Universität des Saarlandes
Fachrichtung 6.2 – Informatik

Diplomarbeit

Drawings as Models of Syntactic Structure: Theory and Algorithms

Mathias Möhl

Februar 2006

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka
Betreuung durch Marco Kuhlmann

Erklärung

Hiermit erkläre ich, Mathias Möhl, an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 21. Februar 2006

Abstract

This thesis introduces drawings, a class of mathematical structures suitable to represent the syntactic structure of natural language sentences. Drawings are simple structures, and independent of any grammar formalism or generating device. The derived structures of many grammar formalisms can be interpreted as a drawing. Structures derived by different formalisms can thus be compared on a common base.

Drawings are not restricted to projective analyses. We present two measures for the level of non-projectivity of drawings: gap degree and well-nestedness. With these two measures we characterise the class of drawings that corresponds to the derivations of Tree Adjoining Grammar (TAG).

Furthermore, a description language for well-nested drawings is presented and complemented with an algorithm that enumerates all drawings for an expression of that language.

Acknowledgements

I would like to thank Professor Dr. Gert Smolka for proposing this interesting subject to me, and for enabling me to work in the pleasant atmosphere of the Programming Systems Lab.

I also want to thank my supervisor Marco Kuhlmann. Most of the results of this thesis originated from extensive discussions with Marco; the weekly meetings were always a rich source of new ideas and motivation. Beyond scientific questions, his comprehensive support covered also help with typesetting in LaTeX, and with English phrasing.

I am grateful to Alexander Koller who read drafts of Chapter 7 and provided me with helpful comments on saturation algorithms.

Furthermore, I want to thank Manuel Bodirsky, Ralph Debusmann, Robert Grabowski, and Guido Tack for fruitful discussions and support in various aspects. In particular I want to mention that Ralph extended his LaTeX package for the visualisation of drawings to meet my needs.

While working on this thesis, I was funded by the project CHORUS in the Collaborative Research Centre 378 'Resource-Adpative Cognitive Processes' of the Deutsche Forschungsgemeinschaft (DFG).

I also want to thank my parents for their support. Finally, I am grateful to Tina for helping me typesetting some figures and even more for supporting me in a stressful time.

Contents

Chapter 1

Introduction

The analysis of the syntax of natural language sentences is an important subject of computational linguistics. Here, the aim is to generate automatically the syntactic structure for a given sentence. Systems dealing with such problems rely on a grammar that is based on some grammar formalism. The grammar formalism provides methods to characterise the syntactic rules and properties of a certain language. The grammar uses these methods to describe the syntax of a certain language, for example English.

Existing grammar formalisms differ in a variety of aspects. However, the syntactic analyses they produce are often of a similar kind. Most grammar formalisms use either a *phrase structure* or a *dependency structure* to represent the syntactic structure of a sentence.

The phrase structure subdivides the sentence recursively into smaller syntactic units. The sentence: 'Dan loves his girlfriend', for example, could be subdivided into a noun phrase ('Dan'), and a verb phrase ('loves his girlfriend'). The verb phrase can be subdivided into the verb ('loves') and the object ('his girlfriend'), which again could be subdivided.

In contrast to that, the dependency structure represents the dependencies among the words of the sentence. In the sentence 'Dan loves his girlfriend', the words 'Dan' and 'girlfriend' depend on 'loves', and 'his' depends on 'girlfriend'.

In this thesis, we introduce a class of mathematical structures, called *drawings*, that is suitable to represent the dependency structure of a sentence. Drawings are independent of any grammar formalism, and encapsulate the central relations of a dependency structure. It is possible to interpret the structures generated by different grammar formalisms as drawings, and to compare them on that basis.

The focus in this thesis is put on the structural properties of drawings. Drawings are not necessarily projective (dependencies may overlap), and we develop two relaxations of projectivity: gap degree and well-nestedness.

In order to illustrate how the derivation of a grammar formalism is interpreted as a drawing, we explain how derivations in lexicalised Tree Adjoining Grammar (TAG) correspond to drawings. Furthermore, we structurally characterise the drawings that correspond to TAG derivations. This allows us to decide whether a drawing is inducible by a TAG grammar without referring to the derivational machinery that underlies TAG.

Grammar formalisms are usually based on local descriptions of the structures they derive. Since the local description of non-projective drawings is not straightforward, we develop a local description language for well-nested drawings.

The remainder of the thesis is structured as follows:

In Chapter 2 we formalise the notion of drawings and along the way we introduce related mathematical structures like total orders and forests.

Chapter 3 introduces the gap degree as a measure of how non-projective a drawing is. In that context we also answer the question how many gaps a drawing has at most. Furthermore, we present an algorithm to compute the gap degree.

Chapter 4 explains well-nestedness, a second relaxation of projectivity that is orthogonal to the gap degree. We analyse the structure of well-nested drawings, present algorithms to decide whether a drawing is well-nested or not and compare well-nestedness to another property called *planarity*.

Chapter 5 contains a brief introduction to Tree Adjoining Grammar (TAG) and explains how derivations in TAG correspond to drawings. Following, a purely structural characterisation of these TAG drawings is given, based on the structural properties discussed in the previous two chapters.

Chapter 6 presents a description language for well-nested drawings and indicates how this language can be interpreted as an extension of TAG.

Chapter 7 presents an algorithm, that is needed to enumerate all drawings for a given description of the language presented in Chapter 6.

The main contributions of this thesis are

- the formalisation of drawings as a class of mathematical structures to represent the dependency analysis of a sentence,
- the definition of two relaxations of projectivity: gap degree and well-nestedness,
- a purely structural characterisation of TAG derivations based on well-nestedness and gap degree,
- a description language for well-nested drawings.

Chapter 2

Drawings as relational structures

In this chapter, we introduce *drawings*, a class of structures suitable to represent the dependency structure of a sentence. First we describe the general concept of a relational structure, and discuss different kinds of relational structures, in particular forests and total orders. Equipped with this formal basis, we then define drawings as a further kind of relational structure.

2.1 Relational structures

Definition 2.1 (Relational structure) A *relational structure* is a tuple $\mathfrak{G}$ whose first component is a set V, and whose remaining components are relations on V. The set V is referred to as the *carrier* of $\mathfrak{G}$. $\dashv$

Although a relational structure may contain relations of arbitrary arity, in this thesis, we confine ourselves to binary relations. Given a binary relation R on V, we write Rvw (or vRw for relations with infix notation) as a shorthand for $(v, w) \in R$, and use the standard notation and terminology for binary relations (cf. Table 2.1). To add a slightly non-standard concept, we define the *relational image* of an object $v \in V$ under R as the set $Rv := \{ w \in V \mid Rvw \}$. Furthermore, we will make use of a transitive reduction $\tilde{R}$ of a relation R, which is defined as a relation with minimal number of edges such that $\tilde{R}$ and R have the same transitive closure. The transitive reduction is not unique in general, but Aho, Garey and Ullman [1] have shown that the transitive reduction of a directed acyclic graph is unique. Since a relation can be interpreted as a directed graph, it is sufficient to show that a relation is acyclic to ensure that its transitive reduction is unique. We will do this in all cases in which we have to consider a transitive reduction.

Relational structures with only binary relations can be seen as labelled digraphs: the elements of the carrier correspond to nodes, and each element (u, v) of the i-th relation R_i corresponds to an i-labelled edge between u and v. With this visualisation in mind, all the standard graph-theoretic terminology can be applied to relational structures. In this spirit, we refer to the elements of the carrier of a relational structure $\mathfrak{G}$ as the *nodes* of $\mathfrak{G}$, and often refer to elements of its relations as labelled *edges*. We freely mix graph-theoretic and relational terminology as we go along.

$\mathrm{U} := \{ (u,v) \mid u,v \in V \}$	(full relation)
$\mathrm{I} := \{ (v,v) \mid v \in V \}$	(identity relation)
$R_1 \circ R_2 := \{ (u,w) \mid \exists v\colon R_1uv \wedge R_2vw \}$	(composition)
$R^0 := \mathrm{I}$	(0-fold composition)
$R^{n+1} := R \circ R^n$	($(n+1)$-fold composition)
$R^{-1} := \{ (w,v) \mid Rvw \}$	(converse)
$R^+ := \cup_{n\geq 1} R^n$	(transitive closure)
$R^* := \cup_{n\geq 0} R^n$	(reflexive transitive closure)
$R\vert_{V'} := \{ (u,v) \in R \mid u,v \in V' \subseteq V \}$	(domain restriction)

Table 2.1: Notation and terminology for binary relations

Example The relational structure $\mathfrak{G} = (V; R_1, R_2)$ with

$$\begin{aligned} V &= \{a,b,c,d\} \\ R_1 &= \{(a,b),(a,c)\} \\ R_2 &= \{(a,d),(c,d)\} \end{aligned}$$

corresponds to the graph shown in Figure 2.1. Note that in the graph view the relational image of a node corresponds to the set of nodes reachable from that node by a path of length one, following an edge in the corresponding relation. $\dashv$

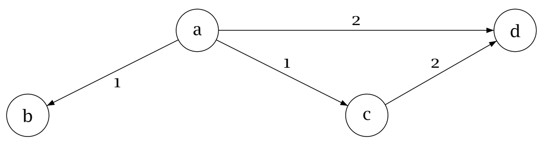


Figure 2.1: An example of a relational structure

2.2 Forests and trees

We want to define a relational structure that represents the dependency structure of a sentence. Using the set of words of a sentence (or – more precisely – the set of word instances) as the carrier of a relational structure, we can model the dependencies among the words with a single relation. An element Rvw in this relation indicates that w depends on v. The dependencies within a sentence usually form a tree or forest.

Definition 2.2 (Forest and tree) A relational structure $(V; R)$ is called a *forest*, if it is acyclic, and each $v \in V$ has at most one R-predecessor. Each node $v \in V$ with no predecessor with respect to R is called a *root*. A forest with exactly one root is called a *tree*.[1] $\dashv$

We will often use the symbol $\lhd$ to refer to the relation R in the preceding definition. We choose infix notation for $\lhd$, so that $u \lhd v$ should be read as "v is an R-successor of u". Furthermore, in linguistic contexts, we will use the familiar genealogical terminology to refer to relations between nodes in a forest: for any nodes $u, v \in V$, if $u \lhd v$, the node v is a *child* of u, and, symmetrically, u is the *parent* of v; for a given node u, the set $(\lhd)^+u$ is called the set of *descendants* of u; the set $(\lhd^{-1})^+u$ is called the set of *ancestors* of u. The set $\lhd^*u$ is referred to as the *yield* of u.

Definition 2.3 (Subtree) Given a forest $(V; \lhd)$ and a node $u \in V$, the relational structure $(\lhd^*u; \lhd|_{\lhd^*u})$ is a tree, the *subtree rooted at* u. $\dashv$

Definition 2.4 (Disjointness) Let $\mathfrak{F} = (V; S)$ be a forest. The *disjointness* relation on $\mathfrak{F}$ is defined as $\perp := \mathrm{U} - \mathrm{I} - S^+ - (S^{-1})^+$.

Two nodes $v, w \in V$ are *disjoint at* $u \in V$, if and only if $\{v, w\} \subseteq S^*u$ and $\{v, w\} \nsubseteq S^*u'$ for all children u' of u. $\dashv$

2.3 Total orders

Total orders are, like forests, a kind of relational structure. They are, for example, suitable to represent the order among the words of a sentence.

Definition 2.5 (Total order) A binary relation R over a set V is said to be *trichotomic*, if and only if for all $a, b \in V$, exactly one of the following conditions holds: Rab, $a = b$, or Rba.

A relational structure $(V; R)$ is a *total order*, if R is trichotomic and transitive. $\dashv$

Orders are often defined as reflexive, antisymmetric and transitive relations in the literature. For our purpose, however, it is more convenient to exclude the reflexive elements from the relation. According to our definition, for example, $<$ (with its usual interpretation) instead of $\leq$ would be the canonical total order over the natural numbers.

In the context of total orders, we will often use the infix symbol $\prec$ for the respective order relation R. Furthermore, we will use the notation $a \preceq b$ as a shorthand for $a \prec b \vee a = b$.

Definition 2.6 (Interval and convex hull) Let $(V; \prec)$ be a total order. For nodes $a, b \in V$ with $a \preceq b$, the set $[a, b] := \{ x \in V \mid a \preceq x \wedge x \preceq b \}$ is the *interval* with endpoints a and b; the empty set is the *empty interval*. Two intervals $[a_1, b_1], [a_2, b_2]$ *overlap*, if and only if $a_1 \prec a_2 \prec b_1 \prec b_2$ or $a_2 \prec a_1 \prec b_2 \prec b_1$. For a given set $V' \subseteq V$ the *convex hull* of V', $\mathcal{H}(V')$, is the smallest interval containing V'. A set $V' \subseteq V$ is said to be *convex*, if and only if $V' = \mathcal{H}(V')$. $\dashv$

[1]In graph theory, the structures such defined are known as rooted trees.

Sets that are not convex have nodes between their elements that do not belong to the set. These nodes form *gaps* in the set. Later on, gaps will be relevant in connection with non-projective parts of dependency structures.

Definition 2.7 (Gap) Let $(V; \prec)$ be a total order. A *gap* in a set $V' \subseteq V$ is a maximal (with respect to set inclusion) non-empty convex set in $\overline{V'} := \mathcal{H}(V') - V'$. The number of gaps of V' is called the *gap degree* of V'. Two different gaps of V are always (set-wise) disjoint. They thus can be ordered from left to right according to $\prec$. The k-th gap of V' according to this order as denoted as $G_k(V')$. The elements of a gap are called *holes*. $\dashv$

Example Let $\Omega = (\{1, 2, 3, ..., 10\}; \prec)$ be the standard total order on the first ten natural numbers. The set $\{1, 3, 4, 7\}$ has the convex hull $[1, 7]$ and two gaps, $\{2\}$ and $\{5, 6\}$. Hence, its gap degree is two. $\dashv$

2.4 Ordered forests

Ordered forests are forests that are extended with a second relation: an order among disjoint nodes. Since only disjoint nodes are ordered, the order is partial.

Definition 2.8 (Partial order) A *partial order* is a relational structure $(V; R)$ where R is a binary relation on V that is transitive and acyclic. $\dashv$

Definition 2.9 (Ordered forest) An ordered forest is a relational structure $(V; \lhd, <)$, such that $(V; \lhd)$ forms a forest and $(V; <)$ is a partial order with $\forall v, w \in V$

I $v \perp w \Leftrightarrow (v < w \vee w < v)$

II $v < w \Rightarrow \lhd^+ v \times \lhd^+ w \subseteq <$ $\dashv$

In some of the following chapters we will need the following non-standard extension of ordered forests.

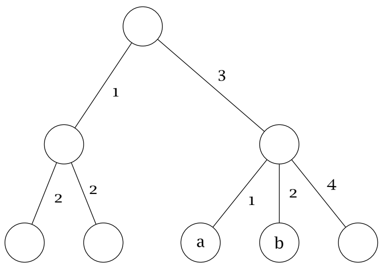
Definition 2.10 (Indexed ordered forest) An indexed ordered forest with k indices is an ordered forest $(V; \lhd, <)$ together with a function $\mathsf{index} : \lhd \to [1, k]$ such that

III $v \lhd w \wedge v \lhd w' \wedge \mathsf{index}(v, w) < \mathsf{index}(v, w') \Rightarrow w < w'$

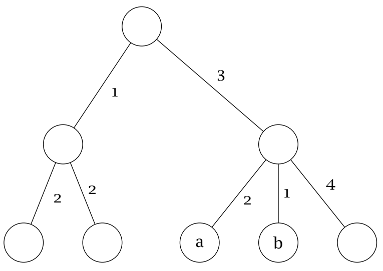
We will use $v \lhd_i w$ as an abbreviation for $v \lhd w \wedge \mathsf{index}(v, w) = i$. $\dashv$

Condition (I) states that the order is defined on all disjoint nodes, condition (II) ensures that the order of two nodes is inherited by their descendants. Equivalently, an ordered forest can be thought of as a forest where only the children of each node and the roots are ordered. All other elements of $<$ that are required in (I) can then be generated by the implication given in (II).

Condition (III) says that the outgoing edges of each node are ordered according to their index.



(a) valid indexing



(b) invalid indexing

Figure 2.2: The left forest has a valid indexing the right violates the condition at the incoming edges of nodes a and b

Example Figure 2.2 shows two indexed ordered forests where the first indexing is valid and the second violates the condition (III) for the indices of the incoming edges of nodes a and b: node a is left of b, but the index of a is larger than the one of b.

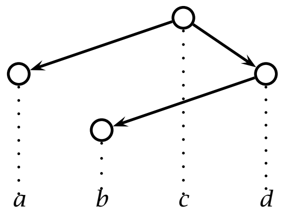
2.5 Drawings

A drawing is a relational structure that is suitable to represent the dependency analysis of a natural language sentence. It consists of two different relations: a tree structure representing dependencies among the words, and a total order representing word order.

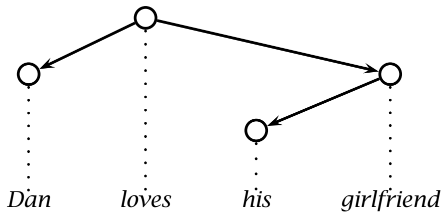
Definition 2.11 (Drawing) A *drawing* is a relational structure $(V; S, \prec)$ in which $(V; S)$ forms a tree, and $\prec$ is a total order on V. $\dashv$

We use the symbols S and $\prec$ for successorship and order in drawings, while we use $\triangleleft$ and $<$ for successorship and order in ordered trees.

Example Figure 2.3(a) shows how we visualise drawings throughout this thesis. The circles and solid edges reflect the tree structure underlying the drawing. The dotted lines



(a) A drawing



(b) A drawing that represents the depencency structure of a sentence

Figure 2.3: Sample drawings

mark the horizontal positions of the nodes with respect to the order in the drawing. The labels at the lower end of the dotted lines give names to the nodes.

Figure 2.3(b) shows how a drawing can represent the dependency structure of a sentence. The nodes of the drawing are the words of the sentence. The tree structure represents the dependencies among the words. For example, the outgoing edges of 'loves' indicate that both 'Dan' and 'girlfriend' depend on 'loves'. The order of the nodes corresponds to the order of the words in the sentence. ⊣

Note that there is an important difference between drawings and ordered trees: the order in drawings is total, whereas in ordered trees, only the order among disjoint nodes is specified (see condition (I) of Definition 2.9). In particular, the order among a node and its children is not specified in an ordered tree. Since the words in a sentence are totally ordered, and in a dependency analysis, each node corresponds to a word, drawings are the right models for the dependency analysis of natural language sentences.

Drawings are simple structures only encapsulating the core notions of a dependency analysis. They do *not* contain additional material specific for a certain grammar formalism, such as additional information that is required during the derivation process of that formalism. In that sense, drawings are a minimal common base of all grammar formalisms that use dependency structures to represent their analyses.

A major aspect of this thesis is the investigation of structural properties of drawings. In particular we identify several – purely structurally motivated – subclasses of drawings. The results gained in this investigation are independent of any grammar formalism, but can be transferred to each formalism that is based on dependency structures. For example, one could discover that the drawings derived by a certain formalism all belong to a certain subclass, or, in other words, all fulfil a certain structural constraint. The results also help to compare different formalisms, if, for example, the class of drawings derivable by one grammar formalism is a superclass of the drawings derivable by another.

The only commonly used structural property of dependency analysis is *projectivity*. It is of particular interest because, on the one hand, many grammar formalisms are only able to derive projective structures, on the other hand, there is evidence that the dependencies within natural language are not always projective.

Definition 2.12 (Projectivity) A drawing is *projective* if and only if all of its subtrees have convex yields. Drawings that are not projective are called *non-projective* drawings. ⊣

Example The drawing in Figure 2.3(b) is projective. The drawing in Figure 2.3(a) is not projective: the yield of the node d consists of the nodes b and d, but $\{b, d\}$ is not convex since $\{c\}$ is a gap in $\{b, d\}$. ⊣

Chapter 3

Gap degree

This chapter analyses drawings with gaps. First we explain what gaps in a drawing are, and introduce the notion of gap degree. Gap degree is a measure of how non-projective a drawing is.

After the introduction of gap degree we will first analyse how many gaps a drawing may have at most, second present an algorithm to compute the gap degree, and third see how gaps are related to derivation mechanisms in different grammar formalisms.

