19

Off-line Constraint Propagation for Efficient HPSG Processing

Walt Detmar Meurers and Guido Minnen

19.1 Introduction

A major goal of a linguist writing HPSG theories is to express very general constraints to capture linguistic phenomena, leaving as much as possible underspecified. When such an HPSG theory is implemented faithfully, either processing is inefficient because only little information is available to guide the constraint resolution process, or the linguistic theory is annotated with information to guide processing. Usually such annotations are provided manually—a very time consuming and error-prone process which can change the original linguistic theory. In this paper we show that it is possible to automatically make a theory more specific at those places where linguistically motivated underspecification would lead to inefficient processing.

An off-line compilation technique called *constraint propagation* is used to improve processing efficiency by means of propagating constraints already expressed in the theory. Programs do not necessarily profit from constraint propagation. For processing grammars, constraint propagation can be very useful, since it makes it possible to process the general constraints expressing linguistic generalizations specified by the linguist, without falling prey to massive nondeterminism. The relevant observation here is that even though certain places in a grammar are underspecified, the grammar does contain enough constraining information—it just needs to be moved to guide processing. Constraint propagation also makes it possible to advance automatically generated

The authors are listed alphabetically.

Lexical and Constructional Aspects of Linguistic Explanation.
Gert Webelhuth, Jean-Pierre Koenig, and Andreas Kathol.

encodings, such as, for example, the definite clause encoding of HPSG grammars introduced by Götz and Meurers (1995, 1997b).

Constraint propagation can be performed on-line (le Provost and Wallace 1993) or it can be used to make programs more specific through off-line compilation (Marriott et al. 1988). In this paper we will focus on the off-line application of constraint propagation. While on-line constraint propagation is more space efficient since information in the code does not need to be duplicated, the off-line process can relieve the run-time from significant overhead.[1] We conjecture that the time-space tradeoff can be exploited by doing off-line constraint propagation *selectively*, i.e., only on those underspecified parts of the grammar which cause processing efficiency to suffer from massive nondeterminism. As such we presuppose that the places in a grammar which will profit from constraint propagation can be located automatically by either exploiting specific properties of the encoding of the grammar or abstract interpretation. The determination of where to perform constraint propagation is also of importance because underspecification can also be used to improve HPSG processing efficiency—see, for example, Kathol 1994, Krieger and Nerbonne 1992, Riehemann 1993 and Frank 1994. Unfortunately, a detailed discussion of this issue is beyond the scope of this paper.

Other techniques to prune the search space that are used in practical natural language processing are *dynamic coroutining*, also referred to as (goal) freezing or delaying, and *static coroutining* by means of Unfold/Fold transformation (Tamaki and Sato 1984). It is important to differentiate between coroutining and constraint propagation: Coroutining changes the way in which the search space is investigated by moving goals through a grammar either on- or off-line. Constraint propagation as conceived in this paper reduces the search space by making the arguments of calls to goals more specific. As we will discuss in Section 19.3, a combination of both techniques can be very useful as constraint propagation can be used to extract restricting information from the definition of goals also in cases where freezing of the call to these goals would hide this information.

This paper is organized as follows: We start with a discussion of two concrete HPSG examples showing how constraint propagation helps improve processing efficiency (Sections 19.2 and 19.3). In Section 19.4 several implementations of constraint propagation algorithms are discussed. Finally, in Section 19.5 we provide some implementation results.

[1] In case an operation for "subtraction" is available for the data structure used, it may be possible to reduce the space cost of the off-line process by eliminating the propagated constraints from their original specification site.

19.2 Efficient processing of ID Schemata

In lexically oriented grammar formalisms like HPSG, the ID schemata specified by the linguist are very schematic since much syntactic information is specified in the lexicon. In faithful implementations this leads to inefficiency in top-down processing because it often is no longer possible to detect locally whether an ID schema applies or not. Consider, for example, the head-adjunct schema and the head-specifier schema of HPSG in Figures 1 and 2.[2]

