Dependency Structures Derived from Minimalist Grammars

Marisa Ferrara Boston[1], John T. Hale[1], and Marco Kuhlmann[2]

[1] Cornell University
[2] Uppsala University

Abstract. This paper provides an interpretation of Minimalist Grammars [16,17] in terms of dependency structures. Under this interpretation, merge operations derive projective dependency structures, and movement operations introduce both non-projectivity and illnestedness. This new characterization of the generative capacity of Minimalist Grammar makes it possible to discuss the linguistic relevance of non-projectivity and illnestedness. This in turn provides insight into grammars that derive structures with these properties.[1]

1 Introduction

This paper investigates the class of dependency structures that Minimalist Grammars (MGs) [16,17] derive. MGs stem from the generative linguistic tradition, and Chomsky's Minimalist Program [1] in particular. The MG formalism encourages a lexicalist analysis in which hierarchical syntactic structure is built when licensed by word properties called "features". MGs facilitate a movement analysis of long-distance dependency that is conditioned by lexical features as well. Unlike unification grammars, but similar to categorial grammar, these features must be cancelled in a particular order specific to each lexical item.

Dependency Grammar [18] is a syntactic tradition that determines sentence structure on the basis of word-to-word connections, or dependencies. DG names a family of approaches to syntactic analysis that all share a commitment to word-to-word connections. [8] relates properties of dependency graphs, such as projectivity and wellnestedness, to language-theoretic concerns like generative capacity.

This paper examines these same properties in MG languages. We do so using a new DG interpretation of MG derivations. This tool reveals that syntactic movement as formalized in MGs can derive the sorts of illnested structures attested in Czech comparatives from the Prague Dependency Treebank 2.0 (PDT) [5]. Previous research in the field indirectly link MGs and DGs: [12] proves the equivalence of MGs with Linear Context-Free Rewriting Systems (LCFRS) [19], and the relation of LCFRS to DGs is made explicit in [8]. This paper provides a

[1] The authors thank Joan Chen-Main, Aravind K. Joshi, and audience members at *Mathematics of Language 11* for discussion and suggestions.

C. Ebert, G. Jäger, and J. Michaelis (Eds.): MOL 10/11, LNCS 6149, pp. 1–12, 2010.

direct connection between the two formalisms. Using this connection, we investigate the linguistic relevance of structural constraints such as non-projectivity and illnestedness based on MGs that induce structures with these properties. The system proposed is not a new formalism, but a technical tool that we use to gain linguistic insight.

Section 2 describes MGs as they are formalized in [17], and Section 3 translates MG operations into operations on dependency structures. Sections 4 and 5 discuss the structural constraints of projectivity and nestedness in terms of MGs.

2 Minimalist Grammars

This section introduces notation particular to the MG formalism. Following [16] and [17], a Minimalist Grammar G is a five-tuple $(\Sigma, F, \mathit{Types}, \mathit{Lex}, \mathcal{F})$. Σ is the vocabulary of the grammar, which can include empty elements. Figure 6 exemplifies the use of empty "functional" elements typical in Chomskyan and Kaynian analyses. There ϵ denotes an empty element that, while syntactically potent, makes no contribution to the derived string. F is a set of *features*, built over a non-empty set of *base features*, which denote the lexical category of the item. If f is a base feature, then $=f$ is a *selection feature*, which selects for complements with base feature f; a prefixed $+$ or $-$ identifies *licensor* and *licensee* features, $+f$ and $-f$ respectively, that license movement. A Minimalist Grammar distinguishes two *types* of structure. The "simple" type, flagged by double colons (::), identifies items fresh out of the lexicon. Any involvement in structure-building creates a "derived" item (:) flagged with a single colon. This distinction allows the first derivation of syntactic composition to be handled differently from later episodes. A *chain* is a triple $\Sigma^* \times \mathit{Types} \times F^*$, and an *expression* is a non-empty sequence of chains. The set of all expressions is denoted by E. The *lexicon* Lex is a finite subset of chains with type ::. The set $\mathcal{F}$ is a set of two generating functions, `merge` and `move`. For simplicity, we focus on MGs that do not incorporate head or covert movement. Table 1 presents the functions in inference-rule form, following [17]. In the table, the juxtaposition st denotes the concatenation of two strings s and t.

