Mildly Context-Sensitive Dependency Languages

Marco Kuhlmann
Programming Systems Lab
Saarland University
Saarbrücken, Germany
kuhlmann@ps.uni-sb.de

Mathias Möhl
Programming Systems Lab
Saarland University
Saarbrücken, Germany
mmohl@ps.uni-sb.de

Abstract

Dependency-based representations of natural language syntax require a fine balance between structural flexibility and computational complexity. In previous work, several constraints have been proposed to identify classes of dependency structures that are well-balanced in this sense; the best-known but also most restrictive of these is projectivity. Most constraints are formulated on fully specified structures, which makes them hard to integrate into models where structures are composed from lexical information. In this paper, we show how two empirically relevant relaxations of projectivity can be lexicalized, and how combining the resulting lexicons with a regular means of syntactic composition gives rise to a hierarchy of mildly context-sensitive dependency languages.

1 Introduction

Syntactic representations based on word-to-word dependencies have a long tradition in descriptive linguistics. Lately, they have also been used in many computational tasks, such as relation extraction (Culotta and Sorensen, 2004), parsing (McDonald et al., 2005), and machine translation (Quirk et al., 2005).

Especially in recent work on parsing, there is a particular interest in *non-projective* dependency structures, in which a word and its dependents may be spread out over a discontinuous region of the sentence. These structures naturally arise in the syntactic analysis of languages with flexible word order, such as Czech (Veselá et al., 2004). Unfortunately, most formal results on non-projectivity are discouraging: While grammar-driven dependency parsers that are restricted to projective structures can be as efficient as parsers for lexicalized context-free grammar (Eisner and Satta, 1999), parsing is prohibitively expensive when unrestricted forms of non-projectivity are permitted (Neuhaus and Bröker, 1997). Data-driven dependency parsing with non-projective structures is quadratic when all attachment decisions are assumed to be independent of one another (McDonald et al., 2005), but becomes intractable when this assumption is abandoned (McDonald and Pereira, 2006).

In search of a balance between structural flexibility and computational complexity, several authors have proposed constraints to identify classes of non-projective dependency structures that are computationally well-behaved (Bodirsky et al., 2005; Nivre, 2006). In this paper, we focus on two of these proposals: the *gap-degree restriction*, which puts a bound on the number of discontinuities in the region of a sentence covered by a word and its dependents, and the *well-nestedness condition*, which constrains the arrangement of dependency subtrees. Both constraints have been shown to be in very good fit with data from dependency treebanks (Kuhlmann and Nivre, 2006). However, like all other such proposals, they are formulated on fully specified structures, which makes it hard to integrate them into a generative model, where dependency structures are composed from elementary units of lexicalized information. Consequently, little is known about the generative capacity and computational complexity of *languages* over restricted non-projective dependency structures.

Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, pages 160–167,
Prague, Czech Republic, June 2007. ©2007 Association for Computational Linguistics

Contents of the paper In this paper, we show how the gap-degree restriction and the well-nestedness condition can be captured in dependency lexicons, and how combining such lexicons with a regular means of syntactic composition gives rise to an infinite hierarchy of mildly context-sensitive languages.

The technical key to these results is a procedure to encode arbitrary, even non-projective dependency structures into trees (terms) over a signature of local order-annotations. The constructors of these trees can be read as lexical entries, and both the gap-degree restriction and the well-nestedness condition can be couched as syntactic properties of these entries. Sets of gap-restricted dependency structures can be described using regular tree grammars. This gives rise to a notion of *regular dependency languages*, and allows us to establish a formal relation between the structural constraints and mildly context-sensitive grammar formalisms (Joshi, 1985): We show that regular dependency languages correspond to the sets of derivations of lexicalized Linear Context-Free Rewriting Systems (LCFRS) (Vijay-Shanker et al., 1987), and that the gap-degree measure is the structural correspondent of the concept of 'fan-out' in this formalism (Satta, 1992). We also show that adding the well-nestedness condition corresponds to the restriction of LCFRS to Coupled Context-Free Grammars (Hotz and Pitsch, 1996), and that regular sets of well-nested structures with a gap-degree of at most 1 are exactly the class of sets of derivations of Lexicalized Tree Adjoining Grammar (LTAG). This result generalizes previous work on the relation between LTAG and dependency representations (Rambow and Joshi, 1997; Bodirsky et al., 2005).

