Spring Cleaning and Grammar Compression: Two Techniques for Detection of Redundancy in HPSG Grammars

Antske Fokkens[a], Yi Zhang[b] and Emily M. Bender[c]

[a] Department of Computational Linguistics, Saarland University
Saarland University Campus, 66123 Saarbrücken, Germany
afokkens@coli.uni-saarland.de
[b] LT-Lab, German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3 / Building D3, 66123 Saarbrücken, Germany
yizhang@dfki.de
[c] Department of Linguistics, University of Washington,
Box 354340, Seattle, WA 98195-4340, USA
ebender@uw.edu

Abstract. This paper presents two approaches that identify which parts of an implemented grammar are used and which parts are computationally inactive. Our results lead to the following insights: even small grammars contain noise due to revised analyses, removing superfluous types from a grammar may help to detect errors in the original grammar and at least half of the types defined in the grammars we investigated do not play a role in the computational process of the grammar.

Keywords: Implemented grammars, HPSG, typed feature structures.

1 Introduction

Grammars of natural language are highly complex objects. This complexity is reflected in implementations of linguistically motivated precision grammars. Understanding the role of a specific element in a broad coverage precision grammar is therefore not always straightforward, even for the engineer who implemented the grammar. Implementations for different phenomena interact, and revisions to the grammar may change the role of existing parts of the grammar. In this paper, we present two techniques for investigating the role that specific types play in implemented HPSG (Pollard and Sag, 1994) grammars. The first, dubbed 'spring cleaning', focuses on identifying portions of the grammar that do not play any role in the set of sentences it recognizes or the structures it assigns to them. Such artifacts can accrue in a grammar because abandoned analyses are not completely removed or because the grammar is built on a cross-linguistic resource but does not use all of the infrastructure that resource provides. Spring cleaning is intended to be used in the course of grammar development and as such must leave the grammar in a state that is still easy to maintain. In contrast, the second technique, 'grammar compression', computes the smallest subset of a type hierarchy that can assign the same structures to the same sentences as the original grammar. In grammar compression, we remove not only the types taken out in spring cleaning, but also those that exist only to express generalizations over their subtypes.

We use these techniques to explore the degree of redundancy in a range of DELPH-IN[1] grammars, including the two grammars of Wambaya (Bender, 2010), the BURGER grammar of Bulgarian (Osenova, 2010), the ManGO grammar[2] of Mandarin Chinese, all built with the LinGO

[1] http://www.delph-in.net/

[2] http://wiki.delph-in.net/moin/MandarinGrammarOnline

Grammar Matrix (Bender *et al.*, 2002; Bender *et al.*, 2010), and two much larger grammars, the English Resource Grammar (Flickinger, 2000) and German Grammar (Müller and Kasper, 2000; Crysmann, 2005).

This paper is structured as follows: First, we describe the overall structure of the grammars under consideration. This section is followed by an overview of the first approach under examination: removing superfluous types from the grammar. Section 4 provides the details of our second investigation of relevant types: maximally reduced computationally equivalent grammars. The next section presents our quantitative results and their implications. Finally, we conclude by suggesting avenues for future research.

2 General structure of DELPH-IN grammars

The grammars under consideration are HPSG-based grammars that can be used for parsing and generation with the LKB (Copestake, 2002) and for parsing with the PET parser (Callmeier, 2002).

The grammars we work with are all written in the DELPH-IN joint reference formalism (Copestake, 2000), known as TDL. TDL grammars consist of one or more files defining a type hierarchy as well as a collection of files defining instances: phrase structure rules, lexical rules, lexical entries, node labels and initial symbols. The instances are manipulated by the parsing and generation algorithms during processing, while the types are used to define the instances. The type hierarchy consists of typed feature structures, i.e., types which are arranged into a subsumption relation and associated with (complex) feature structures.

The definition of a type consists of an identifier for the type, one or more supertypes, and optionally one or more type constraints. Type constraints are feature-path value pairs, where the values are drawn from the type hierarchy. The value of a feature can be atomic (i.e., a type with no further features of its own) or complex (i.e., a type with future features that are appropriate to it). The value of two features may be identified, creating reentrancies. Thus the feature structures are DAGs (cycles are disallowed). Any given feature is associated with exactly one type for which it is appropriate. All other types that bear constraints on the value of that feature must inherit from the type for which the feature is declared appropriate.