`Merge` is a structure building operation that creates a new, derived expression from two expressions ($E \times E \to E$). It is the union of three functions, shown in the upper half of Table 1. Each sub-function applies according to the type and feature of the lexical items to be merged. The three merge operations differ with respect to the types of chains they operate on. If s is simple (::), the `merge1` operation applies. If it is derived (:), the `merge2` operation applies. We write $\cdot$ when the type does not matter. If t has additional features δ, the `merge3` operation must apply regardless of the type of s. The `move` operation is a structure building operation that creates a new expression from an expression ($E \to E$). It is the union of two functions, `move1` and `move2`, provided in the lower half of Table 1. As with the `merge3` operation, `move2` only applies when t has additional features δ.

Table 1. Merge and Move

$$\frac{s :: =f\gamma \qquad t \cdot f, \alpha_1, \ldots \alpha_k}{st : \gamma, \alpha_1, \ldots, \alpha_k}\texttt{merge1}$$

$$\frac{s : =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f, \iota_1, \ldots, \iota_l}{ts : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l}\texttt{merge2}$$

$$\frac{s \cdot =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \iota_1, \ldots, \iota_l}{s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l}\texttt{merge3}$$

$$\frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}\texttt{move1}$$

$$\frac{s \cdot +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{s : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k}\texttt{move2}$$

3 MG Operations on Dependency Trees

In this section we introduce a formalism to derive dependency structures from MGs. Throughout the discussion $\mathbb{N}$ denotes the set of non-negative integers.

3.1 Dependency Trees

DG is typically discussed in terms of directed dependency *graphs*. However, the directed nature of dependency arrows and the single-headed condition [13] allow these graphs to also be viewed as *trees*. We define dependency trees in terms of their nodes, with each node in a dependency tree labeled by an *address*, a sequence of positive integers. We write λ for the empty sequence of integers. Letters u, v, w are variables for addresses, $\mathbf{s}$,$\mathbf{t}$ are variables for sets of addresses, and x, y are variables for sequences of addresses. If u and v are addresses, then the concatenation of the two is as well, denoted by uv. Given an address u and a set of addresses $\mathbf{s}$, we write $\uparrow_u \mathbf{s}$ for the set $\{\, uv \mid v \in \mathbf{s} \,\}$. Given an address u and a sequence of addresses $x = v_1, \ldots, v_n$, we write $\uparrow_u x$ for the sequence $uv_1, \ldots, uv_n$. Note that $\uparrow_u \mathbf{s}$ is a *set* of addresses, whereas $\uparrow_u x$ is a *sequence* of addresses.

A *tree domain* is a set $\mathbf{t}$ of addresses such that, for each address u and each integer $i \in \mathbb{N}$, if $ui \in \mathbf{t}$, then $u \in \mathbf{t}$ (prefix-closed), and $uj \in \mathbf{t}$ for all $1 \leq j \leq i$ (left-sibling closed). A *linearization* of a finite set S is a sequence of elements of S in which each element occurs exactly once. For the purposes of this paper, a *dependency tree* is a pair $(\mathbf{t}, x)$, where $\mathbf{t}$ is a tree domain, and x is a linearization of $\mathbf{t}$. A *segmented dependency tree* is a non-empty sequence $(\mathbf{s}_1, x_1), \ldots, (\mathbf{s}_n, x_n)$, where each $\mathbf{s}_i$ is a set of addresses, each x_i is a linearization of $\mathbf{s}_i$, all sets $\mathbf{s}_i$ are pairwise disjoint, and the union of the sets $\mathbf{s}_i$ forms a tree domain. A pair $(\mathbf{s}_i, x_i)$ is called a *component*, which corresponds to *chains* in Stabler and Keenan's [17] terminology.

An *expression* is a sequence of triples $(c_1, \tau_1, \gamma_1), \ldots, (c_n, \tau_n, \gamma_n)$, where $(c_1, \ldots, c_n)$ is a segmented dependency tree, each τ_i is a type (lexical or derived), and each γ_i is a sequence of features. We write these triples as $c_i :: \gamma_i$ (if the type is lexical), $c_i : \gamma_i$ (if the type is derived), or $c_i \cdot \gamma_i$ (if the type does not matter). We use the letters α and ι as variables for elements of an expression. Given an element $\alpha = ((\mathbf{s}, x), \tau, \gamma)$ and an address u, we write $\uparrow_u \alpha$ for the element $((\uparrow_u \mathbf{s}, \uparrow_u x), \tau, \gamma)$.

