Dependency trees and the strong generative capacity of CCG

Alexander Koller
Saarland University
Saarbrücken, Germany
koller@mmci.uni-saarland.de

Marco Kuhlmann
Uppsala University
Uppsala, Sweden
marco.kuhlmann@lingfil.uu.se

Abstract

We propose a novel algorithm for extracting dependencies from the derivations of a large fragment of CCG. Unlike earlier proposals, our dependency structures are always tree-shaped. We then use these dependency trees to compare the strong generative capacities of CCG and TAG and obtain surprising results: Both formalisms generate the same languages of derivation trees – but the mechanisms they use to bring the words in these trees into a linear order are incomparable.

1 Introduction

Combinatory Categorial Grammar (CCG; Steedman (2001)) is an increasingly popular grammar formalism. Next to being theoretically well-motivated due to its links to combinatory logic and categorial grammar, it is distinguished by the availability of efficient open-source parsers (Clark and Curran, 2007), annotated corpora (Hockenmaier and Steedman, 2007; Hockenmaier, 2006), and mechanisms for wide-coverage semantic construction (Bos et al., 2004).

However, there are limits to our understanding of the formal properties of CCG and its relation to other grammar formalisms. In particular, while it is well-known that CCG belongs to a family of mildly context-sensitive formalisms that all generate the same string languages (Vijay-Shanker and Weir, 1994), there are few results about the *strong* generative capacity of CCG. This makes it difficult to gauge the similarities and differences between CCG and other formalisms in how they model linguistic phenomena such as scrambling and relative clauses (Hockenmaier and Young, 2008), and hampers the transfer of algorithms from one formalism to another.

In this paper, we propose a new method for deriving a dependency tree from a CCG derivation tree for PF-CCG, a large fragment of CCG. We then explore the strong generative capacity of PF-CCG in terms of dependency trees. In particular, we cast new light on the relationship between CCG and other mildly context-sensitive formalisms such as Tree-Adjoining Grammar (TAG; Joshi and Schabes (1997)) and Linear Context-Free Rewrite Systems (LCFRS; Vijay-Shanker et al. (1987)). We show that if we only look at valencies and ignore word order, then the dependency trees induced by a PF-CCG grammar form a regular tree language, just as for TAG and LCFRS. To our knowledge, this is the first time that the regularity of CCG's derivational structures has been exposed. However, if we take the word order into account, then the classes of PF-CCG-induced and TAG-induced dependency trees are incomparable; in particular, CCG-induced dependency trees can be unboundedly non-projective in a way that TAG-induced dependency trees cannot.

The fact that all our dependency structures are *trees* brings our approach in line with the emerging mainstream in dependency parsing (McDonald et al., 2005; Nivre et al., 2007) and TAG derivation trees. The price we pay for restricting ourselves to trees is that we derive fewer dependencies than the more powerful approach by Clark et al. (2002). Indeed, we do not claim that our dependencies are linguistically meaningful beyond recording the way in which syntactic valencies are filled. However, we show that our dependency trees are still informative enough to reconstruct the semantic representations.

The paper is structured as follows. In Section 2, we introduce CCG and the fragment PF-CCG that we consider in this paper, and compare our contribution to earlier research. In Section 3, we then show how to read off a dependency tree from a CCG derivation. Finally, we explore the strong generative capacity of CCG in Section 4 and conclude with ideas for future work.

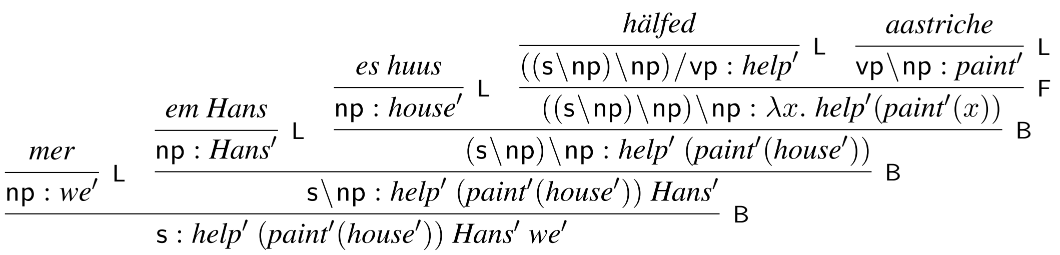

hälfed
aastriche
es huus
em Hans
mer
((s\np)\np)/vp : help′
vp\np : paint′
np : house′
np : Hans′
np : we′
((s\np)\np)\np : λx. help′(paint′(x))
(s\np)\np : help′ (paint′(house′))
s\np : help′ (paint′(house′)) Hans′
s : help′ (paint′(house′)) Hans′ we′
L
F
B

Figure 1: A PF-CCG derivation

2 Combinatory Categorial Grammars

We start by introducing the Combinatory Categorial Grammar (CCG) formalism. Then we introduce the fragment of CCG that we consider in this paper, and discuss some related work.

