let $\beta_1 \leq \beta_2 ... \leq \beta_m = \alpha_1 \leq \alpha_2 ... \leq \alpha_n$.
Now consider some $\alpha_j = ...(c \cdot (a + b))...$ and $\alpha_{j+1} = ...a + (c \cdot b)...$ and the case where $\alpha_1 \leq \alpha_2 ... \leq \alpha_j$ contains some contractions. We then have the following cases:

- all contractions only occur in the context. Then, bring the distribution $\alpha_j \leq \alpha_{j+1}$ to the front.
- $\alpha_{j-1} = (c \cdot p^l \cdot p) \cdot (a + b)$. Then apply multiple distributions and a contraction at the end to get $(c \cdot p^l \cdot p) \cdot (a + b) \leq a + ((c \cdot p^l \cdot p) \cdot b) \leq a + (c \cdot b)$
- $\alpha_{j-1} = c \cdot ((p^l \cdot p) + (a + b))$. Then apply multiple distributions and a contraction at the end to get $c \cdot ((p^l \cdot p) + (a + b)) \leq (p^l \cdot p) + (c \cdot (a + b)) \leq (p^l \cdot p) + (a + (c \cdot b)) \leq a + (c \cdot b)$
- The other cases follow by extrapolation

With this information, we can handle distributivity at the vector (meaning) side by first distributing the vectors, then assigning weights to the vectors, and handle all contractions in the way of the model of [7].

8 Conclusion

We have elaborated on a symmetric extension of the pregroup formalism, intended as a suitable category to be used in extension to the model developed by [7]. The model that we have described so far can, similar to the **FVec** × **P** model, be depicted in a simple diagram:



The pregroup$^+$ category with (d_1) and (d_2) instantiates the *non-planar weakly distributive category* as described by [6]. It is important to note that Cockett and Seely distinguish the case where the two tensors are symmetric, i.e. invariant under permutation of its arguments, in which case (d_3) and (d_4) are induced, but that in the pregroup$^+$ category with the four distributivity postulates, we do not have symmetric tensors, so we cannot classify it as a *symmetric weakly distributive category.* As the upper bound on the weak generative capacity of **LG** is still open, further study may include complexity issues (upper bound for pregroup$^+$). As for parsing possibilities, the theorem of the last section shows that we can split parsing into two parts: applying all possible distributions, and running each result on a nondeterministic pushdown automaton to handle the resulting contractions.

Part II
Displacement Calculus and Generative Capacity

Morrill, Valentìn and Fadda ([25], [8]) introduced the theory of Displacement Calculus, an extension of the Lambek Calculus that typically deals with discontinuities via insertion and wrapping points. We will define a restricted version, so-called *first-order* Displacement Calculus and discuss some of it's possible definitions with respect to how insertion/wrapping points are treated. We will proceed to showing how general first-order Displacement Calculus is capable of describing several mild context-sensitive languages, viz. counting and crossing dependencies. Finally, we will show that general first-order Displacement Calculus is equivalent to well-nested simple Range Concatenation Grammar.

9 Introduction

9.1 Formal Language Theory Beyond the Context-free Boundary

Throughout the last decades, research in computational linguistics has increasingly focused on formalisms that, in terms of generative capacity, go beyond the context-free boundary. Although Context-free Grammar has been well-studied, there are several claims *against the context-freeness* of natural language ([31], [10]). An important phenomenon that can not be directly expressed by Context Free Grammar are crossing dependencies in Dutch and Swiss German. For this reason, there has been a lot of development in the field of computational linguistics to find formalisms that adequately handle these and several other phenomena (viz. counting dependencies) that occur in natural language. In this thesis, we focus on the approach of *Categorial Grammar (CG)* on one side and the approach of *Multiple Context Free Grammar (MCFG)* on the other side. MCFG is a tuple-based extension of Context Free Grammar which has shown to be equivalent to *simple Range Concatenation Grammar (sRCG)*. We will also briefly discuss Tree Adjoining Grammar in the introduction because there is a correspondence between TAG and so-called *well-nested MCFG*.
The outline of this part is as follows: Section 1 comprises a (short) overview of the relevant formalisms and discusses our motivation for using Displacement Calculus, in section 2 we introduce Displacement Calculus and discuss it's *generative capacity*, in section 3 we introduce simple Range Concatenation Grammar as a convenient way to express MCFG. Furthermore, we will discuss the difference between well-nested MCFG and non well-nested MCFG in terms of generative capacity. We will also prove a new normal form for well-nested sRCG. In section 4 we prove the equivalence of first-order Displacement Calculus and well-nested sRCG and give some consequences of our main theorem. Section 5 then returns to the claim made in the first section by providing directions for further research.

10 Exploring the domain of Mild Context-Sensitivity

10.1 TAGs and the introduction of Mild Context-Sensitivity

The notion of Mild Context-Sensitivity was informally introduced by Joshi in [12]. By his definition, a Mildly Context Sensitive Language (MCSL) has the following three properties:

- Limited Crossing Dependencies
- The Constant Growth Property, which states that there is a constant c such that for every $w \in L$, there is a $w' \in L$ with $|w| < |w'| \leq |w| + c$. Informally, this says that the difference between two strings that are closest in length is at most a fixed bound. It ensures that counting and crossing dependencies are allowed, but not languages such as $\{a^{2^n} | n \geq 0\}$.
- Polynomial Parsability

Limited Crossing Dependencies sound somewhat vague, but the idea is that structures occur in language in which dependencies between words cross each other, as in the language $\{a^n b^m c^n d^m | n, m \geq 0\}$. We assume limited to mean having a maximum number of crossing dependencies (one might argue that sentences with 18 crossing dependencies are not really comprehensible by a human).
Joshi et al. also argue in [11] that the so-called MIX_3 language may not be a MCSL, a statement which we will turn to in the next few sections.
Joshi then introduced the formalism known as *Tree Adjoining Grammar (TAG)*, in which tree structures are combined by means of two operations called *substitution* and *adjunction*. Such a grammar consists of tree structures called *elementary trees* and *auxiliary trees*, and the language that is associated with each possible tree structure that can be built from the grammar. We will see in the next subsections why TAG is related to the subject of this part.

10.2 The Categorial Approach

The cornerstone of Categorial Grammar is the Lambek Calculus, originating from the work of Lambek [15] which in 1965 was conjectured to be equivalent in generative capacity with Context Free Grammar [5]. It was not until 1993 that this conjecture was proved by Pentus [26]. In the Categorial

approach as in the case of Context Free Grammar, there were several proposals for extensions of basic Lambek Calculus. These extensions include the multimodal Lambek Calculus ($NL_{\diamond}$) advocated by Moortgat [19], and the Lambek-Grishin Calculus (LG), which was introduced by [20]. A restricted version of the multimodal system, $NL_{\diamond^-}$, in which only non-expanding structural rules are used, was shown to fall in the Context-Sensitive Languages [23], but the exact class of languages that LG describes, is still unknown. Although [22] had tried to prove equivalence of LG and TAG, the result of [17] falsifies the result of the former by showing that LG is capable of generating the intersection of a context-free language and the permutation closure of a context-free language. These languages fall beyond the generative capacity of Tree Adjoining Grammar.

In this part, we will focus on another, more recent proposal by [25] to extend Lambek Calculus to handle discontinuous dependencies in natural language. This work led to a Calculus of Displacement (D) in which strings can be separated in several strings with detached parts. Because of the way strings are separated, an equivalence between what I will call *general first-order Displacement Calculus* and *well-nested simple Range Concatenation Grammar* can be easily constructed. The motivation for this is twofold:

1. We want to compare Categorial Grammar to Generative Grammar.
2. We want to further investigate the role of well-nested MCFG/sRCG, for reasons we will elaborate on in the next few subsections.

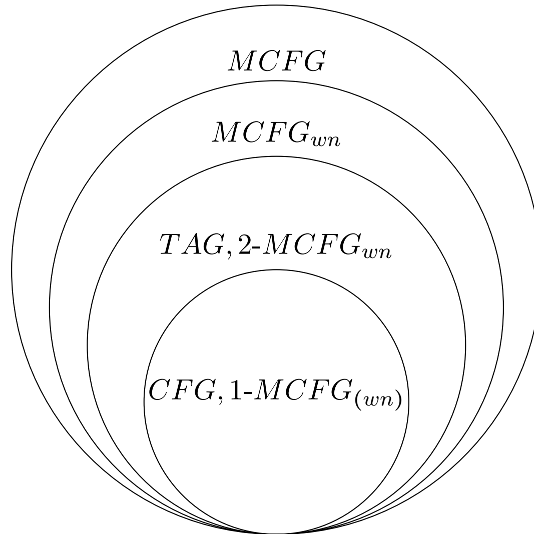
10.3 The Rewriting Approach

At the side of Context Free Grammar we find Multiple Context Free Grammar, which was developed by [30]. Multiple Context Free Grammars manipulate tuples of strings, and exhibit a control property in that the generative capacity grows according to a control parameter. Linear MCFG was shown to be equivalent to simple Range Concatenation Grammar, Linear Context Free Rewriting Systems and set-local MCTAG, a generalizing extension of Tree Adjoining Grammar. The equivalence with simple Range Concatenation Grammar especially is important, because sRCG has been provided with several normal forms, which make the parsing of such grammars very convenient.
Multiple Context Free Grammar were claimed to may be more appropriate to describe the Mild Context-Sensitive Languages. For example, in [10]

who introduced a formalism known as *Literal Movement Grammar*, it is argued that "The class of mildly context-sensitive languages seems to be most adequately approached by [MCFGs]." Because $MCFG$ conforms to two of the three restrictions of Mild Context-Sensitivity. However, linear MCFG is NP-complete, and MCFG in general even is EXPTIME-complete, thus they may not be an appropriate approach to Mild Context-Sensitivity. In this light, [14] investigated a subset of MCFG, so-called *well-nested MCFG* (MCFG$_{wn}$), which do have a polynomial recognition procedure [REF]. Furthermore, by the very recent result of [28], $MIX_3 = \{w \in \{a,b,c\}^* || w_a| = |w_b| = |w_c| \geq 0\}$ language, introduced by [1] is contained in 2-MCFG. Comparing this in retrospect to the claim made in [11]:

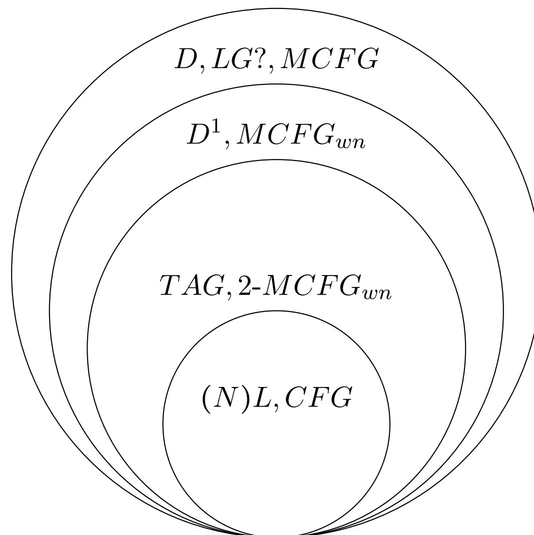
> "MCSGs capture only certain kinds of dependencies, such as nested dependencies and certain limited kinds of crossing dependencies (for example, in subordinate clause constructions in Dutch or some variations of them, but perhaps not in the so-called MIX language)",

we find ourselves in the position to claim that well-nested MCFG may be the most appropriate formalism that describes the class of Mildly Context Sensitive Languages (MCSL). Although it is not known as we speak (that is, a proof has not been provided yet), it is assumed that the MIX_3 language is not in the class of well-nested MCFG, and thus it is sensible to investigate the generative power of *first-order Displacement Calculus* (a variant of Displacement Calculus that puts a restriction on the complexity of types in terms of an order, which we will define in chapter 11), instead of it's higher-order variant, which has been shown to generate the MIX_3 language [8]. Furthermore, it is known that LG as well can generate the MIX_3 language by [17]. Tree Adjoining Grammar coincides with 2-$MCFG_{wn}$ by an indirect encoding of TAG into second-order Abstract Categorial Grammar with a co-order of 3 ˜citedegroote02, and the equivalence of $ACG_{2,3}$ and 2-$MCFG_{wn}$ ([18], [27]). In this sense, we can argue that TAG is the minimal MCSG formalism, whereas $MCFG_{wn}$ is a general notion of the class of MCSL. For some greater clarity, we have included a diagram that depicts a complexity hierarchy of the discussed formalisms.



10.4 The Convergence of Categorial and Generative Grammar

Given the knowledge available about Tree Adjoining Grammar, several Categorial Grammar formalisms and the establishment of Multiple Context Free Grammar and its well-nested variant, we depict the relation between Categorial Grammar versus Rewriting Systems in terms of generative capacity as follows:



Note that for LG, an equivalence has not yet been established. In the last section, I will argue D and $MCFG$ might be equivalent, and that both D and $MCFG$ may be good candidates for a comparison with LG.

11 Displacement Calculus

11.1 Introduction

Glyn Morrill, Oriol Valentìn, Mario Fadda developed the Displacement Calculus as a generalization of the Lambek Calculus. Instead of using structural rules, as is done in Multi-Modal Categorial Grammar and *LG*, Displacement Calculus is based on so-called *Displacement Algebra*, in which multiple wrap operations are defined in order to handle discontinuities in language.

11.2 Displacement Calculus

The key point of Displacement Calculus is the use of a special element, called the "separator". The separator denotes an infixation or extraction point in between words, thus making it possible to intercalate words with appropriate typing into the place of the separator. The underlying algebra therefore needs to contain a special prime element, and is defined as follows:

Definition 3. *A graded syntactic algebra is a free algebra* $(L, +, 0, 1)$ *where:*

- $(L, +, 0)$ *is a monoid*
- $1 \in L$ *is a prime, i.e. has no other factors than* 0 *and itself*

We can sort the syntactical elements of such a graded syntactic algebra by the number of separators it contains, thus obtaining *sort domains* for every i, denoted L_i. We are then ready to define a *displacement algebra*. Here, we will differ from the way the authors define a Displacement Algebra in that we define wrap operations on specific points so that later, we can have types point at specific separator elements.

Definition 4. *A Displacement Algebra is a sorted algebra* $(\{L_i\}_{i \in N}, +, \times_>, \times_<, 0, 1)$ *where:*

- $\{L_i\}_{i \in N}$ *is the partition of* L *into sort domains,*
- $(L, +, 0, 1)$ *is a graded syntactic algebra,*
- $+ : L_i \times L_j \to L_{i+j}$ *is concatenation,*
- $\times_k : L_{i+1} \times L_j \to L_{i+j}$ *with* $k \in \mathbf{N}$ *are wrap operations, i.e.* $s \times_k t$ *denotes the result of replacing the* k*th separator in* s *by* t.