Given an expression d with associated tree domain $\mathbf{t}$, we write $next(d)$ for the minimal positive integer i such that $i \notin \mathbf{t}$.

3.2 Merge

Merge operations allow additional dependency structure to be added to an initially derived tree. These changes are recorded in the manipulation of the dependency tree addresses, as formalized in the previous section. Table 2 provides a dependency interpretation for each of the structure-building rules introduced in Table 1. The $\texttt{merge}_{DG}$ functions create a dependency between two trees, such that the root of the left tree becomes the head of the root of the right tree, where left and right correspond to the trees in the rules. For example, in $\texttt{merge1}_{DG}$ the $\uparrow_1 \mathbf{t}$ notation signifies that $\mathbf{t}$ is now the first daughter of $\mathbf{s}$. Its components are similarly updated.

Table 2. Merge in terms of dependency trees

$$\frac{(\{\lambda\}, \langle\lambda\rangle) :: =f\gamma \qquad (\mathbf{t}, x) \cdot f, \alpha_1, \ldots, \alpha_k}{(\{\lambda\} \cup \uparrow_1 \mathbf{t}, \langle\lambda\rangle \cdot \uparrow_1 x) : \gamma, \uparrow_1 \alpha_1, \ldots, \uparrow_1 \alpha_k}\texttt{merge1}_{DG}$$

$$\frac{(\mathbf{s}, x) : =f\gamma, \alpha_1, \ldots, \alpha_k \qquad (\mathbf{t}, y) \cdot f, \iota_1, \ldots, \iota_l}{(\mathbf{s} \cup \uparrow_i \mathbf{t}, \uparrow_i y \cdot x) : \gamma, \alpha_1, \ldots, \alpha_k, \uparrow_i \iota_1, \ldots, \uparrow_i \iota_l}\texttt{merge2}_{DG}$$

$$\frac{(\mathbf{s}, x) \cdot =f\gamma, \alpha_1, \ldots, \alpha_k \qquad (\mathbf{t}, y) \cdot f\delta, \iota_1, \ldots, \iota_l}{(\mathbf{s}, x) : \gamma, \alpha_1, \ldots, \alpha_k, (\uparrow_i \mathbf{t}, \uparrow_i y) : \delta, \uparrow_i \iota_1, \ldots, \uparrow_i \iota_l}\texttt{merge3}_{DG}$$

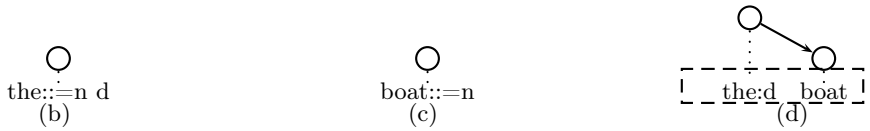
$$\text{where } i = next((\mathbf{s}, x) \cdot =f\gamma, \alpha_1, \ldots, \alpha_k)$$

Applying the $\texttt{merge1}_{DG}$ rule to a simple English grammar creates the dependency tree in Figure (1). Dependency relations between nodes are notated with solid arrows and node labels are notated with dotted lines; the dependency graphs shown in the figures are encoded by the address sets described above in Table 2. A lexicon for these examples is provided in Figure 1(a).

As was mentioned above, the merge rules apply in different contexts depending on the tree types and number of features. $\texttt{Merge1}_{DG}$ can apply in Figure 1 because the selector tree `the` is simple and the selected tree `boat` does not have additional features δ. The entire derived tree forms a single component, denoted by the dashed box. `Merge2` contrasts with `merge1` in the linearized order of the nodes: in this case, the right tree is ordered before the left tree and its children, as in Figure 2(a).

the::=n d boat::n docked::=d=d v where::d -wh ϵ::=v +wh c

(a) Lexicon 1.



(b) (c) (d)

Fig. 1. $\texttt{merge1}_{DG}$ applies to two simple dependency trees

The rules given in Table 2 are a deduction system for *expressions*: sequences of triples whose first components together define a segmented dependency tree. Each component, spanning any number of words, has a feature-sequence associated with it. $\texttt{Merge3}_{DG}$ introduces new, unlinearized components into the derivation. These components are unordered with respect to the other components, though the words within the components are ordered. In Figure 2(b), `docked` and `where` are represented by separate dashed boxes; this indicates that their relative linear order is unknown. $\texttt{Merge3}_{DG}$ contrasts with applications of $\texttt{merge1}_{DG}$ $\texttt{merge2}_{DG}$, where two dependency trees are merged into one, fully-ordered component, demonstrated by Figures 1(d) and 2(a).