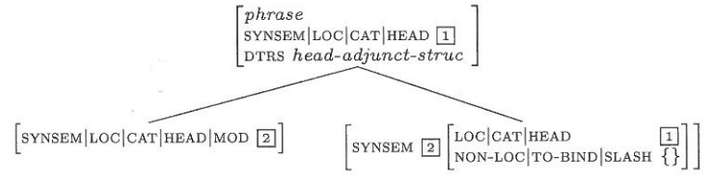

FIGURE 1 The Head-Adjunct ID Schema from Pollard and Sag 1994

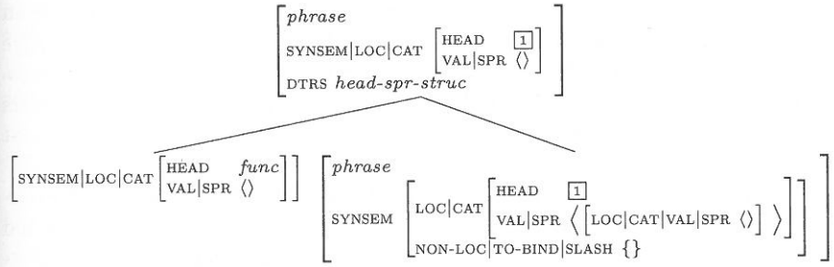

FIGURE 2 The Head-Specifier ID Schema from Pollard and Sag 1994

Due to underspecification, it cannot be determined locally whether the head-adjunct schema can expand specifiers or not. Only upon lexical lookup is it revealed that the head-adjunct schema does not have to be considered for specifiers: The lexicon contains only lexical entries like the one sketched in Figure 3, which specify the category they modify to have a *substantive* head, in this case a *noun*. This specification will

[2] The figures show the head-adjunct schema as expressed in the appendix of Pollard and Sag 1994 and the head-specifier schema from Chapter 9 of the same book - both including the effect of the Head Feature Principle.

therefore always clash with the specification in the head-specifier schema which demands a *functional* head value for the specifier daughter.

$$\begin{bmatrix} word \\ \text{PHON} < kleine > \\ \text{SYNSEM|LOC|CAT|HEAD} \begin{bmatrix} adj \\ \text{MOD|LOC|CAT|HEAD } noun \end{bmatrix} \end{bmatrix}$$

FIGURE 3 The lexical entry for the adjective *kleine*

The sketched efficiency problem seems to suggest that top-down processing is not the right processing strategy to adopt for processing of lexically oriented grammar formalisms. This, however, is not necessarily the case. Strict bottom-up processing means that no filtering information resulting from the start category is made available. To have some guiding information in the case of parsing an extra-logical treatment of the input string can be used, for example, a link relation. However, it is unclear what such a treatment should look like for theories using more elaborate linearization operations. Furthermore, refraining from taking into account information provided by the start category is virtually impossible in the case of generation, and it is not generally clear what an extra-logical treatment of the logical form in a similar fashion as in parsing could look like. There exists an off-line compilation technique called *magic* that allows for filtering given a strict bottom-up processing strategy.[3] However, processing of magic compiled grammars suffers from linguistically motivated underspecification as discussed above just the same.

Returning to the above example, the insight behind constraint propagation is that lifting the common restricting information contained in the lexical entries up into the head-adjunct schema makes it possible to determine locally that there are no modified specifiers in the grammar. In other words, applying constraint propagation to the head-adjunct schema of Figure 1 in a grammar with a lexicon in which the only modifying entries select *substantive* heads, propagates the constraint [SYNSEM|LOC|CAT|HEAD *subst*] into the mother of the head-adjunct schema. The resulting head-adjunct schema shown in Figure 4 is now specific enough to convey immediately that it cannot be used when specifiers need to be licensed.

Note that this way of making grammars more specific is an off-line

[3]See among others, Ramakrishnan 1988. In Minnen 1996 applications of these techniques to natural language processing are discussed.

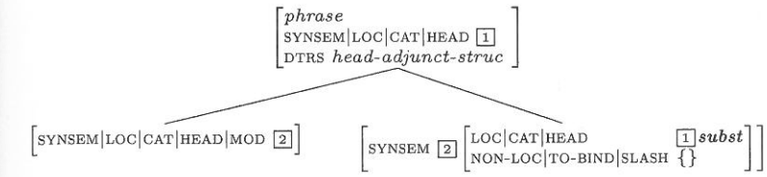

FIGURE 4 The Head-Adjunct ID Schema after constraint propagation

process performed completely automatically. It allows the grammar writer to specify theories in a lexically oriented fashion without any additional procedural specifications.