Structure of the paper The remainder of this paper is structured as follows. Section 2 contains some basic notions related to trees and dependency structures. In Section 3 we present the encoding of dependency structures as order-annotated trees, and show how this encoding allows us to give a lexicalized reformulation of both the gap-degree restriction and the well-nestedness condition. Section 4 introduces the notion of regular dependency languages. In Section 5 we show how different combinations of restrictions on non-projectivity in these languages correspond to different mildly context-sensitive grammar formalisms. Section 6 concludes the paper.

2 Preliminaries

Throughout the paper, we write $[n]$ for the set of all positive natural numbers up to and including n. The set of all strings over a set A is denoted by A^*, the empty string is denoted by ε, and the concatenation of two strings x and y is denoted either by xy, or, where this is ambiguous, by $x \cdot y$.

2.1 Trees

In this paper, we regard trees as terms. We expect the reader to be familiar with the basic concepts related to this framework, and only introduce our particular notation. Let Σ be a set of labels. The set of (finite, unranked) *trees* over Σ is defined recursively by the equation $T_\Sigma := \{\, \sigma(x) \mid \sigma \in \Sigma, x \in T_\Sigma^* \,\}$. The set of *nodes* of a tree $t \in T_\Sigma$ is defined as

$$N(\sigma(t_1 \cdots t_n)) := \{\varepsilon\} \cup \{\, iu \mid i \in [n], u \in N(t_i) \,\}\,.$$

For two nodes $u, v \in N(t)$, we say that u *governs* v, and write $u \trianglelefteq v$, if v can be written as $v = ux$, for some sequence $x \in \mathbb{N}^*$. Note that the governance relation is both reflexive and transitive. The converse of government is called *dependency*, so $u \trianglelefteq v$ can also be read as 'v depends on u'. The *yield* of a node $u \in N(t)$, $\lfloor u \rfloor$, is the set of all dependents of u in t: $\lfloor u \rfloor := \{\, v \in N(t) \mid u \trianglelefteq v \,\}$. We also use the notations $t(u)$ for the label at the node u of t, and t/u for the subtree of t rooted at u. A *tree language* over Σ is a subset of T_Σ.

2.2 Dependency structures

For the purposes of this paper, a *dependency structure* over Σ is a pair $d = (t, x)$, where $t \in T_\Sigma$ is a tree, and x is a list of the nodes in t. We write D_Σ to refer to the set of all dependency structures over Σ. Independently of the governance relation in d, the list x defines a total order on the nodes in t; we write $u \preceq v$ to denote that u *precedes* v in this order. Note that, like governance, the precedence relation is both reflexive and transitive. A *dependency language* over Σ is a subset of D_Σ.

EXAMPLE. The left half of Figure 1 shows how we visualize dependency structures: circles represent nodes, arrows represent the relation of (immediate) governance, the left-to-right order of the nodes represents their order in the precedence relation, and the dotted lines indicate the labelling. □

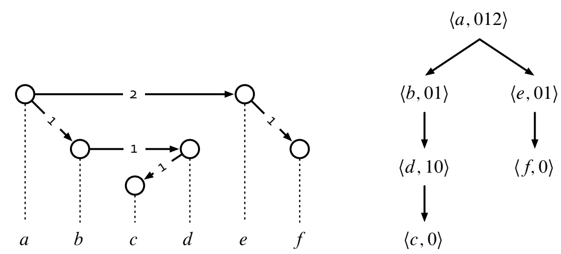


Figure 1: A projective dependency structure

3 Lexicalizing the precedence relation

In this section, we show how the precedence relation of dependency structures can be encoded as, and decoded from, a collection of node-specific order annotations. Under the assumption that the nodes of a dependency structure correspond to lexemic units, this result demonstrates how word-order information can be captured in a dependency lexicon.