3.1 Drawings with gaps

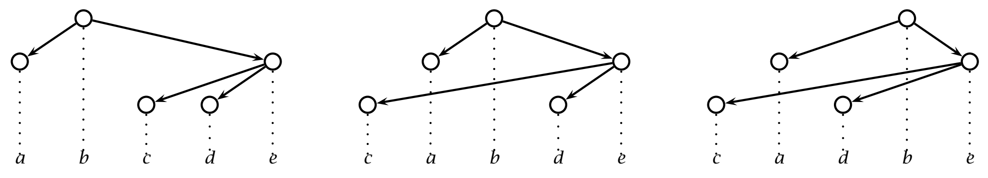
Non-projective drawings have subtrees whose yields are not convex, and therefore have gaps. It thus makes sense to lift the notion of gaps from total orders to drawings.

Definition 3.1 (Gap degree) Let $\mathfrak{D} = (V; S, \prec)$ be a drawing. The gaps of a node $v \in V$ are the gaps of S^*v. The gap degree of $\mathfrak{D}$ is the maximum among the gap degrees of the nodes of $\mathfrak{D}$.

$$(\mathsf{gap\ degree})(\mathfrak{D}) \;:=\; \max_{v \in V} (\mathsf{gap\ degree})(S^*v) \qquad \dashv$$

A gap of a node v may consist of two different kinds of nodes: nodes that dominate v and nodes that are disjoint to it. We call nodes of the first kind *parent holes*, and nodes of the second kind *disjoint holes*. Sometimes we refer to v itself as the *gap host*.

Note that a drawing $\mathfrak{D}$ is projective, if and only if $(\mathsf{gap\ degree})(\mathfrak{D}) = 0$. The gap degree can hence be interpreted as a scale of non-projectivity in the sense that a drawing with gap degree $n + 1$ is 'less projective' than a drawing with gap degree n.



(a) gap degree 0 (b) gap degree 1 (c) gap degree 2

Figure 3.1: Drawings with different gap degrees

Example Figure 3.1 shows three drawings with the same underlying tree structure, but different node orders and different gap degrees. The left drawing is projective and consequently has a gap degree of 0. In the two remaining drawings only the node e has gaps. In the first one it has one gap, $[a, b]$, in the second one it has two gaps, $[a]$ and $[b]$. The drawings thus have a gap degree of 1 and 2, respectively. In both drawings, a is a disjoint hole and b is a parent hole, since a is disjoint to the gap host e and b dominates it. $\dashv$

The gaps of a node v are always subsets of the convex hull of the yield of v. For convenience, we give a name to the convex hull of the yield of v:

Definition 3.2 (Cover) Let $\mathfrak{D} = (V; S, \prec)$ be a drawing. The cover of a node $v \in V$ is defined as $C(v) := \mathcal{H}(S^*v)$. $\dashv$

From this definition an alternative description for the gaps of a node v can be derived: the gaps of v are the maximal non-empty convex sets in $C(v) - S^*v = \overline{S^*v}$. Furthermore, a node has no gaps if its yield equals its cover.

Related work The term 'gap' and the intuition behind it is adopted from Plátek et al. [10]. In contrast to our approach, they define gaps with respect to a certain grammar formalism that generates tree structures. Our definition of gaps can be considered as a purely structural reconstruction of their intuition and makes the term 'gap' independent of any special grammar formalism.

In this section, we have defined gaps in a drawing, and the gap degree. In the remainder of this chapter, we look into certain details that are relevant for drawings with gaps. We start with investigating how many gaps a drawing may have at most.

3.2 The relation between the number of nodes and the number of gaps

In addition to the number of nodes of a drawing, the number of gaps is a second important factor that might influence the complexity of algorithms[1]. However, these two factors are not completely independent. Therefore, this section analyses their relationship. We will see that the tightest general bound on the number of gaps in a drawing with n nodes is $O(n^2)$ and that this can be narrowed to $O(n * \log(n))$ if we do not count identical gaps of different nodes twice.

3.2.1 The number of gaps in a drawing

The number of gaps in a drawing is in $\mathrm{O}(n^2)$:

[1]Note that the gap degree and the number of gaps of a drawing are not the same: The gap degree of a drawing is the maximum of the gap degrees of the nodes, while the number of gaps is the sum of them.

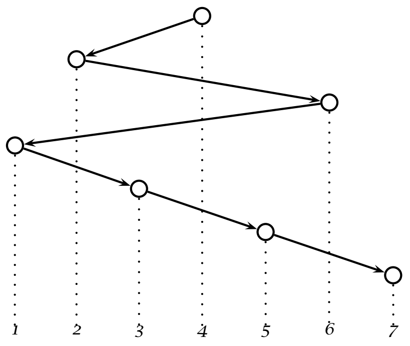


Figure 3.2: A drawing with quadratic number of gaps

Lemma 3.3 Let $\mathfrak{D} = (V; S; \prec)$ be a drawing and $|V| = n$. Then

$$\sum_{v \in V} (\mathsf{gap\ degree})(v) \le n^2$$

□

Proof Since two gaps must always be separated by an element of the yield, for each node $v \in V$ it holds that

$$(\mathsf{gap\ degree})(v) < |S^* v|.$$

With $|S^* v| \le n$ it follows that $(\mathsf{gap\ degree})(v) \le n$ and thus

$$\sum_{v \in V} (\mathsf{gap\ degree})(v) \le n^2$$

■

This upper bound of $O(n^2)$ gaps is tight. To show this, we present a family of drawings $\mathfrak{D}_k$ with $k > 0$, where each drawing $\mathfrak{D}_i$ has $n_i = 2^i - 1$ nodes and $O(n_i^2)$ gaps. The members of the family are defined by the following rules:

$$\begin{array}{rcl}
\mathfrak{D}_1 & := & (\{v\}; \emptyset, \emptyset) \\
\text{and for} & & \mathfrak{D}_k = (V_k; S_k, \prec_k) \text{ with } V_k = \{v_1, \dots, v_n\} \text{ and } v_1 \prec_k v_2 \prec_k \cdots \prec_k v_n \\
D_{k+1} & := & (V_{k+1}; S_{k+1}, \prec_{k+1}) \text{ with} \\
V_{k+1} & := & V_k \cup \{w_1, \dots, w_n, w_{n+1}\} \\
S_{k+1} & := & S_k \cup \{(v_n, w_1)\} \cup \{\, (w_i, w_{i+1}) \mid i \in \{1, \dots, n\}, \prec_{k+1} \} \\
\prec_{k+1} & s.t. & w_1 \prec_{k+1} v_1 \prec_{k+1} w_2 \prec_{k+1} v_2 \prec_{k+1} \cdots \prec_{k+1} v_n \prec_{k+1} w_{n+1}
\end{array}$$

As an example, the instance $\mathfrak{D}_3$ is shown in Figure 3.2. The instance $\mathfrak{D}_2$ is obtained if the nodes $1, 3, 5$ and 7 and the corresponding edges are removed from $\mathfrak{D}_3$.

For each instance, the tree structure just consists of one singe path from the root node to a single leaf node. This ensures that the sum of the yields of the nodes is $n + (n-1) + (n-2) + ... + 1 \in O(n^2)$. This is necessary, since the sum of the yields is an upper bound on the sum of the gaps (see the proof of Lemma 3.3). The order is chosen such that a maximum number of gaps is reached. The total number of gaps of a drawing $\mathfrak{D}_k$ is

$$0 + 1 + 2 + ... + (2^{k-1} - 2) + (2^{k-1} - 1) + (2^{k-1} - 2) + (2^{k-1} - 3) + ... + 1 + 0,$$

where the i-th component of the sum corresponds to the i-th node on the path from the root to the leaf. The sum can be reordered to

$$(0 + (2^{k-1} - 1)) + (1 + (2^{k-1} - 2)) + ... + ((2^{k-1} - 2) + 1)$$

$$= (2^{k-1} - 1) * (2^{k-1} - 1) = (2^{k-1} - 1)^2 = (\frac{1}{2}2^k - 1)^2$$

Since $\mathfrak{D}_k$ has $2^k - 1$ nodes, the number of gaps grows quadratically with the number of nodes.

3.2.2 The number of gap sets

If we look at the quadratic number of gaps of the drawings $\mathfrak{D}_k$, we observe that many of these gaps consist of the same nodes. The singleton set that consists only of the root of $\mathfrak{D}_3$ (see Figure 3.2), for example, is a gap of the nodes $1, 2, 3$ and 6. An algorithm that represents a gap just by the set of its nodes would represent these four gaps with just one element, or in other words would not distinguish between them. Regarding the complexity of such an algorithm, it is interesting to know the number of different gaps: the number of node sets that form a gap for some node. We will call these sets *gap sets*. Similar to the upper bound on the number of gaps, we now determine an upper bound on the number of gap sets of a drawing. To reach this goal, we first show some relevant lemmata.

Lemma 3.4 For each drawing $\mathfrak{D} = (V; S; \prec)$ the following holds:

$$|\bigcup_{v \in V} (\text{gap sets})(v)| = \sum_{(v,w) \in S} |(\text{gap sets})(w) - (\text{gap sets})(v)| \qquad \square$$

This means basically that if we are interested in the number of different gaps of a drawing, it is sufficient to count the gaps that differ from the gaps of its parent for each node.

Proof Root nodes have no gap sets, and hence it is sufficient to count only gaps of nodes that have a parent. With regard to these nodes, we must only take care to count each gap set exactly once. For each node w, it suffices to count each gap set, except the ones that are also gap sets of w's parent, since nodes that are disjoint to w will have no common gap sets with w, and any ancestor of v only has a common gap set with w if v also has this gap set. ∎

Lemma 3.5 Let $\mathfrak{D} = (V; S; \prec)$ be a drawing, $v, w \in V$ and $(v, w) \in S$. Then

$$|(\text{gap sets})(w) - (\text{gap sets})(v)| \le |S^*v - S^*w| \qquad \square$$

The Lemma states that the number of gaps of a node w that are not equal to any gaps of w's parent v is limited by the number of nodes in the yield of v that are not in the yield of w. Thus, not only $|S^*w|$, but also $|S^*v - S^*w|$ is an upper bound to the number of w's gaps, if we do not want to count identical gaps twice.

Proof Each gap of w must contain at least one node from S^*v if it should not also be a gap of v. Since a gap of w cannot contain nodes from S^*w, the node must be from $S^*v - S^*w$. Since two gaps of w cannot have a common node, $|S^*v - S^*w|$ is an upper bound on the gaps of w that are no gaps of v. ■

Now we have all the necessary details to give an upper bound on the number of gap sets:

Lemma 3.6 Let $\mathfrak{D} = (V; S; \prec)$ be a drawing and $|V| = n$. Then

$$|\bigcup_{v \in V} (\text{gap sets})(v)| \le n * \log(n) \qquad \square$$

Proof Due to Lemma 3.4 the following holds:

$$|\bigcup_{v \in V} (\text{gap sets})(v)| = \sum_{(v,w) \in S} |(\text{gap sets})(w) - (\text{gap sets})(v)|$$

A first bound on this is

$$\sum_{(v,w) \in S} |(\text{gap sets})(w) - (\text{gap sets})(v)| \le \sum_{(v,w) \in S} |(\text{gap sets})(w)| < \sum_{(v,w) \in S} |S^*w|,$$

and a second bound is given by Lemma 3.5:

$$\sum_{(v,w) \in S} |(\text{gap sets})(w) - (\text{gap sets})(v)| \le \sum_{(v,w) \in S} |S^*v - S^*w|.$$

If we increase $|S^*w|$ for any node w, the first bound will get weaker and the second bound will get stronger. If we decrease $|S^*w|$, it will be the other way around. The maximum is thus reached when both bounds are equal, that is $|S^*w| = |S^*v - S^*w|$. The optimum is consequently a balanced binary tree. In such a balanced tree, the following holds:

$$\sum_{(v,w) \in S} |S^*w| \le n * \log(n)$$

The statement holds since the yields of nodes with a common height in the tree are disjoint, and thus all together they cover no more than n elements. A balanced tree with n nodes has a height of $\log(n)$. If each height level contributes at most n to the sum, the sum of all yields is bounded by $n * \log(n)$. ■

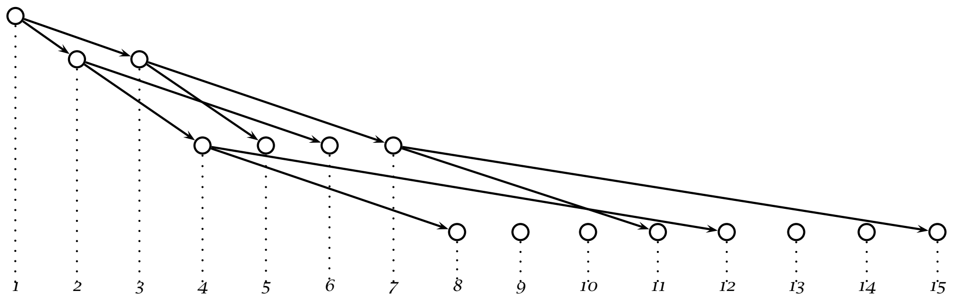


Figure 3.3: A drawing with $O(n * \log(n))$ different gaps (some edges omitted to maintain readability)

This bound is asymptotically tight as the following family $\mathfrak{D}'_k$, $k > 0$, of drawings shows:

$$\mathfrak{D}'_k := (\{v_1, v_2, \dots, v_{2^k-1}\}; S, \prec) \text{ with}$$
$$v_1 \prec v_2 \prec \cdots \prec v_{2^k-1}$$
$$(v_i, v_j) \in S \iff i = 2^m + r \text{ with } r < 2^m \wedge (j = 2^{m+1} + r \vee j = 2^{m+1} + i)$$

As an example, $\mathfrak{D}'_4$ is shown in Figure 3.3; instance $\mathfrak{D}'_3$ is obtained by removing nodes 8 to 15 and their incoming edges.

The order of the nodes is such that between two nodes of each yield another node is placed. Each node v, except the root, then has $|S^* v| - 1$ gaps. The sum of the yields of all nodes with some height h in the tree is at least $\frac{1}{2} * n$, the nodes of height h hence contribute $O(\frac{1}{2} * n) = O(n)$ gaps. Thus, overall, there are $O(n * \log(n))$ gaps for all $\log(n)$ different heights together. Furthermore, gaps of nodes always differ from the gaps of their parents: each child combines two of the gaps of its parent into another gap.

3.2.3 Consequences for algorithms

We have seen that a drawing with n nodes in the worst case has n^2 gaps. Any algorithm that does some computation with the gaps and treats each gap separately hence cannot have a better worst case complexity than $O(n^2)$. The only chance to achieve a better worst case complexity is to treat several gaps together in one operation. One possible way is to consider the gap sets instead of the gaps itself, such that all gaps consisting of the same gap set are treated together in one step. With this idea, the complexity can be reduced to $O(n * \log(n))$.

3.3 An algorithm to compute the gap degree

The gap degree can be computed in a simple bottom-up traversal over the tree structure of the drawing. At each node u, first u's yield is computed, and then the number of its

$x \wedge y$	bitwise and
$x \vee y$	bitwise or
$x = y$	equality test
$\sim x$	bitwise inverse
$x << n$	bitshift n bits to the left
$\mathsf{RBound}(x)$	Most-significant bit in x
$\mathsf{LBound}(x)$	Least-significant bit in x
$\mathsf{Singleton}(p)$	BV in which only position p is set to 1
$\mathsf{Prefix}(p)$	BV covering all positions $\leq p$
$\mathsf{Suffix}(p)$	BV covering all positions $\geq p$

Table 3.1: Operations on bit vectors

gaps. This number of gaps is compared with the numbers computed in the recursive calls for each of u's children, and the maximum is passed upwards in the tree. The value for the root is then the gap degree of the drawing.

One efficient implementation uses bitvectors to represent the yields of the nodes: the bits of such a vector correspond to the nodes of the drawing. A bit set to one means that the corresponding node is contained in the respective yield. The necessary operations on bit vectors are shown in Table 3.1. The first five operations are atomic. The remaining ones can be computed in constant time, as shown by Daniels and Meurers [4].

Using bitvectors, the yield of a node can be computed as the disjunction of the yields of its children, and the singleton vector where only the bit of the node itself is set to one. Since the yield of each node is involved only in the computation of the yield of its parent, there are $O(n)$ disjunctions necessary to compute the n yields of all nodes.

The number of gaps in a yield represented by a bitvector Y is then computed as follows:

```
1: count := 0
2: $X := Y \wedge \sim (Y << 1)$
3: **while** $X \neq$ zero-vector **do**
4:     count := count + 1
5:     $X := X \wedge (\sim \mathsf{RBound}(X))$
6: **end while**
7: return count − 1
```

In line 2 a bitvector X is computed, such that in X only the bit immediately left of each gap, and the last bit of the yield Y are set to one. In each iteration of the loop, one of these bits is removed and counted, until all bits are set to zero. If Y has g gaps, the while loop is iterated $g + 1$ times, and since all other operations need constant time, the computation of g gaps of Y costs $O(g)$ time. Consequently, the time that the algorithm needs to count all gaps in the drawing equals the number of gaps in the drawing. This is bounded by n^2 and even tighter by $n * (\text{gap degree})(\mathfrak{D})$.

Summarising all costs, we have $O(n)$ time for the computation of all yields and $O(n * (\text{gap degree})(\mathfrak{D}))$ for the computation of the gap degrees of all nodes. For projective drawings ($(\text{gap degree})(\mathfrak{D}) = 0$) the complexity hence is $O(n)$; for non-projective drawings the complexity is $O(n * (\text{gap degree})(\mathfrak{D})) \leq O(n^2)$.

The algorithm can be modified such that for a given constant k, it decides whether the given drawing has a gap degree of at most k or not. In this case, the traversal can be aborted as soon as any visited node has more than k gaps. The complexity is then $O(k * n)$; deciding if a drawing has a gap degree of at most 2 for example needs $O(2 * n) = O(n)$ time.

3.4 Gap restrictions in grammar formalisms

In this section, we will analyse how far different grammar formalisms are able to derive a dependency analysis that contains gaps. We first show how strictly lexicalised Context-Free Grammars induce drawings, and argue why these drawings are always projective. Then we show how strictly lexicalised *Multiple Context-Free Grammars* are able to derive drawings with gaps.

Context-Free Grammar Since we want to show how grammar formalisms are able to derive drawings with gaps, we first have to state more precisely how the derivation in a certain grammar formalism corresponds to a drawing. First we explain how a derivation within a strictly lexicalised Context-Free Grammar can be interpreted as a drawing. A Context-Free Grammar is called strictly lexicalised, if each production rule has exactly one terminal symbol on its right-hand side. The terminal symbols are the nodes of the drawing. Due to the lexicalisation, each node of the derivation tree has one corresponding terminal symbol. The derivation tree builds the tree structure of the drawing, the order among the nodes is the order among the terminals in the derived string.

The drawing in Figure 3.1(a) can, for example, be derived by a grammar with non-terminal symbols $\{A, B, C, D, E\}$, terminal symbols $\{a, b, c, d, e\}$, start symbol B and the following production rules:

$$p_1 : B \to AbE \qquad p_2 : E \to CDe \qquad p_3 : A \to a \qquad p_4 : C \to c \qquad p_5 : D \to d$$

From the perspective of a drawing, each of these rules describes the adjacencies of one node. Rule p_1, for example, states that the node b has two children: one is the root of a subtree of type A and the other one is the root of a subtree of type E. Furthermore, the order among these two subtrees and the node b is specified: the subtree of type A is left of b, the one of type E right of it. Rule p_2 states that node e is the root of a subtree of type E and that it has two children whose corresponding subtrees have types C and D, respectively. Here, the subtree of type C is to the left of the subtree of type D, and e is to the right of both of them.

Multiple Context-Free Grammar The drawing in Figure 3.1(b) cannot be derived by a Context-Free Grammar. This is due to the fact that a subderivation within a Context-Free Grammar always contributes a convex substring of the finally derived string. This implies for the corresponding drawing, that each subtree may only contribute a convex set of nodes to the node set of the drawing. Since the yields of nodes with gaps are not convex, Context-Free Grammars are only able to derive projective drawings. In Figure 3.1(b), for example, the yield of node e consists of two convex subsets, $\{c\}$ and $\{d, e\}$, but a context-free rule like p_2 does not allow any external material to be inserted between C and D.