19.3 Efficient processing of the lexicon

Constraint propagation can also be applied to optimize automatically generated lexicons. In Meurers and Minnen 1997 a compiler is described which translates a set of HPSG lexical rules and their interaction into definite relations used to constrain lexical entries. This, so-called, *covariation approach* uses the generalizations captured by lexical rules for processing and makes it possible to deal with the infinite lexicon proposed in many recent HPSG theories. Most of the current HPSG analyses of Dutch, German, Italian, and French fall into this category. This is, for example, the case for all proposals working with verbal lexical entries which raise the arguments of a verbal complement (Hinrichs and Nakazawa 1989) that also use lexical rules such as the Complement Extraction Lexical Rule (Pollard and Sag 1994) to operate on those raised elements. Also an analysis treating adjunct extraction via lexical rules (van Noord and Bouma 1994) results in an infinite lexicon.

The linguist inputs the lexical rules used in his/her theory. On the basis of this specification and the signature of the proposed grammar, the covariation compiler automatically deduces the transfer of properties which were left unspecified in the lexical rule provided by the linguist. The compiler then uses the lexical rules and lexical entries to produce a definite clause encoding of lexical rules and their possible interaction. The resulting lexicon consists of extended lexical entries calling an interaction predicate encoding the entries which can be derived by lexical rule applications. Figure 5 shows an example for an extended lexical entry: a simplified entry for a German auxiliary using argument raising in the style of Hinrichs and Nakazawa (1989).

The call to the interaction predicate encodes the possible sequences of lexical rule applications. For a simple theory with a Complement Extraction Lexical Rule (CELR) and a Finitivization Lexical Rule (FINLR)

```
extended_lex_entry([OUT]):-
    [ PHON    < können >
      VFORM   bse
      SUBCAT  < [ VFORM   bse
                  SUBCAT  [1]
                  CONT    [2] ] | [1] >
      CONT    [ können'
                ARG     [2] ] ],
    interaction_0( [OUT] ).
```

FIGURE 5 The extended lexical entry for the modal auxiliary *können* ('can')

the slightly simplified interaction predicate looks as shown in Figure 6 on page 305.[4] The encoding in Figure 6 already contains the deduced transfer information in the call to the lexical rule predicates; for example, the PHON, VFORM, and CONT values are transferred to the CELR by adding the corresponding structure sharings to the [IN] tag and to the [AUX] tag appearing in the call to the `celr/2` predicate. Regarding the notation in the figure, a variable tag and a feature specification in the same argument slot are intended to be unified.

The automatically obtained encoding of lexical rule application in lexical entries shown in the above figures is not very efficient since before execution of the call to the interaction predicate it is unknown which information of the base lexical entry ends up in a derived lexical entry. One is therefore forced to execute the call to the interaction predicate directly when the lexical entry is used during processing, independent of the processing strategy used. Otherwise there is no information available to restrict the search space of a generation or parsing process.

Off-line constraint propagation can be used to avoid this by factoring out the information which is common to all solutions for the called interaction predicate. This is accomplished by computing the most specific generalization of these solutions and lifting this common information into the extended lexical entries. Let C be the common information, and $D_1, \ldots, D_k$ the solutions for the interaction predicate called. Then by distributivity we factor out C in $(C \wedge D_1) \vee \ldots \vee (C \wedge D_k)$ to obtain $C \wedge (D_1 \vee \ldots \vee D_k)$, where the D are assumed to contain no further common factors. The result of performing constraint propagation on the extended lexical entry for *können* is given in Figure 7 on page 306. In the next section we investigate in more detail how this result is achieved. Delaying the call to an interaction predicate as in van Noord and Bouma 1994 by freezing the recursive application of a lexical rule on the basis of

[4] The lexical rules in Figure 6 are simplified versions of the CELR (Pollard and Sag 1994, 378) and the Third-Singular Inflectional Rule (Pollard and Sag 1987, 210).


```
interaction_0([IN] [ PHON [1]
                     VFORM [2]
                     CONT [3] ], [OUT]):-
        celr([IN],[AUX]),
        interaction_0([AUX] [ PHON [1]
                              VFORM [2]
                              CONT [3] ], [OUT]).

interaction_0([IN] [ PHON [1]
                     SUBCAT [2]
                     SLASH [3]
                     CONT [4] ], [OUT]):-
        finlr([IN],[AUX]),
        interaction_1([AUX] [ PHON [1]
                              SUBCAT [2]
                              SLASH [3]
                              CONT [4] ], [OUT]).

interaction_0([OUT],[OUT]).
interaction_1([OUT],[OUT]).

celr( [ VFORM   bse
        SUBCAT  < [1] | [2] >
        SLASH   [3] ],
      [ SUBCAT  [2]
        SLASH   < [1] | [3] > ] ).

finlr( [ PHON   [1]
         VFORM  bse ],
       [ PHON   [2]
         VFORM  fin ] ) :- third_fin([1],[2]).

third_fin(können,kann).
...
```