3.1 Projective structures

Lexicalizing the precedence relation of a dependency structure is particularly easy if the structure under consideration meets the condition of *projectivity*. A dependency structure is projective, if each of its yields forms an interval with respect to the precedence order (Kuhlmann and Nivre, 2006).

In a projective structure, the interval that corresponds to a yield $\lfloor u \rfloor$ decomposes into the singleton interval $[u, u]$, and the collection of the intervals that correspond to the yields of the immediate dependents of u. To reconstruct the global precedence relation, it suffices to annotate each node u with the relative precedences among the constituent parts of its yield. We represent this 'local' order as a string over the alphabet $\mathbb{N}_0$, where the symbol 0 represents the singleton interval $[u, u]$, and a symbol $i \neq 0$ represents the interval that corresponds to the yield of the ith direct dependent of u. An *order-annotated tree* is a tree labelled with pairs $\langle \sigma, \omega \rangle$, where σ is the label proper, and ω is a local order annotation. In what follows, we will use the functional notations $\sigma(u)$ and $\omega(u)$ to refer to the label and order annotation of u, respectively.

EXAMPLE. Figure 1 shows a projective dependency structure together with its representation as an order-annotated tree. □

We now present procedures for encoding projective dependency structures into order-annotated trees, and for reversing this encoding.

Encoding The representation of a projective dependency structure (t, x) as an order-annotated tree can be computed in a single left-to-right sweep over x. Starting with a copy of the tree t in which every node is annotated with the empty string, for each new node u in x, we update the order annotation of u through the assignment $\omega(u) := \omega(u) \cdot 0$. If $u = vi$ for some $i \in \mathbb{N}$ (that is, if u is an inner node), we also update the order annotation of the parent v of u through the assignment $\omega(v) := \omega(v) \cdot i$.

Decoding To decode an order-annotated tree t, we first linearize the nodes of t into a sequence x, and then remove all order annotations. Linearization proceeds in a way that is very close to a pre-order traversal of the tree, except that the relative position of the root node of a subtree is explicitly specified in the order annotation. Specifically, to linearize an order-annotated tree, we look into the local order $\omega(u)$ annotated at the root node of the tree, and concatenate the linearizations of its constituent parts. A symbol i in $\omega(u)$ represents either the singleton interval $[u, u]$ $(i = 0)$, or the interval corresponding to some direct dependent ui of u $(i \neq 0)$, in which case we proceed recursively. Formally, the linearization of u is captured by the following three equations:

$$
\begin{aligned}
\mathrm{lin}(u) &= \mathrm{lin}'(u, \omega(u)) \\
\mathrm{lin}'(u, i_1 \cdots i_n) &= \mathrm{lin}''(u, i_1) \cdots \mathrm{lin}''(u, i_n) \\
\mathrm{lin}''(u, i) &= \textbf{if } i = 0 \textbf{ then } u \textbf{ else } \mathrm{lin}(ui)
\end{aligned}
$$

Both encoding and decoding can be done in time linear in the number of nodes of the dependency structure or order-annotated tree.

3.2 Non-projective structures

It is straightforward to see that our representation of dependency structures is insufficient if the structures under consideration are non-projective. To witness, consider the structure shown in Figure 2. Encoding this structure using the procedure presented above yields the same order-annotated tree as the one shown in Figure 1, which demonstrates that the encoding is not reversible.

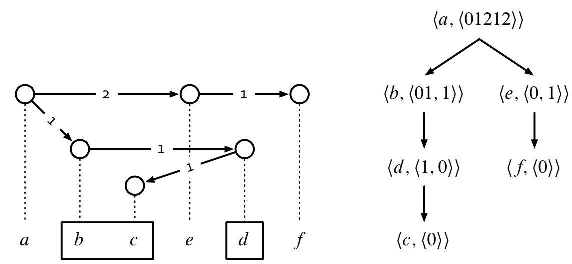


Figure 2: A non-projective dependency structure

Blocks In a non-projective dependency structure, the yield of a node may be spread out over more than one interval; we will refer to these intervals as *blocks*. Two nodes v, w belong to the same block of a node u, if all nodes between v and w are governed by u.