One formalism that allows a subderivation to contribute several convex substrings to the derived string is Multiple Context-Free Grammar[2] [11]. Here, a nonterminal X may consist of several parts $X_1, \ldots, X_k$, and each of them contributes a convex part of the derived string. The derivations for the different parts $X_1, \ldots, X_k$ are carried out in parallel; the left- and right-hand sides of the rules consist of tuples to account for this parallelism. In a Multiple Context-Free Grammar, the rules p_1 and p_2 can be replaced with

$$p_1' : B \to E_1 A b E_2 \qquad p_2' : (E_1, E_2) \to (C, De)$$

With these additional rules, the drawing in Figure 3.1(b) is derivable. The rule p_1' says that b has two children whose subtrees have type A and E, respectively, and a part of the subtree of type E is left of the subtree of type A, and a second part is right of b. Rule p_2' then specifies that C is contained in the first part of the subtree of type E, and D and e are contained in the second part. By splitting up a subtree into several parts, it is possible to place gaps between these parts. The arity of the tuples indicates the gap degree: In a 2-tuple there is only one position for a gap (between the two tuple components) and hence the nodes whose corresponding production rules consist of 2-tuples have a gap degree of at most one. In order to derive the drawing in Figure 3.1(c), which has a gap degree of two, we need a rule with 3-tuples:

$$p_1'' : B \to E_1 A E_2 b E_3 \qquad p_2'' : (E_1, E_2, E_3) \to (C, D, e)$$

Summarising, we can say that a grammar formalism must allow a subderivation to contribute non-convex parts of the derived string if the corresponding drawing should have gaps. Multiple Context-Free Grammar is a formalism that allows this with the help of several parallel derivations that are all together represented in one production rule. Multiple Context-Free Grammar is just one among many tuple-based extensions of Context-Free Grammar. The ideas presented in this section are similarly applicable

[2]Sometimes the formalism meant here is also called *Linear* Multiple Context-Free Grammar

for the more general class of Linear Context-Free Rewriting Systems which, for example, subsumes Head Grammars, Tree Adjoining Grammars and Multicomponent Tree Adjoining Grammars [12].

Chapter 4

Well-nestedness

4.1 Well-nested drawings

In the last chapter, we have presented the gap degree as a relaxation of projectivity. In this chapter, we introduce well-nestedness, a second relaxation of projectivity. Well-nestedness is orthogonal to the gap degree in the sense that for each gap degree there exist both drawings that are well-nested, and drawings that are not.

Later on, we will show that drawings that are induced by TAG derivations are always well-nested. In the same manner as the gap degree, well-nestedness originates from a property of ordered sets that can be generalised to drawings.

Definition 4.1 (Interleaving sets) Let $(V; \prec)$ be a total order. Two sets $V_1, V_2 \subseteq V$ *interleave*, if and only if there are nodes $l_1, r_1 \in V_1$ and $l_2, r_2 \in V_2$ such that $[l_1, r_1]$ and $[l_2, r_2]$ overlap. We call l_1, l_2, r_1 and r_2 the *witnessing nodes*. $\dashv$

Definition 4.2 (Well-nestedness) Let T_1 and T_2 be subtrees in a drawing $\mathfrak{D} = (V; S, \prec)$, and let the node sets of T_1 and T_2 be disjoint. We say that T_1 and T_2 *interleave*, if and only if the node sets of T_1 and T_2 interleave. $\mathfrak{D}$ is *well-nested*, if and only if it does not contain any interleaving subtrees. $\dashv$

Figure 4.1 shows examples for interleaving and non-interleaving subtrees. The subtrees shown in Figure 4.1(a) are placed side by side. In projective drawings, subtrees are always placed side by side, but well-nested drawings also allow the alternative shown in Figure 4.1(b): subtrees are also non-interleaving, if one is placed within a gap of the other. Subtrees interleave if a part of one subtree is contained in a gap of the other, and another part is located outside this gap. This is shown in Figure 4.1(c), where l_2 is in the first gap of l_1, but r_2 is outside. Summarising, one can say that in projective drawings,

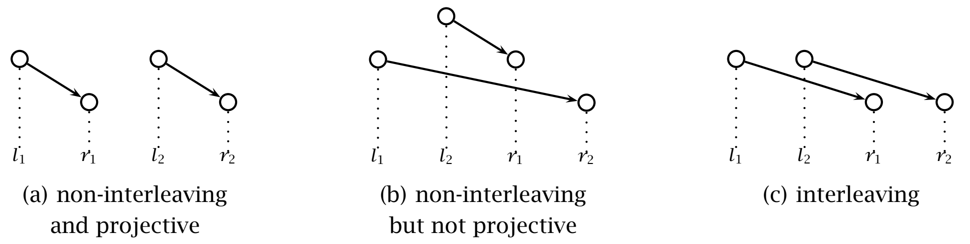
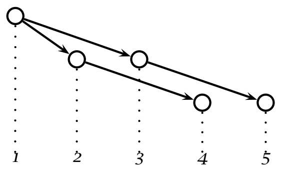


Figure 4.1: Examples for non-interleaving and interleaving subtrees



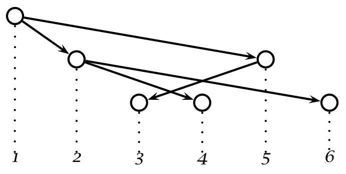


Figure 4.2: Drawings that are not well-nested

subtrees are always placed side by side, and in well-nested drawings it is also allowed to place one subtree in the gap of another one.

An alternative characterisation of interleaving subtrees T_1 and T_2 is that at least one node from T_1 is a hole in T_2 and the other way around: witnessing nodes l_1, r_1 from T_1 and l_2, r_2 from T_2 always correspond to such gaps because with $l_1 \prec l_2 \prec r_1 \prec r_2$ the node l_2 is a hole in T_1 and r_1 is a hole in T_2.

Example The drawings in Figure 3.1 on page 9 are all well-nested; the drawings in Figure 4.2 are not well-nested.

In the left drawing in Figure 4.2, the subtree rooted at node 2 interleaves with the subtree rooted at 3, since $[2,4]$ overlaps with $[3,5]$ or from the alternative perspective since $3 \in S^*3$ is a gap of 2 and $4 \in S^*2$ is a gap of 3.

The right drawing in Figure 4.2 shows that it is not sufficient to check whether the largest intervals within two subtrees (that is their covers) overlap. Although the covers of 2 and 5 do not overlap, their subtrees contain overlapping intervals: $[2,4]$ overlaps with $[3,5]$. The intuitive argument, why this drawing is not well-nested, is that 3 is contained in the first gap of 2, but 5 is not. In this case 5 is also located in a gap of 2, but not in the same gap as 3. $\dashv$

4.2 Algorithms to test well-nestedness

In this section, we present two different algorithms that decide whether a given drawing is well-nested or not.

4.2.1 Optimising a naïve approach

The naïve algorithm that decides if a drawing is well-nested, tests for each pair of subtrees, whether they interleave. Since a drawing with n nodes has also n different subtrees, this naïve algorithm performs $O(n^2)$ tests. Each single test involves the search for witnessing nodes which, naïvely implemented, increases the complexity even more. Based on this naïve approach, an efficient algorithm can be obtained by first minimising the number of tests that are carried out, and second finding an efficient method to perform every single test.

The algorithm that is presented in the following only needs constant time for each interleave test, but in the worst case still performs $O(n^2)$ of these tests. However, in most cases the number of tests is much lower, in particular for drawings where the outdegree of the nodes is bounded by some constant k, only $O(k^2 * n)$ tests are performed. For a binary tree the algorithm, for example, performs $O(4 * n)$ tests.

The following lemma says that it is not necessary to check for all pairs of subtrees whether they interleave or not. It is sufficient to check whether the subtrees of sibling nodes interleave:

Lemma 4.3 For a drawing $\mathfrak{D} = (V; S, \prec)$ it holds that if there are two nodes v, w with interleaving subtrees, then there also exist two *sibling* nodes v' and w' whose subtrees interleave. □

PROOF Let $l_1, r_1 \in S^*v$ and $l_2, r_2 \in S^*w$ be the witnessing nodes of the interleaving subtrees rooted at v and w. Let v and w be disjoint at u. Then u has two children v' and w' with $v \in S^*v'$ $w \in S * w'$. Then also $l_1, r_2 \in S^*v'$ and $l_2, r_2 \in S^*w'$. Consequently the subtrees rooted at v' and w' interleave with witnessing nodes l_1, l_2, r_1, r_2. ■

Due to this lemma, the algorithm only needs to test the subtrees of sibling nodes for interleaving. All other tests for interleaving subtrees would be redundant. The next lemma shows how each of these tests can be performed efficiently.

Lemma 4.4 Let $\mathfrak{D} = (V; S; \prec)$ be a drawing and let $v_1, v_2 \in V$ be two disjoint nodes. Then the subtrees rooted at v_1 and v_2 interleave, if and only if $C(v_1) \cap S^*v_2 \neq \emptyset$ and $C(v_2) \cap S^*v_1 \neq \emptyset$. □

PROOF Since v_1 and v_2 are disjoint, $C(v_1) \cap S^*v_2 \neq \emptyset$ implies that some node in the yield of v_2 is a hole in $C(v_1)$. Symmetrically $C(v_2) \cap S^*v_1 \neq \emptyset$ implies that some node in the yield of v_1 is a hole in $C(v_2)$. Consequently the two subtrees interleave. ■

The algorithm works as follows. In a bottom-up traversal through the tree structure of the drawing, for each node u we compute $C(u)$ and S^*u, and then check for each pair v_1, v_2 of u's children whether the conditions $C(v_1) \cap S^*v_2 \neq \emptyset$ and $C(v_2) \cap S^*v_1 \neq \emptyset$ are fulfilled. If we find such a pair, we stop and report that the drawing is not well-nested; if the traversal reaches the root without finding one, the drawing is well-nested.

Correctness and complexity The correctness of the algorithm follows directly from Lemmata 4.3 and 4.4. The complexity depends on the implementation.

In an efficient implementation, we represent the yields and covers as bitvectors and use the operations of Table 3.1 in Section 3.3. As in the algorithm used to compute the gap degree, the computation of all n yields can then altogether be done in $O(n)$ time. Furthermore, the cover of a node can be computed in constant time once its yield Y is computed:

$$C(Y) := \mathsf{Suffix}(\mathsf{LBound}(Y)) \wedge \mathsf{Prefix}(\mathsf{RBound}(Y))$$

Also the tests $C(v_1) \cap S^*v_2 \neq \emptyset$ and $C(v_2) \cap S^*v_1 \neq \emptyset$ can be done in constant time, since only conjunction and equality test are necessary. The complexity thus depends just on the number of tests that must be carried out. Since only nodes with a common parent node are tested, the worst case occurs when all nodes (except the root itself) have the same parent[1]. In this case $(n-1)^2$ tests must be carried out, and the worst case complexity of the algorithm is thus $O(n^2)$.

However, if we consider only drawings where the outdegree of the nodes is bounded by some constant k, at most k^2 tests must be carried out among the children of each node. Overall less than $k^2 * n$ tests are then necessary and the complexity is reduced to $O(k^2 * n) = O(n)$.

4.2.2 An alternative test for well-nestedness

Now we present a second algorithm that checks whether a drawing is well-nested or not. This algorithm offers an alternative characterisation of well-nestedness, which is based on *gap graphs*. The gap graph for a drawing $\mathfrak{D}$ makes the relation between gaps and their gap hosts explicit.

Definition 4.5 Let $\mathfrak{D} = (V; S, \prec)$ be a drawing. The *gap graph* for $\mathfrak{D}$ is the graph $\mathcal{G}(\mathfrak{D}) := (W, E)$ with $W = V$ and $E = S \cup \{\, (v, w) \mid v \in \overline{S^*u} \wedge u \perp v \,\}$ $\dashv$

In other words, $\mathcal{G}(\mathfrak{D})$ is an extension of the tree relation underlying $\mathfrak{D}$ by additional edges between (disjoint[2]) holes and their gap hosts. It will be convenient to have a special name to refer to these additional edges; we will call them *gap edges*.

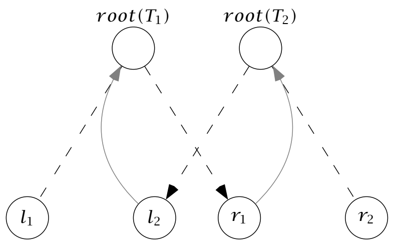
Given a gap graph for a drawing, the question of whether this drawing is well-nested can be answered in time linear in the number of nodes of the drawing. However, since a drawing may have a quadratic number of gaps (as we have shown in Chapter 3) the gap graph may have a quadratic number of edges, which makes its construction quite costly. The algorithm presented here might therefore be interesting, for example, in scenarios where a drawing is constructed or modified step by step, and well-nestedness must be checked frequently during this process. The gap graph can then easily be adopted each time the drawing is modified, and each single well-nestedness test is cheap.

The algorithm is based on the following theorem.

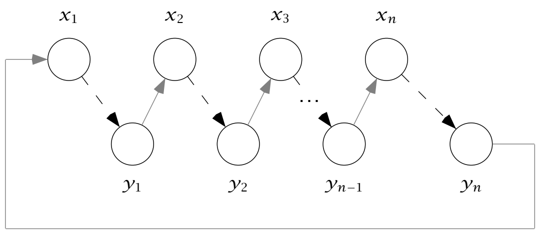
Theorem 4.6 Let $\mathfrak{D}$ be a drawing, and let $\mathcal{G}(\mathfrak{D})$ be the gap graph for $\mathfrak{D}$. Then $\mathfrak{D}$ is well-nested if and only if $\mathcal{G}(\mathfrak{D})$ is acyclic. □

[1] If all nodes except the root have the same parent, the drawing is always well-nested, since each subtree consists of only one node and cannot interleave with any other subtree. A more practical worst case would be that each of these subtrees consists of two nodes. Then the root has still $O(\frac{1}{2} * n) = O(n)$ children and their subtrees possibly interleave.

[2] It suffices to consider only additional edges from disjoint holes, since the algorithm we present is only concerned with transitive dominance in the gap graph and additional edges from parent holes are always subsumed by a path in the tree structure.



(a) The cycle of interleaving subtrees



(b) Normal form of a cycle in a gap graph

Figure 4.3: Situation in the proofs of Lemmata 4.7 and 4.8. Gray edges denote gap edges, dotted black edges denote sequences of tree edges.

Since the existence of a cycle in a graph with n nodes can be checked in $O(n)$, for a given gap graph, well-nestedness can also be checked in linear time. The theorem follows directly from Lemmata 4.7 and 4.9 below.

Lemma 4.7 Let $\mathfrak{D}$ be a drawing, and $\mathcal{G}(\mathfrak{D})$ be the gap graph for $\mathfrak{D}$. If $\mathfrak{D}$ is not well-nested, $\mathcal{G}(\mathfrak{D})$ contains a cycle. □

Proof If $\mathfrak{D}$ is not well-nested, we have the following situation: $\mathfrak{D}$ contains two disjoint subtrees T_1, T_2 with nodes $l_1, r_1 \in T_1$ and $l_2, r_2 \in T_2$ and without loss of generality $l_1 \prec l_2 \prec r_1 \prec r_2$. Then l_2 is in a gap of the root of T_1 and r_1 is in a gap of the root of T_2. The corresponding gap edges $(l_2, \mathit{root}(T_1))$ and $(r_1, \mathit{root}(T_2))$ together with the paths from $\mathit{root}(T_1)$ to r_1 and from $\mathit{root}(T_2)$ to l_2 build a cycle as shown in Figure 4.3(a). ■

To prove the other direction of Theorem 4.6, we first prove

Lemma 4.8 If a gap graph contains a cycle, it also contains a cycle in which all nodes reached by a gap edge are pairwise disjoint. □

Proof Each cycle c in a gap graph G contains a sub-cycle c' of the form depicted in Figure 4.3(b). Assume that there are two distinct nodes x_i and x_j in c' that are both reached by a gap edge, but are not disjoint. More specifically, without loss of generality, assume that $x_i \lhd^* x_j$. The graph G then contains a path p from x_i to x_j using tree edges only. We can therefore find a new cycle d in G that is smaller than c' in the sense that it contains at least one gap edge less than c'. Since each application of this procedure eliminates a non-disjointness link between two nodes reached by a gap edge, it will eventually yield a cycle in which all nodes reached by a gap edge are pairwise disjoint.■

Lemma 4.9 Let $\mathfrak{D}$ be a drawing, and let $\mathcal{G}(\mathfrak{D})$ be the gap graph for $\mathfrak{D}$. If $\mathcal{G}(\mathfrak{D})$ contains a cycle, $\mathfrak{D}$ cannot be well-nested. □

Proof by contradiction. Let $\mathfrak{D}$ be a well-nested drawing, and let $\mathcal{G}(\mathfrak{D})$ be a gap graph for $\mathfrak{D}$ containing a cycle. By Lemma 4.8, we may assume that the cycle in $\mathcal{G}(\mathfrak{D})$ has the form depicted in Figure 4.3(b), where all of the x_i are pairwise disjoint. Each path $x \cdots yx'$ in the cycle translates into the requirement that $C(x) \subset C(x')$: y is a hole in the yield of x' because of the gap edge between y and x', and since x and x' are disjoint, the assumed well-nestedness of $\mathfrak{D}$ tells us that y together with all other elements of $C(x)$ must form a convex proper subset of $C(x')$ because otherwise the subtree rooted at x interleaves with the subtree rooted at x'. By transitivity of set inclusion, we thus arrive at the requirement that $C(x) \subset C(x)$, which is a contradiction. ∎

4.3 The gap forest

In this section, we describe a structure called gap forest. The existence of gap forests is a major property of well-nested drawings, and offers a deeper insight into their nature. We first introduce the notion of a gap forest independently of drawings, and in the second step show that the nodes of drawings have corresponding gap forests.

4.3.1 Gap forests in sets

The basic idea behind gap forests is that we can represent the relations among different subsets of an ordered set as a relational structure or graph. Doing this we identify each subset with a node in the graph, and consider two binary relations on these nodes: the 'being in the gap' relation among the sets, and an order among them which is simply the order of their nodes lifted to sets. This order is partial and defined only for pairs of sets where all elements of the one set are before all elements of the second. Under certain conditions this structure is an ordered forest.

Definition 4.10 Let $(V; \prec)$ be a total order and $V_1, \dots, V_k$ disjoint, non-interleaving subsets of V, and let Γ be a bijection from some set W to $\{V_1, \dots, V_k\}$. Then the corresponding gap forest is the relational structure $(W; \triangleleft, <)$ with

$$\begin{aligned}
\triangleleft^+ &:= \{ (v, w) \mid v, w \in W \wedge \overline{\Gamma(v)} \supseteq \Gamma(w) \} \\
< &:= \{ (v, w) \mid v, w \in W \wedge \Gamma(v) \times \Gamma(w) \subseteq \prec \} \\
\triangleleft &:= \mathsf{transitive-reduction}(\triangleleft^+)
\end{aligned}$$

and a function *index* which assigns to each edge $(v, w) \in \triangleleft$ an index i indicating that $\Gamma(w)$ is located in the i-th gap of $\Gamma(v)$. Note that $\Gamma(w)$ is located in exactly one gap of $\Gamma(v)$, since the sets are non-interleaving.

Furthermore, the transitive reduction of $\triangleleft^+$ is unique since $\triangleleft^+$ is acyclic (see the first paragraph of Section 2.1). ⊣

Lemma 4.11 The gap forest is an indexed ordered forest. □

Proof Collecting all the properties in the definition of a forest (Definition 2.2), an ordered forest (Definition 2.9), and an indexed ordered forest (Definition 2.10), we have to show that (a) $(W; \lhd)$ is acyclic and each node has at most one $\lhd$ predecessor; (b) $<$ is transitive and acyclic and, furthermore, satisfies the conditions (I) and (II) of the definition of ordered forests (Definition 2.9); (c) condition (III) of the definition of an indexed ordered forest (Definition 2.10) is satisfied.

(a) Because $\supset$ is acyclic, also $\lhd^+$ and, furthermore, $\lhd$ is acyclic. To show that each node has at most one predecessor, consider a node w with $v \lhd^+ w$ and $v' \lhd^+ w$ and $v \neq v'$. Then $\Gamma(w)$ is in a gap of both $\Gamma(v)$ and $\Gamma(v')$. Since $\Gamma(v)$ and $\Gamma(v')$ do not interleave and are disjoint, one of them must be in the gap of the other. It follows that either $v \lhd^+ v'$ or $v' \lhd^+ v$, and hence either (v, w) or (v', w) is not contained in the transitive reduction $\lhd$.

(b) $<$ inherits the transitivity and acyclicity directly from $\prec$.