As in the Lambek(-Grishin) Calculus, the Displacement Calculus acts on typed expressions, therefore, having defined the underlying algebra, we need a type system:

Definition 5. *The set of types Tp of a displacement algebra D is constructed with $(\bullet, /, \backslash, I, \odot, \downarrow_k, \uparrow_k, J)$ with the following interpretations into D:*

$$
\begin{array}{ll}
[A \bullet B] = \{s_1 + s_2 | s_1 \in [A] \& s_2 \in [B]\} & [A \odot_k B] = \{s_1 \times_k s_2 | s_1 \in [A] \& s_2 \in [B]\} \\
[A\backslash B] = \{s_2 | \forall s_1 \in [A] : s_1 + s_2 \in [B]\} & [A \downarrow_k B] = \{s_2 | \forall s_1 \in [A] : s_1 \times_k s_2 \in [B]\} \\
[B/A] = \{s_1 | \forall s_2 \in [A] : s_1 + s_2 \in [B]\} & [B \uparrow_k A] = \{s_1 | \forall s_2 \in [A] : s_1 \times_k s_2 \in [B]\} \\
[I] = \{0\} & [J] = \{1\}
\end{array}
$$

We define the sort of a type as the number of separators it contains. Furthermore we add inductive definitions of type order and type length, since we need it later on.

Definition 6. *The length function $len : Tp \to N$ is defined as follows:*

$$
\begin{array}{l}
len(I) = len(J) = len(p) = 1 \text{ where } p \text{ is a primitive type} \\
len(A \bullet B) = len(A\backslash B) = len(B/A) = \\
len(A \odot B) = len(A \downarrow_k B) = len(B \uparrow_k A) = len(A) + len(B)
\end{array}
$$

Definition 7. *The order function $ord : Tp \to N$ is defined as follows:*

$$
\begin{array}{l}
ord(I) = ord(J) = ord(p) = 0 \text{ where } p \text{ is a primitive type} \\
ord(A \bullet B) = ord(A \odot B) = max(ord(A), ord(B)) \\
ord(A\backslash B) = ord(A \downarrow_k B) = max(ord(A) + 1, B) \\
ord(B/A) = ord(B \uparrow_k A) = max(ord(B), ord(A + 1))
\end{array}
$$

We use $\alpha|_k\beta$ to denote the result of replacing the kth separator in α by β. Furthermore, the image $\overrightarrow{A}$ is defined as:

$$
\overrightarrow{A} = \begin{cases} A & \text{if } sA = 0; \\ A\underbrace{\{[\ ], ..., [\ ]\}}_{sA} & \text{if } sA > 0. \end{cases}
$$

Because at this time, we are only interested in types of order 1, we can do with the *first-order fragment* of the Displacement Calculus, which only has introduction rules for $\bullet$ and $\odot_k$ and elimination rules for $/, \backslash$ and $\uparrow_k, \downarrow_k$:

Definition 8. *The (syntactic) natural deduction system of the first-order fragment of the Displacement Calculus looks as follows:*

$$\frac{\begin{array}{c}\vdots\\ \alpha : A\end{array} \quad \begin{array}{c}\vdots\\ \gamma : A\backslash C\end{array}}{\alpha + \gamma : C}\, E\backslash \qquad \frac{\begin{array}{c}\vdots\\ \gamma : C/A\end{array} \quad \begin{array}{c}\vdots\\ \alpha : A\end{array}}{\gamma + \alpha : C}\, E/$$

$$\frac{\begin{array}{c}\vdots\\ \alpha : A\end{array} \quad \begin{array}{c}\vdots\\ \beta : B\end{array}}{\alpha + \beta : A \bullet B}\, I\bullet \qquad \frac{}{0 : I}\, Ax.I$$

$$\frac{\begin{array}{c}\vdots\\ \alpha : A\end{array} \quad \begin{array}{c}\vdots\\ \gamma : A \downarrow_k C\end{array}}{\alpha|_k\gamma : C}\, E\downarrow_k \qquad \frac{\begin{array}{c}\vdots\\ \gamma : C \uparrow_k A\end{array} \quad \begin{array}{c}\vdots\\ \alpha : A\end{array}}{\gamma|_k\alpha : C}\, E\uparrow_k$$

$$\frac{\begin{array}{c}\vdots\\ \alpha : A\end{array} \quad \begin{array}{c}\vdots\\ \beta : B\end{array}}{\alpha|_k\beta : A \odot_k B}\, I\odot_k \qquad \frac{}{1 : J}\, Ax.J$$

We denote the first-order fragment of $\mathbf{D}$ by $\mathbf{D}^1$.

Definition 9. *A* $\boldsymbol{D}$*-grammar* G *is a tuple* (Σ, δ, S) *where:*

- Σ *is a finite set of words*
- δ *is a function assigning types to words in* Σ
- S *is the distinguished start type*

We will write $\mathbf{D} \;\Rightarrow w : S$ to express that $w : S$ is a theorem of $\mathbf{D}$.
We are ready to define both the string and tree languages for Displacement Grammar.

Definition 10. *Let* $G = (\Sigma, \delta, S)$ *be a* $\boldsymbol{D}$*-grammar. We define string and tree languages:*

- *The* string language *of* G *is defined as* $L_S(G) = \{w : S | w \in \Sigma^* \text{and } \boldsymbol{D} \Rightarrow w : S\}$.

Natural deduction proofs can be viewed as trees that have axioms as leaves and the theorem as the root. Each deduction rule that is used then corresponds to edges directing from the conclusion vertex to one or more premise vertices. We use this obvious encoding to define the tree language for Displacement Grammar:

Definition 11. *Let* $G = (\Sigma, \delta, S)$ *be a* $\boldsymbol{D}$*-grammar and let* $w = w_1 ... w_n$ *be a word over* Σ^**. A derivation tree of* G *with respect to* w *is a binary tree* $D = (V, l, E, r)$ *such that:*

- *There are exactly n leaves $v_1...v_n$ with $l(v_i) = w_i : \delta(w_i)$.*
- *For every internal vertex v_0, for either it's one daughter vertex v_1 or it's two daughter vertices v_1, v_2 it holds that $l(v_0)$ is the conclusion of some deduction rule applied to respectively either v_1 or v_1, v_2.*
- $l(r) = w : S$

The tree language *of G is defined as $L_T(G) = \{D | D$ is a derivation tree of G for some $w \in \Sigma^*\}$.*

Displacement Calculus has a clear control parameter: the maximum sort allowed in any sequent. [25] distinguish only the cases of **1D** and **2D**. I will generalize this in order to show what kind of structures can be recognized at each level of displacement. Notation will be **nD** for a particular n. Furthermore we define the *order* of a **D**-grammar G as $ord(G) = max(ord(\delta(t)))$ with t ranging over Σ. In other words, the order of such a grammar equals the maximum order of the types. The *length* of a **D**-grammar G is furthermore defined as $len(G) = max(len(\delta(t)))$ with t ranging over Σ, i.e. the length of a grammar is the maximum length of it's types. Then, we can define the first-order fragment of Displacement Grammar as those grammars D such that $ord(D) = 1$.

11.3 The generative capacity of first-order Displacement Calculus

In [8], some expressivity results on the Displacement Calculus were presented. In particular, it is shown that **D** can recognize the permutation closure of any context-free language, and that for every n, there exists a **D**-grammar that recognizes $MIX_n = \{w \in \{a_1...a_n\}^* || a_1|_w = |a_2|_w = ... = |a_n|_w\}$. Also, they give grammars for the non-context free languages $\{a^n b^n c^n | n \geq 1\}$ and $\{ww | w \in \{a, b\}^+\}$. Note that these languages all are recognized in **1D**. Here, I will improve results by showing that in $\mathbf{D}^1$, the cube language, i.e. $\{www | w \in \{a, b\}^+\}$, and $\{wwww | w \in \{a, b\}^+\}$ can be recognized in **2D**, as can 4 or 5 dependencies and crossing dependencies. Generalizing, I will show that $\{w^n | w \in \{a, b\}^+\}$ can be recognized in $\frac{n}{2}\mathbf{D}^1$ if n is even, and $\frac{n+1}{2}\mathbf{D}^1$ if n is odd. Also, I will give a generalization for dependencies and crossing dependencies.

11.3.1 The Copy and Double Copy Languages

The grammar from [8] for the copy language $\{ww | w \in \{a, b\}^+\}$ is as follows:

$$
\begin{gathered}
S' = S \odot_1 I \\
a : A; J\backslash(A\backslash S); J\backslash(S \downarrow (A\backslash S)) \\
b : B; J\backslash(B\backslash S); J\backslash(S \downarrow (B\backslash S))
\end{gathered}
$$

The idea is that we first can derive $a + 1 + a : S$ or $b + 1 + b : S$ so that $a + a : S \odot I$ ($b + b : S \odot I$) are in the copy language, but that we can derive $1 + a : S \downarrow (A\backslash S)$ and intercalate this structure into a derived S structure, obtaining for example $b + 1 + a + b : A\backslash S$. The type given for a ensures that we must also concatenate another a to the left, finally deriving $a + b + a + b : S \odot I$, which is also in the copy language. It is clear that these steps can be repeated. As an example, the following is a natural deduction proof of $a + b + a + b : S \odot_1 I$:

$$
\dfrac{0 : I \qquad \dfrac{a : A \qquad \dfrac{\dfrac{b : B \qquad \dfrac{1 : J \quad b : J\backslash(B\backslash S)}{1 + b : B\backslash S}\,E\backslash}{b + 1 + b : S}\,E\backslash \qquad \dfrac{1 : J \quad a : J\backslash(S \downarrow_1 (A\backslash S))}{1 + a : S \downarrow_1 (A\backslash S)}\,E\backslash}{b + 1 + a + b : A\backslash S}\,E\downarrow_1}{a + b + 1 + a + b : S}\,E\backslash}{a + b + a + b : S \odot_1 I}\,I\odot_1
$$

So what about the double copy language $\{www | w \in \{a, b\}^+\}$? If we were to use only one separator, we could derive structures like $a + b + a + b + b + a \in \{www^R | w \in \{a, b\}^+\}$, but that's still a mildly context-sensitive pattern. Inspired by set-local multi component tree adjoining grammar (MCTAG), we want to use two separators as insertion points but ensure that we have to synchronously intercalate two structures. This can be done in **2D** by introducing new types that act as a 'lock' for intercalation, that is to say, one intercalation causes the derived structure to be of a type that can only be unlocked by another intercalation. The grammar looks as follows:

$$
\begin{gathered}
S' = (S \odot_2 I) \odot_1 I \;\; a : A; J\backslash(A\backslash(J\backslash(A\backslash S))); J\backslash(S \downarrow_> (A\backslash T)); J\backslash(T \downarrow_< S) \\
b : B; J\backslash(B\backslash(J\backslash(B\backslash S))); J\backslash(S \downarrow_> (B\backslash U)); J\backslash(U \downarrow_< S)
\end{gathered}
$$

As an example, the second assignment for b allows the derivation of $b + 1 + b + 1 + b : S$, so that $b + b + b : (S \odot I) \odot I$ is in the cube language. Now, we can intercalate $1 + a : S \downarrow_> (A\backslash T)$ to derive $a + b + 1 + a + b + 1 + b : T$. The only way to 'unlock' this structure is to intercalate $1 + a : T \downarrow_< S$, resulting in $a + b + 1 + a + b + 1 + a + b : S$, causing $a + b + a + b + a + b : (S \odot I) \odot I$ to be recognized. A natural deduction proof for $a + b + a + b + a + b : (S \odot_2 I) \odot_1 I$ looks as follows:

$$\dfrac{1 : J \quad b : J\backslash(B\backslash(J\backslash(B\backslash S)))}{1 + b : B\backslash(J\backslash(B\backslash S))}\ E\backslash$$

$$\dfrac{b : B \quad 1 + b : B\backslash(J\backslash(B\backslash S))}{b + 1 + b : J\backslash(B\backslash S)}\ E\backslash$$

$$\dfrac{1 : J \quad b + 1 + b : J\backslash(B\backslash S)}{1 + b + 1 + b : B\backslash S}\ E\backslash$$

$$\dfrac{b : B \quad 1 + b + 1 + b : B\backslash S}{b + 1 + b + 1 + b : S}\ E\backslash$$

$$\dfrac{1 : J \quad a : J\backslash(S \downarrow_1 (A\backslash T))}{1 + a : S \downarrow_1 (A\backslash T)}\ E\backslash$$

$$\dfrac{b + 1 + b + 1 + b : S \quad 1 + a : S \downarrow_1 (A\backslash T)}{b + 1 + a + b + 1 + b : A\backslash T}\ E \downarrow_1$$

$$\dfrac{b + 1 + a + b + 1 + b : A\backslash T}{a + b + 1 + a + b + 1 + b : T}\ E\backslash$$

$$\dfrac{1 : J \quad a : J\backslash(T_2 \downarrow_2 S)}{1 + a : T_2 \downarrow_2 S}\ E\backslash$$

$$\dfrac{a + b + 1 + a + b + 1 + b : T \quad 1 + a : T_2 \downarrow_2 S}{a + b + 1 + a + b + 1 + a + b : S}\ E \downarrow_2$$

$$\dfrac{\dfrac{a + b + 1 + a + b + 1 + a + b : S}{a + b + 1 + a + b + a + b : S \odot_2 I}\ I\odot_2}{a + b + a + b + a + b : (S \odot_2 I) \odot_1 I}\ I\odot_1$$

This pattern can be generalized to $\{w^n | w \in \{a, b\}^+\}$ for $(n-1)\mathbf{D}$ assuming we can replace any separator, i.e. not just the outermost ones. For greater clarity, we added superscripts to denote the ith occurrence of a basic type, whereas subscripts denote different types:

$$a : A; J^n\backslash(A^n\backslash ... J^1\backslash(A^1\backslash S)); J\backslash(S \downarrow_1 (A\ T_1)); J\backslash(T_i \downarrow_i S) \text{ for } 2 \geq i \leq n$$
$$b : B; J^n\backslash(B^n\backslash ... J^1\backslash(B^1\backslash S)); J\backslash(S \downarrow_1 (B\ U_1)); J\backslash(U_i \downarrow_i S) \text{ for } 2 \geq i \leq n$$

The idea of this generalization is that we first can derive $a+(1+a)^{(n-1)} : S$ and we can intercalate $1 + a$ $(1 + b)$ $n - 1$ times of which the very first intercalation starts at the leftmost separators and also concatenates a a (b) to the left, resulting in $a + a + (1 + a + a)^{n-1} : S$ $(b + a + (1 + b + a)^{n-1} : S)$. Setting the start symbol to $S(\odot I)^{n-1}$, such a grammar exactly recognizes n copies of a word over $\{a, b\}^+$. Figure 2 shows what a natural deduction proof for w^n looks like.