3.3 Move

The move operation does not create or destroy dependencies in the derived tree. It re-orders the nodes, and reduces the number of components in the tree by one. Table 3 defines these rules in terms of dependency trees.

Table 3. `Move` in terms of dependency trees

$$\frac{(\mathbf{s}, x) : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, (\mathbf{t}, y) : -f, \alpha_{i+1}, \ldots, \alpha_k}{(\mathbf{s} \cup \mathbf{t}, yx) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}\texttt{move1}_{DG}$$

$$\frac{\mathbf{s} \cdot +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, \mathbf{t} : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{\mathbf{s} : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \mathbf{t} : \delta, \alpha_{i+1}, \ldots, \alpha_k}\texttt{move2}_{DG}$$

Figure 3 demonstrates the $\texttt{move1}_{DG}$ operation on a simple structure. The `where` node is not only reordered to the front of the tree, but it also becomes part of the ϵ node's component. Note that for this example we use an ϵ to denote the structural position that has the `+wh` feature. This follows from standard linguistic practice; in some languages, this position can be marked by an overt lexical item. In English, it is not. Both ϵ and overt lexical items can have licensor features.

Unlike the previous `merge` and `move` operations described, $\texttt{move2}_{DG}$ does not change the dependency structure or linearization of the tree. $\texttt{Move2}_{DG}$ applies when the licensee component has additional features that require further movements. Its sole purpose is to cancel the licensor and licensee features; this feature

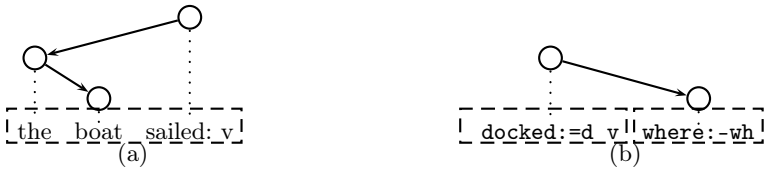


Fig. 2. $\texttt{merge2}_{DG}$ and $\texttt{merge3}_{DG}$

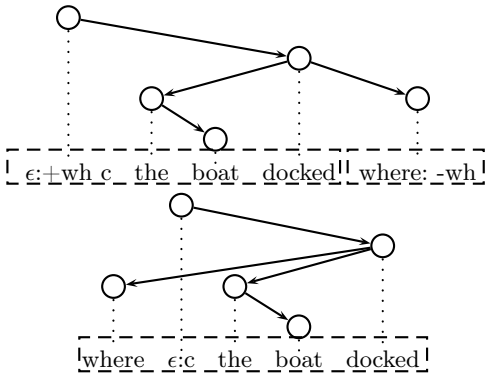


Fig. 3. $\texttt{move1}_{DG}$

cancellation is necessary to preserve the linguistic intuition of the intermediate derivation step. Figure 4 demonstrates $\texttt{Move2}_{DG}$. Although the $+wh$ and $-wh$ features are canceled, the linear order is unaffected and the additional licensee feature $-t$ on `where` remains.

Following [16] and [17], we restrict movement with the Shortest Move Condition (SMC), defined in (1).

(1) None of $\alpha_1, ..., \alpha_{i-1}, \alpha_{i+1}, ..., \alpha_k$ has $-f$ as its first feature.

Adoption of the SMC guarantees a version of MGs that are weakly equivalent to LCFRS [2].

The rules above derive connected dependency structures. This is demonstrated by induction on the derived structure: Single-node trees (i.e., simple lexical items) vacuously satisfy connectedness. All merge rules create dependencies between trees, and movements do not destroy any already-created dependencies. Therefore, a dependency structure at any derivation step will be connected.

Provided that every expression in the lexicon has exactly one base feature, the dependency trees derived from MGs will not contain multi-headed nodes (i.e., nodes with multiple parents). This single-headedness proof follows straightforwardly from two lemmas concerning the role of base features in expressions $E = (c_1, \tau_1, \gamma_1), \ldots, (c_n, \tau_n, \gamma_n)$. Lemma 1 asserts the unique existence of a base feature f in the first feature sequence γ_1. Lemma 2 denies the existence of base features in later components.