Condition (I) is satisfied:

The first equivalence holds since $\Gamma(v)$ and $\Gamma(w)$ are disjoint and non-interleaving.

$$\begin{array}{lll} & v \perp w & \\ \iff & C(\Gamma(v)) \cap C(\Gamma(w)) = \emptyset & \\ \iff & C(\Gamma(v)) \times C(\Gamma(w)) \subseteq \prec \;\vee\; C(\Gamma(w)) \times C(\Gamma(v)) \subseteq \prec & \text{(covers are convex)} \\ \iff & \Gamma(v) \times \Gamma(w) \subseteq \prec \;\vee\; \Gamma(w) \times \Gamma(v) \subseteq \prec & \\ \iff & v < w \vee w < v & \text{(definition of } <\text{)} \end{array}$$

Condition (II) is satisfied:

$$\begin{array}{llll} v < w & \Rightarrow & \Gamma(v) \times \Gamma(w) \subseteq \prec & \text{(definition of } <\text{)} \\ & \Rightarrow & C(\Gamma(v)) \times C(\Gamma(w)) \subseteq \prec & \\ & \Rightarrow & \lhd^+ v \times \lhd^+ w \subseteq < & (\text{since } \forall v' \in \lhd^+ v . \Gamma(v') \subseteq C(\Gamma(v))\ ) \end{array}$$

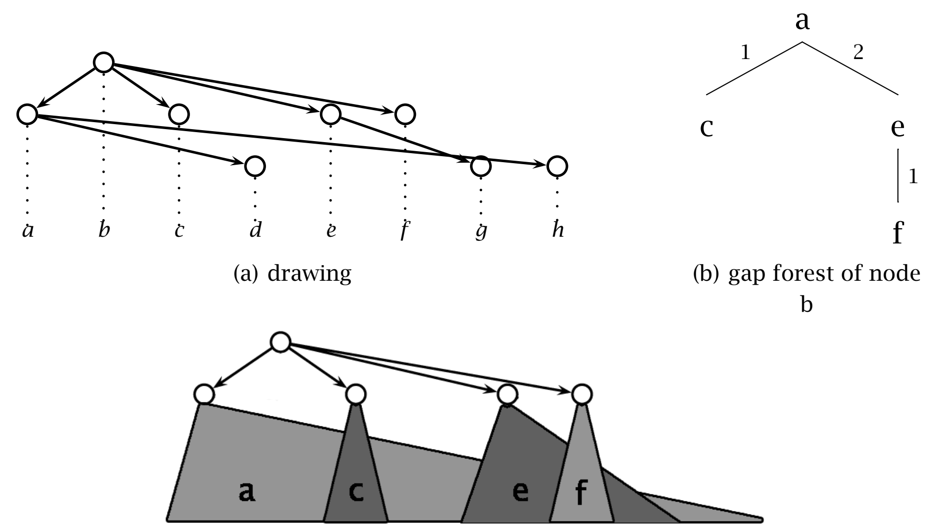
(c) Condition (III) is satisfied: if a node v has two children w, w' and $\mathit{index}(v, w) = i$, $\mathit{index}(v, w') = j$ and $i < j$ then $\Gamma(v)$ has $\Gamma(w)$ in its i-th gap and $\Gamma(w')$ in its j-th gap. With $i < j$ all elements of $\Gamma(w)$ are left of all elements of $\Gamma(w)$, and consequently $w < w'$.

■

4.3.2 Gap forests in drawings

So far we have defined gap forests for arbitrary sets, only with the restriction that these sets are disjoint and do not interleave. For a node v of a well-nested drawing the yields of v's children are sets which fulfil these requirements: obviously the yields are disjoint, and since the drawing is well-nested, they do not interleave.

Definition 4.12 (Gap forest) Let $\mathfrak{D} = (V; S, \prec)$ be a well-nested drawing and $v \in V$. The gap forest of v is the gap forest $(Sv; \lhd, <)$ with $\forall w \in Sv . \Gamma(w) = C(w)$. ⊣



(a) drawing

(b) gap forest of node b

(c) relative positions of the children of b

Figure 4.4: A drawing together with the gap forest of node b

The gap forest of a node v represents the relation among the subtrees that were spanned from v's children: If one child w dominates another child w' in the gap forest, then the whole yield of w' is located in a gap of w, and the index of the first edge on the path from w to w' indicates which gap it is exactly. If on the other hand w and w' are disjoint in the gap forest, the order among them is identical to the order of their yields in the drawing.

Example Figure 4.4 shows a drawing together with the gap forest of its root node b. Since the gap forest of b represents the relative positions of the subtrees of b's children, these relative positions are visualised in Figure 4.4(c). Node b has four nodes in its gap forest: a, c, e and f. The outgoing edges from node a indicate that the subtree rooted at c is contained in the first gap of a and the subtrees rooted at e and f are contained in the second gap of it. Furthermore, the subtree rooted at f is contained in the first gap of e.

The gap forests of the other nodes are much simpler: node a has two children, its gap forest hence consists of two nodes, d and h where d is left of h because $C(d) \times C(h) = \{(d, h)\} \subseteq \prec$. The gap forest of node e only consists of one node, the gap forests of the leaves of the drawing have empty node sets.

Note that the gap forest for a node v only exists if the yields of v's children do not interleave. This is always the case in well-nested drawings, but drawings which are not well-nested always have sibling nodes whose yields interleave (see Lemma 4.3). This implies that a drawing is well-nested if and only if corresponding gap forests exist for all of its nodes.

4.4 Gap sets in well-nested drawings

In Section 3.2, we have shown that a drawing has at most $O(n^2)$ gaps, and at most $O(n * \log(n))$ different gaps or gap sets. The worst case example for the $O(n^2)$ bound is well-nested, but the worst case example that we presented for the $O(n * \log(n))$ gap sets (see Figure 3.3) is not. In fact, the question whether we can also find a family of well-nested drawings with $O(n * \log(n))$ gap sets must be answered in the negative. If we consider only well-nested drawings, the bound of $n * \log(n)$ is no more tight. In this section, we therefore compute a new tight bound for the number of gap sets in a well-nested drawing: well-nested drawings have at most $O(n)$ gap sets.

Lemma 4.13 Let $\mathbb{D} = (V; S; \prec)$ be a well-nested drawing and $|V| = n$. Then

$$|\bigcup_{v \in V} (\text{gap sets})(v)| \leq n$$

□

Proof

$$\begin{aligned}
|\bigcup_{v \in V} (\text{gap sets})(v)| &= \sum_{(v,w) \in S} |(\text{gap sets})(w) - (\text{gap sets})(v)| \\
&= \sum_{v \in V} |(\bigcup_{w \in Sv} (\text{gap sets})(w)) - (\text{gap sets})(v)| \\
&\leq \sum_{v \in V} |Sv| \\
&\leq n
\end{aligned}$$

The first equality holds due to Lemma 3.4 and the first ≤ due to the postponed Lemma 4.14. ∎

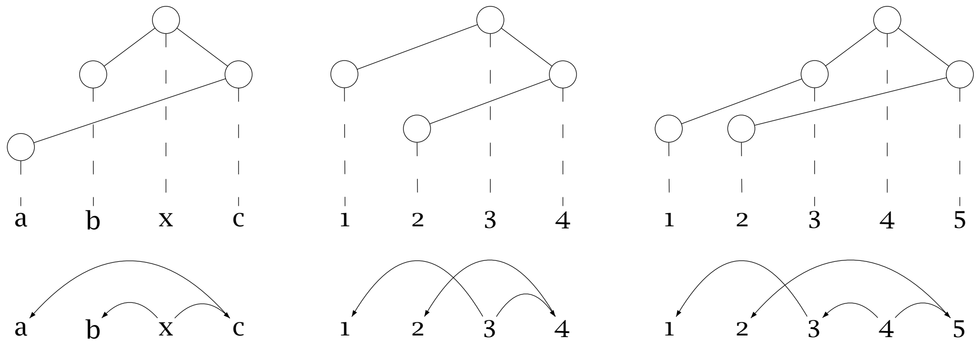
Lemma 4.14 Let $\mathbb{D} = (V; S; \prec)$ be a *well-nested* drawing and $v \in V$. Then

$$|(\bigcup_{w \in Sv} (\text{gap sets})(w)) - (\text{gap sets})(v)| \leq |Sv|.$$

□

Proof Let $Sv = w_1, \ldots, w_m$. Since we only count the gap sets of $w_1, \ldots, w_m$ that are not also gaps sets of v itself, it suffices to consider for each w_i the gaps sets that contain elements of the yield of some sibling of w_i, or contain the parent v. At most one of all these gap sets contains v but no elements of the yields of sibling nodes. The number of gap sets that contain material from sibling nodes can be measured based on the gap forest of v: since $\mathbb{D}$ is well-nested, the gap forest of v exists, and each of these gap sets corresponds to an edge in this forest. The lemma then follows from the fact that a forest with m nodes has at most $m - 1$ edges. ∎

The bound of $O(n)$ different gaps in a well-nested drawing with n nodes is tight: each drawing of the family $\mathbb{D}_k$ (see Figure 3.2 on page 11) with n nodes is well-nested and has $\lfloor \frac{n}{2} \rfloor$ different gaps: the leftmost node alone has already $\lfloor \frac{n}{2} \rfloor \in O(n)$ gaps.



(a) planar and well-nested
(b) not planar but well-nested
(c) not planar and not well-nested

Figure 4.5: Examples of planar and not planar drawings. The upper part contains the drawings and the lower part the same structures in link grammar style in which the crossing edges are visible.

4.5 Planarity

Well-nestedness is a relaxation of projectivity in the sense that the set of projective drawings is a proper subset of the set of well-nested drawings. In this section, we will compare well-nestedness to another relaxation of projectivity, called *planarity*.

Planarity has been defined by Yli-Jyrä [13]. It originates from the work on link grammar, and differs from the concept of planarity known from graph theory: while in graph theory, a graph is planar if it can be drawn in a plane without crossing edges, the analysis of a sentence is called planar if the dependency links can be drawn in the *half-plane* above the sentence without crossing edges. This notion of planarity can be adopted to drawings as shown in Figure 4.5.

Making planarity formal Yli-Jyrä defines planarity informally as

> Planarity of a dependency tree is the requirement that the links do not cross when drawn above the sentence

and formally as follows: a drawing $(V; S, \prec)$ is planar if and only if for all nodes a, b, c in V the following holds: $a \prec b \prec c \wedge (Sac \vee Sca) \Rightarrow S^*ab \vee S^*ba \vee S^*cb \vee S^*bc$ [3]. The formal definition and the informal one unfortunately do not coincide, as shown by the examples in Figures 4.5(a) and 4.5(b).

The drawing in Figure 4.5(a) is planar according to the informal definition but the condition of the formal definition does not hold. In Figure 4.5(b) this condition is satisfied for all possible a, b and c, but the drawing is not planar according to the informal definition.

[3]Yli-Jyrä uses the notation $\prec^*$ instead of $\prec$, *a linked b* instead of $Sab \vee Sba$ and *a linked* b* instead of $S^*ab \vee S^*ba$

In order to compare planarity and well-nestedness on a proper formal background, we need to give a new formal definition of planarity that coincides with the informal definition of Yli-Jyrä. The following is a straightforward formalisation of the informal description:

Definition 4.15 A drawing $(V; S, \prec)$ is planar if and only if there do not exist $a, b, c, d \in V$ with the following properties: $a \prec b \prec c \prec d$ and $Sac \vee Sca$ and $Sbd \vee Sdb$ $\dashv$

If such nodes a, b, c and d exist, the edge between a and c and the one between b and d must cross. On the other hand, if there are some crossing edges, then the nodes that are connected by these edges always form witnessing nodes a, b, c and d.

Planarity and well-nestedness Planarity and well-nestedness are two closely related concepts. Every planar drawing is well-nested, but the reverse statement does not hold. The similarity between both properties becomes apparent in the following lemma, which characterises planarity in the same way as well-nestedness is defined (see Definition 4.2).

Lemma 4.16 A drawing $\mathfrak{D} = (V; S, \prec)$ is planar if and only if there do not exist two sets $V_1, V_2 \subset V$ such that

(a) $V_1 \cap V_2 = \emptyset$

(b) $\forall v, w \in V_1.(v, w) \in (S|_{V_1} \cup S^{-1}|_{V_1})^*$ and $\forall v, w \in V_2.(v, w) \in (S|_{V_2} \cup S^{-1}|_{V_2})^*$

(c) V_1 and V_2 interleave (with respect to $\prec$)

We call sets V_1 and V_2 that satisfy conditions (a) and (b) *disjoint connected components.* If they also satisfy (c), they are called interleaving disjoint connected components. $\square$

Proof Let a, b, c, d be nodes of $\mathfrak{D}$ as specified in Definition 4.15. Then $V_1 = \{a, c\}$ and $V_2 = \{b, d\}$ are disjoint connected components and interleave.

To proof the backward direction, consider two arbitrary disjoint interleaving connected components of a drawing. These interleaving connected components have at least one pair of crossing edges. The four nodes at the endpoints of those two crossing edges then fulfil the properties of the nodes a, b, c, d stated in Definition 4.15. The drawing is thus not planar. $\blacksquare$

In contrast to well-nestedness, planarity does not prohibit disjoint interleaving subtrees, but disjoint interleaving connected components in the tree structure. Since each subtree is also a connected component, it is not weaker than well-nestedness. In other words: each planar drawing is well-nested.

Planarity is even stricter than well-nestedness, because two disjoint connected components V_1, V_2 cannot always be extended to two disjoint subtrees: if one node from V_1 dominates a node in V_2, each tree containing V_1 also contains nodes of V_2, and the subtrees are no more disjoint. An example for this case is shown in Figure 4.5(b). $V_1 = \{1, 3\}$

and $V_2 = \{2, 4\}$ are interleaving connected components, but they cannot be extended to disjoint interleaving subtrees: V_1 can only be extended to the subtree with node set $\{1, 2, 3, 4\}$, which is no more disjoint to $\{1, 2\}$.

Multi-planarity Multi-planarity is another relaxation of planarity that is proposed by Yli-Jyrä [13]: a graph is m-planar, if its edge set can be partitioned into m parts in such a way that each of the m subgraphs is planar. Each drawing with n nodes has $n - 1$ edges and is consequently at least $(n - 1)$-planar. If we want to show that a drawing is not m-planar for some $m < n - 1$, we have to consider each partition of the edge set into m parts, and have to show that in each of these partitions at least one subgraph is not planar. Therefore, determining the minimal m for a given drawing to be m-planar is an optimization problem, and finding a solution to this problem may be quite costly. In contrast to that, well-nestedness is purely structurally motivated, and easy to check.

We can conclude that well-nestedness is a relaxation of planarity in the sense that the set of well-nested drawings is a proper superset of the set of planar drawings. Multi-planarity is another relaxation of planarity but of a quite different nature.

Chapter 5

TAG-drawings

In this chapter, we define a certain class of drawings called TAG drawings. The key idea here is that each drawing of that class corresponds to a derivation in a lexicalised Tree Adjoining Grammar (TAG) [6]. We first give an overview on TAG. Then we explain how the correspondence between TAG derivations and drawings looks like, and characterise the class of TAG drawings as the class of well-nested drawings with a gap degree of at most one. The major relevance of this result is that we provide a purely structural characterisation of 'TAG-ness', which is completely independent of the grammar formalism itself. Given the analysis of a sentence in form of a drawing, we can decide whether this analysis is describable by some TAG grammar without looking at the operational machinery that underlies TAG derivations.

The results of this chapter have been published as joint work with Manuel Bodirsky and Marco Kuhlmann [2].

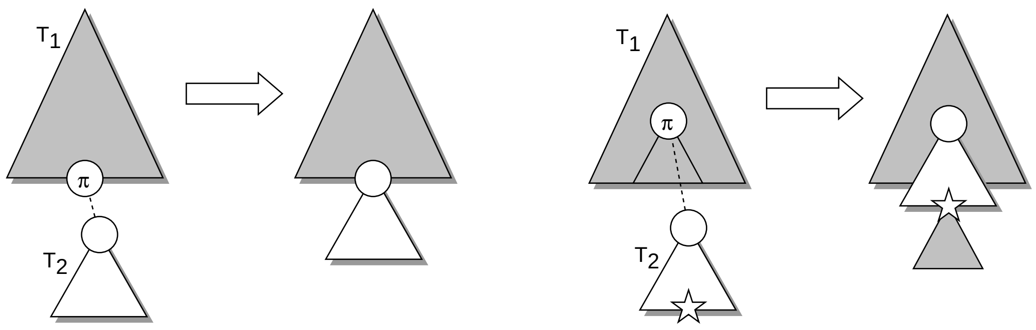
5.1 Tree Adjoining Grammar

Tree Adjoining Grammar is a grammar formalism whose derivations manipulate tree structures. This section gives a brief overview on the formalism.

The building blocks of a TAG grammar are called *elementary trees;* they are ordered trees in which each node has one of three types: *anchor* (or *terminal node*), *non-terminal node*, or *foot node*. Anchors and foot nodes must be leaves; non-terminal nodes may be either leaves or inner nodes. As a further restriction, each elementary tree can have one foot node at most. Foot nodes are usually represented with a $\star$-symbol. Elementary trees without a foot node are called *initial trees*; non-initial trees are called *auxiliary trees*. A TAG grammar is *strictly lexicalised*, if each of its elementary trees contains exactly one anchor.

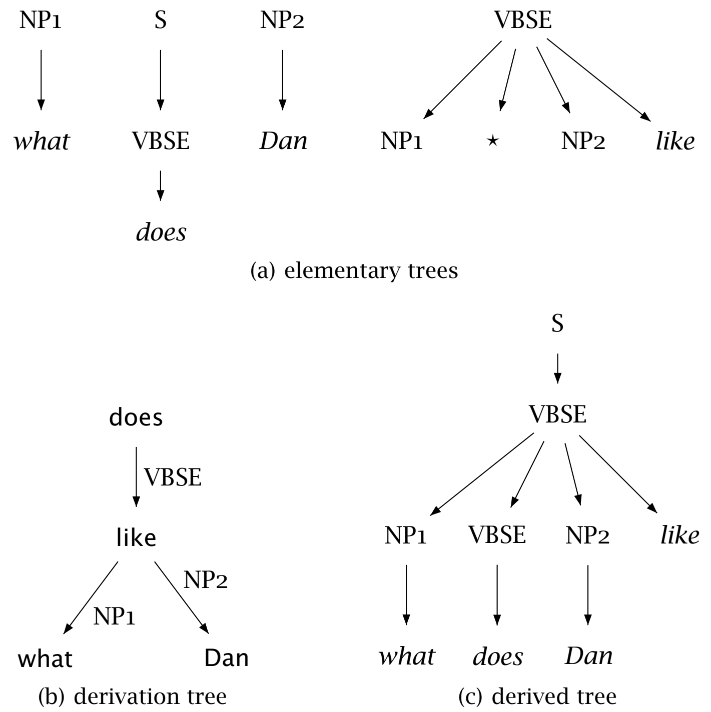
Example Figure 5.2(a) shows some elementary trees. The nodes 'what', 'does', 'Dan' and 'like' are anchors. They are all leaves, and, since the grammar is strictly lexicalised, each tree has exactly one anchor. The first three trees are initial trees, the last one is an auxiliary tree because it has a foot node. $\dashv$

Trees in TAG can be combined using two operations (see Figure 5.1): *Substitution* combines a tree structure T_1 with an initial tree T_2 by identifying a non-terminal leaf node π of T_1 with the root node of T_2 (see Figure 5.1(a)). *Adjunction* identifies an inner node π



(a) substitution (b) adjunction

Figure 5.1: The combining operations for TAG elementary trees



(a) elementary trees

(b) derivation tree (c) derived tree

Figure 5.2: An example for a TAG derivation

of a tree T_1 with the root node of an auxiliary tree T_2; the subtree of T_1 rooted at π is excised from T_1 and inserted below the foot node of T_2 (see Figure 5.1(b)). Combing operations are disallowed at root and foot nodes.

TAG *derivation trees* record information about how tree structures were combined during a derivation. Formally, they can be seen as unordered trees whose nodes are labelled with elementary trees, and whose edges are labelled with the nodes at which the combining operations took place. If v is a node in a derivation tree, we write $\ell(v)$ for the label of v (that is, the corresponding elementary tree). When we visualise derivation trees, we usually only use the anchor of an elementary tree as the node label, not the whole tree. In order to indicate the difference, we use a Sans serif font for the nodes of the derivation tree (for example 'like' represents the node of the derivation tree and '*like*' the anchor of the corresponding elementary tree).

In the derivation tree, an edge (v_1, v_2) with label π signifies that the elementary trees $\ell(v_1)$ and $\ell(v_2)$ where combined at node π. If $\ell(v_2)$ is an initial tree, this combining operation is a substitution, otherwise it is an adjunction.

Example A sample derivation tree is shown in Figure 5.2(b). The edge between the nodes 'does' and 'like' indicates that the auxiliary tree of 'like' is adjoined into the initial tree of 'does' at the node VBSE. The remaining two edges represent substitutions, since the elementary tree of 'what' and 'Dan' are initial trees. ⊣

TAG *derived trees* represent results of derivations; we write $drv(D)$ for the derived tree corresponding to a derivation tree D. Derived trees are ordered trees made up from the accumulated material of the elementary trees participating in the derivation. In particular, each TAG derivation induces a mapping ρ that maps each node v in D to the root node of $\ell(v)$ in $drv(D)$. In strictly lexicalised TAGs, a derivation also induces a mapping α that maps each node v in D to the anchor of $\ell(v)$ in $drv(D)$.