An example TDL type definition, drawn from the Wambaya grammar, is shown in Figure 1. The string before the symbol `:=` is the type identifier (`noun-lex`). After `:=`, four supertypes are specified. The remainder of the type definition (enclosed in square brackets) provides the type constraints. `#spr` labels a reentrancy between the values of two features.[3]

```
noun-lex := basic-noun-lex & basic-one-arg &
            no-hcons-lex-item & non-wh-lex &
  [ SYNSEM.LOCAL.CAT [ MC na, HEAD.MOD <>,
                       VAL [ SPR < #spr & [ LOCAL.CAT.HEAD det, OPT + ] >,
                             COMPS < >, SUBJ < >, SPEC < > ]],
    ARG-ST < #spr > ] .
```

Figure 1: Sample type definition from Wambaya grammar

The parsing and generation algorithms require that the type hierarchy be a bounded complete partial order (BCPO), i.e., for any two types that share subtypes, there must be one unique most general such type, called the 'greatest lower bound (glb)'. However, grammar engineers are not required to make sure that the type hierarchy has this property. Rather, the glb types are created programmatically at compile time: if any two types have more than one mutual subtype, the grammar compiler creates an additional type, as shown in Figure 2.

[3] Note that the $< >$ notation is syntactic sugar that simplifies the expression of lists. Lists are underlying treated as feature structures with `FIRST` and `REST` as appropriate features.

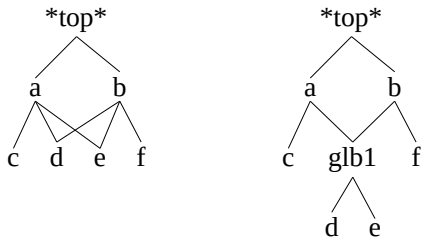

Figure 2: Sample hierarchy as written by grammar engineer (left) and augmented with glb type by compiler (right)

When working with the LKB, the grammar is compiled when it is loaded into the system. The grammar engineer can then interact with the grammar, using it to parse and generate but also exploring the type hierarchy through a GUI. With PET, grammar compilation is handled via a program called 'flop', resulting in a file storing the compiled version of the grammar. The parser, called 'cheap', reads in the flopped grammar for runtime parsing.

3 Spring cleaning: Removing superfluous types

This section describes a process for removing types that do not have an impact on the competence of the grammar. By **competence**, we mean the set of analyses that the grammar provides for any possible input strings; for strings the grammar does not recognize, this will be the empty set. In practice, we can only test this competence with some finite set of strings. In the experiments reported below, we use the testsuites associated with each grammar.

As explained in the previous section, the LKB and PET parsers and LKB generator take instances as starting points for parsing and generation. Instances thus have a direct influence on the competence of the grammar. There are several ways in which types may be **relevant**, i.e., play a role for instances. The first and most direct way is if a type is an **instantiated type**, meaning that it either has an instance or has a subtype that has an instance. Instantiated types thus directly define the instances of the grammar. The second way a type has an impact on instances is by defining **features** or **values** that are part of the definition of one or more instantiated types. Finally, a type may be relevant to instances, because it defines a **lower bound** between two relevant types. In this case, the type influences the possibility of unification, i.e. if a type t forms the only lower bound between types t_1 and t_2, t permits unification between t_1 and t_2, which can no longer take place when t is removed from the grammar.

Types that do not influence instances in any of the three possible ways defined above do not have an impact on the competence of the grammar. Our 'spring cleaning' algorithm goes through the type definition files and identifies such irrelevant types. This process takes place in two stages. In the first stage, we identify types that are necessary for the definition of the instantiated types. This means types that either (i) are instantiated types, (ii) define values of features of instantiated types, or (iii) introduce features used to define an instantiated type. Errors occurring in this stage can be tested at **compile time** of the grammar: if a type is removed that is required for the definition of an instantiated rule or lexical entry, the grammar will not load in LKB. Similarly, the PET 'flop' command will fail (and report an error) on a grammar missing such a type.