2.1 CCG

Combinatory Categorial Grammar (Steedman, 2001) is a grammar formalism that assigns *categories* to substrings of an input sentence. There are *atomic* categories such as s and np; and if A and B are categories, then $A\backslash B$ and A/B are *functional* categories representing a constituent that will have category A once it is combined with another constituent of type B to the left or right, respectively. Each word is assigned a category by the lexicon; adjacent substrings can then be combined by *combinatory rules*. As an example, Steedman and Baldridge's (2009) analysis of Shieber's (1985) Swiss German subordinate clause *(das) mer em Hans es huus hälfed aastriiche* ('(that) we help Hans paint the house') is shown in Figure 1.

Intuitively, the arguments of a functional category can be thought of as the syntactic valencies of the lexicon entry, or as arguments of a function that maps categories to categories. The core combinatory mechanism underlying CCG is the composition and application of these functions. In their most general forms, the combinatory rules of (forward and backward) application and composition can be written as in Figure 2. The symbol $|$ stands for an arbitrary (forward or backward) slash; it is understood that the slash before each B_i above the line is the same as below. The rules derive statements about triples $w \vdash A : f$, expressing that the substring w can be assigned the category A and the semantic representation f; an entire string counts as grammatical if it can be assigned the start category s. In parallel to the combination of substrings by the combinatory rules, their semantic representations are combined by functional composition.

We have presented the composition rules of CCG in their most general form. In the literature, the special cases for $n = 0$ are called forward and backward *application*; the cases for $n > 0$ where the slash before B_n is the same as the slash before B are called *composition of degree* n; and the cases where $n > 0$ and the slashes have different directions are called *crossed composition of degree* n. For instance, the F application that combines *hälfed* and *aastriche* in Figure 1 is a forward crossed composition of degree 1.

2.2 PF-CCG

In addition to the composition rules introduced above, CCG also allows rules of substitution and type-raising. Substitution is used to handle syntactic phenomena such as parasitic gaps; type-raising allows a constituent to serve syntactically as a functor, while being used semantically as an argument. Furthermore, it is possible in CCG to restrict the instances of the rule schemata in Figure 2—for instance, to say that the application rule may only be used for the case $A = \mathsf{s}$. We call a CCG grammar *pure* if it does not use substitution, type-raising, or restricted rule schemata. Finally, the argument categories of a CCG category may themselves be functional categories; for instance, the category of a VP modifier like *passionately* is $(\mathsf{s}\backslash\mathsf{np})\backslash(\mathsf{s}\backslash\mathsf{np})$. We call a category that is either atomic or only has atomic arguments a *first-order category*, and call a CCG grammar *first-order* if all categories that its lexicon assigns to words are first-order.

In this paper, we only consider CCG grammars that are pure and first-order. This fragment, which we call PF-CCG, is less expressive than full CCG, but it significantly simplifies the definitions in Section 3. At the same time, many real-world CCG grammars do not use the substitution rule, and type-raising can be compiled into the grammar in the sense that for any CCG grammar, there is an equivalent CCG grammar that does not use type-raising and assigns the same semantic representations to

$$\frac{(a, A, f) \text{ is a lexical entry}}{a \vdash A : f}\ \mathsf{L}$$

$$\frac{v \vdash A/B : \lambda x.\, f(x) \qquad w \vdash B \mid B_n \mid \ldots \mid B_1 : \lambda y_1, \ldots, y_n.\, g(y_1, \ldots, y_n)}{vw \vdash A \mid B_n \mid \ldots \mid B_1 : \lambda y_1, \ldots, y_n.\, f(g(y_1, \ldots, y_n))}\ \mathsf{F}$$

$$\frac{v \vdash B \mid B_n \mid \ldots \mid B_1 : \lambda y_1, \ldots, y_n.\, g(y_1, \ldots, y_n) \qquad w \vdash A\backslash B : \lambda x.\, f(x)}{vw \vdash A \mid B_n \mid \ldots \mid B_1 : \lambda y_1, \ldots, y_n.\, f(g(y_1, \ldots, y_n))}\ \mathsf{B}$$

Figure 2: The generalized combinatory rules of CCG

each string. On the other hand, the restriction to first-order grammars is indeed a limitation in practice. We take the work reported here as a first step towards a full dependency-tree analysis of CCG, and discuss ideas for generalization in the conclusion.