For nodes v in derivation trees D of strictly lexicalised TAGs we define

$$\begin{aligned} \mathit{derived}(v) &:= \{\, \alpha(u) \mid v \lhd^* u \text{ in } D \,\} \quad \text{and} \\ \mathit{yield}(v) &:= \{\, \pi \mid \pi \text{ is an anchor and } \rho(v) \lhd^* \pi \text{ in } drv(D) \,\}\,. \end{aligned}$$

The set $\mathit{derived}(v)$ contains those anchors in $drv(D)$ that are contributed by the partial derivation starting at $\ell(v)$; $\mathit{yield}(v)$ contains those anchors that are dominated by the root node of $\ell(v)$.

Example Figure 5.2(c) shows the TAG derived tree that is obtained when the elementary trees of Figure 5.2(a) are combined according to the derivation tree of Figure 5.2(b).

For the node 'like' of this derivation tree derivation, we have

$$\begin{aligned} \alpha(\mathsf{like}) &= \mathit{like} \\ \rho(\mathsf{like}) &= \mathrm{VBSE} \\ \mathit{derived}(\mathsf{like}) &= \{\mathit{what}, \mathit{Dan}, \mathit{like}\} \\ \mathit{yield}(\mathsf{like}) &= \mathit{derived}(\mathsf{like}) \cup \{\mathit{does}\}\,. \end{aligned}$$

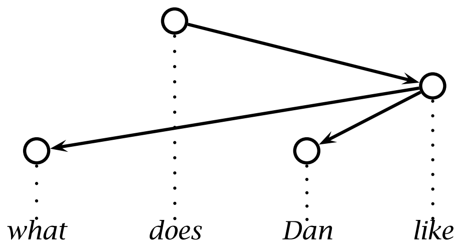


Figure 5.3: The TAG drawing that corresponds to the TAG derivation shown in Figure 5.2

⊣

5.2 The correspondence between TAG derivations and drawings

In this section, we show how a derivation of a strictly lexicalised TAG grammar can be interpreted as a drawing.

In a TAG derivation, the derivation tree represents the dependencies among the participating elementary trees. Since in a lexicalised TAG, each elementary tree has exactly one anchor, a derivation tree also represents the dependencies among these anchors, that is among the words of the sentence that is derived. If we complement this tree with the word order of the derived sentence, we obtain a drawing.

Definition 5.1 (TAG drawing) Let $D = (V; \lhd)$ be a derivation tree for a strictly lexicalised TAG. A drawing $\mathfrak{D} = (V'; S, \prec)$ is the TAG drawing corresponding to D if and only if

- $V' = \{\, \alpha(v) \mid v \in V \,\}$
- $S = \{\, (\alpha(v), \alpha(w)) \mid (v, w) \in \lhd \,\}$
- $\prec$ is the order of the leaves of $drv(D)$ ⊣

The order is defined as the order of the leaves in the derived tree since this represents the word order of the derived sentence in TAG derivations.

Example Figure 5.3 shows the TAG drawing of the TAG derivation shown in Figure 5.2. The tree structure of the drawing corresponds to the tree structure of the derivation tree shown in Figure 5.2(b), while the node order is the leaf order of the derived tree shown in Figure 5.2(c). ⊣

At this point, the question arises whether each drawing is a TAG drawing. This question must be answered to the negative. The class of TAG drawings is a proper subclass of the class of all drawings, as we will show in the subsequent sections.

5.3 TAG drawings have a gap degree of at most one

Gaps in TAG drawings correspond to adjunctions in TAG derivations: during an adjunction, disjoint material – namely the material that is inserted at the foot node – is placed in between the nodes of a subtree and causes a gap. Since each elementary tree has one foot node at most, the gap degree of TAG drawings is limited to one.

Example Consider the derivation shown in Figure 5.2. The elementary tree of 'like' is adjoined into the elementary tree of 'does', and during this operation, the anchor '*does*' is placed in between the dependents of 'like' (NP1 and NP2), although it does not depend on it. Consequently, in the induced drawing (see Figure 5.3), the node '*does*' is a gap in the subtree rooted at '*like*', and, furthermore, it is the only gap of it, since the elementary tree of 'like' has only the one foot node where '*does*' is placed during the adjunction. $\dashv$

We now make this intuition precise.

Lemma 5.2 Let D be a TAG derivation tree and let v be a node in D. Then

(a) $\mathit{derived}(v) \subseteq \mathit{yield}(v)$

(b) $\mathit{yield}(v) - \mathit{derived}(v)$ is convex

(c) $\mathit{derived}(v)$ contains at most one gap. $\square$

PROOF (a) Each $a \in \mathit{derived}(v)$ is either the anchor of $\ell(v)$ or has been derived from $\ell(v)$ in one or more steps. In both cases a is dominated by the root node of $\ell(v)$ in the derived tree, since during adjunction or substitution operations, the root of the tree in which other material is adjoined or substituted remains the topmost node. Thus $a \in \mathit{yield}(v)$.

(b) Define $G := \mathit{yield}(v) - \mathit{derived}(v)$ and let a_l and a_r be the leftmost and rightmost anchor in G, respectively (assuming that G is non-empty). The only way by which an anchor can have entered G is by an adjunction of $\ell(v)$ into some other elementary tree (see Figure 5.1(b)). Now assume that G was not convex, i.e. there is an anchor $a \notin G$ such that $a_l \prec a \prec a_r$. Since $\mathit{yield}(v)$ is convex, a must be an element of $\mathit{derived}(v)$. Since both a_l and a_r are dominated by the foot node of $\ell(v)$, a is dominated by that node as well. This is a contradiction: neither can an anchor be dominated by the foot node of its own elementary tree (the foot node always is a leaf), nor can the foot node be the starting node of a sub-derivation (substitution and adjunction are disallowed at foot nodes). Thus, G is convex.

(c) The third item follows from the preceding two and from the observation that $\mathit{yield}(v)$ is convex. $\blacksquare$

Corollary 5.3 With Lemma 5.2 (c) each subtree of a TAG drawing has at most one gap. Consequently TAG drawings have a gap degree of at most one. $\square$

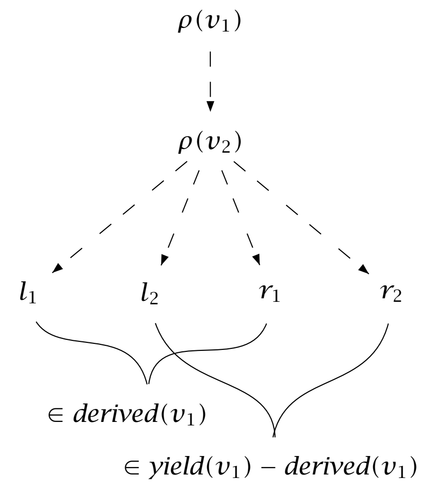


Figure 5.4: The TAG derived tree in the proof of Lemma 5.4

5.4 TAG drawings are well-nested

The gap restriction alone does not yet characterise the class of TAG drawings exactly. A second important restriction is that each TAG drawing is well-nested. The cause of this restriction has its origin again in the fact that gaps originate from adjunctions. In a drawing that is not well-nested, two subtrees exist so that a part of the first is contained in a gap of the second and vice versa. On the TAG side, this implies that each of the two trees must be adjoined into the other one. This is not possible, since the derivation tree is acyclic.

Lemma 5.4 TAG drawings are well-nested. □

PROOF Let D be a derivation tree. Imagine the TAG drawing for D, and assume that it contains two interleaving subtrees T_1 and T_2 with witnessing nodes $l_1 \prec l_2 \prec r_1 \prec r_2$. We will show that this leads to a contradiction. Let v_1 and v_2 be the nodes of the derivation tree whose anchors $\alpha(v1)$ and $\alpha(v_2)$ are the roots of T_1 and T_2, respectively. The witnessing nodes define two overlapping intervals in the leaves of $drv(D)$: $[l_1, r_1]$ and $[l_2, r_2]$. Let π be an anchor present in both of these intervals. Since π is dominated in $drv(D)$ by both $\rho(v_1)$ and $\rho(v_2)$, either $\rho(v_1)$ also dominates $\rho(v_2)$, or vice versa. We assume that $\rho(v_1)$ dominates $\rho(v_2)$, the other case is symmetric. The situation is then as shown in Figure 5.4: $yield(v_1) \supseteq yield(v_2)$, $l_1, r_1 \in derived(v_1)$ and $l_2, r_2 \in yield(v_1) - derived(v_1)$ (l_2 and r_2 are not in $derived(v_1)$ since they are derived from v_2 which is disjoint to v_1 in D). With $l_2 \prec r_1 \prec r_2$ it follows that $yield(v_1) - derived(v_1)$ is not convex. This contradicts Lemma 5.2 (b). ■

5.5 Constructing a TAG grammar for a drawing

The final step in our characterisation of TAG drawings is to show that each well-nested drawing with a gap degree of at most one is a TAG drawing. We give a constructive proof in form of an algorithm that takes a well-nested drawing with gap degree of at most one as input, and returns a TAG grammar whose only derivation corresponds to that drawing.

A TAG grammar consists of a set of elementary trees. The task of the algorithm is hence to generate for each node of the drawing an elementary tree so that their combination (according to the derivation tree, which equals the tree structure of the drawing) results in a derivation that corresponds to the drawing. These elementary trees are based on an extended form of the gap forests of the nodes.

5.5.1 Extending the gap forest

The gap forest of a node u as presented in Section 4.3 describes the relation among the yields of the children of u. If one of these children – call it w – dominates another child w' in the gap forest, the yield of w' is contained in a gap of w. Hence, the subtrees below the outgoing edges of w in the gap forest represent the material within w's gaps that is contributed by its siblings. However, there might also be other material within w's gaps; for example, w's parent could be a hole in $C(w)$. We will now define an extended form of gap forests, in which all 'being in the gap' relations are represented, not only the ones among sibling nodes. For this purpose, we add further nodes to the gap forest of a node u that represent the node u itself and the gaps of u.

Definition 5.5 (Extended gap forest) Let $\mathfrak{D} = (V; S, \prec)$ be a well-nested drawing, and $u \in V$ be a node with n gaps. Then the extended gap forest of u is the gap forest $(Su \cup \{\mathit{self}, \star_1, \dots, \star_n\}; \lhd, <)$ with

$$\Gamma(x) := \begin{cases} \{u\} & \text{if } x = \mathit{self} \\ G_i(u) & \text{if } x = \star_i \text{ for some } i \in [1, n] \\ C(x) & \text{otherwise} \end{cases}$$

Compared to the original definition of a gap forest, we have only added a few nodes: the node *self* represents the singleton set $\{u\}$, and the $\star$-nodes represent the gaps of u. As before, the sets are disjoint and do not interleave (the new sets are all convex and hence cannot interleave with any disjoint sets). In fact, the new extension only adds some additional leaves to the forest structure: since the new sets are all convex, they do not have any gaps, and accordingly no dependents in the extended gap forest.

Example Figure 5.5 shows a drawing together with the extended gap forests of two of its nodes. The extended gap forest of node 3 consists of the children of 3 (that is 1, 4 and 5), the *self* node, and one $\star$-node, since 3 has one gap. The outgoing edges from

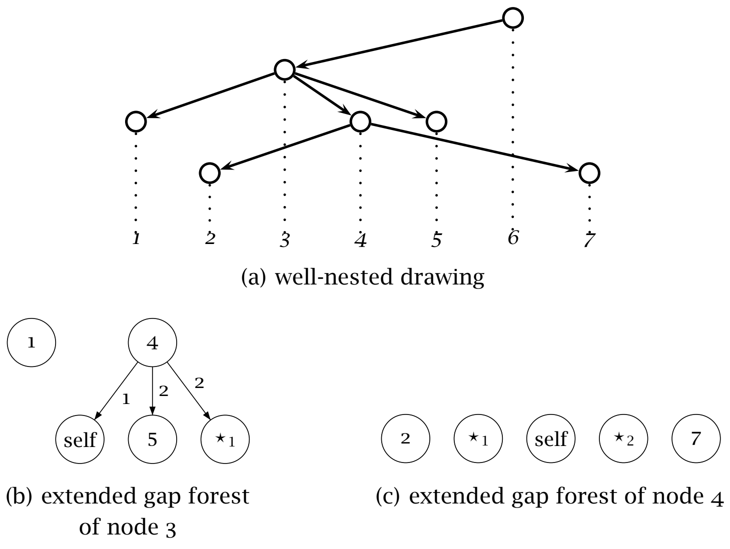

1
2
3
4
5
6
7
(a) well-nested drawing
1
4
1
2
2
self
5
⋆1
(b) extended gap forest of node 3
2
⋆1
self
⋆2
7
(c) extended gap forest of node 4

Figure 5.5: A well-nested drawing together with the extended gap forests of the nodes 3 and 4

node 4 indicate that 4 has two gaps: its parent 3 makes up the first gap, and the second gap consists of the yield of 5 and the gap of node 3, where the yield of 5 is left of the gap of 3. The yield of 1 is not in the gap of any of its sibling nodes. It is located left of all the other sets, and builds a second root in the extended gap forest.

All elements of the extended gap forest of node 4 have no gaps at all. Consequently all nodes are leaves (and simultaneously roots). They are ordered from left to right in the way their associated sets are ordered in the drawing. ⊣

5.5.2 The algorithm

We want to construct TAG elementary trees that correspond to the nodes of a well-nested drawing with gap degree of at most one. These trees must be such that we can interpret the tree structure of the drawing as a derivation tree, and compose the elementary trees of the different nodes in a way that the order of the leaves in the derived tree is identical to the order of the nodes in the drawing. This is done by Algorithm 1.

In each iteration, the algorithm generates one elementary tree. First, the extended gap forest of the respective node v is computed (line 3), and its nodes are renamed (lines 4–8): the anchor *self* is replaced by the node v itself (line 4) to ensure that the leaves of the derived tree equal the nodes of the drawing, and the nodes for the children of v are replaced by corresponding nonterminals (lines 6–8). These nonterminals offer the adjunction and substitution sites for the elementary trees of the children of v. The nonterminals for the different nodes are generated via a function NT, which is a bijection between the nodes of the drawing and a set of nonterminal symbols. The function NT

Algorithm 1 TAG-GRAMMAR$(V, S, \prec)$

```
 1: $Lexicon := \emptyset$
 2: for all $v \in V$ do
 3:   $\mathfrak{F} :=$ extended gap forest$(v)$
 4:   rename self by $v$ in $\mathfrak{F}$
 5:   rename $\star_1$ by $\star$ in $\mathfrak{F}$
 6:   for all $w \in Sv$ do
 7:     rename $w$ by $\mathsf{NT}(w)$ in $\mathfrak{F}$
 8:   end for
 9:   $\mathfrak{T} :=$ tree with nodes and edges from $\mathfrak{F}$ and root node $\mathsf{NT}(v)$
10:   $Lexicon := Lexicon \cup \{\mathfrak{T}\}$
11: end for
```

is arbitrary, with the only restriction that the set of nonterminal symbols must be disjoint to the node set of the drawing: the nodes of the drawing equal the anchors of the elementary trees, and these must be distinguishable from the nonterminal nodes.

After the renaming steps, the forests are transformed into trees by simply adding an additional node that dominates all the roots of the forest (line 9). This root of the elementary tree is labelled with the nonterminal of v, such that the elementary tree can be adjoined or substituted into the elementary tree of the parent of v. If v is the root of the drawing, then $\mathsf{NT}(v)$ is the start symbol of the grammar.

The algorithm shows that the *self* node and $\star$-nodes of extended gap forests correspond to the anchors and the foot nodes of elementary trees, respectively. To point out this similarity, in the following we will use the terms anchor and foot node also for extended gap forests: the *self* node is the anchor of an extended gap forest; the $\star$-nodes are the foot nodes.

Example Figure 5.6(a) shows a drawing $\mathfrak{D}$, and Figure 5.6(c) shows the corresponding elementary trees generated by the algorithm. Since 2 and 4 are children of 3 in $\mathfrak{D}$, their elementary trees must be adjoined or substituted into the elementary tree of 3. Consequently, the elementary tree of 3 contains nodes $\mathsf{NT}(2)$ and $\mathsf{NT}(4)$ where these operations can take place. If we compose all elementary trees, we obtain the derived tree shown in Figure 5.6(b), whose leaf order equals the order of $\mathfrak{D}$.

If we compare the elementary trees in Figure 5.6(c) to the extended gap forests they are based on (see Figure 5.6(d)), we observe that always a new root is added and the nodes are renamed. In the elementary tree of node 4, for example, the new root is $\mathsf{NT}(4)$, the node 2 is renamed as $\mathsf{NT}(2)$ and *self* as 4. The $\star_i$ nodes are replaced by foot nodes, $\star$ and since all TAG drawings have a gap degree of at most one, each elementary tree has at most one foot node.

As a second example, we can again consider the drawing for the sentence 'What does

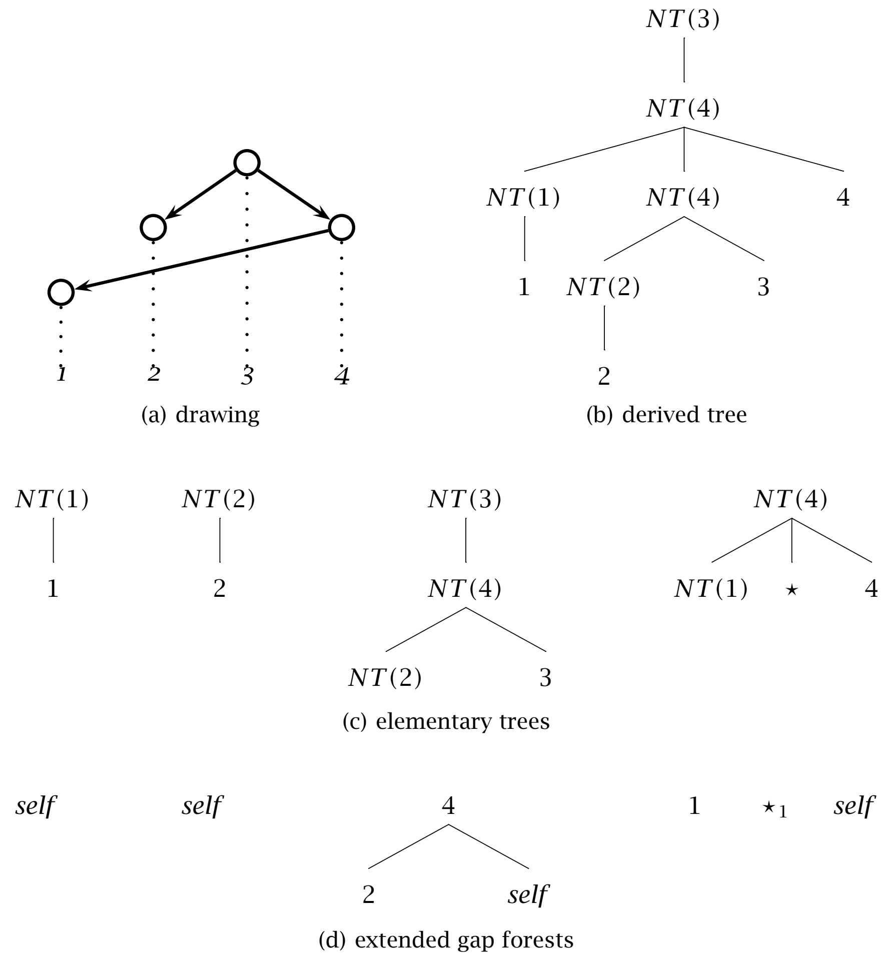


Figure 5.6: A drawing $\mathfrak{D}$ together with the corresponding elementary trees generated by a call of TAG-GRAMMAR($\mathfrak{D}$), the extended gap forests they are based on, and the derived tree that is obtained by combining the elementary trees.

Dan like' (see Figure 5.3). The corresponding elementary trees shown in Figure 5.2(a) are obtained with a function NT such that

$$\begin{aligned} \mathsf{NT}(\textit{what}) &= \text{NP1} \\ \mathsf{NT}(\textit{does}) &= \text{S} \\ \mathsf{NT}(\textit{Dan}) &= \text{NP2} \\ \mathsf{NT}(\textit{like}) &= \text{VBSE} \end{aligned}$$

⊣

5.5.3 Correctness

We have to give arguments for two different aspects of the algorithm: first, the produced structures must be valid elementary trees, and second, they must be combinable such that the resulting leaf order equals the order of the drawing.

The first aspect is easy: we must only ensure that anchor and foot-node are leaves, and that there is exactly one anchor and at most one foot node. All these properties are inherited from the *self* and the $\star_1$ nodes in the extended gap forest.