Example. Consider the nodes b, c, d in the structures depicted in Figures 1 and 2. In Figure 1, these nodes belong to the same block of b. In Figure 2, the three nodes are spread out over two blocks of b (marked by the boxes): c and d are separated by a node (e) not governed by b. □

Blocks have a recursive structure that is closely related to the recursive structure of yields: the blocks of a node u can be decomposed into the singleton $[u, u]$, and the blocks of the direct dependents of u. Just as a projective dependency structure can be represented by annotating each yield with an order on its constituents, an unrestricted structure can be represented by annotating each block.

Extended order annotations To represent orders on blocks, we extend our annotation scheme as follows. First, instead of a single string, an annotation $\omega(u)$ now is a tuple of strings, where the kth component specifies the order among the constituents of the kth block of u. Second, instead of one, the annotation may now contain multiple occurrences of the same dependent; the kth occurrence of i in $\omega(u)$ represents the kth block of the node ui.

We write $\omega(u)_k$ to refer to the kth component of the order annotation of u. We also use the notation $(i\#k)_u$ to refer to the kth occurrence of i in $\omega(u)$, and omit the subscript when the node u is implicit.

Example. In the annotated tree shown in Figure 2, $\omega(b)_1 = (0\#1)(1\#1)$, and $\omega(b)_2 = (1\#2)$. □

Encoding To encode a dependency structure (t, x) as an extended order-annotated tree, we do a post-order traversal of t as follows. For a given node u, let us represent a constituent of a block of u as a triple $i : [v_l, v_r]$, where i denotes the node that contributes the constituent, and v_l and v_r denote the constituent's leftmost and rightmost elements. At each node u, we have access to the singleton block $0 : [u, u]$, and the constituent blocks of the immediate dependents of u. We say that two blocks $i : [v_l, v_r]$, $j : [w_l, w_r]$ can be *merged*, if the node v_r immediately precedes the node w_l. The result of the merger is a new block $ij : [v_l, w_r]$ that represents the information that the two merged constituents belong to the same block of u. By exhaustive merging, we obtain the constituent structure of all blocks of u. From this structure, we can read off the order annotation $\omega(u)$.

Example. The yield of the node b in Figure 2 decomposes into $0 : [b, b]$, $1 : [c, c]$, and $1 : [d, d]$. Since b and c are adjacent, the first two of these constituents can be merged into a new block $01 : [b, c]$; the third constituent remains unchanged. This gives rise to the order annotation $\langle 01, 1\rangle$ for b. □

When using a global data-structure to keep track of the constituent blocks, the encoding procedure can be implemented to run in time linear in the number of blocks in the dependency structure. In particular, for projective dependency structures, it still runs in time linear in the number of nodes.

Decoding To linearize the kth block of a node u, we look into the kth component of the order annotated at u, and concatenate the linearizations of its constituent parts. Each occurrence $(i\#k)$ in a component of $\omega(u)$ represents either the node u itself ($i = 0$), or the kth block of some direct dependent ui of u ($i \neq 0$), in which case we proceed recursively:

$$\begin{aligned} \mathrm{lin}(u,k) &= \mathrm{lin}'(u, \omega(u)_k) \\ \mathrm{lin}'(u, i_1 \cdots i_n) &= \mathrm{lin}''(u, i_1) \cdots \mathrm{lin}''(u, i_n) \\ \mathrm{lin}''(u, (i\#k)_u) &= \textbf{if } i = 0 \textbf{ then } u \textbf{ else } \mathrm{lin}(ui, k) \end{aligned}$$

The root node of a dependency structure has only one block. Therefore, to linearize a tree t, we only need to linearize the first block of the tree's root node: $\mathrm{lin}(t) = \mathrm{lin}(\varepsilon, 1)$.