$$\frac{1:J \quad b:J\backslash(B\backslash ...J\backslash(B\backslash S))}{1+b:B\backslash ...J\backslash(B\backslash S)}\; E\backslash$$

$$\frac{b:B \quad 1+b:B\backslash ...J\backslash(B\backslash S)}{b+1+b:...J\backslash(B\backslash S)}\; E\backslash$$

$$\vdots$$

$$b+1+b+1+...+1+b:S$$

$$\frac{1:J \quad a:J\backslash(S\downarrow_1 (A\backslash T_1))}{1+a:S\downarrow_1 (A\backslash T_1)}\; E\backslash$$

$$\frac{b+1+b+1+...+1+b:S \quad 1+a:S\downarrow_1 (A\backslash T_1)}{b+1+a+b+1+...+1+b:A\backslash T_1}\; E\downarrow_1$$

$$\frac{a:A \quad b+1+a+b+1+...+1+b:A\backslash T_1}{a+b+1+a+b+1+b+1...+1+b:T_1}\; E\backslash$$

$$\frac{1:J \quad a:J\backslash(T_1\downarrow_2 T_2)}{1+a:T_1\downarrow_2 T_2}\; E\backslash$$

$$\frac{a+b+1+a+b+1+b+1...+1+b:T_1 \quad 1+a:T_1\downarrow_2 T_2}{a+b+1+a+b+1+a+b+1+b+1+...+1+b:T_2}\; E\downarrow_2$$

$$\vdots$$

$$a+b+1+a+b+1+...+1+a+b+1+b:T_{n-1}$$

$$\frac{1:J \quad a:J\backslash(T_{n-1}\downarrow_{n-1} S)}{1+a:T_{n-1}\downarrow_{n-1} S}\; E\backslash$$

$$\frac{a+b+1+a+b+1+...+1+a+b+1+b:T_{n-1} \quad 1+a:T_{n-1}\downarrow_{n-1} S}{a+b+1+a+b+1+...+1+a+b:S}\; E\downarrow_{n-1}$$

$$\frac{0:I \quad a+b+1+a+b+1+...+1+a+b:S}{a+b+a+b+1+a+...+1+a+b:S\odot_1 I}\; I\odot_1$$

$$\vdots$$

$$a^1+b^1+...+a^n+b^n:(...(S\odot_1 I)\odot_2 I)...\odot_{n-1} I$$

Figure 2: A natural deduction proof for $(ab)^n$

11.3.2 Counting and Crossing Dependencies

It may have become clear that each separator may carry in its type 'local' and 'global' left and right leaves. In this sense, it is not surprising that counting dependencies up to 4 dependencies can be recognized in **1D**, for one separator can have two local left and right leaves for the inner dependencies that intersperse the outer dependencies, which are carried as 'global' left and right leaves. Up to four crossing dependencies can recognized by first counting the left and third dependency, and then interspersing the second and fourth dependencies. The grammars for four counting dependencies $\{a^n b^n c^n d^n\}$ and crossing dependencies respectively are as follows:

$$\begin{array}{cc} S' = B \odot_1 I & S' = D \odot_1 \\ a : A & a : A/J; (C \downarrow A)/J \\ b : ((A\backslash(B/D))/C)/J; ((B \downarrow_1 (A\backslash(B/D)))/C)/J & b : J\backslash(C \downarrow B); J\backslash(D \downarrow B) \\ c : C & c : A\backslash C \\ d : D & d : B\backslash D \end{array}$$

Note that our grammars differ from the ones introduced in [8]. This is only for the sake of simplicity, since the generalizations will be somewhat easier to read. A typical derivation of 4 crossing dependencies starts with c to derive $b + 1 + c : C$, then concatenating to d and a respectively to derive $a + b + 1 + c + d : A$. One then can go on to derive $b + 1 + c : A \downarrow C$ and intercalate this in the place of the separator. Because the start type is $A \odot I$, it is necessary to concatenate to the 'global' a and d left and right structures to derive an accepted string. See Figure 3 for a natural deduction proof. For counting dependencies, the idea is to first derive $a^n + 1 + c^n : C$ and at a point intercalating $1 + b : C \downarrow B$ and concatenating to d, producing $a^n + 1 + b + c^n + d : D$. With $D \odot_1 I$ as start symbol, this string can be accepted or we can intercalate another $1 + b$ and concatenate the result to d. So the first dependency is counted, and once the second dependency is inserted, the previous counting is lost and we go on counting the second dependency.

$$
\frac{\frac{\frac{a:A \quad \frac{\frac{\frac{b:((A\backslash(B/D))/C)/J \quad 1:J}{b+1:(A\backslash(B/D))/C}\,E/ \quad c:C}{b+1+c:A\backslash(B/D)}\,E/}{}}{a+b+1+c:B/D}\,E\backslash \quad d:D}{a+b+1+c+d:B}\,E/ \quad \frac{\frac{b:((B\downarrow_1(A\backslash(B/D)))/C)/J \quad 1:J}{b+1:(B\downarrow_1(A\backslash(B/D)))/C}\,E/ \quad c:C}{b+1+c:B\downarrow_1(A\backslash(B/D))}\,E/}{a+b+b+1+c+c+d:A\backslash(B/D)}\,E\downarrow_1
$$

$$
\frac{\frac{\frac{a:A \quad a+b+b+1+c+c+d:A\backslash(B/D)}{a+a+b+b+1+c+c+d:B/D}\,E\backslash \quad d:D}{a+a+b+b+1+c+c+d+d:B}\,E/ \quad 0:I}{a+a+b+b+c+c+d+d:B\odot_1 I}\,I\odot_1
$$

Figure 3: A natural deduction proof for $a^n b^n c^n d^n$

Again, these structures can be generalized: **nD** can recognize up to $2n+2$ counting dependencies and up to $n+1$ crossing dependencies. The following grammars generalize the above ones:

Counting Dependencies:

$S' = (...((A_2 \odot_1 I^1) \odot_1 I^2)...) \odot_1 I^{\frac{m-2}{2}}$ ($m-1$ if m is odd)
$a_1 : A_1$
$a_i : A_i$ for i is odd and $1 < i < m$
$a_m : A_m$

$a_2 : (A_1 \backslash (A_2/A_m))/A_{m-1})/J^1)/A_{m-2})/A_{m-3})/J^2)/.../A_4)/A_3)/J^{\frac{m-2}{2}}$

$a_2 : ((A_2 \downarrow_1 U_1)/A_3)/J$
$a_k : ((U_{\frac{k}{2}-1} \downarrow_{\frac{k}{2}} U_{\frac{k}{2}})/A_{k+1})/J$ for k is even and $1 < k < m$
$a_{m-2} : ((U_{\frac{m-2}{2}-1} \downarrow_{\frac{m-2}{2}} (A_1 \backslash (A_2/A_m)))/A_{m-1})/J$ if m is even
$a_{m-1} : ((U_{\frac{m-1}{2}-1} \downarrow_{\frac{m-1}{2}} (A_1 \backslash A_2))/A_{m-1})/J$ if m is odd

Crossing Dependencies: Let n be the number of crossing dependencies. Let $m = 2n$. Then the following grammar describes n crossing dependencies.

$S' = (...((A_2 \odot_1 I^1) \odot_1 I^2)...) \odot_1 I^{\frac{m-2}{2}}$
$a_1 : (...((((A_1/A_{m-1})/J)/A_{m-3})/J)/.../A_3)/J$
$a_i : A_i$ for i is odd and $1 < i \leq m-1$

$a_1 : (A_1 \downarrow_1 U_1)/J$
$a_k : (U_{\frac{k-1}{2}} \downarrow_{\frac{k+1}{2}} U_{\frac{k-1}{2}})/J$ for k is odd and $1 < k < m-3$
$a_{m-3} : ((U_{\frac{m-4}{2}} \downarrow_{\frac{m-2}{2}} A_1)/J)/A_{m-1}$

$a_2 : (A_1 \downarrow_1 T_1)/J$
$a_2 : (A_2 \downarrow_1 T_1)/J$
$a_l : (T_{\frac{l}{2}} \downarrow_{\frac{l}{2}+1} T_{\frac{l}{2}+1})/J$ for l is even and $2 < l < m-2$
$a_{m-2} : (T_{\frac{m-2}{2}-1} \downarrow_{\frac{m-2}{2}} (A_2/A_m))/J$

The general derivation structures as natural deduction proofs for counting dependencies are depicted in Figure 4:

$$\dfrac{a_2 : (\ldots(A_1\backslash(A_2/A_n))/A_{n-1})/J^1)/A_{n-2})/A_{n-3})/J^2)/\ldots/A_4)/A_3)/J^{\frac{n-2}{2}} \qquad 1 : J}{a_2 + 1 : ((\ldots(A_1\backslash(A_2/A_n))/A_{n-1})/J^1)/A_{n-2})/A_{n-3})/J^2)/\ldots/A_4)/A_3} \; E/$$

$$\dfrac{a_2 + 1 : ((\ldots(A_1\backslash(A_2/A_n))/A_{n-1})/J^1)/A_{n-2})/A_{n-3})/J^2)/\ldots/A_4)/A_3 \qquad a_3 : A_3}{a_2 + 1 + a_3 : ((\ldots(A_1\backslash(A_2/A_n))/A_{n-1})/J^1)/A_{n-2})/A_{n-3})/J^2)/\ldots/A_4} \; E/$$

$$\vdots$$

$$\dfrac{a_1 : A_1 \qquad a_2 + 1 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} : A_1\backslash(A_2/A_n)}{a_1 + a_2 + 1 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} : A_2/A_n} \; E\backslash$$

$$\dfrac{a_1 + a_2 + 1 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} : A_2/A_n \qquad a_n : A_n}{a_1 + a_2 + 1 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_n : A_2} \; E/$$

$$\dfrac{a2 : ((A_2 \downarrow_1 U_1)/A_3)/J \qquad 1 : J}{a2 + 1 : (A_2 \downarrow_1 U_1)/A_3} \; E/ \qquad \dfrac{a2 + 1 : (A_2 \downarrow_1 U_1)/A_3 \qquad a3 : A_3}{a2 + 1 + a3 : A_2 \downarrow_1 U_1} \; E/$$

$$\dfrac{a_1 + a_2 + 1 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_n : A_2 \qquad a2 + 1 + a3 : A_2 \downarrow_1 U_1}{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_n : U_1} \; E\downarrow_1$$

$$\dfrac{a_4 : ((U_1 \downarrow_2 U_2)/A_5)/J \qquad 1 : J}{a_4 + 1 : (U_1 \downarrow_2 U_2)/A_5} \qquad \dfrac{a_4 + 1 : (U_1 \downarrow_2 U_2)/A_5 \qquad a_5 : \ldots}{a_4 + 1 + a_5 : U_1 \downarrow_2 U_2}$$

$$\dfrac{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_n : U_1 \qquad a_4 + 1 + a_5 : U_1 \downarrow_2 U_2}{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + a_5 + a_5 + 1 + \ldots + 1 + a_{n-1} + a_n : U_2} \; E\downarrow$$

$$\vdots$$

$$a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n : U_{\frac{n-2}{2}}$$

$$\dfrac{a_{n-2} : ((U_{\frac{n-2}{2}-1} \downarrow_{\frac{n-2}{2}} (A_1\backslash(A_2/A_n)))/A_{n-1})/J \qquad \overline{1 : J}\; Ax.}{a_{n-2} + 1 : (U_{\frac{n-2}{2}} \downarrow_{\frac{n-2}{2}} (A_1\backslash(A_2/A_n)))/A_{n-1}} \; E/$$

$$\dfrac{a_{n-2} + 1 : (U_{\frac{n-2}{2}} \downarrow_{\frac{n-2}{2}} (A_1\backslash(A_2/A_n)))/A_{n-1}}{a_{n-2} + 1 + a_{n-1} : U_{\frac{n-2}{2}-1} \downarrow_{\frac{n-2}{2}} (A_1\backslash(A_2/A_n))} \; E/$$

$$\vdots$$

$$\dfrac{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_n : U_{\frac{n-2}{2}} \qquad a_{n-2} + 1 + a_{n-1} : U_{\frac{n-2}{2}-1} \downarrow_{\frac{n-2}{2}} (A_1\backslash(A_2/A_n))}{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n : A_1\backslash(A_2/A_n)} \; E\downarrow_{\frac{n-2}{2}}$$

$$\dfrac{a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n : A_1\backslash(A_2/A_n)}{a_1 + a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n : A_2/A_n} \; E\backslash$$

$$\dfrac{a_1 + a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n : A_2/A_n}{a_1 + a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n + a_n : A_2} \; E/$$

$$\dfrac{a_1 + a_1 + a_2 + a_2 + 1 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n + a_n : A_2}{a_1 + a_1 + a_2 + a_2 + a_3 + a_3 + a_4 + a_4 + 1 + \ldots + 1 + a_{n-1} + a_{n-1} + a_n + a_n : A_2 \odot_1 I} \; I\odot_1$$

$$\vdots$$

$$a_1 + a_1 + a_2 + a_2 + a_3 + a_3 + a_4 + a_4 + \ldots + a_{n-1} + a_{n-1} + a_n + a_n : (\ldots((A_2 \odot_1 I) \odot_1 I)\ldots) \odot_1 I$$

Figure 4: A natural deduction proof for $a_1^2 \ldots a_n^2$. Because of the size of the deduction, it is split into two parts

12 Well-nested simple Range Concatenation Grammar

In this subsection, we define *well-nested simple Range Concatenation Grammar* or $sRCG_{wn}$ and prove that every $sRCG$ (not only well-nested) can be written in a lexical form which will turn out to be quite similar to Greibach Normal Form for Context-Free Grammar.

12.1 Well-nested simple Range Concatenation Grammar

The following definitions are taken from [13].

Definition 12. *A simple RCG is a tuple* (N, T, V, P, S) *where:*

- N *is a finite set of predicate names with an arity function* $dim : N \to \mathbf{N}$
- T *and* V *are finite disjoint sets of terminals and variables*
- $S \in N$ *is the start symbol with* $dim(S) = 1$
- P *is a finite set of clauses of the form*
  $A(\alpha_1, .., \alpha_{dim(A)}) \to A_1(X^1_1, .., X^1_{dim(A_1)}), .., A_m(X^m_1, .., X^m_{dim(A_m)})$
  for $m \geq 0$ *where:*
  1. $A, A_1, .., A_m \in N$
  2. $X^i_j \in V$ *for* $1 \leq i \leq m, 1 \leq j \leq dim(A_i)$
  3. $\alpha_i \in (T \cup V)^*$ *for* $1 \leq i \leq dim(A)$
- *For all* $c \in P$*, every variable* $X \in V$ *occurring in* c *occurs exactly once in the left-hand side and exactly once on the right-hand side (simple)*

Range Concatenation Grammars as well as Multiple Context-Free Grammars has a natural control parameter. The order of a RCG ($MCFG$) is defined by $max(dim(N))$ where $dim(N) = \bigcup_{i=1}^{|N|}\{dim(n_i)\}$ for $n_i \in N$. We shorten $A(\alpha_1, ..., \alpha_{dim(A)})$ by $A(\overrightarrow{\alpha})$.

Definition 13. *A sRCG R is well-nested if for each clause* $c \in P$ *is holds that the right hand side argument variables are well-nested with respect to their left-hand side ordering.*

A well-nested $sRCG$ will be denoted $sRCG_{wn}$. For example, in a $sRCG_{wn}$, clauses like $A(X, Y, Z, U) \to B(X, U)C(Y, Z)$ are permitted, but clauses like

$A(X,Y,Z,U) \to B(X,Z)C(Y,U)$ are not. $sRCG_{wn}$ is a proper subclass of $sRCG$ and [14] has elaborated well on the difference class $sRCG - sRCG_{wn}$.