Types that permit unification between other types, but play no direct role to other type's definitions, are only relevant at **runtime**. These types are identified in the second stage of the process. After separating types needed to define instantiated types from those that are not, our algorithm checks for relevance at runtime of the latter types. We create a hierarchy h of types that lead to instances and their values. For each type from the original grammar that is not defined in h, our algorithm checks whether it permits the unification of types from h that would otherwise fail. In

order to test this stage for correctness, we use the grammar to parse some set of strings. In addition, we use grammar compression, explained in the next section, to verify if all relevant types are kept in the grammar.[4]

The final output of our algorithm is three sets of files: (i) modified versions of the files of the original grammar, in which the definition files include only those types that are either in *h* or identified as relevant at runtime, (ii) copies of the original definition files and (iii) files that list removed types for each definition file. The structure of the original grammar is preserved by this process. This includes types that are used to capture generalizations so that a given constraint need be stated in only on place. Thus the result is a grammar that is 'cleaned up' with respect to the input without being made more compact in ways that might make it more opaque to the grammar engineer or introduce redundancy at the level of the statement of constraints. As we will see below, exploring which types are eliminated can be illuminating for the grammar engineer as well.

4 Grammar compression: Maximally reduced computationally equivalent grammars

While the 'spring cleaning' algorithm safely removes types that do not have an impact on the competence of the grammar, it does not guarantee the maximal reduction of the grammar. More specifically, the supertypes in an inheritance hierarchy represent generalizations on various levels, but many of them serve no computational purpose, and will never show up in the analysis. This section describes our algorithm that identifies computationally relevant types of the grammar. The compression algorithm removes a superset of types to those reduced by the 'spring cleaning' method, hence can be used to verify the equivalence of the grammars achieved by the less-invasive reduction method from the previous section. The main practical difference between the algorithm applied in this section and the one in the previous section is that this algorithm operates on a compiled grammars, i.e. grammars that have been compiled to be used by the PET parser. The consequence of this difference is that this algorithm can also identify supertypes of instantiated types and relevant values that are not strictly necessary for the grammar from a computational point of view. Note, however, that types that are not computationally relevant can play an important role in the maintainability of the grammar or, from a linguistic point of view, in capturing cross-grammar similarity. We will explain these two differences in the coming subsections. The explanation of the grammar compression algorithm is followed by a formal proof of the equivalence between the original and the maximally reduced grammar.

4.1 Reducing compiled grammars

Before a grammar written in TDL can be interpreted by the PET parser, the grammar has to be compiled. There are several significant differences between a compiled grammar and its original that have an impact on identifying types that are relevant for the grammar. First, as mentioned above, a compiled grammar defines a type hierarchy that is a bounded complete partial order (BCPO), which means that all possible greatest lower bounds of existing compatible types are already included in the inheritance hierarchy. Second, all constraints on each defined type are expanded, including those inherited from supertypes. Moreover, the potential subsumption relations between pairs of types are encoded in a bit-matrix, so the immediate supertype-subtype relations become insignificant so long as the subsumption relations do not change. Because each type definition contains all constraints inherited from supertypes as well as direct information about all types that it subsumes, compatibility between types can be checked by applying bit-vector AND. This procedure allows us to identify types that are not strictly necessary for computational purposes (though they may be useful for reasons of grammar maintainability). For instance, a type that serves to define a set of constraints for all its subtypes is no longer required in a grammar where each subtype already contains these constraints.

[4] Note, however, that neither test (compiling or parsing) ensures that all redundant types have been removed.

This is illustrated in Figure 3, which displays the subsumption table for the BCPO in Figure 2. Each row in this table denotes the types subsumed by the current type. Therefore, even without the direct inheritance information from Figure 2, we can tell from Figure 3 that type *b* is equal to or more general than type *b*, *d*, *e*, *f*, and *glb1*.

	top	*a*	*b*	*c*	*d*	*e*	*f*	*glb1*
top	1	1	1	1	1	1	1	1
a	0	1	0	1	1	1	0	1
b	0	0	1	0	1	1	1	1
c	0	0	0	1	0	0	0	0
d	0	0	0	0	1	0	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	1	0
glb1	0	0	0	0	1	1	0	1