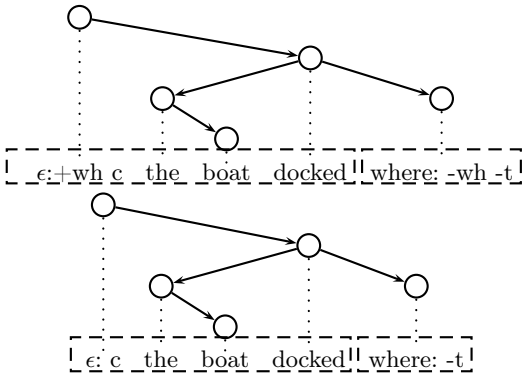


Fig. 4. `move2`$_{DG}$

The dependency structures derived at intermediate steps need not be totally ordered. Because of the `merge3` and `move2` rules, components can be introduced into the dependency structure that have not yet moved to their final order in the structure. However, the usual notion of start category [17] in MGs is a single base feature. This implies that in complete derivations all licensee feature have been checked. The implication guarantees that dependency trees derived using the system in Tables 2 and 3 are totally-ordered.

4 Minimalist Grammars and Block Degree

Projectivity is a constraint on dependency structures that requires subtrees to span intervals. [9] define an interval as the set $[i, j] := \{k \in V | i \leq k \text{ and } k \leq j\}$, where i and j are endpoints, and V is a set of nodes as defined in Section 3.1. Non-projective structures violate this constraint. The node labeled `docked` in Figure 3 spans two intervals: its child spans interval 0 and the node and its other children span intervals 2-4.

Following [8] we use the notion of block degrees to characterize non-projective structures. A tree's block degree is the maximum number of intervals each of its subtrees span. The block degree for Figure 3, repeated in Figure 5, is two: each of the intervals of the node labeled `docked` forms a block. Shaded boxes notate node blocks.

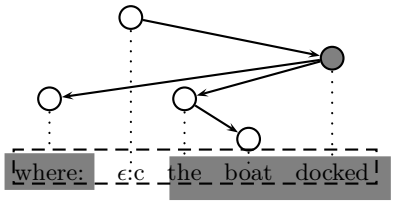


Fig. 5. The block degree of this structure is 2

By construction, $\texttt{merge}_{DG}$ always forms dependency relations between the roots of subtrees. All nodes in the resulting expression are part of the same interval. $\texttt{Move1}_{DG}$ has the potential to create non-projective structures: constituents can move away from the interval that the parent node spans to create a separate constituent block, as demonstrated by Figure 5. In this example, the ϵ element intervenes between `docked` to `where` dependency. As discussed above, in other languages ϵ could be replaced by an overt lexical item. Both types of intervention are considered non-projective in this work.

Because only movements can cause non-projectivity, and because all movements are triggered in the MG framework by a licensor and licensee pair, the block degree of the derived tree is bounded by the number of licensees. In other words, the number of licensees determines the maximum block degree of the structure. This number has previously been identified as an upper bound on the complexity of MGs [11,6]. The coincidence of this result follows from work by [14], who attributed the increased parsing complexity of LCFRS to non-projectivity [8].

5 Minimalist Grammars and Nestedness

A further constraint on the class of dependency structures is wellnestedness [8]. Wellnested structures prohibit the "crossing" of disjoint subtree intervals. Any structure that is not wellnested is said to be illnested, as in Figure 6(b). Here, the subtree spanning `the hearing on the issue` crosses the subtree spanning `scheduled today`. [8] demonstrates that grammars that derive illnested structures are more powerful than grammars that do not, which leads to higher parsing complexity.