In order to define the string and tree languages of $sRCG$, we need to define the *derive* relation which specifies how we apply the clauses to an input string. Because we can have several bindings of variables in the clauses, we need the notion of *ranges* and of *clause instantiations*.

Definition 14. *Let $w = w_1...w_n$ be a word over T^* where $w_i \in T$ for $1 \leq i \leq n$.*

- *The set of all positions is defined as $Pos(w) := \{0, ..., n\}$.*
- *A* range *in w is a pair $\langle l, r\rangle \in Pos(w) \times Pos(w)$ with $l \leq r$.*
- *The yield of a range is defined $\langle l, r\rangle(w) = w_{l+1}...w_r$.*
- *The concatenation of two ranges $\rho_1 = \langle l_1, r_1\rangle$ and $\rho_2 = \langle l_2, r_2\rangle$ is defined as follows:*

$$\rho_1 \cdot \rho_2 = \begin{cases} \langle l_1, r_2\rangle & \text{if } r_1 = l_2; \\ undefined & \text{otherwise.} \end{cases}$$

- *For every $a \in T$, the* ranges *function is defined as $ranges(a, w) = \{\langle i-1, i\rangle | 1 \leq i \leq n, \langle i-1, i\rangle(w) = a\}$*
- *A range vector $\rho \in (Pos(w) \times Pos(w))^k$ is a k-dimensional range vector for w iff $\rho = \langle\langle l_1, r_1\rangle, ..., \langle l_k, r_k\rangle\rangle$ where $\langle l_i, r_i\rangle$ is a range in w for $1 \leq i \leq k$.*

Definition 15. *Let $G = (N, V, T, P, S)$ be an sRCG, let $c = A(\overrightarrow{\alpha}) \to A_1(\overrightarrow{\alpha}_1)...A_m(\overrightarrow{\alpha}_m)$ be a clause in P and let $w = w_1...w_n$ be a word over T^* where $w_i \in T$ for $1 \leq i \leq n$. Furthermore, let $\overrightarrow{\alpha}(i)$ denote the ith argument of $\overrightarrow{\alpha}$.*

- *$Occ_{term} = \{t' | t'$ is an occurrence of some t in the clause $\}$*
- *$Occ_{eps} = \{Eps_i | 1 \leq i \leq dim(A), \overrightarrow{\alpha}(i) = \epsilon\}$*
- *An instantiation with respect to w is a function $f : Occ_{term} \cup V \cup Occ_{eps} \to \{\angle i, j\rangle | i \leq j, i, j \in \mathbf{N}\}$ such that:*
  1. *For all $t' \in Occ_{term}$ that occur in $\overrightarrow{\alpha}$, $f(t')(w) = a$,*
  2. *For all $X \in V$, $f(X) = \langle j, k\rangle$ for some $0 \leq j \leq k \leq n$,*

3. *For all* $Eps \in Occ_{eps}$*, there is a* j *with* $0 \leq j \leq n$ *with* $f(Eps) = \langle j, j \rangle$*. For every* ϵ*-argument* $\overrightarrow{\alpha}(i)$ *we define* $f(\overrightarrow{\alpha}(i)) = f(Eps_i)$*,*
4. *For all adjacent* x, y *in one of the elements of* $\overrightarrow{\alpha}$ *there are* i, j, k *with* $f(x) = \langle i, j \rangle$ *and* $f(y) = \langle j, k \rangle$*. Then we define* $f(xy) = \langle i, k \rangle$*.*

- *If* f *is an instantiation with respect to* c *and* w*, then* $A(f(\overrightarrow{\alpha})) \rightarrow A_1(f(\overrightarrow{\alpha}_1))...A_m(f(\overrightarrow{\alpha}_m))$ *is an* instantiated clause.

To determine whether a string w over T^* is in the language of an *sRCG*, we instantiate the start clause relative to w, and in each derivation step, we replace the left-hand side of an instantiated clause with it's right-hand side. We are now ready to define the string language of an *sRCG*.

Definition 16. *Let* $G = (N, T, V, P, S)$ *be an sRCG. The* string language *of* G *is defined as* $L_S(G) = \{w \mid S(0, |w|) \Rightarrow^* \epsilon\}$

Because we want to compare *strong generative capacity* we need to define the tree language for *sRCG* as well.

Definition 17. *Let* $G = (N, T, V_G, P, S)$ *be a sRCG, and let* $w = w_1...w_n$ *be a word over* T^* *where* $w_i \in T$ *for* $1 \leq i \leq n$*. A derivation tree is a tree* $D = (V, l, E, r)$ *such that:*

- *There are exactly* n *pairwise different leaves* $u_1...u_n$ *with* $l(u_i) = \langle i{-}1, i \rangle$ *for* $1 \leq i \leq n$*, for all other leaves* z *we have* $l(z) = \langle i, i \rangle$ *for some* i *with* $0 \leq i \leq n$*. Furthermore, for each leaf* u *we define r-yield*$(u) = \{l(u)\}$*.*

- *For every internal node* $v_0 \in V$*, for every order* $v_1...v_k$ *of the pairwise different daughters that are internal nodes, if we have that* $l(v_i) = A_i$ *for* $0 \leq i \leq k$*, then for every clause* $A_0(\overrightarrow{\alpha}_0) \rightarrow A_1(\overrightarrow{\alpha}_1)...A_k(\overrightarrow{\alpha}_k)$*, if there is an instantiation* $A_0(\overrightarrow{\rho}_0) \rightarrow A_1(\overrightarrow{\rho}_1)...A_k(\overrightarrow{\rho}_k)$ *with respect to* w*, such that:*

  - $\rho_i \in r\text{-}yield(v_i)$ *for* $1 \leq i \leq k$*,*
  - *There is a daughter* u *of* v_0 *that is a leaf iff either one of the terminals or one of the* ϵ*-arguments in* $\overrightarrow{\alpha}_0$ *is mapped to* $l(u)$ *by this instantiation.*

  then $\overrightarrow{\alpha}_0 \in r\text{-}yield(v_0)$*. Nothing else is in* $r - yield(v_0)$*.*

- $\langle 0, n \rangle \in r\text{-}yield(r)$
- $l(r) = S$

Definition 18. *Let* $G = (N, T, V, P, S)$ *be an sRCG. The* tree language *of* G *is defined as* $L_T(G) = \{D \mid D \text{ is a derivation tree of some } w \in T^*\}$

12.2 A lexicalized normal form for (simple) Range Concatenation Grammar

[3] defines a number of normal forms for Range Concatenation Grammar which naturally extend from known normal forms for Context Free Grammar, including ϵ-free normal form which derives from the determinization of CFG's and binarization, which extends the well-known Chomsky Normal Form for Context Free Grammar. However, it is also known that CFG's can be lexicalized by transformation into the Greibach Normal Form, which is shown in [9]. Here, we will show how to lexicalize simple Range Concatenation Grammar (and thus MCFG and LCFRS as well) by a transformation that is similar to the construction of the Greibach Normal Form for CFG. We begin by defining binary RCG, lexicalizedness for sRCG, and move on to prove the equivalence of sRCG and lexicalized sRCG. Lexicalization is an important property since it can simplify parsing algorithms. Given that Categorial Grammar is by definition always lexicalized, knowing that $sRCG_{wn}$ can be lexicalized plays an important role in proving it's equivalence with first-order Displacement Grammar, which we will turn to in the next section.

Just as any Context Free Grammar can be converted to an equivalent ϵ-free form without useless rules (i.e. clauses that cannot be reached from the start clause), it is also possible to make the same conversion for RCG. Since these normal forms are directly extended from CFG normal forms and they are not directly relevant to the intuition of a lexicalized normal form, we won't elaborate on them here. Rather, we direct the reader to [3] for an overview. We will however, briefly describe Greibach Normal Form for CFG, because it is not immediately obvious how and why this conversion works.

Definition 19. *A Context Free Grammar is in Chomsky Normal Form (CNF) if all the rules are of the form:*

- $A \rightarrow BC$ *with* A, B, C *non-terminals*
- $A \rightarrow a$ *with* A *a non-terminal and* a *a terminal symbol*

Definition 20. *A Context Free Grammar is in Greibach Normal Form (GNF) if all the rules are of the form:*

- $A \rightarrow a\alpha$ *with* A *a non-terminal,* a *a terminal symbol and* α *a possibly non-empty sequence of non-terminals.*

As we can see, a rule of a CFG in GNF is either a terminal rule, or a rule whose right-hand side starts with a terminal. It is not at all obvious how one obtains the Greibach Normal Form from a CFG. The construction however uses Chomsky Normal Form to obtain a lexicalized CFG. In order to lexicalize a CFG, we want to be able to substitute the first right-hand side non-terminal B by a terminal symbol a, but in order to preserve language, we have to replace it by all possible right-hand sides of B-rules. This substitution will therefore only work when all B-rules are already in Greibach Normal Form. A second problem is that a CFG can contain cycles, for example the rules $A \rightarrow BC$ and $B \rightarrow AD$. To overcome these problems, we want to have an ordering on the non-terminals such that each sequence beginning with the lowest element of the ordering always rewrites (we only look at the leftmost symbol on the right-hand side) to either a terminal symbol (GNF) or a non-terminal that is a higher element in the ordering. More formally, we want to have that for each rule $A_i \rightarrow A_j\gamma$, $i < j$. If this is the case, we have ensured that there are no left-cycles and that we can perform the substitions mentioned above. This last fact is true because the rules of the highest element of the ordering must be all terminal clauses, so we can perform a language-preserving substitution that puts all the rules of the single highest element of the ordering in Greibach Normal Form. Then we can do the same until all rules are in GNF.

Note that for a rule $A_i \rightarrow A_k\gamma$ with $i > k$ we can definitely substitute A_k by all possible right-hand sides of A_k rules, and repeat this until each A_i rule is of the form $A_i \rightarrow A_l\delta$ with $i \leq l$. For the cases that an A_i rule is of the form $A_i \rightarrow A_i\delta$, we can use a method called left-recursion elimination (which preserves language) to ensure that all cycles eventually disappear.

Now that we have given an informal explanation of what we need to make the GNF construction work, we can formalize the construction for greater clarity, before we move to a similar construction for $sRCG$. We refer the reader here to [9].

For a CFG $G = (N, T, R, S)$ in CNF, we construct the Greibach Normal Form as follows:

1. Assign a (random) ordering $\{A_1, ..., A_n\}$ on the clauses in R.
2. Modify the clauses in such a way that if $A_i \rightarrow A_j\gamma$, $j \leq k$. This is done as follows:
    - Assume that the rules have been modified such that for $1 \leq i \leq k$,

for each rule $A_i \to A_j\gamma$ it is the case that $i < j$.

- For each rule $A_k \to A_j\gamma$ with $k > j$, replace the rule by a new set of rules with A_j substituted by the right-hand side of each A_j rule. Repeating this at most $k-1$ times, all A_k rules are of the form $A_k \to A_j\gamma$ with $k \leq j$.
- Eliminate left-recursive rules $A_k \to A_k\gamma$.

3. Lexicalize the A_k rules, starting with A_{n-1} and ending with A_1:

   - For each non-terminal rule $A_k \to A_j$ (we have $k < j$), replace the rule by a new set of rules with A_j substituted by the right-hand side of each A_j rule. If we start with A_{n-1} and end with A_1, we ensure that each A_j rule is in Greibach Normal Form, and thus the substitutions ensure that the new A_k rules are in GNF as well.

4. Lexicalize the B_k rules. This is done in the same way as the A_k rules are lexicalized, except that the order does not matter here.

5. Add a new start clause $S' \to S$.

Eliminating left-recursive rules is done by replacing each left-recursive rule with a new rule, involving a new non-terminal, which is right-recursive. Any rule $A \to A\beta$ together with $A \to a$ describes a language $\{a\beta^n | n \geq 0\}$. We can replace these rules by new rules, one of which is right-recursive, while preserving this language. These are $B \to \beta$, $B \to \beta B$, $A \to a$ and $A \to aB$. In general, we replace all A_i rules at a time:

- Each A_i rule is either a left-recursive rule, or the first non-terminal of it's right-hand side is higher in order than A_i, so we have that the A_i rules are divided in rules $A_i \to A_i\beta_1 ... A_i \to A_i\beta_n$ and rules $A_i \to \alpha_1 ... A_i \to \alpha_m$.

- We choose a new non-terminal, B_i and add rules $B_i \to \beta_j$ and $B_i \to \beta_j B$ for $1 \leq j \leq n$.

- We add rules $A_i \to \alpha_j$ and $A_i \to \alpha_j B_i$ for $1 \leq j \leq m$.

Now, we will turn to the lexicalization of (well-nested) simple RCG by a construction that is essentialy very similar to the construction of the Greibach Normal Form for a CFG. We begin with the definition of lexicalized $sRCG$:

Definition 21. *A sRCG* $R = (N, T, V, P, S)$ *is lexicalized (and denoted by* $LsRCG$ *) if each clause* $c \in P$ *except the start clause contains exactly one terminal symbol.*

It is important here to note two important differences between RCG and CFG:

1. In RCG, the order in which the right-hand side is written does not matter for generative capacity, whereas in CFG this order is vitally important. Rather, in an RCG, the order of words is present in the *arguments* of the left-hand side predicate, and their distribution at the right-hand side.

2. In a CFG in Chomsky Normal Form each lexicalized rule contains exactly one terminal symbol, and the resulting CFG in Greibach Normal Form contains only rules that contain exactly one terminal symbol. This restriction does not trivially hold for ϵ-free sRCG without useless rules, although we will need it for we want to compare the *strong* generative capacity of D^1 with $sRCG_{wn}$.

We can easily overcome the second difference: Any clause that contains more than one terminal symbol can be replaced by a set of clauses that each contain exactly one terminal symbol in a language-preserving way. We will call an $sRCG$ with this restriction a *one-terminal* $sRCG$.