Consistent order annotations Every dependency structure over Σ can be encoded as a tree over the set $\Sigma \times \Omega$, where Ω is the set of all order annotations. The converse of this statement does not hold: to be interpretable as a dependency structure, tree structure and order annotation in an order-annotated tree must be *consistent*, in the following sense.

PROPERTY C1: Every annotation $\omega(u)$ in a tree t contains all and only the symbols in the collection $\{0\} \cup \{ i \mid ui \in N(t) \}$, i.e., one symbol for u, and one symbol for every direct dependent of u.

PROPERTY C2: The number of occurrences of a symbol $i \neq 0$ in $\omega(u)$ is identical to the number of components in the annotation of the node ui. Furthermore, the number of components in the annotation of the root node is 1.

With this notion of consistency, we can prove the following technical result about the relation between dependency structures and annotated trees. We write $\pi_\Sigma(s)$ for the tree obtained from a tree $s \in T_{\Sigma\times\Omega}$ by re-labelling every node u with $\sigma(u)$.

PROPOSITION 1. For every dependency structure (t, x) over Σ, there exists a tree s over $\Sigma \times \Omega$ such that $\pi_\Sigma(s) = t$ and $\mathrm{lin}(s) = x$. Conversely, for every consistently order-annotated tree $s \in T_{\Sigma\times\Omega}$, there exists a uniquely determined dependency structure (t, x) with these properties. □

3.3 Local versions of structural constraints

The encoding of dependency structures as order-annotated trees allows us to reformulate two constraints on non-projectivity originally defined on fully specified dependency structures (Bodirsky et al., 2005) in terms of syntactic properties of the order annotations that they induce:

Gap-degree The *gap-degree* of a dependency structure is the maximum over the number of discontinuities in any yield of that structure.

EXAMPLE. The structure depicted in Figure 2 has gap-degree 1: the yield of b has one discontinuity, marked by the node e, and this is the maximal number of discontinuities in any yield of the structure. □

Since a discontinuity in a yield is delimited by two blocks, and since the number of blocks of a node u equals the number of components in the order annotation of u, the following result is obvious:

PROPOSITION 2. A dependency structure has gap-degree k if and only if the maximal number of components among the annotations $\omega(u)$ is $k + 1$. □

In particular, a dependency structure is projective iff all of its annotations consist of just one component.

Well-nestedness The well-nestedness condition constrains the arrangement of subtrees in a dependency structure. Two subtrees $t/u_1, t/u_2$ *interleave*, if there are nodes $v_l^1, v_r^1 \in t/u_1$ and $v_l^2, v_r^2 \in t/u_2$ such that $v_l^1 \prec v_l^2 \prec v_r^1 \prec v_r^2$. A dependency structure is *well-nested*, if no two of its disjoint subtrees interleave. We can prove the following result:

PROPOSITION 3. A dependency structure is well-nested if and only if no annotation $\omega(u)$ contains a substring $i \cdots j \cdots i \cdots j$, for $i, j \in \mathbb{N}$. □

EXAMPLE. The dependency structure in Figure 1 is well-nested, the structure depicted in Figure 2 is not: the subtrees rooted at the nodes b and e interleave. To see this, notice that $b \prec e \prec d \prec f$. Also notice that $\omega(a)$ contains the substring 1212. □

4 Regular dependency languages

The encoding of dependency structures as order-annotated trees gives rise to an encoding of dependency languages as tree languages. More specifically, dependency languages over a set Σ can be encoded as tree languages over the set $\Sigma \times \Omega$, where Ω is the set of all order annotations. Via this encoding, we can study dependency languages using the tools and results of the well-developed formal theory of tree languages. In this section, we discuss dependency languages that can be encoded as regular tree languages.

4.1 Regular tree grammars

The class of regular tree languages, REGT for short, is a very natural class with many characterizations (Gécseg and Steinby, 1997): it is generated by regular tree grammars, recognized by finite tree automata, and expressible in monadic second-order logic. Here we use the characterization in terms of grammars. Regular tree grammars are natural candidates for the formalization of dependency lexicons, as each rule in such a grammar can be seen as the specification of a word and the syntactic categories or grammatical functions of its immediate dependents.