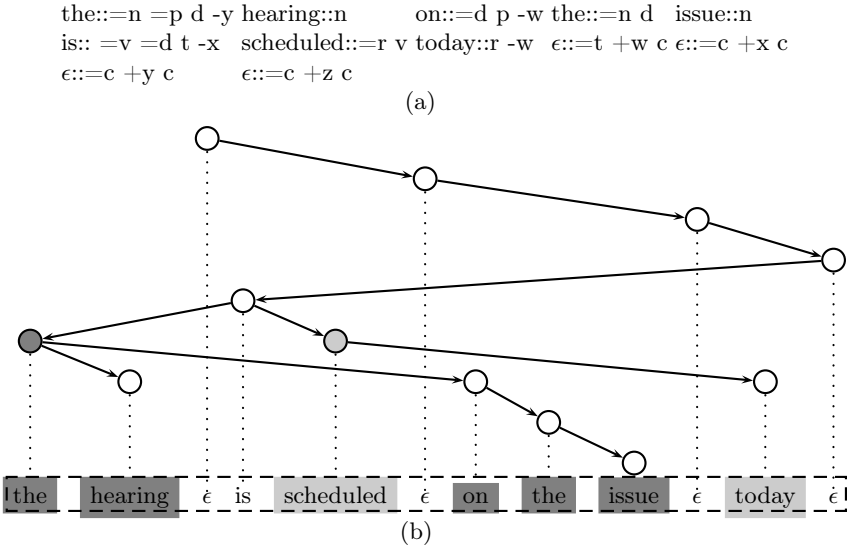
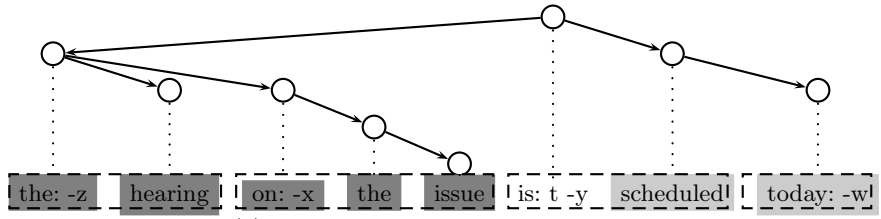
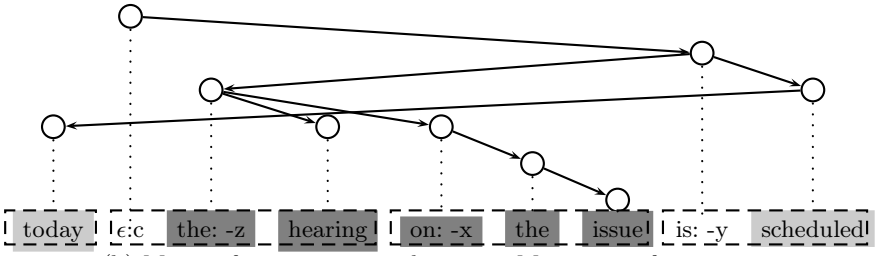


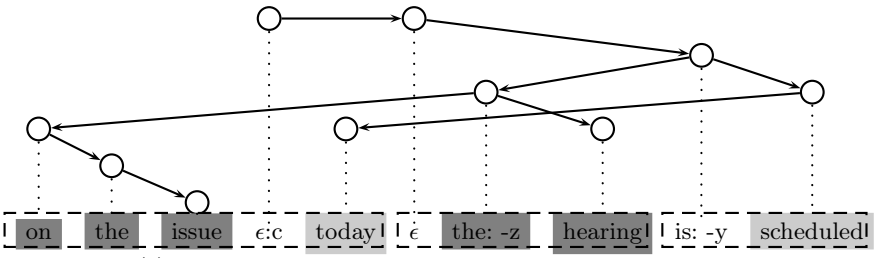
Fig. 6. MGs derive illnested dependency structures



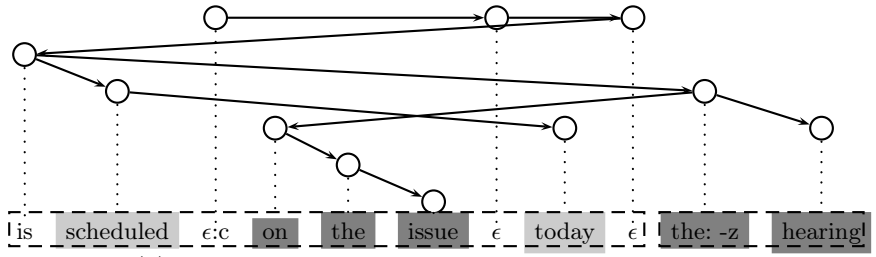
(a) Merged structure before movement.