2.3 Related work

The main objective of this paper is the definition of a novel way in which dependency trees can be extracted from CCG derivations. This is similar to Clark et al. (2002), who aim at capturing 'deep' dependencies, and encode these into annotated lexical categories. For instance, they write $(\mathsf{np}_i\backslash\mathsf{np}_i)/(\mathsf{s}\backslash\mathsf{np}_i)$ for subject relative pronouns to express that the relative pronoun, the trace of the relative clause, and the modified noun phrase are all semantically the same. This means that the relative pronoun has multiple parents; in general, their dependency structures are not necessarily trees. By contrast, we aim to extract only dependency *trees*, and achieve this by recording only the fillers of syntactic valencies, rather than the semantic dependencies: the relative pronoun gets two dependents and one parent (the verb whose argument the modified np is), just as the category specifies. So Clark et al.'s and our dependency approach represent two alternatives of dealing with the tradeoff between simple and expressive dependency structures.

Our paper differs from the well-known results of Vijay-Shanker and Weir (1994) in that they establish the *weak* equivalence of different grammar formalisms, while we focus on comparing the derivational structures. Hockenmaier and Young (2008) present linguistic motivations for comparing the strong generative capacities of CCG and TAG, and the beginnings of a formal comparison between CCG and spinal TAG in terms of Linear Indexed Grammars.

3 Induction of dependency trees

We now explain how to extract a dependency tree from a PF-CCG derivation. The basic idea is to associate, with every step of the derivation, a corresponding operation on dependency trees, in much the same way as derivation steps can be associated with operations on semantic representations.

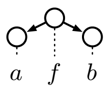
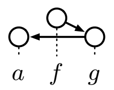
3.1 Dependency trees

When talking about a dependency tree, it is usually convenient to specify its tree structure and the linear order of its nodes separately. The tree structure encodes the valency structure of the sentence (immediate dominance), whereas the linear precedence of the words is captured by the linear order.

For the purposes of this paper, we represent a *dependency tree* as a pair $d = (t, s)$, where t is a ground term over some suitable alphabet, and s is a linearization of the nodes (term addresses) of t, where by a linearization of a set S we mean a list of elements of S in which each element occurs exactly once (see also Kuhlmann and Möhl (2007)). As examples, consider

$$(f(a, b), [1, \varepsilon, 2]) \quad \text{and} \quad (f(g(a)), [1 \cdot 1, \varepsilon, 1])\,.$$

These expressions represent the dependency trees

$d_1 =$  and $d_2 =$  .

Notice that it is because of the separate specification of the tree and the order that dependency trees can become non-projective; d_2 is an example.

A *partial dependency tree* is a pair (t, s) where t is a term that may contain variables, and s is a linearization of those nodes of t that are not labelled with variables. We restrict ourselves to terms in which each variable appears exactly once, and will also prefix partial dependency trees with λ-binders to order the variables.

$$\frac{e = (a, A \mid A_m \cdots \mid A_1) \text{ is a lexical entry}}{a \vdash A \mid A_m \cdots \mid A_1 : \lambda x_1, \dots, x_m.\ (e(x_1, \dots, x_m), [\varepsilon])}\ \mathsf{L}$$

$$\frac{v \vdash A \mid A_m \cdots \mid A_1/B : \lambda x, x_1, \dots, x_m.\ d \qquad w \vdash B \mid B_n \cdots \mid B_1 : \lambda y_1, \dots, y_n.\ d'}{vw \vdash A \mid A_m \cdots \mid A_1 \mid B_n \cdots \mid B_1 : \lambda y_1, \dots, y_n, x_1, \dots, x_m.\ d[\, x := d' \,]_F}\ \mathsf{F}$$

$$\frac{w \vdash B \mid B_n \cdots \mid B_1 : \lambda y_1, \dots, y_n.\ d' \qquad v \vdash A \mid A_m \cdots \mid A_1 \backslash B : \lambda x, x_1, \dots, x_m.\ d}{wv \vdash A \mid A_m \cdots \mid A_1 \mid B_n \cdots \mid B_1 : \lambda y_1, \dots, y_n, x_1, \dots, x_m.\ d[\, x := d' \,]_B}\ \mathsf{B}$$

Figure 3: Computing dependency trees in CCG derivations

3.2 Operations on dependency trees

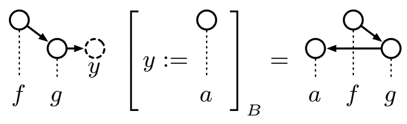
Let t be a term, and let x be a variable in t. The result of the substitution of the term t' into t for x is denoted by $t[\, x := t' \,]$. We extend this operation to dependency trees as follows. Given a list of addresses s, let xs be the list of addresses obtained from s by prefixing every address with the address of the (unique) node that is labelled with x in t. Then the operations of *forward and backward concatenation* are defined as

$$\begin{aligned}(t, s)[\, x := (t', s') \,]_F &= (t[\, x := t' \,], s \cdot xs')\,,\\ (t, s)[\, x := (t', s') \,]_B &= (t[\, x := t' \,], xs' \cdot s)\,.\end{aligned}$$