If we combine the trees, we obtain the correct leaf order because of the construction of the extended gap forests. The nodes in the gap forest are ordered according to the sets they are associated with by Γ (as defined in Definition 5.5). For each node in the elementary tree, this set consists of the anchors that are contributed by the subderivation starting at this node. Consequently, one node $\mathsf{NT}(v)$ dominates another node $\mathsf{NT}(w)$ in an elementary tree, if the leaves contributed by $\mathsf{NT}(w)$ must be placed in between the leaves contributed by $\mathsf{NT}(v)$. On the other hand, $\mathsf{NT}(v)$ and $\mathsf{NT}(w)$ are placed side by side if the leaves contributed by $\mathsf{NT}(v)$ and the leaves contributed by $\mathsf{NT}(w)$ are located side by side in the derived tree.

The correctness of the algorithm establishes the following:

Lemma 5.6 Each well-nested drawing with a gap degree of at most one is a TAG drawing. □

Corollary 5.3 and Lemmata 5.4 and 5.6 together imply:

Theorem 5.7 A drawing is a TAG drawing if and only if it is well-nested and has a gap degree of at most one. □

In this section we have seen that strictly lexicalised TAG can be interpreted as a formalism that describes sets of drawings. Furthermore, we have shown that the class of TAG drawings is the class of well-nested drawings with a gap degree of at most one. In the next chapter we develop a description language for well-nested drawings with an arbitrary gap degree. We also indicate how this description language can be interpreted as an extension of TAG.

Chapter 6

A description language for well-nested drawings

In this chapter, we present a description language for well-nested drawings. We will start with analysing the grammar formalisms we have discussed in the previous chapters.

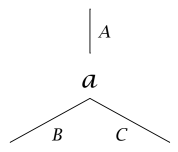
6.1 Locality

In the course of this thesis, we have already discussed some grammar formalisms that are suitable to describe sets of drawings. In Section 3.4, we have shown how lexicalised Context-Free Grammars (CFGS) induce drawings, and in Chapter 5, we have explained the correspondence between lexicalised TAG derivations and drawings. In this section, we shall analyse how both formalisms can describe drawings locally.

There is an important similarity between CFG and TAG: the descriptions within these formalisms are local in a sense that a description of a drawing is split up into several small units. Each of these units describes one local fragment of a drawing. These units are usually called lexicon entries. In a CFG, lexicon entries are production rules, in TAG, lexicon entries are elementary trees. Each lexicon entry describes a part of the tree structure, and a part of the order of the drawing. The tree structure is usually described with the concept of *valencies*. The local description of the order differs in TAG and CFG. This difference is a reason for the higher expressivity of TAG.

Local description of the tree structure The tree structure is described by giving each node v a *type*, and by specifying the types of its children. The types of the children of v are also called the *valencies* of v. The type of a node v can also be considered as an edge label for the incoming edge of v. Each lexicon entry then describes a fragment of the tree structure like the one shown in Figure 6.1(a): one node with its type and its valencies. Two lexicon entries can be combined if the type of the one is a valency of the other.

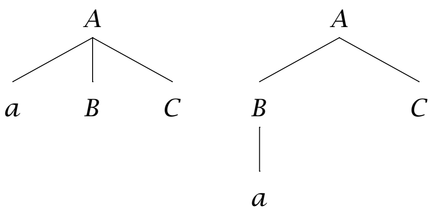
Tree structures are described similarly in lexicalised CFG and lexicalised TAG. Both context-free production rules and TAG elementary trees consist of nonterminals and one terminal v. One dedicated nonterminal represents the type of v, the other nonterminals represent the valencies of v. In the context-free production rules, the left-hand side nonterminal represents the type of v, and the right-hand side nonterminals represent the valencies of it. In a TAG elementary tree, the root represents the type of v, and the other nonterminals are the valencies. Examples for such production rules and



(a) local part of the tree structure

$$
\begin{aligned}
A &\rightarrow aBC \\
A &\rightarrow BaC
\end{aligned}
$$

(b) context-free rules



(c) TAG elementary trees

Figure 6.1: A local part of the tree structure of a drawing and several context-free rules and TAG elementary trees that describe it

elementary trees are given in Figure 6.1(b) and 6.1(c), respectively. They all describe the tree fragment shown in Figure 6.1(a), but specify different local orders for it.

Local description of the order In CFG rules, the local order is described by the order among the terminal and the nonterminals on the right-hand side of the rule. In the first rule in Figure 6.1(b), for example, the node a is located before the two subtrees represented by B and C; in the second rule, a is placed in between them.

In TAG elementary trees, the order is encoded in a forest structure that underlies the elementary tree: if we leave out the root of the elementary tree (which specifies the type of v and is not relevant for the order), we obtain a forest that can be interpreted as an extended gap forest (according to Definition 5.5). The anchor represents the *self*-node of the extended gap forest, and the nonterminals represent the yields of the children. Their relative position within the forest represents their local order. While in context-free production rules all valencies have to be ordered from left to right, in the gap forest there are two alternatives: either two valencies are ordered side by side, or one is dominated by the other. In the first case, the subtrees represented by the different valencies are ordered side by side, in the second case, one is placed within a gap of the other.

Context-free production rules can also be interpreted as a restricted form of TAG elementary trees with a different notation: the root of the tree is written on the left-hand side of the rule, the gap forest dominated by it on the right-hand side. As a restriction, the gap forest may only consist of roots, that is all nodes must be ordered from left to right. For example, the first production rule shown in Figure 6.1(b) specifies the same information as the first elementary tree shown in Figure 6.1(c).

Limitations of CFG and TAG So far we have seen that CFG and TAG use the same mechanism to specify the tree structure of a drawing, but TAG has an extended method to specify the local order. This has also consequences for the expressivity of both formalisms: whereas CFG only describes projective drawings, TAG is able to describe well-nested drawings with a gap degree of at most one. The gap restriction is due to the fact that the gap forest underlying a TAG elementary tree may have one foot node at most.

In the remainder of this chapter, we will investigate how this expressivity can be further extended. The idea is to eliminate the gap restriction by allowing arbitrary gap forests as a specification of the order within one lexicon entry. There are several questions arising within that context:

(a) Does the combination of the local orders described by the gap forests always result in a total order?

(b) Can the local descriptions within different lexicon entries always be combined, or is it possible that they contradict each other?

Since these questions are independent of the way how the tree structure of the drawing is described, we assume that the tree structure is already given, and at each of the nodes one extended gap forest is annotated. One can assume that the tree structure is built out of one lexicon entry for each node, containing the type and valencies of this node, but we abstract away this detail in the next sections, since our questions are only concerned with the local description of the order.

In the next section, we shall analyse the structural properties of the gap forests of the different nodes of a drawing, and shall ask whether they are completely independent of each other or not. This will answer question (b). Then we will present an algorithm that constructs a corresponding drawing for a given tree structure with annotated extended gap forests for all nodes. This will answer question (a).

6.2 Structural characterisation of extended gap forests

Not every indexed ordered forest is a valid gap forest. In this section we analyse the structure of gap forests and hence characterise which structure indexed ordered forests must have if they describe the local order within a lexicon entry.

6.2.1 The structure of a single extended gap forest

Lemma 6.1 Let $\mathbb{D} = (V; S, \prec)$ be a drawing and $v \in V$. Then the extended gap forest $(W; \lhd, <)$ of v has the following properties:

(a) The node set contains an anchor and possibly foot nodes:
$W = Sv \cup \{\mathit{self}, \star_1, \dots, \star_k\}$ for some $k \in \mathbb{N}$

(b) The anchor and the foot nodes are leaves: $\lhd \mathit{self} = \lhd \star_1 = \cdots \lhd \star_k = \emptyset$

(c) The foot nodes are ordered according to their index: $\forall i, j \in [1, k]. i < j \Rightarrow \star_i < \star_j$

(d) If a node has an i-th gap (for $i > 1$), it has also an $(i-1)$-th gap:
$\forall w, w' \in W. w \lhd_i w' \wedge i > 1 \Rightarrow \exists w''. w \lhd_{i-1} w''$

(e) A foot node is never the leftmost or rightmost node in the gap forest:
$\nexists i \in [1,k].(\forall w \in W - \{\star_i\}.w < \star_i) \vee (\forall w \in W - \{\star_i\}.\star_i < w)$

(f) Two gaps are never next to each other:
$\forall \star_i, \star_j \in W$ with $i \neq j$.
$\exists w \in W, R \in \{<,>,\lhd_1,\dots,\lhd_k,\rhd_1,\dots,\rhd_k\}.wR\star_i \wedge \neg wR\star_j$ □

Most of the properties are obvious, only the last two need some explanations. Property (e) holds since a foot node represents a gap of v, and a gap is always surrounded by material of the yield of v. This material is also represented by nodes in the extended gap forest. Property (f) formalises that two gaps must never be next to each other: there must always be some material between two gaps. This material must belong to some node in the extended gap forest, and this node has a different relation to $\star_i$ than to $\star_j$.

6.2.2 Structural dependencies among forests of different nodes

The extended gap forests of the nodes of a drawing are not completely independent of each other. Not every combination of gap forests for the different nodes describes a valid total order. This is a crucial point for our description language: if we want to combine two lexicon entries, we do not only have to take care whether the type of the one is a valency of the other, but in addition, we have to take care that the two corresponding gap forests match. Fortunately, the dependencies among the different gap forests are quite limited. The gap forest of a node v matches the gap forest of its child w, if the number of foot nodes in the gap forest of w equals the number of different outgoing edge indices of the node w in the gap forest of v.

Lemma 6.2 Let $(V;S,\prec)$ be a drawing, $v,w \in V$ with Svw and let $(W^v;\lhd^v,<^v)$ and $(W^w;\lhd^w,<^w)$ be the extended gap forests of v and w respectively. Then

$$\lhd_i^v w \neq \emptyset \iff \star_i \in W^w$$

Proof $\lhd_i^v w \neq \emptyset \iff w$ has an i-th gap $\iff \star_i \in W^w$ ■

This dependency exists only because the extended gap forests of both the parent and the child node depend on the gap degree of the child. For a grammar formalism, this means that a lexicon entry for some node v does not only require a certain type for its children, but also a certain gap degree. Considering the gap degree as a part of the type is an elegant way to take care of this requirement.

6.3 Constructing the drawing for a description

In the previous section, we have identified necessary properties for the extended gap forests of a drawing. Now we will show that they are also sufficient in the following sense: a description in form of a tree with annotated indexed ordered forests for all nodes has a corresponding drawing if the indexed ordered forests have the properties specified in the Lemmata 6.1 and 6.2. We will show this with an algorithm that takes such a description as input and generates the (only) corresponding drawing. Since each well-nested drawing has gap forests for all nodes, the algorithm also implies that each well-nested drawing is describable in such a setup.

The algorithm constructs the drawing bottom-up. For each node v, it constructs an intermediate drawing that represents the part of the final drawing which is dominated by v. Furthermore, the order of this intermediate drawing is extended with some additional placeholders $\sqcup_1, \dots, \sqcup_k$ that represent the gaps of v. At these placeholders the parent of v inserts nodes during the construction of its drawing.

The drawing for v is constructed as follows. Let $Sv = \{w_1, \dots, w_n\}$, and let,for each $i \in \{1, \dots, n\}$, $(V_i; S_i, \prec_i)$ be the drawing of w_i, and $\prec'_i$ the extension of the order such that $\prec_i \cup \prec'_i$ is a total order among the nodes of the drawing and the placeholders. Then the drawing for v is $(V; S, \prec)$ with

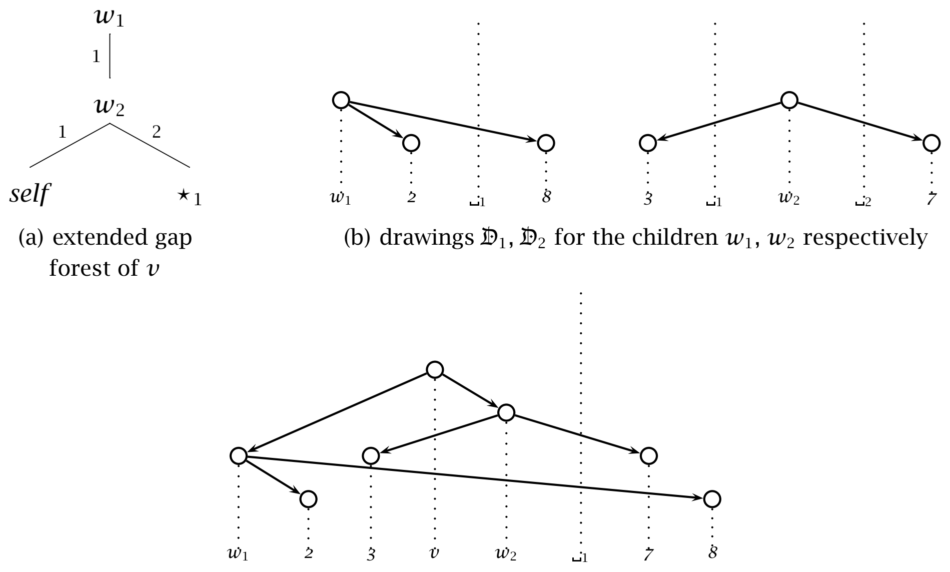
$$\begin{aligned} V &= \{v\} \cup \bigcup_{i \in [1,n]} V_i \\ S &= \{\, (v, w_i) \mid i \in 1, \dots, n \,\} \cup \bigcup_{i \in [1,n]} S_i \end{aligned}$$

The order $\prec \cup \prec'$ is defined by a left-to-right depth-first traversal over the extended gap forest of v. During this traversal, the nodes and the placeholders are enumerated from left to right. Each node w of the extended gap forest contributes the nodes $\mathsf{Con}(w, j)$ at its j-th visit, where $\mathsf{Con}(w, j)$ is defined as follows:

$$\begin{aligned} \mathsf{Con}(\mathit{self}, 1) &:= \{v\} \\ \mathsf{Con}(\ \star_i, 1) &:= \{\sqcup_i\} \\ \mathsf{Con}(\ w_i, j) &:= \begin{cases} \{\, u \in V_i \mid u \prec'_i \sqcup_1 \,\} & \text{if } j = 1 \\ \{\, u \in V_i \mid \sqcup_{j-1} \prec'_i u \prec'_i \sqcup_j \,\} & \text{if } 1 < j < (\mathsf{gap\ degree})(w_i) \\ \{\, u \in V_i \mid \sqcup_{j-1} \prec'_i u \,\} & \text{if } j = (\mathsf{gap\ degree})(w_i) \end{cases} \end{aligned}$$

The anchor contributes v, and the foot nodes contribute the new placeholders. Inner nodes always represent a child w_i of v. They contribute the nodes of the drawing of this child step by step. At the first visit, the nodes before the first placeholder are contributed; at the second visit, the nodes between the first and the second placeholder, and so on.

If an inner node of the extended gap forest has several outgoing edges with the same index, the descent to them counts as one visit. In other words: The j-th visit of an inner node w is between descending to the outgoing edges with index $j - 1$ and descending to the edges with index j.



(a) extended gap forest of v

(b) drawings $\mathbb{D}_1$, $\mathbb{D}_2$ for the children w_1, w_2 respectively

(c) resulting drawing for v

Figure 6.2: A sample iteration of the algorithm: the drawing for node v is constructed from the drawings for v's children and its extended gap forest

Lemma 6.2 guarantees that the number of placeholders for each inner node in the extended gap forest matches up exactly with the number of different outgoing edge indices. Lemma 6.1 ensures the rest of the necessary details, for example, that anchor and foot nodes are leaves, and that there are always nodes between two placeholders.

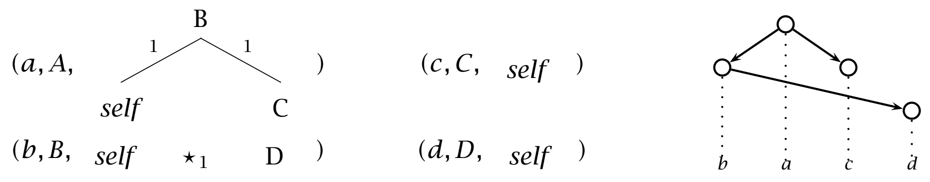
Example An example iteration of the algorithm is shown in Figure 6.2. The drawing for node v is constructed out of the drawings for v's children and its extended gap forest. The corresponding traversal over the extended gap forest visits the nodes as follows:

current node	contributed part of order
w_1	$\mathsf{Con}(w_1, 1) = \{w_1, 2\}$
w_2	$\mathsf{Con}(w_2, 1) = \{3\}$
self	$\mathsf{Con}(\mathit{self}, 1) = \{v\}$
w_2	$\mathsf{Con}(w_2, 2) = \{w_2\}$
$\star_1$	$\mathsf{Con}(\star_1, 1) = \{\sqcup_1\}$
w_2	$\mathsf{Con}(w_2, 3) = \{7\})$
w_1	$\mathsf{Con}(w_1, 2) = \{8\})$

$\dashv$

6.4 The description language

In the previous sections we have shown that each well-nested drawing is uniquely describable as a tree structure with an extended gap forest for each node. The description language we are going to describe now hence describes such a structure.



(a) lexicon entries (b) described drawing

Figure 6.3: Some lexicon entries and a drawing that is described by their combination.

An expression of the language simply consists of a set of lexicon entries.

Definition 6.3 (Lexicon entry) Let V be a set of nodes and T a set of types. Then a lexicon entry is a tuple $(v, t, \mathfrak{F})$. We call $v \in V$ the node, and $t \in T$ the type of the lexicon entry. The structure $\mathfrak{F}$ is an indexed ordered forest with node set $\{\mathit{self}, \star_1, \dots, \star_g\} \cup \mathit{val}$, where the elements of $\mathit{val} \in \mathfrak{P}(T)$ are called the valencies of v, and $g \in \mathbb{N}$ the gap degree of v. Furthermore, $\mathfrak{F}$ must satisfy the conditions of Lemma 6.1. ⊣

The forest $\mathfrak{F}$ represents the gap forest of v, but since within the lexicon entry, the children of v are not known, their respective nodes in $\mathfrak{F}$ are represented by the valencies of v.

A drawing $\mathfrak{D}$ is described by an expression of the description language, if for each node v of $\mathfrak{D}$ a lexicon entry can be found such that

- the valencies of v are the types of the children of v;
- the gap forest of v equals the forest $\mathfrak{F}$ of its lexicon entry, modulo the fact that the latter contains the types of the children of v rather than the children themselves.

In order to enumerate all drawings described by a set of lexicon entries, one has to enumerate all possibilities to combine these entries. In oder to decide whether a node v described by a lexicon entry l_1 can be the parent of a node w with lexicon entry l_2, it suffices to check whether the type of w is within the (open) valencies of v, and whether in the indexed ordered forest of v, the type of w has outgoing edges with indices $1, \dots, g_w$, where g_w is the gap degree of w. The second condition ensures that the property stated in Lemma 6.2 is satisfied. This guarantees that after generating a complete tree structure, a corresponding drawing exists. This drawing can be computed with the algorithm of Section 6.3.

Example Some sample lexicon entries are shown in Figure 6.3(a). The lexicon entries can be combined such that the drawing in Figure 6.3(b) is obtained. The tree structure results from the valency information. For example, the lexicon entry for node a contains the valency information that a has two children with respective types B and C. The order of the drawing is computed according to the algorithm of Section 6.3. ⊣

6.5 An extension of TAG

The description language which we presented in the previous section can also be interpreted as an extension of TAG. This extension can describe well-nested drawings with arbitrary gap degree. In this section we briefly sketch this extension.

The information contained in a lexicon entry of a node v can be represented as an elementary tree: the type t of v is the root node, the rest of the tree consists of the extended gap forest $\mathfrak{F}$. The anchor *self* is replaced by the node v that represents the anchor of the elementary tree. A sample elementary tree and its corresponding lexicon entry are shown in Figure 6.5(a) and (b).

In contrast to a TAG elementary tree, such an extended elementary trees now may have an arbitrary number of foot nodes. Furthermore, all edges except the outgoing edges of the root are indexed, since they originate from an indexed ordered forest. In fact, the outgoing edges of the root are of a different nature compared to the other edges: whereas the other edges describe the local order contained in the lexicon entry, the outgoing edges from the root are only an artefact of our syntax. They transform the forest into a tree and allow us to visualise the combination of two lexicon entries as a combining operation over trees.

In TAG, there are two kinds of combining operations: substitution inserts a tree without foot node at a leaf, and adjunction inserts a tree with one foot node at an inner node. In the extension, these two operations are only special cases of a more general combining operation. If a node v of a tree T_1 has outgoing edges with labels $1, \dots, k$, a tree T_2 with nodes $\star_1, \dots, \star_k$ can be inserted at v (see the condition of Lemma 6.2). The subtrees below v with some label i are then placed below the i-th foot node of T_2. The general scheme of this operation for trees with two foot nodes is given in Figure 6.4.