Figure 3: Example of subsumption table in a compiled grammar

Our grammar compression algorithm goes through the instantiated types of the grammar and marks the instantiated type itself plus the type value of each expanded constraint. Our grammar compression algorithm goes through the instantiated types of the grammar and marks the type value of each expanded constraint. Once this is done, it further marks the glbs of the marked types to complete the sub-BCPO. The unmarked types which are identified as having no computational impact on the grammar are a superset of those identified by our spring cleaning algorithm, including both types that are not related to instantiated types in any manner, as well as types that may play a role in representing linguistic generalizations and in maintainability of the grammar, but could be removed from the grammar in a compression process where constraints are pushed down on subtypes. The maximally reduced grammar then will only contain the marked types, with the constraints on the removed types specified on the subtypes directly. We provide a proof in the next subsection that such grammar reduction does not change the grammar behavior.

4.2 Proof of equivalence

Definition 1. *For a TFS grammar G, an* analysis *is a TFS created by unifying the TFSes of instances defined in G.*

Definition 2. *TFS grammars G and G′ are* A-equivalent *iff they always produce the same type feature structure analyses for any inputs.*

Note that the notion of *A-equivalence* is similar to the *tree-language equivalence* for CFG or TAG where two grammars not only yield the same sequences, but also the complex structures that produce those sequences.

Assuming a type inheritance hierarchy T is a bounded complete partial order (BCPO), we define a reduced type inheritance hierarchy $R(T)$ as an order-preserving hierarchy containing all the types that (i) are directly mentioned in one or more instances in G; or (ii) serve as the greatest lower-bound (glb) in T for the types included in $R(T)$. A reduced TFS grammar $G(R(T))$ contains the same instances in $G(T)$, and a reduced hierarchy $R(T)$ from T.

Theorem 1. *Given a BCPO type hierarchy* $T = \langle \mathcal{T}, \wedge_T \rangle$ *and its reduction* $R(T) = \langle \mathcal{T}', \wedge_{T'} \rangle$ $(\mathcal{T}' \subseteq \mathcal{T})$*, if two types from* $R(T)$ *are compatible, their greatest lower bound is the same, i.e.,* $\forall t_1, t_2 \in \mathcal{T}', t_1 \wedge_{T'} t_2 = t_1 \wedge_T t_2$*,*

The proof is obvious from the definition, due to the glb-preserving nature of the mapping of types from $R(T)$ to T.

Theorem 2. $G(T)$ *and* $G(T')$ *are* A-Equivalent *if T' = R(T).*

Proof. An analysis with grammar $G(T)$ is produced by unifications with instances. The analysis with $G(T')$ involves the unification of the same set of instances. Since the unification of types mentioned in the instances yields the same glb (Theorem 1), the unifications of the same instances on T and $R(T)$ produce the same TFSes. □

5 Results

5.1 Spring cleaning Matrix based grammars

We tested our algorithm by removing superfluous types from the BURGER grammar of Bulgarian [bul] (Osenova, 2010), the ManGO grammar[5] of Mandarin [cmn] and two grammars of Wambaya [wmb] (Bender, 2010). The Wambaya grammars represent two branches of development on the same grammar, built around two separate analyses of the auxiliaries in the language: an (older) grammar with an argument composition analysis ('arg-comp') and a new grammar which uses a verbal cluster approach ('aux+vc').

We checked whether the reduced grammar was equivalent to the original grammar by comparing the results of the spring cleaning algorithm to those of the grammar compression algorithm, which we have proven to lead to an equivalent grammar. If the reduced grammar contains all types of the compressed grammar, it is guaranteed that all types that have instances as well as types that influence their ability to unify are present in the grammar. This only leaves the question of whether all required supertypes were correctly identified by the algorithm. This can be verified by compiling the grammar, since missing supertypes lead to compilitation errors. We verified the two biggest spring cleaned grammars, the Wambaya grammars, this way, which both passed the test.

In addition, we verified equivalence by parsing associated testsuites with the original and newly created grammar. The set of syntactic trees and semantic interpretations produced for each grammar were compared using [incr tsdb()] (Oepen, 2001). Equivalence was confirmed for all grammars, though not immediately with the Wambaya grammars. As discussed below (§5.2), we discovered that the value hierarchy associated a semantic feature was not a BCPO in the grammars as written. Once this was fixed, we were able to confirm equivalence. As mentioned in Section 1, all four grammars are further developments of grammars originally created by the LinGO Grammar Matrix customization system. This means that types identified as being superfluous can be divided in two main categories: language specific types and types from the cross-linguistic component.