Formally, a *(normalized) regular tree grammar* is a construct $G = (N_G, \Sigma_G, S_G, P_G)$, in which N_G and Σ_G are finite sets of non-terminal and terminal symbols, respectively, $S_G \in N_G$ is a dedicated start symbol, and P_G is a finite set of productions of the form $A \to \sigma(A_1 \cdots A_n)$, where $\sigma \in \Sigma_G$, $A \in N_G$, and $A_i \in N_G$, for every $i \in [n]$. The *(direct) derivation relation* associated to G is the binary relation $\Rightarrow_G$ on the set $T_{\Sigma_G \cup N_G}$ defined as follows:

$$\frac{t \in T_{\Sigma_G \cup N_G} \qquad t/u = A \qquad (A \to s) \in P_G}{t \Rightarrow_G t[u \mapsto s]}$$

Informally, each step in a derivation replaces a non-terminal-labelled leaf by the right-hand side of a matching production. The *tree language* generated by G is the set of all terminal trees that can eventually be derived from the trivial tree formed by its start symbol: $L(G) = \{ t \in T_{\Sigma_G} \mid S_G \Rightarrow_G^* t \}$.

4.2 Regular dependency grammars

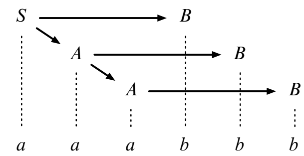
We call a dependency language *regular*, if its encoding as a set of trees over $\Sigma \times \Omega$ forms a regular tree language, and write REGD for the class of all regular dependency languages. For every regular dependency language L, there is a regular tree grammar with terminal alphabet $\Sigma \times \Omega$ that generates the encoding of L. Similar to the situation with individual structures, the converse of this statement does not hold: the consistency properties mentioned above impose corresponding syntactic restrictions on the rules of grammars G that generate the encoding of L.

PROPERTY C1′: The ω-component of every production $A \to \langle \sigma, \omega \rangle (A_1 \cdots A_n)$ in G contains all and only symbols in the set $\{0\} \cup \{ i \mid i \in [n] \}$.

PROPERTY C2′: For every non-terminal $X \in N_G$, there is a uniquely determined integer d_X such that for every production $A \to \langle \sigma, \omega \rangle (A_1 \cdots A_n)$ in G, d_{A_i} gives the number of occurrences of i in ω, d_A gives the number of components in ω, and $d_{S_G} = 1$.

It turns out that these properties are in fact sufficient to characterize the class of regular tree grammars that generate encodings of dependency languages. In but slight abuse of terminology, we will refer to such grammars as *regular dependency grammars*.

EXAMPLE. Figure 3 shows a regular tree grammar that generates a set of non-projective dependency structures with string language $\{ a^n b^n \mid n \geq 1 \}$. □



$$\begin{aligned}
S &\to \langle a, \langle 01 \rangle \rangle (B) \mid \langle a, \langle 0121 \rangle \rangle (A, B) \\
A &\to \langle a, \langle 0, 1 \rangle \rangle (B) \mid \langle a, \langle 01, 21 \rangle \rangle (A, B) \\
B &\to \langle b, \langle 0 \rangle \rangle
\end{aligned}$$

Figure 3: A grammar for a language in REGD(1)

5 Structural constraints and formal power

In this section, we present our results on the generative capacity of regular dependency languages, linking them to a large class of mildly context-sensitive grammar formalisms.

5.1 Gap-restricted dependency languages

A dependency language L is called *gap-restricted*, if there is a constant $c_L \geq 0$ such that no structure in L has a gap-degree higher than c_L. It is plain to see that every regular dependency language is gap-restricted: the gap-degree of a structure is directly reflected in the number of components of its order annotations, and every regular dependency grammar makes use of only a finite number of these annotations. We write REGD(k) to refer to the class of regular dependency languages with a gap-degree bounded by k.