The concatenation operations combine two given dependency trees (t, s) and (t', s') into a new tree by substituting t' into t for some variable x of t, and adding the (appropriately prefixed) list s' of nodes of t' either before or after the list s of nodes of t. Using these two operations, the dependency trees d_1 and d_2 from above can be written as follows. Let $d_a = (a, [\varepsilon])$ and $d_b = (b, [\varepsilon])$.

$$\begin{aligned}d_1 &= (f(x, y), [\varepsilon])[\, x := d_a \,]_F[\, y := d_b \,]_F\\ d_2 &= (f(x), [\varepsilon])[\, x := (g(y), [\varepsilon]) \,]_F[\, y := d_a \,]_B\end{aligned}$$

Here is an alternative graphical notation for the composition of d_2:



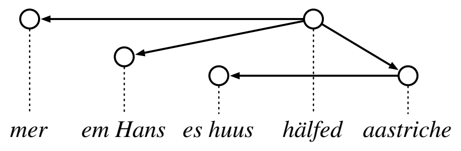
In this notation, nodes that are not marked with variables are positioned (indicated by the dotted projection lines), while the (dashed) variable nodes dangle unpositioned.

3.3 Dependency trees for CCG derivations

To encode CCG derivations as dependency trees, we annotate each composition rule of PF-CCG with instructions for combining the partial dependency trees for the substrings into a partial dependency tree for the larger string. Essentially, we now combine partial dependency trees using forward and backward concatenation rather than combining semantic representations by functional composition and application. From now on, we assume that the node labels in the dependency trees are CCG lexicon entries, and represent these by just the word in them.

The modified rules are shown in Figure 3. They derive statements about triples $w \vdash A : p$, where w is a substring, A is a category, and p is a lambda expression over a partial dependency tree. Each variable of p corresponds to an argument category in A, and vice versa. Rule L covers the base case: the dependency tree for a lexical entry e is a tree with one node for the item itself, labelled with e, and one node for each of its syntactic arguments, labelled with a variable. Rule F captures forward composition: given two dependency trees d and d', the new dependency tree is obtained by forward concatenation, binding the outermost variable in d. Rule B is the rule for backward composition. The result of translating a complete PF-CCG derivation δ in this way is always a dependency tree without variables; we call it $d(\delta)$.

As an example, Figure 4 shows the construction for the derivation in Figure 1. The induced dependency tree looks like this:



For instance, the partial dependency tree for the lexicon entry of *aastriiche* contains two nodes: the root (with address ε) is labelled with the lexicon entry, and its child (address 1) is labelled with the

$$\dfrac{\dfrac{\textit{mer}}{(\textit{mer},[\varepsilon])}\,\mathsf{L} \qquad \dfrac{\dfrac{\textit{em Hans}}{(\textit{Hans},[\varepsilon])}\,\mathsf{L} \qquad \dfrac{\dfrac{\textit{es huus}}{(\textit{huus},[\varepsilon])}\,\mathsf{L} \qquad \dfrac{\dfrac{\textit{hälfed}}{\lambda x,y,z.\ (\textit{hälfed}(x,y,z),[\varepsilon])}\,\mathsf{L} \qquad \dfrac{\textit{aastriiche}}{\lambda w.\ (\textit{aastriiche}(w),[\varepsilon])}\,\mathsf{L}}{\lambda w,y,z.\ (\textit{hälfed}(\textit{aastriiche}(w),y,z),[\varepsilon,1])}\,\mathsf{F}}{\lambda y,z.\ (\textit{hälfed}(\textit{aastriiche}(\textit{huus}),y,z),[11,\varepsilon,1])}\,\mathsf{B}}{\lambda z.\ (\textit{hälfed}(\textit{aastriiche}(\textit{huus}),\textit{Hans},z),[2,11,\varepsilon,1])}\,\mathsf{B}}{(\textit{hälfed}(\textit{aastriiche}(\textit{huus}),\textit{Hans},\textit{mer}),[3,2,11,\varepsilon,1])}\,\mathsf{B}$$

Figure 4: Computing a dependency tree for the derivation in Figure 1

variable x. This tree is inserted into the tree from *hälfed* by forward concatenation. The variable w is passed on into the new dependency tree, and later filled by backward concatenation to *huus*. Passing the argument slot of *aastriiche* to *hälfed* to be filled on its left creates a non-projectivity; it corresponds to a crossed composition in CCG terms. Notice that the categories derived in Figure 1 mirror the functional structure of the partial dependency trees at each step of the derivation.