Table 1 presents the results for all grammars. The types removed from Wambaya (arg-comp) form a subset of those removed from Wambaya (aux+vc).

Source	Wambaya (arg-comp)	Wambaya (aux+vc)	BURGER	ManGO
Multilingual Component	178	178	138	131
Language Specific	17	26	226	62
Total	195	204	364	213

Table 1: Number of redundant types identified for each grammar

In general, most of the removed types come from the multilingual core. BURGER forms the only exception, but this result is misleading. The bulk of the language specific types removed from BURGER are lexical types (183 of 226). The lexicon of the publicly available version of BURGER[6] is relatively small (BURGER only has 183 lexical items), and a grammar with a

[5] `http://wiki.delph-in.net/moin/MandarinGrammarOnline`

[6] http://www.bultreebank.org/BURGER/index.html

bigger lexicon currently exists (Osenova, p.c.). Taking this in consideration, it is likely that the real number of superfluous language specific types in BURGER lies around 40.

It is expected that most removed types would be multilingual types for several reasons. First, the multilingual core may contain types that are not relevant for the particular language. Second, the grammars we investigated are relatively small. This has two consequences that may lead to a relatively high number of superfluous multilingual types as compared to language specific types. The multilingual core may, to begin with, contain types that support phenomena that the grammar does not yet cover. In addition, a small to medium sized grammar will not have too many revised analyses that would lead to noise in the grammar.

5.2 Further observations from Spring cleaning

In order to get a clearer idea of the way the Grammar Matrix multilingual component is used, we compared the set of types removed from each grammar. We looked at which types where removed from all grammars, from two grammars or just one grammar. All types were classified according to whether they correspond to a phenomenon not covered by the grammar (Not Covered), whether they represent a phenomenon that does not occur cross-linguistically generally (Not Applicable), or represent an abandoned analysis (Abandoned). Note that the types from the first category may potentially be used in the grammar in the future. The Grammar Matrix contains one type that serves as an example and is not meant to be used (Example). Table 2 presents this classification.[7]

Grammars where removed	Not Covered	Not applicable	Example	Abandoned	Total
Wambaya, BURGER, ManGO	63	29	1	6	99
Wambaya, BURGER	2				2
BURGER, ManGO	18	6			24
Wambaya	10	36		15	61
BURGER	12	1		2	15
ManGO	4	4			8
Total	109	76	1	23	209

Table 2: Classification of redundant types from LinGO Grammar Matrix

As mentioned above, we parsed associated testsuites of each grammar with the original version of the grammar, as well as the reduced grammar in order to verify the equivalence between the two grammars. This additional test lead to another, unforeseen, benefit of our investigation for the Wambaya grammars. Namely, we identified missing types in the hierarchy of the semantic values for person and number. The reduced grammars produced the same number of readings and the same trees as the original grammars, but revealed differences in semantic representations for a small set of analyses. Upon closer inspection, it turned out that some semantic feature values of the feature PERNUM (person and number) had glb-types (greatest lower bound types) as their value. These types are automatically introduced by the LKB when compiling the grammar to make sure the type hierarchy is bounded complete partially ordered (BCPO). They may play a role as a syntactic component of a derivation, but they should not appear in semantic representations. The LKB uses numbers to assign each added glb-type a unique name. These numbers, and thus the name given to a specific type, may differ if the size of the grammar changes. Because of different number of the glb-types representing person and number values in Wambaya, we were able to identify the missing types and improve the Wambaya grammars.

[7] The grammars were not based on the same version of the Grammar Matrix, leading to some divergence. Specifically, five of the types removed from BURGER and ManGO because they are not cross-linguistically valid were not present in the Grammar Matrix that Wambaya is based on.

5.3 Grammar Compression

We ran our grammar compression algorithm on the ERG, GG, JaCY, Wambaya (arg-comp) and BURGER grammars. The results are presented in Table 3.