Linear Context-Free Rewriting Systems Gap-restricted dependency languages are closely related to Linear Context-Free Rewriting Systems (LCFRS) (Vijay-Shanker et al., 1987), a class of formal systems that generalizes several mildly context-sensitive grammar formalisms. An LCFRS consists of a regular tree grammar G and an interpretation of the terminal symbols of this grammar as linear, non-erasing functions into tuples of strings. By these functions, each tree in $L(G)$ can be evaluated to a string.

EXAMPLE. Here is an example for a function:

$$f(\langle x_1^1, x_1^2 \rangle, \langle x_2^1 \rangle) = \langle a x_1^1, x_2^1 x_1^2 \rangle$$

This function states that in order to compute the pair of strings that corresponds to a tree whose root node is labelled with the symbol f, one first has to compute the pair of strings corresponding to the first child of the root node ($\langle x_1^1, x_1^2\rangle$) and the single string corresponding to the second child ($\langle x_2^1\rangle$), and then concatenate the individual components in the specified order, preceded by the terminal symbol a. $\square$

We call a function *lexicalized*, if it contributes exactly one terminal symbol. In an LCFRS in which all functions are lexicalized, there is a one-to-one correspondence between the nodes in an evaluated tree and the positions in the string that the tree evaluates to. Therefore, tree and string implicitly form a dependency structure, and we can speak of the *dependency language* generated by a lexicalized LCFRS.

Equivalence We can prove that every regular dependency grammar can be transformed into a lexicalized LCFRS that generates the same dependency language, and vice versa. The basic insight in this proof is that every order annotation in a regular dependency grammar can be interpreted as a compact description of a function in the corresponding LCFRS. The number of components in the order-annotation, and hence, the gap-degree of the resulting dependency language, corresponds to the *fan-out* of the function: the highest number of components among the arguments of the function (Satta, 1992).[1] A technical difficulty is caused by the fact that LCFRS can swap components: $f(\langle x_1^1, x_1^2\rangle) = \langle a x_1^2, x_1^1\rangle$. This commutativity needs to be compiled out during the translation into a regular dependency grammar.

We write LLCFRL(k) for the class of all dependency languages generated by lexicalized LCFRS with a fan-out of at most k.

PROPOSITION 4. $\text{REGD}(k) = \text{LLCFRL}(k+1)$ $\square$

In particular, the class REGD(0) of regular dependency languages over projective structures is exactly the class of dependency languages generated by lexicalized context-free grammars.

EXAMPLE. The gap-degree of the language generated by the grammar in Figure 3 is bounded by 1. The rules for the non-terminal A can be translated into the following functions of an equivalent LCFRS:

$$\begin{aligned} f_{\langle a,\langle 0,1\rangle\rangle}(\langle x_1^1\rangle) &= \langle a, x_1^1\rangle \\ f_{\langle a,\langle 01,21\rangle\rangle}(\langle x_1^1, x_1^2\rangle, \langle x_2^1\rangle) &= \langle a x_1^1, x_2^1 x_1^2\rangle \end{aligned}$$

The fan-out of these functions is 2. $\square$

[1] More precisely, gap-degree = fan-out − 1.

5.2 Well-nested dependency languages

The absence of the substring $i \cdots j \cdots i \cdots j$ in the order annotations of well-nested dependency structures corresponds to a restriction to 'well-bracketed' compositions of sub-structures. This restriction is central to the formalism of Coupled-Context-Free Grammar (CCFG) (Hotz and Pitsch, 1996).

It is straightforward to see that every CCFG can be translated into an equivalent LCFRS. We can also prove that every LCFRS obtained from a regular dependency grammar with well-nested order annotations can be translated back into an equivalent CCFG. We write $\text{REGD}_{wn}(k)$ for the well-nested subclass of REGD(k), and LCCFL(k) for the class of all dependency languages generated by lexicalized CCFGs with a fan-out of at most k.