3.4 Semantic equivalence

The mapping from derivations to dependency trees loses some information: different derivations may induce the same dependency tree. This is illustrated by Figure 5, which provides two possible derivations for the phrase *big white rabbit*, both of which induce the same dependency tree. Especially in light of the fact that our dependency trees will typically contain fewer dependencies than the DAGs derived by Clark et al. (2002), one could ask whether dependency trees are an appropriate way of representing the structure of a CCG derivation.

However, at the end of the day, the most important information that can be extracted from a CCG derivation is the semantic representation it computes; and it is possible to reconstruct the semantic representation of a derivation δ from $d(\delta)$ alone. If we forget the word order information in the dependency trees, the rules F and B in Figure 3 are merely η-expanded versions of the semantic construction rules in Figure 2. This means that $d(\delta)$ records everything we need to know about constructing the semantic representation: We can traverse it bottom-up and apply the lexical semantic representation of each node to those of its subterms. So while the dependency trees obliterate some information in the CCG derivations (particularly its associative structure), they are indeed appropriate representations because they record all syntactic valencies and encode enough information to recompute the semantics.

4 Strong generative capacity

Now that we know how to see PF-CCG derivations as dependency trees, we can ask what sets of such trees can be generated by PF-CCG grammars. This is the question about the strong generative capacity of PF-CCG, measured in terms of dependency trees (Miller, 2000). In this section, we give a partial answer to this question: We show that the sets of PF-CCG-induced *valency trees* (dependency trees without their linear order) form regular tree languages, but that the sets of dependency trees themselves are irregular. This is in contrast to other prominent mildly context-sensitive grammar formalisms such as Tree Adjoining Grammar (TAG; Joshi and Schabes (1997)) and Linear Context-Free Rewrite Systems (LCFRS; Vijay-Shanker et al. (1987)), in which both languages are regular.

4.1 CCG term languages

Formally, we define the language of all dependency trees generated by a PF-CCG grammar G as the set

$$L_D(G) = \{\, d(\delta) \mid \delta \text{ is a derivation of } G \,\}\,.$$

Furthermore, we define the set of *valency trees* to be the set of just the term parts of each $d(\delta)$:

$$L_V(G) = \{\, t \mid (t,s) \in L_D(G) \,\}\,.$$

By our previous assumption, the node labels of a valency tree are CCG lexicon entries.

We will now show that the valency tree languages of PF-CCG grammars are *regular tree languages* (Gécseg and Steinby, 1997). Regular tree languages are sets of trees that can be generated by *regular tree grammars*. Formally, a regular tree grammar (RTG) is a construct $\Gamma = (N, \Sigma, S, P)$, where N is an alphabet of non-terminal symbols, Σ is an alphabet of ranked term constructors called terminal symbols, $S \in N$ is a distinguished start symbol, and P is a finite set of production rules of the form $A \to \gamma$, where $A \in N$ and γ is a term over Σ and N, where the nonterminals can be used

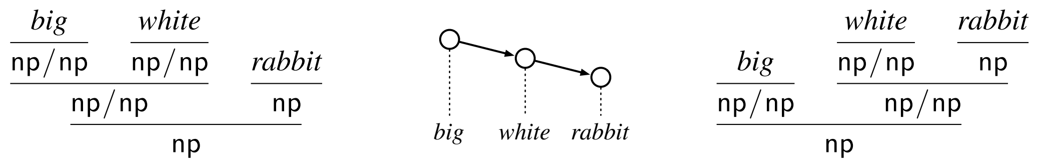


Figure 5: Different derivations may induce the same dependency tree

as constants. The grammar Γ generates trees from the start symbol by successively expanding occurrences of nonterminals using production rules. For instance, the grammar that contains the productions $S \to f(A, A)$, $A \to g(A)$, and $A \to a$ generates the tree language $\{\, f(g^m(a), g^n(a)) \mid m, n \geq 0 \,\}$.

We now construct an RTG $\Gamma(G)$ that generates the set of valency trees of a PF-CCG G. For the terminal alphabet, we choose the lexicon entries: If $e = (a, A \mid B_1 \ldots \mid B_n, f)$ is a lexicon entry of G, we take e as an n-ary term constructor. We also take the atomic categories of G as our nonterminal symbols; the start category s of G counts as the start symbol. Finally, we encode each lexicon entry as a production rule: The lexicon entry e above encodes to the rule $A \to e(B_n, \ldots, B_1)$.