Grammar	required types	defined types	reduction rate
ERG	4030	7772	48.1%
GG	5211	10778	51.7%
JaCY	1232	2497	50.7%
Wambaya (arg-comp)	751	3400	77.9%
BURGER	3325	5280	37.0%

Table 3: Superfluous Types according to Grammar Compression

Our results show that all tested grammars contain a great number of types that do not have an impact on the behavior of the grammar. Furthermore, a significant difference is observed between the grammars, with the large grammars like ERG, GG and JaCY having about half of their types playing a computational role in the grammar's competence, and the Wambaya grammar using only slightly above 20%. The high amount of unused types in Wambaya can be explained by the fact that it contains the multilingual component from the Grammar Matrix. The Matrix-based BURGER grammar, on the other hand, implements a sophisticated morphological system with over 2700 inflectional rules, outweighing the unused types provided by the Matrix core grammar.

The results presented in Section 5.1 showed that most superfluous types in the Wambaya grammar are found in the multilingual component. Moreover, the multilingual core of the Grammar Matrix makes heavily use of generic types used to define more specific types further down the hierarchy. The purpose of this structure is to make the core more modular and hence provide more flexibility to ensure cross-linguistic applicability. In a monolingual grammar, the grammar engineer can restrict this usage to generic types known to be useful for the particular language.

6 Conclusion and Future Work

We have presented two approaches that investigate which types in an HPSG grammar are used. The first approach examines the type hierarchy as it is defined in TDL. It identifies types that do not define properties of any instantiated types, nor influence their ability to unify and removes such types from the grammar. As such, this approach respects the original structure in the grammar. The second approach looks at the minimum number of types that is required to create a grammar that is equivalent to the grammar that is defined in TDL.

One can learn a variety of things by examining the types that are removed, both for monolingual projects and for cross-linguistic projects. For monolingual projects, cleaning out the unused types improves grammar maintainability. In a multilingual context, systematically exploring what gets used and what doesn't is useful feedback for developers of the multilingual core. These insights shed light both on which analyses aren't working crosslinguistically and on where Matrix users are apparently failing to discover relevant infrastructure provided by the Matrix.

In future work, we plan to extend our spring cleaning algorithm to identify constraints that do not affect the grammar's competence. Furthermore, we aim to continue our investigation of the Grammar Matrix by comparing the formal properties of type hierarchies. For this investigation, we plan to compare hierarchies that represent a grammar at different stages of its development, which should lead to more exact insights on how multilingual components are used in individual grammars.

References

Bender, Emily M. 2010. Reweaving a grammar for Wambaya: A case study in grammar engineering for linguistic hypothesis testing. *Linguistic Issues in Language Technology*, 3(3), 1–34.

Bender, Emily M., Scott Drellishak, Antske Fokkens, Laurie Poulson, and Safiyyah Saleem. 2010. Grammar customization. *Research on Language & Computation*, 8(1), 23–72.

Bender, Emily M., Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In John Carroll, Nelleke Oostdijk, and Richard Sutcliffe, eds., *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pp. 8–14, Taipei, Taiwan.

Callmeier, Ulrich. 2002. Preprocessing and encoding techniques in PET. In Stephan Oepen, Daniel Flickinger, J. Tsujii, and Hans Uszkoreit, eds., *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*. CSLI Publications, Stanford, CA.

Copestake, Ann. 2000. Appendix: Definitions of typed feature structures. *Natural Language Engineering*, 6, 109–112.

Copestake, Ann. 2002. *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, CA.

Crysmann, Berthold. 2005. Relative clause extraposition in German: An efficient and portable implementation. *Research on Language and Computation*, 3(1), 61–82.

Flickinger, Dan. 2000. On building a more efficient grammar by exploiting types. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG), 15 – 28.

Müller, Stefan and Walter Kasper. 2000. HPSG analysis for German. In Wolfgang Wahlster, ed., *Verbmobil: Foundations of Speech-to-Speech translation*, pp. 238 – 253, Berlin, Germany. Springer.

Oepen, Stephan. 2001. [incr tsdb()] — competence and performance laboratory. Technical report, DFKI, Saarbrücken, Germany.

Osenova, Petya. 2010. BUlgarian Resource Grammar – Efficient and Realistic (BURGER). ms, Stanford University.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. The University of Chicago Press and CSLI Publications, Chicago, IL and Stanford, CA.