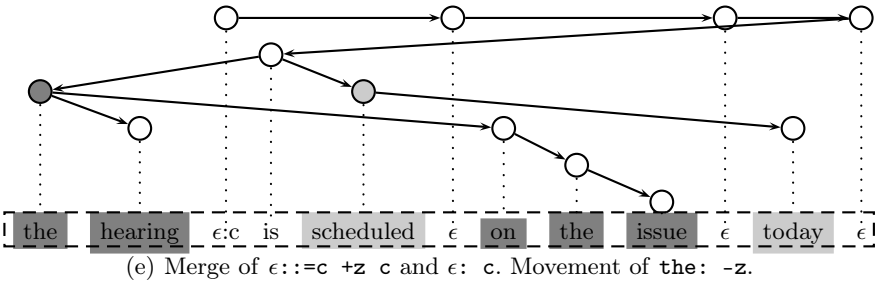
(b) Merge of ϵ`::=v +w c` and `is: -y`. Movement of `today: -w`.



(c) Merge of ϵ`::=c +x c` and ϵ`: c`. Movement of `on: -x`.



(d) Merge of ϵ`::=c +y c` and ϵ`: c`. Movement of `is: -y`.



(e) Merge of ϵ`::=c +z c` and ϵ`: c`. Movement of `the: -z`.

Fig. 7. Derivation of illnested English structure

We prove that MGs are able to derive illnested structures by example. The grammar in Figure 6(a) derives the illnested English structure in Figure 6(b). The result is a 1-illnested structure, the lowest level of illnestedness in the characterization of [10]. Not all mildly context-sensitive formalisms can derive illnested structures. For example, TAGs can only generate wellnested structures with a block-degree of at most two [8]. Our proof demonstrates that MGs derive structures with higher block degrees, which have a potential for illnested structures. This allows MGs to generate the same string languages as LCFRS, which also generate illnested structures [15].

The illnested structure in Figure 6(b) is also interesting from a linguistic perspective. It represents a case of noun-complement clause extraposition [4], where the complement `on the issue` is extraposed from the determiner phrase `the hearing`. The additional extraposition of the adverb `today` from the verb phrase `is scheduled` leads to the illnested final structure. Several analyses of extraposition are put forth in the literature, but here we choose Kayne's [7] "stranding analysis", where a series of leftward movements leads to modifier stranding. These movements are each motivated by empty functional categories that could be overt in other possible human languages. In this lexicon, first the adverb is moved by the licensor $+w$ (Figure 7(a)), followed by the prepositional phrase, the verb phrase, and finally the noun phrase, as in Figures 7(b) through 7(e).

The analysis of the illnested structure in Figure 6(b) in terms of extraposition provides a first step towards an understanding of the linguistic relevance of illnested structures. This is not only useful for the analysis of dependencies in formal grammars, but also the analysis of extraposition in linguistics.

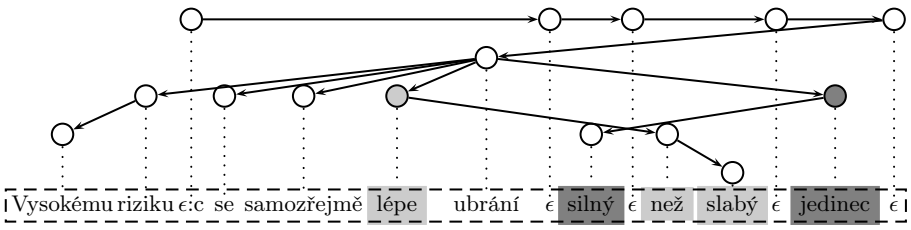
Investigating the linguistic qualities of illnested structures cross-linguistically also shows promise. For example, treebanks from languages with freer word order, such as Czech, tend to have more illnested structures [8]. The sentence in Figure 8 is sentence number Ln94209_45.a/18 from the PDT 2.0[2]. An English gloss of the sentence is "A strong individual will obviously withstand a high risk better than a weak individual".[3] This particular example is a comparative construction (better X than Y) [3], which can give rise to illnestedness in Czech. The MG acknowledges syntactic relationships between the comparative construction and the adjectives *weak* and *strong*. The specific analysis stays close to the Kaynian tradition in supposing empty categories that intertwine with the dependencies from two different subtrees. Other constructions that cause illnestedness in the PDT are subject complements, verb-nominal predicates, and coordination (Kateřina Veselá, p.c.). At least in the case of Czech, this evidence suggests that the expressive power of a syntactic movement rule (rather than just attachment) is required.

[2] Punctuation is removed to simplify the diagram.