Let us look at our running example to see how this works. Representing the lexicon entries as just the words for brevity, we can write the valency tree corresponding to the CCG derivation in Figure 4 as $t_0 = \textit{hälfed}(\textit{aastriiche}(\textit{huus}), \textit{Hans}, \textit{mer})$; here *hälfed* is a ternary constructor, *aastriiche* is unary, and all others are constants. Taking the lexical categories into account, we obtain the RTG with

$$\begin{aligned} \mathsf{s} &\to \textit{hälfed}(\mathsf{vp}, \mathsf{np}, \mathsf{np}) \\ \mathsf{vp} &\to \textit{aastriiche}(\mathsf{np}) \\ \mathsf{np} &\to \textit{huus} \mid \textit{Hans} \mid \textit{mer} \end{aligned}$$

This grammar indeed generates t_0, and all other valency trees induced by the sample grammar.

More generally, $L_V(G) \subseteq L(\Gamma(G))$ because the construction rules in Figure 3 ensure that if a node v becomes the i-th child of a node u in the term, then the result category of v's lexicon entry equals the i-th argument category of u's lexicon entry. This guarantees that the i-th nonterminal child introduced by the production for u can be expanded by the production for v. The converse inclusion can be shown by reconstructing, for each valency tree t, a CCG derivation δ that induces t. This construction can be done by arranging the nodes in t into an order that allows us to combine every parent in t with its children using only forward and backward application. The CCG derivation we obtain for the example is shown in Figure 6; it is a derivation for the sentence *das mer em Hans hälfed es huus aastriiche*, using the same lexicon entries. Together, this shows that $L(\Gamma(G)) = L_V(G)$. Thus:

Theorem 1 *The sets of valency trees generated by PF-CCG are regular tree languages.* □

By this result, CCG falls in line with context-free grammars, TAG, and LCFRS, whose sets of derivational structures are all regular (Vijay-Shanker et al., 1987). To our knowledge, this is the first time the regular structure of CCG derivations has been exposed. It is important to note that while CCG derivations themselves can be seen as trees as well, they do *not* always form regular tree languages (Vijay-Shanker et al., 1987). Consider for instance the CCG grammar from Vijay-Shanker and Weir's (1994) Example 2.4, which generates the string language $a^n b^n c^n d^n$; Figure 7 shows the derivation of $aabbccdd$. If we follow this derivation bottom-up, starting at the first c, the intermediate categories collect an increasingly long tail of $\backslash\mathsf{a}$ arguments; for longer words from the language, this tail becomes as long as the number of cs in the string. The infinite set of categories this produces translates into the need for an infinite nonterminal alphabet in an RTG, which is of course not allowed.

4.2 Comparison with TAG

If we now compare PF-CCG to its most prominent mildly context-sensitive cousin, TAG, the regularity result above paints a suggestive picture: A PF-CCG valency tree assigns a lexicon entry to each word and says which other lexicon entry fills each syntactic valency. In this respect, it is the analogue of a TAG derivation tree (in which the lexicon entries are elementary trees), and we just saw that PF-CCG and TAG generate the same tree languages. On the other hand, CCG and TAG are weakly equivalent (Vijay-Shanker and Weir, 1994), i.e. they generate the same linear word orders. So one could expect that CCG and TAG also induce the same dependency trees. Interestingly, this is not the case.

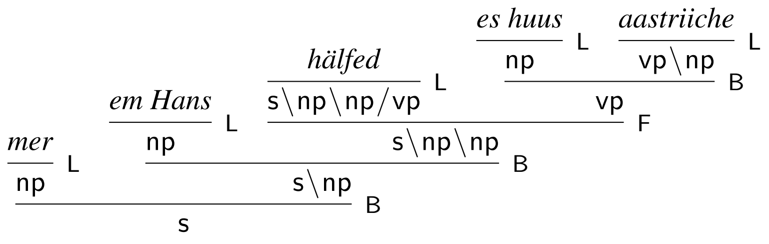


Figure 6: CCG derivation reconstructed from the dependency tree from Figure 4 using only applications

We know from the literature that those dependency trees that can be constructed from TAG derivation trees are exactly those that are well-nested and have a block-degree of at most 2 (Kuhlmann and Möhl, 2007). The block-degree of a node u in a dependency tree is the number of ‘blocks’ into which the subtree below u is separated by intervening nodes that are not below u, and the block-degree of a dependency tree is the maximum block-degree of its nodes. So for instance, the dependency tree on the right-hand side of Figure 8 has block-degree two. It is also well-nested, and can therefore be induced by TAG derivations.