Definition 22. *A sRCG* $R = (N, T, V, P, S)$ *is a* one-terminal $sRCG$ *(and denoted* $1T$*-*$sRCG$*)if each clause* $c \in P$ *except the start clause contains at most one terminal symbol.*

We will now turn to proving that given an $sRCG$ G, we can construct an equivalent one-terminal $sRCG$ and an equivalent $LsRCG$. Because the constructions use two operations, namely substitution and elimination of left-recursion, we start by defining these operations and show that the string language of an $sRCG$ is *invariant under substitution* and *invariant under elimination of left-recursion.*

Definition 23. *Let* $R = (N, T, V, P, S)$ *be a sRCG. Let* $c = A_0(\overrightarrow{\alpha_0}) \rightarrow A_1(\overrightarrow{\alpha_1})...A_n(\overrightarrow{\alpha_n})$ *be a clause in* P *and let* A_1 *be the predicate that will be substituted. We define then a substitution as the replacement of* c *by all possible instantiations of an* A_1 *clause. That is, for all* A_1 *clauses of the form* $A_1(\overrightarrow{\beta}) \rightarrow B_1(\overrightarrow{\beta_1})...B_m(\overrightarrow{\beta_m})$*, we add a new clause* $c' = A_0(\alpha_0') \rightarrow B_1(\overrightarrow{\beta_1})...B_m(\overrightarrow{\beta_m})A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$ *where* $\overrightarrow{\alpha_0'}$ *denotes* $\overrightarrow{\alpha_0}$ *with variables from*

$\overrightarrow{\alpha_1}$ *that occur in* $\overrightarrow{\alpha_0}$ *replaced by the variables in* $\overrightarrow{\beta}$. *Finally, we remove the original* A_0 *clause.*

Example. Let R be a *sRCG* given by the following clauses:

$$\begin{aligned}
&S(XYZ) \to A(X,Y,Z)\\
&A(XY,Z,U) \to B(X,Y)C(Z,U)\\
&B(XY,ZU) \to D(X,U)E(Y,Z)\\
&B(X,YZ) \to E(X,Y)F(Z)\\
&D(X,Y) \to B(X,Y)
\end{aligned}$$

Then, we substitute B in the A clause, giving a new set of clauses:

$$\begin{aligned}
&S(XYZ) \to A(X,Y,Z)\\
&A(XT_1YT_2,Z,U) \to D(X,T_2)E(T_1,Y)C(Z,U)\\
&A(XYT_1,Z,U) \to E(X,Y)F(T_1)C(Z,U)\\
&B(XY,ZU) \to D(X,U)E(Y,Z)\\
&B(X,YZ) \to E(X,Y)F(Z)\\
&D(X,Y) \to B(X,Y)
\end{aligned}$$

■

Lemma 1. *The string language of an sRCG is invariant under substitution.*

Proof. Let $R = (N,T,V,P,S)$ be an *sRCG*, and let $R' = (N,T,V',P',S)$ be an *sRCG* in which one clause $c \in P$ has been substituted by it's left-most right-hand side argument. Let $c = A_0(\overrightarrow{\alpha_0}) \to A_1(\overrightarrow{\alpha_1})...A_n(\overrightarrow{\alpha_n})$ and let $A_1(\overrightarrow{\beta}) \to B_1(\overrightarrow{\beta_1})...B_m(\overrightarrow{\beta_m})$ be all the A_1 clauses.
Now, let $w = w_1...w_n \in L(R)$, and let it's rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ include a clause instantiation $A_0(\rho_{\mathbf{0}}) \to A_1(\rho_{\mathbf{1}})...A_n(\rho_{\mathbf{n}})$. Then, in the next step, a clause instantiation $A_1(\rho_{\mathbf{1}}) \to B_1(\sigma_{\mathbf{1}})...B_m(\sigma_{\mathbf{m}})$ for some A_1 clause is used. Then there is a clause instantiation

$$A_0(\rho_{\mathbf{0}}) \to B_1(\sigma_{\mathbf{1}})...B_m(\sigma_{\mathbf{m}})A_2(\rho_{\mathbf{2}})...A_n(\rho_{\mathbf{n}})$$

for R'. Because this holds for every clause that is in P'-P, we may conclude that there exists a rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ for R' and thus, that $w \in L(R')$.

For the converse, let $w = w_1...w_n \in L(R')$, and let it's rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ include a clause instantiation

$$Inst = A_0(\rho_{\mathbf{0}}) \rightarrow B_1(\sigma_{\mathbf{1}})...B_m(\sigma_{\mathbf{m}})A_2(\rho_{\mathbf{2}})...A_n(\rho_{\mathbf{n}})$$

for a clause $c \in P'$-P.
Let $A_0(\alpha_0) \rightarrow A_1(\alpha_1)...A_n(\alpha_n)$ and $A_1(\beta) \rightarrow B_1(\beta_1)...B_m(\beta_m)$ be the clauses that produced $A_0(\alpha'_0) \rightarrow B_1(\beta_1)...B_m(\beta_m)A_2(\alpha_2)...A_n(\alpha_n)$ of which $Inst$ is an instantiation. Then there are clause instantiations

$$A_0(\rho_{\mathbf{0}}) \rightarrow A_1(\rho_{\mathbf{1}})...A_n(\rho_{\mathbf{n}})$$

and

$$A_1(\rho_{\mathbf{1}}) \rightarrow B_1(\sigma_{\mathbf{1}})...B_m(\sigma_{\mathbf{m}})$$

for R. Because this holds for every clause $c \in P'$-P, we may conclude that there exists a rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ for R and thus, that $w \in L(R)$. We conclude that $L(R) = L(R')$. For inverse substitution, the proof is the symmetric analog of the one given. ∎

Definition 24. *Let $R = (N,T,V,P,S)$ be a sRCG. Let Let $A_0(\overrightarrow{\alpha_0}) \rightarrow A_0(\overrightarrow{\alpha_1})A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$ be the form of all left-recursive A_0 clauses in P. Let $A_0(\overrightarrow{\beta_0}) \rightarrow \gamma$ be the form of the remaining A_0 clauses. We then define elimination of left-recursion as the replacement of all the left-recursive clauses by new (right-recursive) clauses in which a new predicate is used. That is, we choose a new nonterminal B_0 with $dim(B_0) = dim(A_0)$ and:*

- *For each left-recursive clause $A_0(\overrightarrow{\alpha_0}) \rightarrow A_0(\overrightarrow{\alpha_1})...A_n(\overrightarrow{\alpha_n})$, we add two new B_0 clauses, namely*

$$B_0(\overrightarrow{\beta_0}) \rightarrow A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})B_0(\overrightarrow{\alpha_1})$$
$$B_0(\overrightarrow{\beta'_0}) \rightarrow A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$$

  where $\overrightarrow{\beta'_0}$ denotes $\overrightarrow{\beta_0}$ with all occurrences of variables in $\overrightarrow{\alpha_1}$ deleted.

- *For each of the remaining clauses $A_0(\overrightarrow{\beta_0}) \rightarrow \gamma$, we add for each left-recursive clause $A_0(\overrightarrow{\alpha_0}) \rightarrow A_0(\overrightarrow{\alpha_1})A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$ the clause $A_0(\beta'_0) \rightarrow \gamma B_0(\overrightarrow{\beta_1})$ where $\overrightarrow{\beta_1}$ denotes the vector of variables with $dim(B_0)$ as it's length and $\overrightarrow{\beta'_0}$ denotes $\overrightarrow{\beta_0}$ with the variables of $\overrightarrow{\beta_1}$ inserted at corresponding places.*

- *We remove all left-recursive clauses $A_0(\overrightarrow{\alpha_0}) \rightarrow A_0(\overrightarrow{\alpha_1})A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$.*

Example. Let R be a *sRCG* given by the following clauses:

$$\begin{array}{l} S(XY) \rightarrow A(X,Y) \\ A(XY,Z) \rightarrow A(X,Z)C(Y) \\ A(X,Y) \rightarrow D(X,Y) \\ A(X,YZ) \rightarrow E(X,Y,Z) \end{array}$$

Then, we introduce B' and add new A and B' clauses, giving a new set of clauses:

$$\begin{array}{l} S(XY) \rightarrow A(X,Y) \\ B'(X,\epsilon) \rightarrow C(X) \\ B'(XY,Z) \rightarrow C(Y)B'(X,Z) \\ A(XT_1,T_2Y) \rightarrow D(X,Y)B'(T_1,T_2) \\ A(XT_1,T_2YZ) \rightarrow E(X,Y,Z)B'(T_1,T_2) \\ A(X,Y) \rightarrow D(X,Y) \\ A(X,YZ) \rightarrow E(X,Y,Z) \end{array}$$

■

Lemma 2. *The string language of an sRCG is invariant under elimination of left-recursion.*

Proof. Let $R = (N,T,V,P,S)$ be an *sRCG* containing at least one useful left-recursive clause in P, and let $R' = (N,T,V',P',S)$ be an *sRCG* in which c has been eliminated by the construction given in the definition of elimination of left-recursion. Let $A_0(\overrightarrow{\alpha_0}) \rightarrow A_0(\overrightarrow{\alpha_1})A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$ be the form of all left-recursive A_0 clauses and let $A_0(\overrightarrow{\beta_0}) \rightarrow \gamma$ be the form of the remaining A_0 clauses. Furthermore, let

$$\begin{array}{c} B_0(\overrightarrow{\beta_0}) \rightarrow A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})B_0(\overrightarrow{\alpha_1}) \\ B_0(\overrightarrow{\beta_0'}) \rightarrow A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n}) \\ A_0(\overrightarrow{\beta_0'}) \rightarrow \gamma B_0(\overrightarrow{\beta_1}) \\ A_0(\overrightarrow{\beta_0}) \rightarrow \gamma \end{array}$$

be the clauses produced by the elimination operation.

Now, let $w = w_1...w_n \in L(R)$, and let it's rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ include a clause instantiation $A_0(\rho_{\mathbf{0}}) \rightarrow A_0(\rho_{\mathbf{1}})A_2(\rho_{\mathbf{2}})...A_n(\rho_{\mathbf{n}})$. For the next step, either a left-recursive A_0 clause or a remaining A_0 clause is instantiated and used in the derivation, so we have either one of two types of instantiations

$$A_0(\rho_{\mathbf{1}}) \to A_0(\sigma_{\mathbf{1}})A_2(\sigma_{\mathbf{2}})...A_n(\sigma_{\mathbf{n}})$$
$$A_0(\rho_{\mathbf{1}}) \to \gamma$$

So we see that we have a number of copies $A_2(\sigma_{\mathbf{2}})...A_n(\sigma_{\mathbf{n}})......A_2(\rho_{\mathbf{2}})...A_n(\rho_{\mathbf{n}})$ at the right-hand side, until finally, some remaining clause is used and we have some γ to the left. Then, the following instantiations can be made in R':

$$A_0(\rho_{\mathbf{0}}) \to \gamma B_0(\eta_{\mathbf{0}})$$
$$B_0(\overrightarrow{\eta_0}) \to A_2(\overrightarrow{\eta_2})...A_n(\overrightarrow{\eta_n})B_0(\overrightarrow{\eta_1})$$
$$\vdots$$
$$B_0(\overrightarrow{\beta_0}) \to A_2(\overrightarrow{\alpha_2})...A_n(\overrightarrow{\alpha_n})$$

We may conclude that there exists a rewriting sequence $S(\langle 0,n\rangle) \Rightarrow^* \epsilon$ in R' for w and thus, that $w \in L(R')$.

For the converse, let $w = w_1...w_n \in L(R')$ and let it's rewriting sequence $S(0,n) \Rightarrow^* \epsilon$ include a clause instantiation

$$Inst = B_0(\eta_{\mathbf{0}}) \to A_2(\eta_{\mathbf{2}})...A_n(\eta_{\mathbf{n}})B_0(\eta_{\mathbf{1}})$$

Because this clause is reachable (otherwise, it could not be used in a correct rewriting sequence), it must be preceded by a clause instantiation $A_0(\rho_{\mathbf{0}}) \to \gamma B_0(\eta_{\mathbf{0}})$ and it finally precedes (because the clause is right-recursive) an instantiation $B_0(\sigma_{\mathbf{0}}) \to A_2(\sigma_{\mathbf{2}})...A_n(\sigma_{\mathbf{n}})$. So we see that we have some γ followed by a number of copies $A_2(\sigma_{\mathbf{2}})...A_n(\sigma_{\mathbf{n}})......A_2(\rho_{\mathbf{2}})...A_n(\rho_{\mathbf{n}})$. Then, the following instantiations can be made in R:

$$A_0(\rho_{\mathbf{1}}) \to A_0(\sigma_{\mathbf{1}})A_2(\sigma_{\mathbf{2}})...A_n(\sigma_{\mathbf{n}})$$
$$A_0(\rho_{\mathbf{1}}) \to \gamma$$

We may conclude that there exists a rewriting sequence $S(\langle 0,n\rangle) \Rightarrow^* \epsilon$ in R for w and thus, that $w \in L(R)$.
We conclude that $L(R) = L(R')$. For the inverse case, the proof is the symmetric analog of the one given.

■

Lemma 3. *For every ϵ-free sRCG $R = \{N,T,V,P,S\}$ there is an equivalent one-terminal sRCG $R' = \{N',T,V',P',S\}$.*

Proof.

- The set of terminals and the start symbol are the same for R and R'.
- The set of non-terminals and the set of variables of R are added new non-terminals and variables as we construct P'.
- The set of clauses P' is constructed by replacing each clause $c \in P$ that contains more than one terminal symbol with a set of new clauses, in the following way:
  Let $c = A(\alpha_1, ..., \alpha_j) \to A_1(X_1, ..., X_n)...A_m(Y_1, ..., Y_k)$ be a clause containing more than one terminal symbol, and denote all occurring terminal symbols b_0 through b_r (b_0 being the leftmost occurring terminal symbol and b_r being the rightmost one). For each occurring terminal symbol b_i for $1 \leq i \leq r$, add a new clause $B_i(b_i) \to \epsilon$ and replace c by $c' = A(\beta_1, ..., \beta_2) \to A_1(X_1, ..., X_n)...A_m(Y_1, ..., Y_k)B_1(T_1)...B_r(T_r)$ where $\beta_1, ..., \beta_j$ denote $\alpha_1, ..., \alpha_j$ with each b_i replaced by its corresponding variable T_i.

Because the construction only applies consecutive inverse substitutions, by lemma 1 we have that $L(R) = L(R')$.

■

Example. Consider the following $sRCG_{wn}$ for the double copy language $\{w^3 | w \in \{a, b\}^*\}$:

$$
\begin{aligned}
&S(XYZ) \to A(X, Y, Z)\\
&A(aX, aY, aZ) \to A(X, Y, Z)\\
&A(bX, bY, bZ) \to A(X, Y, Z)\\
&A(a, a, a) \to \epsilon\\
&A(b, b, b) \to \epsilon
\end{aligned}
$$

We replace all clauses but the start clause with new clauses. As an example, for the second clause $A(aX, aY, aZ) \to A(X, Y, Z)$ we add new rules $B(a) \to \epsilon$ and $C(a) \to \epsilon$ and replace the original clause by $A(aX, T_1Y, T_2Z) \to A(X, Y, Z)B(T_1)C(T_2)$.
The resulting grammar looks as follows:

$S(XYZ) \to A(X,Y,Z)$
$A(aX,T_1Y,T_2Z) \to A(X,Y,Z)B(T_1)C(T_2)$
$A(bX,T_1Y,T_2Z) \to A(X,Y,Z)D(T_1)E(T_2)$
$A(a,T_1,T_2) \to F(T_1)G(T_2)$
$A(b,T_1,T_1) \to H(T_1)I(T_2)$
$B(a) \to \epsilon$
$C(a) \to \epsilon$
$D(b) \to \epsilon$
$E(b) \to \epsilon$
$F(a) \to \epsilon$
$G(a) \to \epsilon$
$H(b) \to \epsilon$
$I(b) \to \epsilon$

■

Lemma 4. *For every sRCG $R = \{N,T,V,P,S\}$ with $\epsilon \notin L(R)$, there is an equivalent lexicalized sRCG $R' = \{N',T,V,P',S'\}$.*

Proof. The idea of the conversion is similar to that of the Greibach Normal Form for CFG's. We construct R' from the ϵ-free form, with useless rules eliminated. Essentially, the construction alters the set of non-terminals and the set of clauses. These resulting sets are denoted N' and P' respectively, and are assumed to become clear from the following construction:

1. Assign an ordering $\{A_1,...,A_n\}$ on the clauses in P.
2. Modify the clauses in such a way that if $A_j(\alpha_1,...,\alpha_m) \to A_k(X_1,...,X_n)\gamma$, $j \leq k$.
3. Eliminate left-recursive clauses $A_k \to A_k\gamma$, thereby introducing new clauses B_k.
4. Lexicalize the clauses, starting with A_{n-1} and ending with A_1.
5. Lexicalize the B_k clauses.
6. Add a new start clause $S'(X) \to S(X)$.