Example An example for this combining operation is shown in Figure 6.5. In this example, tree T_2 is inserted into T_1 at node B. The material below the edges with index 1 is placed below the first foot node of T_2, the material below the edge with index 2 is placed below the second foot node. Note that the outgoing edges of the root nodes of the elementary trees are not indexed, since they do not belong to the gap forest. As in TAG, no combining operations may take place at root nodes, and hence the indices are not needed. The tree in Figure 6.5(d) has no edge indices at all, since there is no inner node left where a combining operation could take place. $\dashv$

6.6 Towards some underspecification

So far, we were mostly concerned with the description of single drawings. Sets of drawings with a similar structure are described with underspecification. Underspecification in the description of the tree structure means, for example, that the valencies of a node

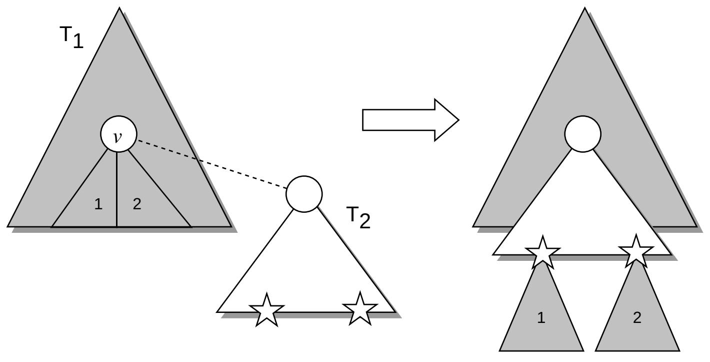
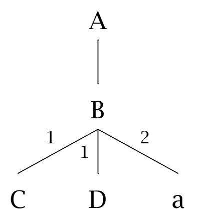
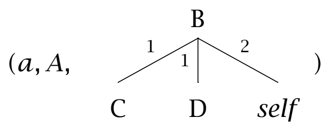


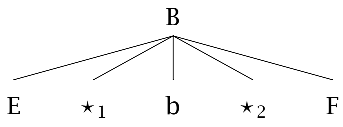
Figure 6.4: Combining operation for trees with two foot nodes



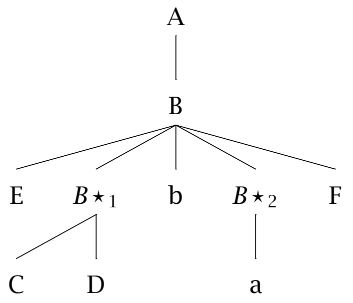
(a) extended elementary tree T_1



(b) corresponding lexicon entry for T_1



(c) extended elementary tree T_2



(d) result of the combining operation

Figure 6.5: An example for the combining operation of the TAG extension

are not completely fixed. Underspecification in the order means, that there are several alternatives to order a node and its valencies.

In CFG, TAG, and the description language we presented in Section 6.4, the order and tree structure in a single lexicon entry is always completely specified. Underspecification is realised in these systems as enumeration of the different alternatives. For example, the two context-free rules of Figure 6.1(b) on page 44 together state that the node a has two valencies B and C, where C is rightmost, but the order among a and B is not specified.

Instead of enumerating all alternatives, a description language can also offer a syntax that is powerful enough to leave some aspects of a lexicon entry underspecified. In the next chapter, we develop a language for the underspecified description of gap forests. By replacing the forests in the lexicon entries of our description language by such an underspecified description, it is possible to specify only partially the order within a single lexicon entry.

Chapter 7

A saturation algorithm for constraints on ordered forests

In this chapter, we introduce a constraint language for ordered forests and a saturation algorithm that decides in *NP* time whether an expression of that language is consistent or not. It can also be used to enumerate all solutions, that is, to enumerate all ordered forests described by the expression. The second half of the chapter is concerned with an extension to describe indexed ordered forests.

The algorithm is related to saturation algorithms that have been developed to solve dominance constraints. The first of these algorithms was presented by Koller, Niehren and Treinen [9]; extended versions for richer constraint languages have been proposed by Duchier and Niehren [5] and Koller [7].

7.1 Introduction to saturation algorithms

The algorithm is formulated with the help of saturation rules that successively enlarge a given constraint φ until it describes a solution, or an inconsistency is detected. The constraint φ is a conjunction of atomic formulas. A saturation rule is applicable, if φ contains all atoms of the precondition, and no atom of the conclusion. If a rule is applied, one atom of its conclusion is added to φ. There are two different kinds of rules: propagation rules and distribution rules.

Propagation rules have only one atom in their conclusion, and are hence deterministic. They state that any solution satisfying the precondition also satisfies the conclusion. Some propagation rules have no atom of the constraint language but the special item *fail* as their conclusion. This represents the fact that any constraint satisfying the precondition is inconsistent.

Distribution rules have more than one atom in their conclusion, and each solution that satisfies the precondition satisfies at least one of them. If we only want to check the consistency of a constraint, we nondeterministically choose a satisfiable alternative, if one exists. If we want to enumerate all solutions, we have to treat all alternatives separately, since the extension of φ with any atom of the conclusion may lead to a solution.

The saturation algorithm applies the rules until no more new atoms can be added. The constraint is then called saturated. If *fail* is derived, the constraint is inconsistent, otherwise the final constraint describes a solution. The order in which we apply the rules does not matter (as long as we are not concerned with efficiency), since the rules are monotonic in the sense that only new atoms are added. Furthermore, the preconditions only require the existence of some atoms, not their absence, and consequently a rule whose preconditions are fulfilled remains applicable no matter which other rules are applied in between.

Saturation algorithms of that kind can be implemented in a constraint programming system. Such a system applies the nondeterministic distribution rules only if no more deterministic propagation rules are applicable. This strategy is efficient since it mimimises the search tree that is generated by the distribution rules.

7.2 Syntax and semantics

We want to describe ordered forests with the constraint language. Ordered forests consist of nodes and two binary relations on it: successorship for the forest represented by $\lhd$, and the order $<$ among disjoint nodes. The nodes are represented by the variables of the constraint. More precisely, there is a one to one correspondence between the variables of the constraint and the nodes of the forest: distinct variables describe distinct nodes, and each node must correspond to a variable. To keep things simple, we assume the identity of variables in the tree and nodes in the constrained forest.

A constraint φ ranging over variables $Vars(\varphi) = \{x, y, \dots\}$ is a set of atoms of the form xRy, where $R \subseteq \{<, >, \lhd^+, \rhd^+\}$. The set R represents a disjunction. An atom xRy has the semantics that for at least one $r \in R$, xry holds in the constrained forest. Note that only transitive successorship and not immediate successorship is expressible.

7.3 Saturation rules

The set of saturation rules is shown in Figure 7.1. With the help of these rules, the saturations and solved forms of a constraint can be computed.

Definition 7.1 (Saturation and solved form) A saturation of a constraint φ is a constraint φ' that is obtained by applying all inference rules exhaustively to φ. We call φ' a solved form, if it is failure-free, that is no rule with *fail* as conclusion is applicable on it. ⊣

The saturation rules can be distinguished according to their role within the inference system. In particular, there are two main classes of rules. The rules INIT, INTERSECTION, FAIL, and CHOICE build a kind of core structure. They are mainly concerned with the general fact that the atoms of the constraints represent disjunctions, and they are independent of the concrete semantics of the describes relations. In contrast to that, the remaining rules ensure certain properties of the relations. These remaining rules hence characterise the class of structures that we want to accept as solutions, and are more specific to the problem that is solved by the algorithm.

Propagation rules:

$$\textsc{init}\ \frac{}{x\{<,>,\lhd^+,\rhd^+\}y}\ x \neq y$$

$$\textsc{intersection}\ \frac{xR_1y \qquad xR_2y}{xRy}\ R = R_1 \cap R_2$$

$$\textsc{fail}\ \frac{x\emptyset y}{\mathit{fail}}$$

$$\textsc{order trans}\ \frac{x\{<\}y \qquad y\{<\}z}{x\{<\}z} \qquad \frac{x\{\lhd^+\}y \qquad y\{\lhd^+\}z}{x\{\lhd^+\}z}\ \textsc{dom trans}$$

$$\textsc{order to dom (1)}\ \frac{x\{<\}y \qquad x\{\lhd^+\}x'}{x'\{<\}y} \qquad \frac{x\{<\}y \qquad y\{\lhd^+\}y'}{x\{<\}y'}\ \textsc{order to dom (2)}$$

$$\textsc{order inverse (1)}\ \frac{x\{>\}y}{y\{<\}x} \qquad \frac{x\{<\}y}{y\{>\}x}\ \textsc{order inverse (2)}$$

$$\textsc{dom inverse (1)}\ \frac{x\{\rhd^+\}y}{y\{\lhd^+\}x} \qquad \frac{x\{\lhd^+\}y}{y\{\rhd^+\}x}\ \textsc{dom inverse (2)}$$

Distribution rules:

$$\textsc{choice}\ \frac{xRy}{xR_1y \vee xR_2y}\ R = R_1 \uplus R_2$$

Figure 7.1: The set of saturation rules

We now prove some properties of our saturation rules to simplify the subsequent proofs for termination, soundness and completeness of the system. We first address the properties that are ensured by the core rules INIT, INTERSECTION, FAIL, and CHOICE.

Lemma 7.2 Let φ be a failure-free constraint on which the rules INTERSECTION and FAIL are not applicable. Then for all x,y there are no disjoint sets R_1,R_2 such that $xR_1y \in \varphi$ and $xR_2y \in \varphi$. □

PROOF Let $xR_1y \in \varphi$ and $xR_2y \in \varphi$ for some disjoint sets R_2, R_2. Since INTERSECTION is not applicable, also $x\emptyset y \in \varphi$. This would make the FAIL rule applicable and contradicts the assumption that φ is failure-free. ■

Lemma 7.3 Let φ be a constraint on which the rule CHOICE is not applicable. Then for each atom $xRy \in \varphi$, there exists an $r \in R$ such that $x\{r\}y \in \varphi$. □

PROOF We prove the statement by induction over $|R|$. If $|R| = 1$ then $R = \{r\}$ and hence the atom xRy itself is the atom $x\{r\}y$. For $|R| = k + 1$ we know that there exist R_1 and R_2 with $R_1 \uplus R_2 = R$ and either $xR_1y \in \varphi$ or $xR_2y \in \varphi$ since otherwise CHOICE would be applicable. Without loss of generality we assume $xR_1y \in \varphi$. By induction hypothesis, there exists a singleton set $\{r\}$ with $r \in R_1$ and $x\{r\}y$. Since $R_1 \subset R$, also $r \in R$. ■

Lemma 7.4 Let φ be a failure-free constraint where none of the rules INIT, INTERSECTION, FAIL and CHOICE is applicable. Then for each pair of variables x,y with $x \neq y$ there exists exactly one $r \in \{<, >, \lhd^+, \rhd^+\}$ such that $x\{r\}y \in \varphi$. □

PROOF We first show by contradiction that at most one such r exists. Assume the existence of r_1 and r_2 with $r_1 \neq r_2$ such that both $x\{r_1\}y$ and $x\{r_2\}y$ are atoms in φ. Since $\{r_1\}$ and $\{r_2\}$ are disjoint this contradicts Lemma 7.2.

Now we show that at least one r exists such that $x\{r\}y \in \varphi$. Since INIT is not applicable, $x\{<, >, \lhd^+, \rhd^+\}y \in \varphi$. Hence by Lemma 7.3 for at least one $r \in \{<, >, \lhd^+, \rhd^+\}$ $xry \in \varphi$. ■

So far we have shown general properties that are independent of the semantics of the relations $<$, $>$, $\lhd^+$, and $\rhd^+$. Now we show how certain properties of these relations are guaranteed by the inference rules.

Lemma 7.5 A transitive, irreflexive relation does not contain cycles. □

PROOF Let a relation r be transitive and irreflexive. Assume a cycle x_1rx_2, x_2rx_3, …, x_nrx_1. Transitivity implies x_1rx_1 which is not possible since r is irreflexive. ■

Corollary 7.6 The relation $\{\lhd^+\}$ does not contain a cycle for any saturated failure-free constraint φ. □

Proof $\{\triangleleft^+\}$ is transitive since it is saturated under DOM TRANS. It is irreflexive since any atom $x\{\triangleleft^+\}x$ implies via DOM INVERSE the existence of an atom $x\{\triangleright^+\}x$, which contradicts Lemma 7.2. ∎

Corollary 7.7 The relation $\{<\}$ does not contain a cycle for any saturated failure-free constraint φ. □

Proof $\{<\}$ is transitive since it is saturated under ORDER TRANS and it is irreflexive since any atom $x\{<\}x$ implies via ORDER INVERSE the existence of an atom $x\{>\}x$, which contradicts Lemma 7.2. ∎

Corollary 7.8 Since $\{\triangleleft^+\}$ is acyclic, the transitive reduction of $\{\triangleleft^+\}$ is unique (see the first paragraph in Section 2.1). □

7.4 Termination and soundness

Termination guarantees that a saturation of a constraint can be computed with a finite number of saturation steps.

Lemma 7.9 (Termination) For a constraint with n variables the algorithm performs at most $O(n^2)$ saturation steps. □

Proof Each saturation step adds one atom xRy that was not there before. For n variables, there are at most $16 * n^2$ of these atoms, since there are 16 subsets of $\{<, >, \triangleleft^+, \triangleright^+\}$, and thus 16 possible values for R. Since no variables are added and no atoms are deleted during the saturation, the algorithm terminates after at most $16 * n^2$ saturation steps. ∎

Soundness ensures that inconsistency is only reported for constraints that have no solution.

Lemma 7.10 (Soundness) If all saturations of a constraint φ contain *fail*, φ is unsatisfiable. □

Proof We prove the contrapositive: if φ is satisfiable, it has a saturation that does not contain *fail*.

Each propagation rule is sound in the sense that the atoms in the precondition imply the atom in the conclusion. Hence, if during the saturation process a propagation rule with conclusion c is applied on a satisfiable constraint φ, then $\varphi \wedge c$ is also satisfiable.

The distribution rules have the property that, if the precondition holds, at least one of the conclusions holds, too. Consequently, if a distribution rule with conclusions $c_1, \ldots, c_n$ is applied on a satisfiable constraint φ, there is one c_i such that $\varphi \wedge c_i$ is satisfiable. Hence, in the search tree that is spanned by the distribution rules, there is at least one path that always remains satisfiable. Furthermore, the search tree is finite due to Lemma 7.9. The last node of the path that always remains satisfiable then represents a saturation that is satisfiable and hence does not contain *fail*. ∎

7.5 Completeness

Completeness is the converse of soundness: If a constraint φ is unsatisfiable, all saturations of φ contain *fail*. An equivalent formulation, as the contrapositive, is: if φ has a solved form φ' then φ is satisfiable. Since φ is satisfiable if φ' is satisfiable, it suffices to show:

Theorem 7.11 (Completeness) Each constraint in solved form is satisfiable. □

Proof The proof will be split into three parts. First we construct a solution for an arbitrary solved form φ, second we show that the solution is an ordered forest, and third we show that this forest satisfies the constraint.

1. (*Construction of a possible solution*)
The solution is the ordered forest $(Vars(\varphi); \triangleleft, <)$ with

$$\begin{array}{lcl} \triangleleft^+ & := & \{\,(x,y) \mid x\{\triangleleft^+\}y \in \varphi\,\} \\ < & := & \{\,(x,y) \mid x\{<\}y \in \varphi)\,\} \\ \triangleleft & := & \text{transitive-reduction}(\triangleleft^+) \end{array}$$

This is well-defined, if and only if the transitive reduction of $\triangleleft^+$ is unique. This is the case due to Corollary 7.8.

2. (*Ordered forest*) To ensure that our constructed solution is an ordered forest, we show that

(a) $\triangleleft$ is acyclic.

(b) each node has at most one $\triangleleft$ predecessor.

(c) $<$ is transitive.

(d) $<$ is acyclic.

(e) $v \perp w \Leftrightarrow (v < w \vee w < v)$

(f) $v < w \Rightarrow \forall v' \in \triangleleft^+ v, \forall w' \in \triangleleft^+ w. v' < w'$

(a) and (b) ensure that $(V; \triangleleft)$ forms a forest, (c) and (d) ensure that $(V; <)$ is a partial order and (e) and (f) are the two additional constraints (I) and (II) in the definition of ordered forests (Definition 2.9).

(a) $\triangleleft$ is acyclic, since $\triangleleft^+$ is acyclic as we showed in Corollary 7.6.

(b) Assume a node u has two $\triangleleft$ predecessors, that is $v \triangleleft u \wedge w \triangleleft u$. Then $v\{\triangleleft^+\}u \in \varphi$ and $w\{\triangleleft^+\}u \in \varphi$, and furthermore $v\{\triangleleft^+\}w \notin \varphi$ and $v\{\triangleright^+\}w \notin \varphi$. Due to Lemma 7.4, for exactly one $r \in \{<, >, \triangleleft^+, \triangleright^+\}$, we have that $v\{r\}w \in \varphi$; the remaining two alternatives are $v\{<\}w \in \varphi$ or $v\{>\}w \in \varphi$.

We first consider $v\{<\}w \in \varphi$. The rule ORDER TO DOM (1) infers from $v\{<\}w \in \varphi$ and $v\{\lhd^+\}u \in \varphi$, that $u\{<\}w \in \varphi$. Together with $w\{\lhd^+\}u \in \varphi$ this implies via DOM INVERSE also $u\{\rhd^+\}w \in \varphi$, and hence contradicts Lemma 7.4.

The case $v\{>\}w \in \varphi$ is obtained symmetrically to $v\{<\}w \in \varphi$ by swapping v and w and applying ORDER INVERSE.

(c) Assume $u < v$ and $v < w$. Then $u\{<\}v \in \varphi$ and $v\{<\}w \in \varphi$. By ORDER TRANS also $u\{<\}w \in \varphi$ and hence $u < w$.

(d) The acyclicity of $<$ is shown in Corollary 7.7.

(e) $v \perp w \Leftrightarrow (v < w \vee w < v)$ is equivalent to the fact that an atom $v\{<\}w$ or $w\{<\}v$ is in φ if and only if no atom $v\{\lhd^+\}w$ or $w\{\lhd^+\}v$ is in φ. With the rules DOM INVERSE and ORDER INVERSE we can reduce this to the proposition that $v\{<\}w$ or $v\{>\}w$ is in φ if and only if no atom $v\{\lhd^+\}w$ or $v\{\rhd^+\}w$ is in φ, which is subsumed by Lemma 7.4.

(f) Let $v < w$ and let v',w' be arbitrary nodes such that $v \lhd^+ v'$ and $w \lhd^+ w'$. Consequently, $v\{<\}w \in \varphi$, and furthermore $v\{\lhd^+\}v' \in \varphi$ and $w\{\lhd^+\}w' \in \varphi$, since $\{\lhd^+\}$ is transitive due to DOM TRANS. By ORDER TO DOM 1 also $v'\{<\}w \in \varphi$, and by ORDER TO DOM 2 then $v'\{<\}w' \in \varphi$. It follows that $v' < w'$.

3. (*Solution*) We show that the constructed solution satisfies the constraint by demonstrating that each single atom is satisfied. Due to Lemma 7.3 it suffices to show that the atoms xRy with $|R| = 1$ are satisfied: any other atom xRy is satisfied if $xR'y$ is satisfied for some $R' \subseteq R$.

There are four different kinds of atoms xRy with $|R| = 1$: $x\{<\}y$, $x\{\lhd^+\}y$, $x\{>\}y$ and $x\{\rhd^+\}y$. The first two are satisfied by construction of $<$ and $\lhd$, respectively. The atoms of the third and fourth kind are satisfied since the rules ORDER INVERSE and DOM INVERSE imply the existence of constraints with the same semantics that are of kind one or two, respectively. ∎

7.6 Extension to the indexed case

In this section, we discuss how the system can be extended to handle indexed ordered forest. We will use terms like *core system* or *core syntax* to refer to parts of the original system without the extensions.

7.6.1 The new syntax and semantics

We extend our syntax by adding an index to the $\lhd^+$ relation. The new atoms are of the form xRy with $x, y \in Vars(\varphi)$ and $R \subseteq \{<, >, \lhd_1^+, \dots, \lhd_k^+, \rhd_1^+, \dots, \rhd_k^+\}$ for some $k \in \mathbb{N}$.

An indexed ordered forest $\mathfrak{F}$ satisfies an atom $x\{\triangleleft_i^+\}y$, if the node x is an ancestor of y in $\mathfrak{F}$, and the first edge on the path from x to y has index i. An atom $x\{\triangleright_i^+\}y$ has the same semantics as $y\{\triangleleft_i^+\}x$.