Things are different for the dependency trees that can be induced by PF-CCG. Consider the left-hand dependency tree in Figure 8, which is induced by a PF-CCG derivation built from words with the lexical categories a / a, b \ a, b \ b, and a. While this dependency tree is well-nested, it has block-degree *three*: The subtree below the leftmost node consists of three parts. More generally, we can insert more words with the categories a / a and b \ b in the middle of the sentence to obtain dependency trees with arbitrarily high block-degrees from this grammar. This means that unlike for TAG-induced dependency trees, there is no upper bound on the block-degree of dependency trees induced by PF-CCG—as a consequence, there are CCG dependency trees that cannot be induced by TAG.

On the other hand, there are also dependency trees that can be induced by TAG, but not by PF-CCG. The tree on the right-hand side of Figure 8 is an example. We have already argued that this tree can be induced by a TAG. However, it contains no two adjacent nodes that are connected by an edge; and every nontrivial PF-CCG derivation must combine two adjacent words at least at one point during the derivation. Therefore, the tree cannot be induced by a PF-CCG grammar. Furthermore, it is known that all dependency languages that can be generated by TAG or even, more generally, by LCRFS, are regular in the sense of Kuhlmann and Möhl (2007). One crucial property of regular dependency languages is that they have a bounded block-degree; but as we have seen, there are PF-CCG dependency languages with unbounded block-degree. Therefore there are PF-CCG dependency languages that are not regular. Hence:

Theorem 2 *The sets of dependency trees generated by PF-CCG and TAG are incomparable.* □

We believe that these results will generalize to full CCG. While we have not yet worked out the induction of dependency trees from full CCG, the basic rule that CCG combines adjacent substrings should still hold; therefore, every CCG-induced dependency tree will contain at least one edge between adjacent nodes. We are thus left with a very surprising result: TAG and CCG both generate the same string languages and the same sets of valency trees, but they use incomparable mechanisms for linearizing valency trees into sentences.

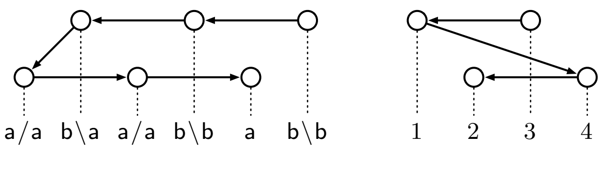


Figure 8: The divergence between CCG and TAG

4.3 A note on weak generative capacity

As a final aside, we note that the construction for extracting purely applicative derivations from the terms described by the RTG has interesting consequences for the weak generative capacity of PF-CCG. In particular, it has the corollary that for any PF-CCG derivation δ over a string w, there is a permutation of w that can be accepted by a PF-CCG derivation that uses only application—that is, every string language L that can be generated by a PF-CCG grammar has a context-free sublanguage L' such that all words in L are permutations of words in L'.

This means that many string languages that we commonly associate with CCG cannot be generated

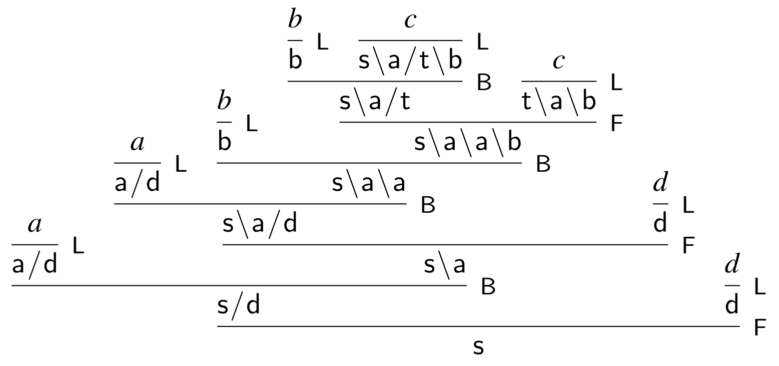


Figure 7: The CCG derivation of *aabbccdd* using Example 2.4 in Vijay-Shanker and Weir (1994)

by PF-CCG. One such language is $a^n b^n c^n d^n$. This language is not itself context-free, and therefore any PF-CCG grammar whose language contains it also contains permutations in which the order of the symbols is mixed up. The culprit for this among the restrictions that distinguish PF-CCG from full CCG seems to be that PF-CCG grammars must allow all instances of the application rules. This would mean that the ability of CCG to generate non-context-free languages (also linguistically relevant ones) hinges crucially on its ability to restrict the allowable instances of rule schemata, for instance, using slash types (Baldridge and Kruijff, 2003).