Given that we have assigned an ordering on the clauses, we can modify the clauses such that if $A_j(\alpha_1,...,\alpha_m) \to A_k(X_1,...,X_n)\gamma$, $j \leq k$ as follows:

For k from A_1 to A_n:

1. Assume that for $1 \le i \le k$, $A_i(\alpha_1, ..., \alpha_m) \to A_j(X_1, ..., X_p)\gamma$ only if $j > i$.
2. If $A_k(\alpha_1, ..., \alpha_m) \to A_j(X_1, ..., X_p)\gamma \in P$ and $j < k$, remove this clause and add clauses with $A_j(X_1, ..., X_p)$ substituted with the righthand side of each A_j clause. Note that when a A_j clause is lexicalized, we move these terminal symbols to A_k in the logical sense.
3. If we repeat 2) at most $k - 1$ times, every clause is of the form

   $$A_k(\alpha_1, ..., \alpha_m) \to A_j(X_1, ..., X_o)$$

   with $j \ge k$.
4. If an A_k clause is left-recursive, i.e. it is of the form $A_k(\alpha_1, ..., \alpha_m) \to A_k(X_1, ..., X_m)\gamma$, there must also be at least one terminal A_k clause (else the A_k clause is useless and thus would not exist). We replace all left-recursive clauses for a k by eliminating left-recursion as described in definition 24.

Now we have that each A_n clause must be a terminal clause, and thus is lexicalized since the construction starts off with an ϵ-free $SRCG$. Each A_{n-1} clause is either terminal or has A_n as the leftmost predicate in the righthand side. Replace the latter clauses by each possible substitution of variables by the terminal symbols in A_n. Repeat this for $An - 2$ until A_1. Now we can do the same for each B_i clause.

Now we show that $L(R) = L(R')$. Because the construction only applies consecutive substitutions and eliminations of left-recursion, we have by lemma 1 and lemma 2 that $L(R) = L(R')$.

∎

Example. Consider the following $sRGC_{wn}$ grammar, which is a slight modification of a linguistic example drawn from [13]:

$$\begin{array}{l} S(XYZ) \to A(X,Z)B(Y) \\ A(X,YZ) \to A(X,Y)C(Z) \\ A(X,Y) \to D(X)E(Y) \\ B(b) \to \epsilon \\ C(c) \to \epsilon \\ D(d) \to \epsilon \\ E(e) \to \epsilon \end{array}$$

Step 1: We assume an ordering, say $\{B, D, E, A, S, C\}$.
Step 2: Now we modify the clauses such that if $A_j(\alpha_1, ..., \alpha_m) \to A_k(X_1, ..., X_n)\gamma$, $j \leq k$:

- We begin with the B clauses. As these are all terminal clauses, we continue. The D and E clauses are also terminal.
- The second A clause is well-ordered, whereas the first is not. So we replace the second clause with $A(d, Y) \to E(Y)$. Again, we replace the obtained clause with a new one, and obtain $A(d, e) \to \epsilon$.
- The second A clause is left-recursive, so we replace it by new A clauses $B'(X, YZ) \to C(Z)B'(X, Y)$ and $B'(\epsilon, Z) \to C(Z)$. Furthermore, for the first A clause we add $A(dX, eY) \to B'(X, Y)$.
- The S clause is not well-ordered, so we replace it by two new clauses $S(dYeU) \to C(U)B(Y)$ and $S(dXYeZ) \to B'(X, Z)B(Y)$.

Step 3: We lexicalize all non-lexicalized clauses. In this case, these are $B'(X, YZ) \to C(Z)B'(X, Y)$ and $B'(\epsilon, Z) \to C(Z)$, which we replace by $B'(X, Yc) \to B'(X, Y)$ and $B'(\epsilon, c) \to \epsilon$ respectively.
The grammar now looks as follows:

$$
\begin{aligned}
&S(dYeU) \to C(U)B(Y) \\
&S(dXYeZ) \to B'(X, Z)B(Y) \\
&A(dX, eY) \to B'(X, Y) \\
&A(d, e) \to \epsilon \\
&B'(X, Yc) \to B'(X, Y) \\
&B'(\epsilon, c) \to \epsilon \\
&B(b) \to \epsilon \\
&C(c) \to \epsilon \\
&D(d) \to \epsilon \\
&E(e) \to \epsilon
\end{aligned}
$$

■

13 The equivalence of first-order Displacement Grammar and well-nested simple Range Concatenation Grammar

The idea of the correspondence between $sRCG_{wn}$ and $\mathbf{D}^1$ is to interpret the rewriting clauses as abstract derivation possibilities where the typing of terminals force off exactly those derivations that are possible in the $sRCG_{wn}$: the left hand side of each clause is the conclusion, the right hand side predicates are premisses. Predicate names become types, the separation of arguments in predicates are interpreted as separator locations. Given the structures of the premisses, we need to make it possible to derive exactly the conclusion by rightly assigning types to the terminals involved. So terminal clauses (those with an empty right hand side) become axioms, and since the start clauses are not lexicalized, we only reassign the types of the start symbol in a $sRCG$ to include the removal of insertion points and the concatenation of some premisses.

We compare $\mathbf{D}$ to well-nested $sRCG$, because ill-nested clauses such as $C(XYZUc) \to A(X,Z)B(Y,U)$ are not directly interpretable: we would have as premisses $X+1+Z : A$ and $Y+1+U : B$ and as the conclusion $X+Y+Z+U+c : C$, so we would need to chew off the U, then intercalate $Y+1$ into $X+1+Z$ and concatenate U again. This is not evidently possible because we do not know the type of U. However, a clause such as $C(XYZUc) \to A(X,U)B(Y,Z)$ can be interpreted, by simple assigning $((A \odot B) \odot I)\backslash C$ to c.

Lemma 5. *For every n-$sRCG_{wn}$ R, there is a weakly equivalent $(n-1)$ $\boldsymbol{D}$-grammar D.*

Proof. We show that for every lexicalized $LsRCG_{wn}$ R, there exists a weakly equivalent $\mathbf{D}$-grammar D. Let $R = (N, T, V, P, S)$ be a $sRCG_{wn}$ having the above mentioned properties. We construct $D = (\Sigma, \delta, S')$ as follows:

- $\Sigma = T$
- S' is obtained from the starting clause in R. For the start clause $S(X_1...X_n) \to A_1(X_i, ..., X_j)A_2(X_k, ...X_l)...A_m(X_o, ..., X_p)$, we add to S' the type that denotes the operations needed to convert $X_i + 1 + X_{i+1}...1 + X_j : A_1; ...; X_o + 1 + ... + 1 + X_p : A_m$ into $X_1 + X_2 + ... +$

X_n. This can be intercalating 0's everywhere and concatenating A_1 to A_m or intercalating one of the right hand side predicates into one another since we have a well-nestedness restriction. (e.g. we could have $S(XYZU) \to A(X,U)B(Y,Z)$ such that we add to S' the type $(A \odot (B \odot I))$

- The typing δ is obtained from the clauses in R. Since we already have converted the start clause, we can divide the remaining clauses into terminating clauses (clauses with an empty right hand side) and non-terminating clauses (clauses with a non-empty right hand side).

- For each terminating clause $A(a) \to \epsilon$, we add the pair (a, A) to δ.

- For each non-terminating clause $A_0(\overrightarrow{\alpha_0}) \to A_1(\overrightarrow{\alpha_1}))...A_n(\overrightarrow{\alpha_n})$ we want to interpret the clause as part of a deduction, in which the induction hypothesis tells us that we already have a deduction for each of the premisses of the form $\alpha_i' : A_i$ for $1 \leq i \leq n$ and where we want to deduce $\alpha_0' : A_0$ where α_i' denotes α_i, but with separators interspersed between each dimension. Furthermore, we have one terminal $a \in \overrightarrow{\alpha_0}$ for which we want to derive a typing with which the deduction will succeed. Graphically, this looks as follows:

$$\begin{array}{c} \alpha_1' : A_1 \quad ... \quad \alpha_n' : A_n \quad a : A' \\ \vdots \\ \alpha_0' : A_0 \end{array}$$

and we need to derive the correct typing for A'. It is important to make the following observation at this point:
The rules of a LsRCG$_{wn}$ are such that can only be intercalations of predicate tuples inside other tuples (because of well-nestedness) and concatenations. Furthermore, clauses with tuples are in fact clauses with several continuous substrings. This observation shows us that to make a deduction like the graphically depicted one above, we can do with intercalations, concatenation and the use of separators, and thus that it is possible to give a (unique) typing for a such that exactly this deduction can be made in first-order Displacement Grammar. For α_0', a is either the lefmost symbol of the lefmost argument of α_0' or the rightmost symbol of the rightmost argument, or it is somewhere in between. The type for a then becomes respectively either A_0/B or $B\backslash A_0$ or $B \downarrow_k A$ or $(B \downarrow_k A)/J$ or $J\backslash(B \downarrow_k A)$ (these last two cases are needed when a intercalates while keeping the dimension intact, thus leaving a separator in the place of separator) where B is the type for

α_0' with a removed, say α_0''. Then, we need to derive the B type by reverse engineering α_0''. Here we can say that it is built up by either the concatenation of two parts, or the intercalation of two parts, resulting in $B \coloneqq (C \bullet D)|C \odot_k D|C \odot_k (D \bullet J)|C \odot_k (J \bullet D)$, the last two choices again, needed when a part intercalates while keeping the dimension intact. We repeat these steps to finally find a typing for a. We must however note that $\odot_k$ is not by definition associative in our definition of Displacement Calculus (consider for example $(A \odot_1 B) \odot_1 C$ vs. $A \odot_1 (B \odot_1 C)$ with $a+1+d+1+e : A$ and $b : B, c : C$), so that there can be a *choice* of typings. For example, to go from $X+1+W : B, Y+1+V : C, Z+1+U : D$ to $X+Y+Z+1+U+V+W : A$, we could have either $A \coloneqq B \odot_1 (C \odot_1 D)$ or $A \coloneqq (B \odot_1 C) \odot_1 D$, strictly giving two different derivation trees but the same string yield. It is for this reason that we cannot compare *strong*, but can compare *weak* generative capacity.

Now we need to show that $L(R) = L(D)$.

($\Rightarrow$) Assume $w = w_1...w_n \in L(R)$. Then there exists a rewriting sequence $S(\langle 0,n \rangle) \Rightarrow^* \epsilon$. We need to show that there exists a natural deduction proof for w with the grammatical assignments of D. By construction, each w_i where $1 \leq i \leq n$ is in the lexicon, and has a typing that directly corresponds to the predicate name of some terminal clause in R for w_i. By construction, we also know that we can give a typing to each terminal symbol that is present in a clause in R such that we mimic the derivation structure of this clause so that if we have the premisses, we can derive it's left-hand side predicate. Since we have all type assignments that correspond to each terminal clause, and every clause instantiated in the rewriting sequence must eventually lead to a terminal clause, and we have type assignments such that every clause can be mimicked, we may conclude that there exists a natural deduction proof for w in D, and thus that $w \in L(D)$.

($\Leftarrow$) Assume $w = w_1...w_n \in L(D)$. Then there exists a natural deduction proof for w in D. Since all type assignments in D make it possible to derive exactly the yield of each possible instantiated clause, we may conclude that there exists a rewriting sequence $S(\langle 0,n \rangle) \Rightarrow^* \epsilon$ for R and thus that $w \in L(R)$. We conclude that $L(R) = L(D)$ and end our proof. ■

Example. Consider the $sRCG_{wn}$ $R = (\{S,A\},\{a,b\},\{X,Y\},P,S)$ with P as follows:

$$S(XY) \to A(X,Y)$$
$$A(aX,aY) \to A(X,Y)$$
$$A(bX,bY) \to A(X,Y)$$
$$A(a,a) \to \epsilon$$
$$A(b,b) \to \epsilon$$

We have that $L(R) = \{ww|w \in \{a,b\}^+\}$, i.e. the single copy language. We now show how to construct an equivalent $1D^1$ grammar. First, we convert R to a one-terminal $sRCG$, and obtain the following clauses:

$$S(XY) \to A(X,Y)$$
$$A(aX,TY) \to A(X,Y)A'(T)$$
$$A(bX,TY) \to A(X,Y)B'(T)$$
$$A(a,T) \to A'(T)$$
$$A(b,T) \to B'(T)$$
$$A'(a) \to \epsilon$$
$$B'(b) \to \epsilon$$

We construct a D-grammar $G = (T, \delta, S)$ as follows:

- The start symbol is defined as $S := A \odot_1 I$, since we remove one dimension by reversing the start clause.
- We assign new types for each of the six remaining clauses:
  1. For the terminal clauses, we add to δ (a, A') and (b, B').
  2. $A(aX,TY) \to A(X,Y)A'(T)$ denotes the following abstract derivation:

     $$\dfrac{X+1+Y : A \quad T : A' \quad a : A_0}{a+X+1+T+Y : A}$$

     We see that to obtain the conclusion, we have to intercalate $1+T$ and concatenate the result to a to the left. So the typing for a becomes $A_0 := A/B$, where B is the type of $X+1+T+Y$. This is the result of intercalation of T together with a concatenated separator, hence $B := A \odot_1 (J \bullet A')$. So we add a type assignment $(a, A/(A \odot_1 (J \bullet A')))$ to δ.
  3. For $A(bX,TY) \to A(X,Y)B'(T)$, we add the type assignment $(b, A/(A \odot_1 (J \bullet B')))$ to δ.
  4. $A(a,T) \to A'(T)$ denotes the following derivation:

$$\begin{array}{c} T : A' \quad a : A_1 \\ \vdots \\ a+1+T : A \end{array}$$

The type assignment for a becomes $A_1 := (A/A')/J$ (note here that we could as well have given $A_1 := A/(J \bullet A')$), and we add to δ $(a, (A/A')/J)$.