We can assume that $k \leq |Vars(\varphi)|$, since a forest with $|Vars(\varphi)|$ nodes has less than $|Vars(\varphi)|$ edges, and hence, a constraint with more than $|Vars(\varphi)|$ different edge indices is always unsatisfiable.

We define a translation of a constraint φ into a constraint of the core syntax as follows:

$$\begin{aligned}
[\![\varphi]\!] &:= \bigcup\{\, \{x[\![R]\!]y\} \mid xRy \in \varphi \,\} \\
[\![R]\!] &:= \bigcup\{\, [\![r]\!] \mid r \in R \,\} \\
[\![<]\!] &:= \{<\} \\
[\![\triangleleft_i^+]\!] &:= \{\triangleleft^+\}
\end{aligned}$$

The translation does nothing else than removing all indices from the symbols $\triangleleft_i^+$ such that they match the old syntax.

7.6.2 The new set of saturation rules

The new saturation rules are shown in Figure 7.2. The only new rule is INDEX, which ensures a valid indexing. All other rules remain unchanged or have only minor modifications to be suitable for the extended syntax (Modified rules are marked with a prime, for example, INIT' instead of INIT).

The coherence of the core system and the new system is expressed in the following lemma:

Lemma 7.12 If a constraint φ is saturated and failure-free with respect to the new saturation rules, the constraint $[\![\varphi]\!]$ is saturated and failure free with respect to the core system. □

PROOF If $[\![\varphi]\!]$ would not be saturated and failure-free, at least one rule must be applicable to add new atoms or indicate a failure. But for every rule in the core system there is a corresponding rule in the new system that only differs in the additional indices. We show the argumentation in detail for the rules DOM TRANS and FAIL, all other cases are analogous.

Assume that DOM TRANS would be applicable in $[\![\varphi]\!]$. Then the atoms $x\{\triangleleft^+\}y \in [\![\varphi]\!]$ and $y\{\triangleleft^+\}z \in [\![\varphi]\!]$ exist. These atoms can only exist if there are atoms $x\{\triangleleft_i^+\}y \in \varphi$ and $y\{\triangleleft_j^+\}z \in \varphi$ that were translated into the corresponding atoms of $[\![\varphi]\!]$. Hence, DOM TRANS' would be applicable to φ, which contradicts the assumption that φ is saturated and failure-free.

Assume that FAIL is applicable in $[\![\varphi]\!]$. Then $x\emptyset y \in [\![\varphi]\!]$ for some x, y. This atom can only exist in $[\![\varphi]\!]$ if it also exists in φ. But then FAIL would be applicable to φ, which contradicts the fact that φ is failure-free. ■

Propagation rules:

$$\text{INIT'}\ \frac{}{x\{<,>,\lhd_1^+,\ldots,\lhd_k^+,\rhd_1^+,\ldots,\rhd_k^+\}y}\ x \neq y$$

$$\text{INTERSECTION}\ \frac{xR_1y \qquad xR_2y}{xRy}\ R = R_1 \cap R_2$$

$$\text{FAIL}\ \frac{x\emptyset y}{\mathit{fail}}$$

$$\text{ORDER TRANS}\ \frac{x\{<\}y \qquad y\{<\}z}{x\{<\}z} \qquad \frac{x\{\lhd_i^+\}y \qquad y\{\lhd_j^+\}z}{x\{\lhd_i^+\}z}\ \text{DOM TRANS'}$$

$$\text{ORDER TO DOM' (1)}\ \frac{x\{<\}y \qquad x\{\lhd_i^+\}x'}{x'\{<\}y} \qquad \frac{x\{<\}y \qquad y\{\lhd_i^+\}y'}{x\{<\}y'}\ \text{ORDER TO DOM' (2)}$$

$$\text{ORDER INVERSE (1)}\ \frac{x\{>\}y}{y\{<\}x} \qquad \frac{x\{<\}y}{y\{>\}x}\ \text{ORDER INVERSE (2)}$$

$$\text{DOM INVERSE' (1)}\ \frac{x\{\rhd_i^+\}y}{y\{\lhd_i^+\}x} \qquad \frac{x\{\lhd_i^+\}y}{y\{\rhd_i^+\}x}\ \text{DOM INVERSE' (2)}$$

$$\text{INDEX}\ \frac{x\{\lhd_i^+\}y \qquad x\{\lhd_j^+\}z}{y\{<\}z}\ i < j$$

Distribution rules:

$$\text{CHOICE}\ \frac{xRy}{xR_1y \vee xR_2y}\ R = R_1 \uplus R_2$$

Figure 7.2: The set of saturation rules for the indexed case

7.6.3 Termination and soundness

For the extended system, we again have to give termination and soundness arguments.

Lemma 7.13 (Termination) For a constraint with n variables, the algorithm performs at most $O(n^4)$ saturation steps. □

Proof The proof is analogous to the termination of the core system (see Lemma 7.9). Instead of $4^2 = 16$ different relations between two variables x and y, we now have $(2 * n + 2)^2$ different possible relations, since the set $\{<, >, \lhd_1^+, \dots, \lhd_n^+, \rhd_1^+, \dots, \rhd_n^+\}$ has $2 * n + 2$ elements. With n^2 different possibilities to choose a pair (x, y), the number of atoms xRy is then $n^2 * (2 * n + 2)^2 = 4n^4 + 8n^3 + 4n^2 \in O(n^4)$.

Lemma 7.14 (Soundness) If all saturations of the inference system contain *fail*, the constraint is unsatisfiable. □

Proof The proof is analogous to the proof for the core system (see Lemma 7.10), since all new saturation rules are also sound in the sense that their preconditions imply their conclusion. ■

7.6.4 Completeness

With the help of the translation of a constraint φ into a constraint $[\![\varphi]\!]$ of the core system, it is possible to reuse the completeness proof of the core system. We therefore do not need to care about all the details that remained the same, but concentrate on the things that changed with the extension.

Theorem 7.15 (Completeness) Each constrained in solved form is satisfiable. □

Proof The completeness proof again has the same structure as in the proof for the core system (Theorem 7.11): we construct a solution for a saturated failure-free constraint φ, show that it describes a (now indexed) ordered forest, and finally show that this forest satisfies the constraint.

1. (*Construction of a solution*) The solution for φ is an ordered forest together with an indexing function. We define the ordered forest $(\mathit{Vars}(\varphi); \lhd, <)$ to be the solution of $[\![\varphi]\!]$ with respect to the core system. This solution exists due to Lemma 7.12. The indexing function which is now additionally required is defined as

$$\mathit{index}(v, w) := i \iff v \lhd w \wedge v\{\lhd_i^+\}w \in \varphi$$

This defines a function, since there are no i, i' with $v\{\lhd_i^+\}w \in \varphi$, $v\{\lhd_{i'}^+\}w \in \varphi$ and $i \neq i'$ due to Lemma 7.2.

2. (*Indexed ordered forest*) The fact that $(\mathit{Vars}(\varphi); \lhd, <)$ is an ordered forest follows from the fact that it is a solution for $[\![\varphi]\!]$. It remains to show that *index* is a valid indexing

function. The only restriction for *index* is condition (III) of the definition of an indexed ordered forest (see page 6). It is fulfilled due to the following implications.

$$\begin{aligned} & v \lhd w \wedge v \lhd w' \wedge \mathsf{index}(v,w) < \mathsf{index}(v,w') \\ \Rightarrow\ & \exists i,j \in \mathbb{N}. i < j \wedge v\{\lhd_i^+\}w \in \varphi \wedge v\{\lhd_j^+\}w' \in \varphi \\ \Rightarrow\ & w\{<\}w' \in \varphi \\ \Rightarrow\ & w < w' \end{aligned}$$

The second implication follows from the application of the rule INDEX.

3. (*Solution*) We have to show that each atom of φ is satisfied by the solution. As in the completeness proof of the core system, Lemma 7.3 implies that it suffices to show that atoms xRy with $|R| = 1$ are satisfied.

Atoms of kind $x\{<\}y$ or $y\{>\}x$ are satisfied since they are also atoms of $[\![\varphi]\!]$, and the forest is a solution for $[\![\varphi]\!]$. For atoms of kind $x\{\rhd_i^+\}y$, due to DOM INVERSE' there exists a corresponding item of kind $x\{\lhd_i^+\}y$ that needs to be satisfied and has the same semantics. It remains to show that the atoms of kind $x\{\lhd_i^+\}y$ are satisfied.

For an atom $x\{\lhd_i^+\}y$, there is a licensing path in the forest because $x\{\lhd^+\}y \in [\![\varphi]\!]$, and the forest is also a solution of $[\![\varphi]\!]$. We must show that the first edge of this path has index i. We first consider the case that $x \lhd y$ is an edge in the forest (that is, the path has length one). Then the edge has index i by definition of the indexing function. If, on the other hand, x only transitively dominates y in the forest structure and the first edge on this path has index i' there are atoms $x\{\lhd^+\}z_1, z_1\{\lhd^+\}z_2, \dots, z_n\{\lhd^+\}y$ in $[\![\varphi]\!]$ for some $z_1, \dots, z_n$ and corresponding atoms $x\{\lhd_{i'}^+\}z_1, z_1\{\lhd_j^+\}z_2, \dots, z_n\{\lhd_k^+\}y$ in φ. By iteratively applying DOM TRANS', we infer $x\{\lhd_{i'}^+\}z_1 \in \varphi$, $x\{\lhd_{i'}^+\}z_2 \in \varphi$, …, and finally also $x\{\lhd_{i'}^+\}y \in \varphi$. By Lemma 7.2 it follows that $i = i'$, since both $x\{\lhd_{i'}^+\}y$ and $x\{\lhd_i^+\}y$ are elements of φ. ∎

7.7 Further extensions

In our description language for well-nested drawings (Chapter 6), we want to use the constraint language to describe the extended gap forests of the nodes. In that context, we have special kinds of nodes, namely the foot nodes and the anchor. Furthermore, the forests must have the structural properties stated in Lemma 6.1 (see page 45). In other words the constraints must be interpreted over a restricted domain.

We account for these additional requirements with two modifications:

- The syntax is extended with constants for foot nodes and the anchor.
- Additional saturation rules ensure that each solution fulfils the required structural properties.

The syntax is simply extended with some constants $Cons(\varphi) = \{self, \star_1, \dots, \star_l\}$; atoms have now the form xRy with $x, y \in Vars(\varphi) \cup Cons(\varphi)$. The constant *self* represents the anchor, each constant $\star_i$ represents a foot node of the forest. The additional

$$\textsc{self leaf}\ \frac{self\{\lhd_i^+\}w}{fail}$$

$$\textsc{gap leaf}\ \frac{\star_i\{\lhd_j^+\}w}{fail}$$

$$\textsc{gaps ordered}\ \frac{}{\star_i\{<\}\star_j}\ i<j$$

$$\textsc{index dense}\ \frac{w\{\lhd_i^+\}w'}{\bigvee_{w''\in Vars(\varphi)\cup Cons(\varphi)} w\{\lhd_{i-1}^+\}w''}\ i>1$$

$$\textsc{gaps separated}\ \frac{}{\bigvee_{w\in Vars(\varphi)\cup Cons(\varphi)}\ \bigvee_{r\in R_{\text{full}}} (\star_i\{r\}w \wedge \star_j(R_{\text{full}}-\{r\})w)}\ i\neq j$$

$$\textsc{gap not leftmost}\ \frac{\forall v \in (Vars(\varphi)\cup Cons(\varphi)-\{\star_i\}).\ \star_i\{<\}v}{fail}$$

$$\textsc{gap not rightmost}\ \frac{\forall v \in (Vars(\varphi)\cup Cons(\varphi)-\{\star_i\}). v\{<\}\star_i}{fail}$$

Figure 7.3: Additional saturation rules

rules are shown in Figure 7.3. In the rules the symbol R_{full} is used as an abbreviation for the set $\{<,>,\lhd_1^+,\ldots,\lhd_k^+,\rhd_1^+,\ldots,\rhd_k^+\}$.

Termination and soundness are almost not affected by the extensions.

Lemma 7.16 (Termination) For a constraint φ with $n = |Vars(\varphi)\cup Cons(\varphi)|$, the algorithm performs at most $O(n^4)$ saturation steps. □

Proof The proof is analogous to the proof of Lemma 7.13. ■

Lemma 7.17 (Soundness) If all saturations of the inference system contain *fail*, the constraint is unsatisfiable. □

Proof The proof is analogous to the proof for the core system (see Lemma 7.10), since all new saturation rules are also sound in the sense that their preconditions imply their conclusion. The implications of the new rules hold due to the structural properties of gap forests stated in Lemma 6.1. The rules self leaf and gap leaf are sound due to Lemma 6.1(b), gap ordered due to Lemma 6.1(c), index dense due to Lemma 6.1 (d), gaps separated due to Lemma 6.1 (f) and gap not leftmost and gap not rightmost due to part (e) of Lemma 6.1. ■

Lemma 7.18 (Completeness) Each constraint φ in solved form has a valid gap forest as solution. □

PROOF The solution $(W; \lhd, <)$ has a node set $W = \mathit{Vars}(\varphi) \cup \mathit{Cons}(\varphi)$ and $\lhd$, $<$ and the indexing function are constructed in the same way as without the extension (see proof of Theorem 7.15). We only need to show that this forest has the structural properties (b)–(f) stated in Lemma 6.1.

(b) $\lhd \mathit{self} = \lhd \star_1 = \cdots \lhd \star_k = \emptyset$ holds since otherwise φ would contain atoms that correspond to the outgoing edges of *self* or the $\star$-nodes and SELF LEAF or GAP LEAF would derive failure.

(c) $\forall i, j \in [1,k]. i < j \Rightarrow \star_i < \star_j$ holds since by GAPS ORDERED the constraint φ contains corresponding atoms $\star_i\{<\}\star_j$.

(d) $\forall w, w' \in W. w \lhd_i w' \wedge i > 1 \Rightarrow \exists w''. w \lhd_{i-1} w''$ holds since by INDEX DENSE, the constraint φ contains corresponding atoms $w\{\lhd^+_{i-1}\}w''$.

(e) $\nexists i \in [1,k].(\forall w \in W - \{\star_i\}. w < \star_i) \vee (\forall w \in W - \{\star_i\}. \star_i < w)$ also holds: if there would exist such an i, either for all $w \in W - \{\star_i\}$ there would exist atoms $w\{<\}\star_i$ in φ, or for all of them there would exist atoms $\star_i\{<\}w$ in φ. In the first case, GAP NOT RIGHTMOST would infer failure, in the second case, GAP NOT LEFTMOST would infer failure.

(f) $\forall \star_i, \star_j \in W$ with $i \neq j. \exists w \in W, R \in \{<, >, \lhd_1, \ldots, \lhd_k\}. wR\star_i \wedge \neg wR\star_j$ holds due to rule GAPS SEPARATED. This rule nondeterministically chooses R and w and adds corresponding atoms to φ. ■

7.8 Towards more efficient algorithms

Due to its nondeterministic nature, we do not claim the algorithm presented in this chapter to be efficient. Nevertheless, it offers deep insights in the nature of the problem that it solves, and provides a basis for the development of faster algorithms. As mentioned in the beginning of this chapter, the algorithm is similar to saturation algorithms that were developed to solve dominance constraints. The experience that was gained with these algorithms helped to identify a subclass of dominance constraints, called normal dominance constraints, and to develop deterministic polynomial algorithms for this subclass (see Koller, Mehlhorn and Niehren [8]).

There are good chances that polynomial algorithms of that kind can also be developed for the problem that we discussed here. A first important step towards this direction was made by Bodirsky and Kutz [3], where a deterministic polynomial algorithm to check the consistency of partial tree descriptions was proposed. How far this algorithm can be adapted to solve our task is an interesting question for future work. Basically, the algorithm and the underlying description language must be modified in such a way that they can handle indexed ordered forests instead of unordered trees.

Chapter 8

Conclusions and future work

This chapter summarises the contents of the thesis, and indicates promising starting points for future work.

In this thesis, we introduced drawings as models of syntactic structure. After the formal definition of drawings, we mainly investigated structural properties of drawings, and aspects that are concerned with grammar formalisms.

8.1 Structural properties of drawings

We presented two structural properties of drawings: gap degree and well-nestedness. Both of them are relaxations of projectivity.

We developed polynomial algorithms to determine the gap degree of a drawing and to decide whether a drawing is well-nested. These algorithms demonstrate that the properties are not only of theoretical interest, but also computationally tractable. Whereas the computation of the gap degree is linear in the number of nodes, the presented algorithms which test well-nestedness have a quadratic worst case complexity. Further research is needed to investigate whether there also exist algorithms with a better worst case complexity.

A question that we have not addressed, so far, is the linguistic relevance of well-nestedness and gap degree. Are dependencies in natural language sentences usually well-nested? Is there a bound on the gap degree? For many languages there exist dependency treebanks, whose dependency structures can be interpreted as drawings. A statistical evaluation of these treebanks may give answers to these questions.

8.2 Drawings and grammar formalisms

One main contribution of this thesis is the characterisation of TAG drawings as well-nested drawings with a gap degree of at most one. On the one hand, this result shows that well-nestedness and gap degree are properties with theoretical relevance. On the other hand, this result characterises the expressivity of TAG. There is an important property of grammar formalisms: up to what limits they can derive non-projective structures. Our characterisation gives a precise answer to that question for TAG. The characterisation is limited so far to single drawings. Since grammars describe sets of structures, future work must investigate which sets of drawings are describable by a TAG grammar.

As a further benefit, the characterisation of TAG drawings offers a way to compare TAG to other grammar formalisms. For such a comparison, the classes of drawings that are induced by other grammar formalisms must be identified. Possibly other formalisms are also able to derive non-projective drawings or drawings with a higher gap degree than one. This would indicate that these grammar formalisms are more expressive than TAG. Drawings corresponding to some other grammar formalism possibly cannot be characterised by means of well-nestedness and gap degree. Such observations could also lead to the development of further relaxations of projectivity.

Having identified the limitations of the structures derived by a formalism, a further question is how to get over these limitations. We have sketched an extension of TAG that is able to derive drawings with a gap degree higher than one. Since higher expressivity usually comes along with increased parsing complexity, an important open question is whether efficient parsing of this extension is possible.

Bibliography

[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[2] Manuel Bodirsky, Marco Kuhlmann, and Mathias Möhl. Well-nested drawings as models of syntactic structure. In *10th Conference on Formal Grammar and 9th Meeting on Mathematics of Language*, Edinburgh, Scotland, UK, 2005.

[3] Manuel Bodirsky and Martin Kutz. Pure dominance constraints. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2002)*, 2002.

[4] Mike Daniels and W. Detmar Meurers. Improving the efficiency of parsing with discontinuous constituents. In Shuly Wintner, editor, *Proceedings of NLULP'02: The 7th International Workshop on Natural Language Understanding and Logic Programming*, number 92 in Datalogiske Skrifter, pages 49–68, Copenhagen, 2002. Roskilde Universitetscenter.

[5] Denys Duchier and Joachim Niehren. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 326–341. Springer, July 2000.

[6] Aravind Joshi and Yves Schabes. *Handbook of Formal Languages*, volume 3, chapter Tree Adjoining Grammars, pages 69–123. Springer, 1997.

[7] Alexander Koller. *Constraint-based and graph-based resolution of ambiguities in natural language*. PhD thesis, Universität des Saarlandes, 2004.

[8] Alexander Koller, Kurt Mehlhorn, and Joachim Niehren. A polynomial-time fragment of dominance constraints. In *Proceedings of the 38th ACL*, Hong Kong, 2000.

[9] Alexander Koller, Joachim Niehren, and Ralf Treinen. Dominance constraints: Algorithms and complexity. In *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics (LACL '98)*, Grenoble, France. To appear in LNCS, 1998.

[10] Martin Plátek, Tomáš Holan, and Vladislav Kuboň. On relax-ability of word-order by d-grammars. In Cristian Calude, Michael Dinneen, and Smaranda Sburlan, editors, *Combinatorics, Computability and Logic*, Discrete Mathematics and Theoretical Computer Science, pages 159–174. Springer, Berlin, 2001.

[11] Hiroyuki Seki, Takashi Matsumura, Mamoru Fuji, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

[12] David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. PhD thesis, University of Pennsylvania, 1988. Available as Technical Report MS-CIS-88-74 of the Department of Computer and Information Sciences, University of Pennsylvania.

[13] Anssi Yli-Jyrä. Multiplanarity – a model for dependency structures in treebanks. In *Second Workshop on Treebanks and Linguistic Theories*, Mathematical Modelling in Physics, Engineering and Cognitive Sciences, pages 189–200, Växjö, Sweden, 2003.