5 Conclusion

In this paper, we have shown how to read derivations of PF-CCG as dependency trees. Unlike previous proposals, our view on CCG dependencies is in line with the mainstream dependency parsing literature, which assumes tree-shaped dependency structures; while our dependency trees are less informative than the CCG derivations themselves, they contain sufficient information to reconstruct the semantic representation. We used our new dependency view to compare the strong generative capacity of PF-CCG with other mildly context-sensitive grammar formalisms. It turns out that the valency trees generated by a PF-CCG grammar form regular tree languages, as in TAG and LCFRS; however, unlike these formalisms, the sets of dependency trees including word order are *not* regular, and in particular can be more non-projective than the other formalisms permit. Finally, we found new formal evidence for the importance of restricting rule schemata for describing non-context-free languages in CCG.

All these results were technically restricted to the fragment of PF-CCG, and one focus of future work will be to extend them to as large a fragment of CCG as possible. In particular, we plan to extend the lambda notation used in Figure 3 to cover type-raising and higher-order categories. We would then be set to compare the behavior of wide-coverage statistical parsers for CCG with statistical dependency parsers.

We anticipate that our results about the strong generative capacity of PF-CCG will be useful to transfer algorithms and linguistic insights between formalisms. For instance, the CRISP generation algorithm (Koller and Stone, 2007), while specified for TAG, could be generalized to arbitrary grammar formalisms that use regular tree languages—given our results, to CCG in particular. On the other hand, we find it striking that CCG and TAG generate the same string languages from the same tree languages by incomparable mechanisms for ordering the words in the tree. Indeed, the exact characterization of the class of CCG-inducable dependency languages is an open issue. This also has consequences for parsing complexity: We can understand why TAG and LCFRS can be parsed in polynomial time from the bounded block-degree of their dependency trees (Kuhlmann and Möhl, 2007), but CCG can be parsed in polynomial time (Vijay-Shanker and Weir, 1990) without being restricted in this way. This constitutes a most interesting avenue of future research that is opened up by our results.

Acknowledgments. We thank Mark Steedman, Jason Baldridge, and Julia Hockenmaier for valuable discussions about CCG, and the reviewers for their comments. The work of Alexander Koller was funded by a DFG Research Fellowship and the Cluster of Excellence “Multimodal Computing and Interaction”. The work of Marco Kuhlmann was funded by the Swedish Research Council.

References

Jason Baldridge and Geert-Jan M. Kruijff. 2003. Multi-modal Combinatory Categorial Grammar. In *Proceedings of the Tenth EACL*, Budapest, Hungary.

Johan Bos, Stephen Clark, Mark Steedman, James R. Curran, and Julia Hockenmaier. 2004. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th COLING*, Geneva, Switzerland.

Stephen Clark and James Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4).

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building deep dependency structures with a wide-coverage CCG parser. In *Proceedings of the 40th ACL*, Philadelphia, USA.

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In Rozenberg and Salomaa (Rozenberg and Salomaa, 1997), pages 1–68.

Julia Hockenmaier and Mark Steedman. 2007. CCG-bank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Julia Hockenmaier and Peter Young. 2008. Non-local scrambling: the equivalence of TAG and CCG revisited. In *Proceedings of TAG+9*, Tübingen, Germany.

Julia Hockenmaier. 2006. Creating a CCGbank and a wide-coverage CCG lexicon for German. In *Proceedings of COLING/ACL*, Sydney, Australia.

Aravind K. Joshi and Yves Schabes. 1997. Tree-Adjoining Grammars. In Rozenberg and Salomaa (Rozenberg and Salomaa, 1997), pages 69–123.

Alexander Koller and Matthew Stone. 2007. Sentence generation as planning. In *Proceedings of the 45th ACL*, Prague, Czech Republic.

Marco Kuhlmann and Mathias Möhl. 2007. Mildly context-sensitive dependency languages. In *Proceedings of the 45th ACL*, Prague, Czech Republic.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of HLT/EMNLP*.

Philip H. Miller. 2000. *Strong Generative Capacity: The Semantics of Linguistic Formalism*. University of Chicago Press.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Grzegorz Rozenberg and Arto Salomaa, editors. 1997. *Handbook of Formal Languages*. Springer.

Stuart Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343.

Mark Steedman and Jason Baldridge. 2009. Combinatory categorial grammar. In R. Borsley and K. Borjars, editors, *Non-Transformational Syntax*. Blackwell. To appear.

Mark Steedman. 2001. *The Syntactic Process*. MIT Press.

K. Vijay-Shanker and David Weir. 1990. Polynomial time parsing of combinatory categorial grammars. In *Proceedings of the 28th ACL*, Pittsburgh, USA.

K. Vijay-Shanker and David J. Weir. 1994. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6):511–546.

K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th ACL*, Stanford, CA, USA.