5. For $A(b,T) \to B'(T)$, we add the type assignment $(b, (A/B')/J$.

Our final grammar G looks as follows:

$$\begin{array}{l} S' = A \odot_1 I \\ a : A'; (A/A')/J; A/(A \odot_1 (J \bullet A')) \\ b : B'; (A/B')/J; A/(A \odot_1 (J \bullet B')) \end{array}$$

An example derivation of $abab$ looks as follows:

$$\dfrac{0 : I \quad \dfrac{a : A/(A \odot_1 (J \bullet A')) \quad \dfrac{\dfrac{\dfrac{b : (A/B')/J \quad 1 : J}{b+1 : A/B'} E/ \quad b : B'}{b+1+b : A} E\backslash \quad \dfrac{1 : J \quad a : A'}{1+a : J \bullet A'} I\bullet}{b+1+a+b : A \odot_1 (J \bullet A')S} I\odot_1}{a+b+1+a+b : A} E/}{a+b+a+b : A \odot_1 I} I\odot_1$$

■

Before we move to the proof of the converse statement, we define the set of all possible decompositions of types, i.e. for $A\backslash(B\backslash C)$ the set of all possible decompositions is $\{R^{A\backslash(B\ bsC)}, R^{B\ bsC}, R^{C}\}$.

Definition 25. *Let Tp be a finite set of $\boldsymbol{D}^1$ types. We inductively define the set $P(Tp)$:*

1. *Every $A \in Tp$ is in $P(Tp)$,*
2. *If some type $A/B \in P(Tp)$, then $B \in P(Tp)$,*
3. *If some type $B\backslash A \in P(Tp)$, then $B \in P(Tp)$,*
4. *If some type $A \bullet B \in P(Tp)$, then $A, B \in P(Tp)$,*
5. *If some type $A \uparrow_k B \in P(Tp)$, then $B \in P(Tp)$,*
6. *If some type $B \downarrow_k A \in P(Tp)$, then $B \in P(Tp)$,*
7. *If some type $A \odot_k B \in P(Tp)$, then $A, B \in P(Tp)$.*

Lemma 6. *For every n $\boldsymbol{D}^1$-grammar G, there is a strongly equivalent $(n{+}1)$-sRCG$_{wn}$ R.*

Proof. Let $G = (\Sigma, \delta, S)$ be an n **D**-grammar with $ord(D) = 1$. We construct $R = (N, T, V, P, S')$ as follows:

- $N = P(Tp)$ where Tp is the image of δ (for reading comfort, we superscript predicate names on R).
- $T = \Sigma$
- V is generated by the algorithm that produces P.
- P is generated by the following algorithm:

  1. Initialization: $P = \{A(w) \to \epsilon | t \in \Sigma, A \in \delta(w)\}$
  2. Recursion: Until each type is fully decomposed (that is, for 2 to $len(G)$), iteratively add for each clause $c \in P$ new clauses that simulate derivations in D, that is to say, P_{new} is constructed as in Figure 5:

Note that this conversion has a runtime complexity of $O(|\Sigma| \cdot len(G)) = O(|\Sigma|)$ and thus is linear in time. Now we need to show that $L_T(G) = L_T(R)$. ($\Rightarrow$) Let $w = w_1, ..., w_n \in L_T(G)$. By definition, there exists a derivation tree $D = (V, l, E, r)$of G for w. We need to show that there exists a derivation tree of R for w. We construct a derivation tree $D' = (V', l', E', r')$ of R as follows:

$$
\begin{array}{l}
P \cup \\
\{R^B(Y X_1, ..., X_n) \rightarrow R^A(Y) R^{A \backslash B}(X_1, ..., X_n) | R^{A \backslash B}(\alpha_1, ..., \alpha_n) \rightarrow ... \in P\} \cup \\
\{R^B(X_1, ..., X_n Y) \rightarrow R^{B/A}(X_1, ..., X_n) R^A(Y) | R^{B/A}(\alpha_1, ..., \alpha_n) \rightarrow ... \in P\} \cup \\
\{R^{A \bullet B}(X_1, ..., X_n Y_1, ..., Y_m) \rightarrow R^A(X_1, ..., X_n) R^B(Y_1, ..., Y_n) | ... \rightarrow ... R^{A \bullet B}(Z_1 ... Z_k) ... \in P, sort(A) = n-1, sort(B) = \\
m-1, k = sort(A) + sort(B) + 1\} \cup \\
\{R^B(X_1, ..., X_k Y_1, ..., Y_m X_{k+1}, ..., X_n) \rightarrow R^A(X_1, ..., X_n) R^{A \downarrow_k B}(Y_1, ..., Y_m) | R^{A \downarrow_k B}(\alpha_1, ..., \alpha_m) \rightarrow ... \in P\} \cup \\
\{R^B(X_1, ..., X_k Y_1, ..., Y_m X_{k+1}, ..., X_n) \rightarrow R^{B \uparrow_k A}(Y_1, ..., Y_m) R^A(X_1, ..., X_n) | R^{B \uparrow_k A}(\alpha_1, ..., \alpha_m) \rightarrow ... \in P\} \cup \\
\{R^{A \odot_k B}(X_1, ..., X_k Y_1, ..., Y_m X_{k+1}, ..., X_n) \rightarrow R^A(X_1, ..., X_n) R^B(Y_1, ..., Y_m) | ... \rightarrow ... R^{A \odot_k B}(Z_1, ..., Z_k) ... \in P, n > k\} \cup \\
\{R^A(\epsilon, X_1, ..., X_n) \rightarrow R^{J \backslash A}(X_1, ..., X_n) | R^{J \backslash A}(\alpha_1, ..., \alpha_n) \rightarrow ... \in P\} \cup \\
\{R^A(X_1, ..., X_n, \epsilon) \rightarrow R^{A/J}(X_1, ..., X_n) | R^{A/J}(\alpha_1, ..., \alpha_n) \rightarrow ... \in P\} \cup
\end{array}
$$

Figure 5: The recursive step of the construction in lemma 6

- For each w_i for $1 \leq i \leq n$, we add a leaf v_i to V' with $r\text{-}yield(v_i) = \{\langle i-1, i\rangle\}$ and $l'(v_i) = R^{\delta(w_i)}$.
- For each internal vertex $v_0 \in V$ and it's daughter vertices $v_1, v_2 \in V$ (starting from the first parent vertices of the leaves), we have three cases:

  1. It represents a $[E\ \backslash]$ ($[E\ /]$) rule where $l(v_1) = 1 : J$ ($l(v_2) = 1 : J$). In this case, let $r\text{-}yield(v_2') = \{(\langle i_1, j_1\rangle, ..., \langle i_m, j_m\rangle)\}$. Then we add a leaf v_1' to V' with $r\text{-}yield(v_1') = \{\langle i_1, i_1\rangle\}$ and $l'(v_1') = J$. We add a new internal vertex v_0' to V' with edges to v_1', v_2' and with $r\text{-}yield(v_0') = \{(\langle i_1, i_1\rangle, \langle i_1, j_1\rangle, ..., \langle i_m, j_m\rangle)\}$ and $l'(v_0') = l(v_0)$. For the $[E\ /]$ case, we swap v_1' and v_2'.
  2. It represents a standard $[E\ \backslash]$ ($[E\ /]$) rule. In this case, let $r\text{-}yield(v_1') = \{\langle i, j\rangle\}$ and $r\text{-}yield(v_2') = \{\langle j, k\rangle\}$. We add a new internal vertex v_0' with edges to v_1', v_2' and with $r\text{-}yield(v_0') = \{\langle i, k\rangle\}$ and $l'(v_0) = l(v_0)$. Again, the case of $[E\ /]$ is analog.
  3. It represents a $[I\bullet]$ rule where $l(v_1) = 1 : J$ ($l(v_2) = 1 : J$). In this case, let $r\text{-}yield(v_2') = \langle i, j\rangle$. Then we add a leaf v_1' to V' with $r\text{-}yield(v_1') = \{\langle i, i\rangle\}$ and $l'(v_1') = J$. We add a new internal vertex v_0' to V' with edges to v_1', v_2' and with $r\text{-}yield(v_0') = \{(\langle i, i\rangle, \langle i, j\rangle)\}$ and $l'(v_0') = l(v_0)$. For $E\ /$ case, we swap v_1' and v_2'.
  4. It represents a standard $[I\bullet]$ rule. In this case, let $r\text{-}yield(v_1') = \{\langle i, j\rangle\}$ and $r\text{-}yield(v_2') = \{\langle j, k\rangle\}$. We add a new internal vertex v_0' with edges to v_1', v_2' and with $r\text{-}yield(v_0') = \{\langle i, k\rangle\}$ and $l'(v_0) = l(v_0)$.
  5. It represents a $[E\ \downarrow_k]$ rule where $l(v_1) = 0 : I$ ($l(v_2) = 0 : I$). In this case, let $r\text{-}yield(v_2') = \{(\langle i_1, j_1\rangle, ..., \langle i_m, j_m\rangle)\}$. We add a new internal vertex v_0' to V' with an edge to v_2' and with $r\text{-}yield(v_0') = \{(\langle i_1, j_1\rangle, ..., \langle i_k, j_{k+1}\rangle, ...\langle i_m, j_m\rangle)\}$. The $[E\ \uparrow_k]$ is analog.
  6. It represents a standard $[E\ \downarrow_k]$ rule. In this case, let $r\text{-}yield(v_1') = \{(\langle i_1, j_1\rangle, ..., \langle i_m, j_m\rangle)\}$ and $r\text{-}yield(v_2') = \{(\langle p_1, q_1\rangle, ..., \langle p_l, q_l\rangle)\}$. We add a new internal vertex v_0' to V' with edges to v_1, v_2 and with

$$r\text{-}yield(v_0') = \{(\langle i_1, j_1\rangle, ..., \langle i_k, j_k\rangle, \langle p_1, q_1\rangle, ..., \langle p_l, q_l\rangle, \langle i_{k+1}, j_{k+1}\rangle, ..., \langle i_m, j_m\rangle)\}.$$

Note here that $\langle i_k, j_k\rangle, \langle p_1, q_1$ collapses into $\langle i_k, q_1$ if there is not a separator but a terminal symbol, idem for the right side. The $[E\ \uparrow_k]$ is analog.

7. It represents a $[I\odot_k]$ rule. In this case, let

$$r\text{-}yield(v_1') = \{(\langle i_1, j_1\rangle, ..., \langle i_m, j_m\rangle)\}$$
$$r\text{-}yield(v_2') = \{(\langle p_1, q_1\rangle, ..., \langle p_l, q_l\rangle)\}.$$

We add a new internal vertex v_0' to V' with edges to v_1, v_2 and with

$$r\text{-}yield(v_0') =$$
$$\{(\langle i_1, j_1\rangle, ..., \langle i_k, j_k\rangle, \langle p_1, q_1\rangle, ..., \langle p_l, q_l\rangle, \langle i_{k+1}, j_{k+1}\rangle, ..., \langle i_m, j_m\rangle)\}.$$

Note here that $\langle i_k, j_k\rangle, \langle p_1, q_1$ collapses into $\langle i_k, q_1$ if there is not a separator but a terminal symbol, idem for the right side.

Once we have had all vertices of D, we obtain a derivation tree for R where $r\text{-}yield(r') = \langle 0, n\rangle$ and $l'(r') = S'$, and thus conclude that $L_T(G) \subseteq L_T(R)$. ($\Leftarrow$) Let $w = w_1, ..., w_n \in L_T(R)$. By definition, there exists a derivation tree $D = (V, l, E, r)$ of R for w. We need to show that there exists a derivation tree of G for w. Basically, We construct a derivation tree $D' = (V', l', E', r')$ of G as follows:

- For each leaf $v_i \in V$ with $r\text{-}yield(v_i) = \{\langle i-1, i\rangle\}$, we add a lexical leaf v_i' to V' with $l'(v_i)' = w_i : l(v_i)$,
- For each leaf $v \in V$ with $r\text{-}yield(v) = \{\langle i, i\rangle\}$ for some i, add a leaf v' to V' with $l'(v') = 1 : J$,
- For every internal node $v_0 \in V$ that has only one daughter (thus representing a concatenation of ranges) v_1 we add a leaf v_2' with $l'(v_2') = 0 : I$
- For every internal vertex $v \in V$, let $\rho \in r\text{-}yield(v)$. We add an internal vertex v' to V' with $l'(v') = \rho(w) : l(v)$ and we copy the edges of v to v'. In the case that v (v') only has one daughter, say v_1 (v_1'), we have added a new leaf v_2' with $l'(v_2') = 0 : I$ and so we add a new edge from v' to v_2'.

If we have 'copied' the entire derivation tree D into a new tree D' it is obvious that D' is a derivation tree for G, because all derivation steps in R correspond to deduction rules in Displacement Calculus, with the only difference that $0, 1$ are not explicitly present, but because only intercalations

of 0 are represented as a vertex with a single daughter, and only the concatenation of a separator 1 involves ϵ-arguments, we can safely make this construction. We therefore may conclude that $L_T(R) \subseteq L_T(G)$.