PROPOSITION 5. $\text{REGD}_{wn}(k) = \text{LCCFL}(k+1)$ $\square$

As a special case, Coupled-Context-Free Grammars with fan-out 2 are equivalent to Tree Adjoining Grammars (TAGs) (Hotz and Pitsch, 1996). This enables us to generalize a previous result on the class of dependency structures generated by lexicalized TAGs (Bodirsky et al., 2005) to the class of generated dependency languages, LTAL.

PROPOSITION 6. $\text{REGD}_{wn}(1) = \text{LTAL}$ $\square$

6 Conclusion

In this paper, we have presented a lexicalized reformulation of two structural constraints on non-projective dependency representations, and shown that combining dependency lexicons that satisfy these constraints with a regular means of syntactic composition yields classes of mildly context-sensitive dependency languages. Our results make a significant contribution to a better understanding of the relation between the phenomenon of non-projectivity and notions of formal power.

The close link between restricted forms of non-projective dependency languages and mildly context-sensitive grammar formalisms provides a promising starting point for future work. On the practical side, it should allow us to benefit from the experience in building parsers for mildly context-sensitive formalisms when addressing the task of efficient non-projective dependency parsing, at least in the framework of grammar-driven parsing. This may eventually lead to a better trade-off between structural flexibility and computational efficiency than that obtained with current systems. On a more theoretical level, our results provide a basis for comparing a variety of formally rather distinct grammar formalisms with respect to the sets of dependency structures that they can generate. Such a comparison may be empirically more adequate than one based on traditional notions of generative capacity (Kallmeyer, 2006).

Acknowledgements We thank Guido Tack, Stefan Thater, and the anonymous reviewers of this paper for their detailed comments. The work of the authors is funded by the German Research Foundation.

References

Manuel Bodirsky, Marco Kuhlmann, and Mathias Möhl. 2005. Well-nested drawings as models of syntactic structure. In *Tenth Conference on Formal Grammar and Ninth Meeting on Mathematics of Language*, Edinburgh, Scotland, UK.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 423–429, Barcelona, Spain.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 457–464, College Park, Maryland, USA.

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer-Verlag, New York, USA.

Günter Hotz and Gisela Pitsch. 1996. On parsing coupled-context-free languages. *Theoretical Computer Science*, 161:205–233.

Aravind K. Joshi. 1985. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, Cambridge, UK.

Laura Kallmeyer. 2006. Comparing lexicalized grammar formalisms in an empirically adequate way: The notion of generative attachment capacity. In *International Conference on Linguistic Evidence*, pages 154–156, Tübingen, Germany.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL) Main Conference Poster Sessions*, pages 507–514, Sydney, Australia.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 81–88, Trento, Italy.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Human Language Technology Conference (HLT) and Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–530, Vancouver, British Columbia, Canada.

Peter Neuhaus and Norbert Bröker. 1997. The complexity of recognition of linguistically adequate dependency grammars. In *35th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 337–343, Madrid, Spain.

Joakim Nivre. 2006. Constraints on non-projective dependency parsing. In *Eleventh Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 73–80, Trento, Italy.

Chris Quirk, Arul Menezes, and Colin Cherry. 2005. Dependency treelet translation: Syntactically informed phrasal smt. In *43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 271–279, Ann Arbor, USA.

Owen Rambow and Aravind K. Joshi. 1997. A formal look at dependency grammars and phrase-structure grammars. In Leo Wanner, editor, *Recent Trends in Meaning-Text Theory*, volume 39 of *Studies in Language, Companion Series*, pages 167–190. John Benjamins, Amsterdam, The Netherlands.

Giorgio Satta. 1992. Recognition of linear context-free rewriting systems. In *30th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 89–95, Newark, Delaware, USA.

Katerina Veselá, Jiři Havelka, and Eva Hajičova. 2004. Condition of projectivity in the underlying dependency structures. In *20th International Conference on Computational Linguistics (COLING)*, pages 289–295, Geneva, Switzerland.

K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 104–111, Stanford, California, USA.