[3] The authors thank Jiří Havelka (IBM Czech Republic), Kateřina Veselá (Charles University), and E. Wayles Browne (Cornell University) for the translation and linguistic analysis of the illnested structures in the PDT.

Vysokému::Atr	riziku::=Atr Obj -f
se::AuxT	samozřejmě::AuxY
lépe::=AuxC Adv	než::=ExD AuxC -b
slabý::ExD	silný::Atr -d
jedinec::=Atr Sb -a	ubrání::=Obj =Adv =AuxY =AuxT =Sb Pred -e
ϵ::=Pred +a c	ϵ::=c +b c
ϵ::=c +d c	ϵ::=c +e c
ϵ::=c +f c	

(a)



(b)

Fig. 8. An illnested Czech example from the Prague Dependency Treebank

6 Conclusion

This paper provides a definition of MG `merge` and `move` operations in terms of dependency trees, and examines the properties of these operations in terms of both projectivity and nestedness constraints. We find that MGs with movement rules derive illnested structure of exactly the sort required by Czech comparatives and English noun-complement clause extractions.

The work also provides a basis for future research in determining how different types of MG movement, such as head, covert, and remnant movement, interact with dependency constraints and properties like illnestedness. Dependency-generative capacity may also provide a new avenue of research into determining how different types of locality constraints (besides the SMC) interact with generative capacity [2].

References

1. Chomsky, N.: The Minimalist Program. MIT Press, Boston (1995)
2. Gärtner, H.M., Michaelis, J.: Some remarks on locality conditions and Minimalist Grammars. In: Sauerland, U., Gärtner, H.M. (eds.) Interfaces + Recursion = Language?, pp. 161–195. Mouton de Gruyter, Berlin (2007)
3. Goldberg, A.: Constructions at Work: The Nature of Generalization in Language. Oxford University Press, New York (2006)

4. Guéron, J., May, R.: Extraposition and logical form. Linguistic Inquiry 15, 1–32 (1984)
5. Hajič, J., Panevová, J., Hajičová, E., Sgall, P., Pajas, P., Štěpánek, J., Havelka, J., Mikulová, M.: Prague dependency treebank 2.0 (2000)
6. Harkema, H.: A characterization of minimalist languages. In: de Groote, P., Morrill, G., Retoré, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, p. 193. Springer, Heidelberg (2001)
7. Kayne, R.S.: The Antisymmetry of Syntax. MIT Press, Cambridge (1994)
8. Kuhlmann, M.: Dependency structures and lexicalized grammars. Ph.D. thesis, Universität des Saarlandes (2007)
9. Kuhlmann, M., Nivre, J.: Mildly non-projective dependency structures. In: Proceedings of the COLING/ACL 2006, pp. 507–514 (2006)
10. Maier, W., Lichte, T.: Characterizing discontinuity in constituent treebanks. In: Proceedings of the Fourteenth Conference on Formal Grammar (2009) `http://webloria.loria.fr/~degroote/FG09/Maier.pdf`
11. Michaelis, J.: Derivational minimalism is mildly context-sensitive. In: Moortgat, M. (ed.) LACL 1998. LNCS (LNAI), vol. 2014, p. 179. Springer, Heidelberg (2001)
12. Michaelis, J.: On formal properties of Minimalist Grammars. Linguistics in Potsdam (LiP) 13, Universitätsbibliothek Publikationsstelle, Potsdam (2001)
13. Nivre, J.: Inductive Dependency Parsing. In: Text, Speech and Language Technology, Springer, New York (2006)
14. Satta, G.: Recognition of linear context-free rewriting systems. In: Proceedings of the Association for Computational Linguists (ACL), pp. 89–95 (1992)
15. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
16. Stabler, E.P.: Derivational minimalism. In: Retoré, C. (ed.) LACL 1996. LNCS (LNAI), vol. 1328, pp. 68–95. Springer, Heidelberg (1997)
17. Stabler, E.P., Keenan, E.: Structural similarity within and among languages. Theoretical Computer Science 293(2), 345–363 (2003)
18. Tesnière, L.: Éléments de syntaxe structurale. Editions Klincksiek (1959)
19. Vijay-shanker, K., Weir, D.J., Joshi, A.K.: Characterizing structural descriptions produced by various grammatical formalisms. In: Proceedings of the Association for Computational Linguists (ACL), pp. 104–111 (1987)