We conclude that $L_T(G) = L_T(R)$ and thus complete the proof. ∎

Example. $RESP_m = \{a_1^i a_2^i b_1^j b_2^j ... a_{2m-1}^i a_{2m}^i b_{2m-1}^j b_{2m}^j\}$ was shown to be in the difference class $m\text{-}MCFL - m\text{-}MCFL_{wn}$ [29]. As an example of the previous lemma, we show that $RESP_2 \in 3\text{-}MCFL_{wn}$ and argue that this is generalizable to $RESP_m \in (m+1)\text{-}MCFL_{wn}$. Consider $RESP_2 = \{a^n b^n c^m d^m e^n f^n g^m h^m\}$. The following $2\text{-}D^1$ grammar describes $RESP_2$:

$S' = (C \odot_2 I) \odot_1 I$ $a : A; (((T/F)/E)/J)/B$
$b : B; ((T \downarrow_1 (A \backslash (T/F)))/E)/J$
$c : ((T \downarrow_1 (((C/H)/J)/G))/D)/J; ((C \downarrow_1 U)/D)/J$
$d : D$
$e : E$
$f : F$
$g : ((U \downarrow_2 C)/H)/J$
$h : H$

Running the algorithm described in lemma 3, we obtain the following clauses:

Initial stage:

$$R^{A}(a) \rightarrow \epsilon$$
$$R^{((((T/F)/E)/J)/B)}(a) \rightarrow \epsilon$$
$$R^{B}(b) \rightarrow \epsilon$$
$$R^{(((T\downarrow_1(A\backslash(T/F)))/E)/J)}(b) \rightarrow \epsilon$$
$$R^{(((T\downarrow_1(((C/H)/J)/G))/D)/J)}(c) \rightarrow \epsilon$$
$$R^{(((C\downarrow_1 U)/D)/J)}(c) \rightarrow \epsilon$$
$$R^{D}(d) \rightarrow \epsilon$$
$$R^{E}(e) \rightarrow \epsilon$$
$$R^{F}(f) \rightarrow \epsilon$$
$$R^{G}(g) \rightarrow \epsilon$$
$$R^{(((U\downarrow_2 C)/H)/J)}(g) \rightarrow \epsilon$$
$$R^{H}(h) \rightarrow \epsilon$$

The next stage recursively eliminates one $/, \backslash, \downarrow_k, \uparrow_k, \bullet, \odot$:

First step:

$$R^{(((T/F)/E)/J)}(X_1Y_1) \to R^{((((T/F)/E)/J)/B)}(X_1)R^B(Y_1)$$
$$R^{((T\downarrow_1(A\backslash(T/F)))/E)}(X_1,\epsilon) \to R^{(((T\downarrow_1(A\backslash(T/F)))/E)/J)}(X_1)$$
$$R^{((T\downarrow_1(((C/H)/J)/G))/D)}(X_1,\epsilon) \to R^{(((T\downarrow_1(((C/H)/J)/G))/D)/J)}(X_1)$$
$$R^{((C\downarrow_1U)/D)}(X_1,\epsilon) \to R^{(((C\downarrow_1U)/D)/J)}(X_1)$$
$$R^{((U\downarrow_2C)/H)}(X_1,\epsilon) \to R^{(((U\downarrow_2C)/H)/J)}(X_1)$$

Second through last step:

$$R^{((T/F)/E)}(X_1,\epsilon) \to R^{(((T/F)/E)/J)}(X_1)$$
$$R^{(T\downarrow_1(A\backslash(T/F)))}(X_1,X_2Y_1) \to R^{((T\downarrow_1(A\backslash(T/F)))/E)}(X_1,X_2)R^E(Y_1)$$
$$R^{(T\downarrow_1(((C/H)/J)/G))}(X_1,X_2Y_1) \to R^{((T\downarrow_1(((C/H)/J)/G))/D)}(X_1,X_2)R^D(Y_1)$$
$$R^{(C\downarrow_1U)}(X_1,X_2Y_1) \to R^{((C\downarrow_1U)/D)}(X_1,X_2)R^D(Y_1)$$
$$R^{(U\downarrow_2C)}(X_1,X_2Y_1) \to R^{((U\downarrow_2C)/H)}(X_1,X_2)R^H(Y_1)$$
$$R^{(T/F)}(X_1,X_2Y_1) \to R^{((T/F)/E)}(X_1,X_2)R^E(Y_1)$$
$$R^{(A\backslash(T/F))}(Y_1X_1,X_2Y_2) \to R^T(Y_1,Y_2)R^{(T\downarrow_1(A\backslash(T/F)))}(X_1,X_2)$$
$$R^{(((C/H)/J)/G)}(Y_1X_1,X_2Y_2) \to R^T(Y_1,Y_2)R^{(T\downarrow_1(((C/H)/J)/G))}(X_1,X_2)$$
$$R^U(Y_1X_1,X_2Y_2) \to R^C(Y_1,Y_2,Y_3)R^{(C\downarrow_1U)}(X_1,X_2)$$
$$R^C(Y_1,Y_2X_1,X_2Y_3) \to R^U(Y_1,Y_2,Y_3)R^{(U\downarrow_2C)}(X_1,X_2)$$
$$R^T(X_1,X_2Y_1) \to R^{(T/F)}(X_1,X_2)R^F(Y_1)$$
$$R^{(T/F)}(X_1Y_1,X_2) \to R^A(Y_1)R^{(A\backslash(T/F))}(X_1,X_2)$$
$$R^{((C/H)/J)}(X_1,X_2Y_1) \to R^{(((C/H)/J)/G)}(X_1,X_2)R^G(Y_1)$$
$$R^T(X_1,X_2Y_1) \to R^{(T/F)}(X_1,X_2)R^F(Y_1)$$
$$R^{(C/H)}(X_1,X_2,\epsilon) \to R^{((C/H)/J)}(X_1,X_2)$$
$$R^C(X_1,X_2,X_3Y_1) \to R^{(C/H)}(X_1,X_2,X_3)R^H(Y_1)$$
$$R^S(X_1X_2X_3) \to R^C(X_1,X_2,X_3)$$

We can simplify the result by lexicalizing the clauses. We then obtain the following grammar:

$$R^S(XYZ) \to R^C(X,Y,Z)$$
$$R^C(Xc,dYg,hZ) \to R^C(X,Y,Z)$$
$$R^C(Xc,dYg,h) \to R^A(X,Y)$$
$$R^A(aXb,eYf) \to R^A(X,Y)$$
$$R^A(ab,ef) \to \epsilon$$

In this somewhat simpler description of the grammar, it is quite easy to see that we can make a generalization of the grammar, even if we want to have that $i,j \geq 0$. For $RESP_m = \{a_1^i a_2^i b_1^j b_2^j ... a_{2m-1}^i a_{2m}^i b_{2m-1}^j b_{2m}^j | i,j \geq 0\}$, a $(m+1)$-$sRCG_{wn}$ can be depicted as follows:

$S(X_1...X_{m+1}) \to C(X_1, ..., X_{m+1})$
$C(X_1 b_1, b_2 X_2 b_3, ..., b_{m-2} X_m b_{m-1}, b_m X_{m+1}) \to C(X_1, ..., X_m, X_{m+1})$
$C(X_1, ..., X_m, \epsilon) \to A(X_1, ..., X_m)$
$A(a_1 X_1 a_2, a_3 X_2 a_4, ..., a_{2m-1} X_m a_{2m}) \to \epsilon$
$A(\epsilon^1, ..., \epsilon^m) \to \epsilon$

■

Theorem 2. $D^1 L = sRCL_{wn}$

Proof. This follows directly from lemma 2 and lemma 3. ■

Corollary 1. $0D^{(1)}L = L = CFL$

Proof. 1-$sRCG_{wn}$ is trivially equivalent to CFG, and CFL is equivalent to L by [26]. ■

Corollary 2. $1D^1 L = TAL$

14 Conclusion and further directions

We showed some expressivity results for Displacement Grammar which led to the idea of the equivalence between D^1 and $sRCG_{wn}$. To accomplish this result, we showed a lexicalized normal form for simple Range Concatenation Grammar. As we argued in the introduction that well-nested MCFG/sRCG may be the most appropriate way to describe the class of Mildly-Context Sensitive Languages, we claim here that the first-order fragment of general Displacement Calculus has the same status within the Categorial Grammar view, also because it admits the same kind of control property we find in MCFG. Putting this in the light of the result of [28], it could very well be that the difference class $MCFG - MCFG_{wn}$ corresponds to the difference class $\mathbf{D}$–$\mathbf{D}^1$, where $\mathbf{D}$ denotes normal general Displacement Calculus. A first idea is to copy the result of Pentus by applying it to general Displacement Calculus, but it appears not to be possible since Displacement Calculus inherits incompleteness from the Lambek Calculus with respect to the unit ([25], [32]).

Another idea within the Categorial approach concerns the upper bound on generative capacity of the LG calculus, which is as yet still unknown. Several authors have pointed to grammar formalisms that may be good candidates, including *Global Index Grammar* (GIG), introduced by [4], *MCTAG*, and *MCFG*. We think that GIG exceeds the expressive power of LG, since it exceeds MCTAG and MCFG and none of the languages of which we know that they can be generated in *LG*, are not included in MCTAG and MCFG. Furthermore, we think that a comparison between $\mathbf{D}$ and *LG* may be appropriate. As [17] shows, the control that *LG* exhibits for counting dependencies, for example, can be done through transition schemes in both residual families of connectives (if we regard $a_1\backslash a_2$ as a transition from a state a_1 to a_2). Normally, when we give type assignments involving the dual residual connectives $\oslash, \obslash$ (see [21], [17], [24] for further explanation and examples), some parts are able to nest into an arbitrary leaf in the derivation structure through the interaction principles, thus we need to exhibit the control described to direct types to the leaves they should nest into, and in this way structuring the language described. There is an alternative definition of Displacement Calculus where $\uparrow, \downarrow$ can be used to let strings intercalate anywhere in a structure instead of just at a separator position that was pointed out beforehand. The only difference here is that in *LG* the legitimate orders of words is forced off by directing types, whereas in the alternative Displacement calculus, these orders are forced off by pointing

out the strings that may be intercalated.

References

[1] Emmon Bach. Discontinuous constituents in generalized categorial grammars. In *Victoria Burke and James Pustejovsky (eds.) Proceedings of the llth Annual Meeting of the Northeastern Linguistics Society.*, pages l–l2, 1981.

[2] Arno Bastenhof. Tableaux for the Lambek-Grishin calculus. *Computing Research Repository*, abs/1009.3, 2010.

[3] Pierre Boullier. Proposal for a Natural Language Processing Syntactic Backbone. Rapport de recherche RR-3342, INRIA, 1998.

[4] M. Castaño and José M. Castaño. Global index languages, 2004.

[5] Noam Chomsky. Formal properties of grammars. In R. D. Luce, R. Bush, and E. Galanter, editors, *Handbook of Mathematical Psychology*, volume 2, pages 323–418. Wiley, New York, 1963.

[6] J Cockett. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114:133–173, 1997.

[7] Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical foundations for a compositional distributional model of meaning. *Computing Research Repository*, abs/1003.4, 2010.

[8] O. Valentín G. Morrill. On calculus of displacement. In *TAG+10, Proceedings of TAG+Related Formalisms 2010*. University of Yale., 2010.

[9] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, pages 42–52, 1965.

[10] Annius V. Groenink. *Surface Without Structure. Word Order and Tractability Issues in Natural Language Analysis.* PhD thesis, Utrecht University, Utrecht, The Netherlands, 1997.

[11] A. K. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. In P. Sells, S. M. Shieber, and T. Wasow, editors, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, Cambridge, MA, 1991.

[12] Aravind K Joshi. *How much context-sensitivity is required to provide reasonable structural descriptions: Tree adjoining grammars*, pages 206–250. Cambridge University Press, 1985.

[13] L. Kallmeyer. *Parsing Beyond Context-Free Grammars*. Cognitive Technologies. Springer, 2010.

[14] Makoto Kanazawa and Sylvain Salvati. The copying power of well-nested multiple context-free grammars. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martn-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 344–355. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13089-2.29.

[15] Joachim Lambek. The mathematics of sentence structure. *Americal Mathematical Monthly*, 65:154–170, 1958.

[16] Joachim Lambek. Type grammar revisited. In *Logical Aspects of Computational Linguistics*, pages 1–27, 1997.

[17] Matthijs Melissen. The generative capacity of the lambek-grishin calculus: A new lower bound. In Philippe de Groote, Markus Egg, and Laura Kallmeyer, editors, *Proceedings of Formal Grammar 2009*, volume 5591 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2011.

[18] Paola Monachesi, Gerald Penn, Giorgio Satta, Makoto Kanazawa, and Sylvain Salvati. 1 generating control languages with abstract categorial grammars, 2007.

[19] Michael Moortgat. Categorial type logics. In *Handbook of Logic and Language*, pages 93–177. Elsevier, 1997.

[20] Michael Moortgat. Symmetric categorial grammar. *Journal of Philosophical Logic*, 38:681–710, 2009. 10.1007/s10992-009-9118-6.

[21] Michael Moortgat. Symmetric categorial grammar: residuation and galois connections. *Computing Research Repository*, abs/1008.0, 2010.

[22] R. Moot. Lambek grammars, tree adjoining grammars and hyperedge replacement grammars. In *Proceedings of the TAG+ Conference.*, HAL - CCSD, 2008.

[23] Richard Moot. *Proof Nets for Linguistic Analysis*. PhD thesis, Utrecht university, 2002.

[24] Richard Moot. Proof nets for display logic. *Computing Research Repository*, abs/0711.2, 2007.

[25] Glyn Morrill, Oriol Valentín, and Mario Fadda. The displacement calculus. *Journal of Logic, Language and Information*, 20:1–48, 2011. 10.1007/s10849-010-9129-2.

[26] M. Pentus. Lambek grammars are context free. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 429–433, Los Alamitos, California, 1993. IEEE Computer Society Press.

[27] Sylvain Salvati. Encoding second order string acgs with deterministic tree walking transducers. In *Winter Simulation Conference*, 2006.

[28] Sylvain Salvati. MIX is a 2-MCFL and the word problem in $\mathbf{Z}^2$ is solved by a third-order collapsible pushdown automaton. Research report, February 2011.

[29] Hiroyuki Seki and Yuki Kato. On the generative power of multiple context-free grammars and macro grammars. *IEICE - Trans. Inf. Syst.*, E91-D:209–221, February 2008.

[30] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theor. Comput. Sci.*, 88:191–229, October 1991.

[31] Stuart M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985. 10.1007/BF00630917.

[32] O. Valentín. *Discontinuous Lambek calculus.* PhD thesis, Universitat Autnoma de Barcelona, Barcelona, Spain, forthcoming.

A Copy Language and Crossing Dependencies

$\{ww \mid w \in \{a,b\}^+\}$	LG	P^+
$lex(a)$	A	a
$lex(a)$	$(T \oslash (A\backslash S)) \obslash ((T/i) \otimes i)$	$((a^r \cdot s) + t^d) + (t \cdot i^l \cdot i)$
$lex(a)$	$((T/i) \otimes i)\backslash((T \oslash (A\backslash T)) \obslash ((T/i) \otimes i))$	$(i^r \cdot i \cdot t^r) \cdot (((a^r \cdot t) + t^d) + (t \cdot i^l \cdot i))$
$lex(b)$	B	b
$lex(b)$	$(T \oslash (B\backslash S)) \obslash ((T/i) \otimes i)$	$((b^r \cdot s) + t^d) + (t \cdot i^l \cdot i)$
$lex(b)$	$((T/i) \otimes i)\backslash((T \oslash (B\backslash T)) \obslash ((T/i) \otimes i))$	$(i^r \cdot i \cdot t^r) \cdot (((b^r \cdot t) + t^d) + (t \cdot i^l \cdot i))$

$\{a^n b^m c^n d^m\}, n > 0$	LG	P^+
$lex(a)$	A	a
$lex(b)$	B	b
$lex(c)$	$(T \oslash (A\backslash S)) \obslash ((T/i) \otimes i)$	$((a^r \cdot s) + t^d) + (t \cdot i^l \cdot i)$
$lex(c)$	$((T/i) \otimes i)\backslash((T \oslash (A\backslash T)) \obslash ((T/i) \otimes i))$	$(i^r \cdot i \cdot t^r) \cdot (((a^r \cdot t) + t^d) + (t \cdot i^l \cdot i))$
$lex(c)$	$(U \oslash (A\backslash S)) \obslash ((U/i) \otimes i)$	$((a^r \cdot s) + u^d) + (u \cdot i^l \cdot i)$
$lex(c)$	$((T/i) \otimes i)\backslash((U \oslash (A\backslash T)) \obslash ((U/i) \otimes i))$	$(i^r \cdot i \cdot t^r) \cdot (((a^r \cdot t) + u^d) + (u \cdot i^l \cdot i))$
$lex(d)$	$((U/i) \otimes i)\backslash((U \oslash (B\backslash U)) \obslash ((U/i) \otimes i))$	$(i^r \cdot i \cdot u^r) \cdot (((b^r \cdot u) + u^d) + (u \cdot i^l \cdot i))$