FIGURE 6 Encoding sequences of lexical rule application in definite relations

user-specified delay information, can hide important restricting information because it is specified in the definition of the frozen goal. Therefore constraint propagation can be useful, also when coroutining techniques are used.

As discussed in Griffith 1996 an extension of the constraint language with *contexted constraints*, also referred to as dependent or named disjunctions, in certain cases makes it possible to circumvent constraint propagation. Encoding the disjunctive possibilities for lexical rule application using contexted constraints instead of definite clause attachments makes all relevant linguistic information available at lexical look-up. In case of infinite lexica, though, a definite clause encoding of disjunctive possibilities is still necessary and constraint propagation is indispensable for efficient processing (see Section 19.5).

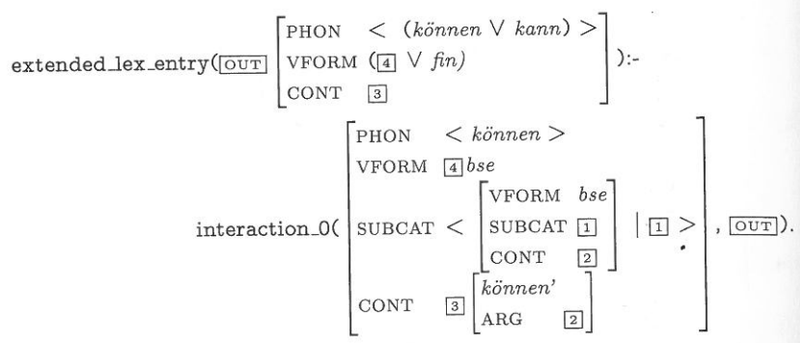

FIGURE 7 The extended lexical entry for *können* after specialized constraint propagation

19.4 Implementing Constraint Propagation

In this section we discuss implementations of some constraint propagation algorithms (in Prolog). We first present constraint propagation using a simple top-down interpreter and point out the problems of this basic algorithm. Subsequently, possible extensions of this interpreter with a, so-called, *branch-and-bound* optimization (le Provost and Wallace 1993) and a depth-bound are discussed. Finally, we show that it is possible to use knowledge about the specific structure of certain encodings to obtain specialized constraint propagation algorithms. In our case, we can exploit our knowledge of the encoding of the lexicon produced by the lexical rule compiler to define a specialized top-down interpreter that relieves us from termination problems related to the covariation encoding of infinite lexicons.

For reasons of exposition, in the remainder of this section, we assume a definite clause representation of an HPSG theory (Götz and Meurers 1995, 1997b) and do not make typed feature structure unification explicit.

19.4.1 Top-down constraint propagation

Consider the predicate `constraint_propagation_on_goal/0` in Figure 8. The predicate `get_goal/0` gets a particular goal on which we want to perform constraint propagation.[5] Subsequently, `generalized_solutions_for_goal/2` is called to produce a possibly more specific instance of this goal. The call to `write_goal/1` replaces the original goal with the possibly more specific

[5] If some kind of abstract interpretation is used to determine the places in a program where underspecification leads to massive nondeterminism, this information can be used to automatically make `get_goal/1` select the relevant goals.


```
constraint_propagation_on_goal:-
    get_goal(Goal),
    generalized_solutions_for_goal(Goal,MoreSpecificGoal),
    write_goal(MoreSpecificGoal).
```

FIGURE 8 A predicate defining simple off-line constraint propagation on a goal

goal obtained. As shown in Figure 9 `generalized_solutions_for_goal/2` computes an instance `GeneralizedSolutionsForGoal` of `Goal` by finding all its solutions with a call to `top_down_interpret/1` and subsequently generalizing over all the solutions.[6]

```
generalized_solutions_for_goal(Goal,GeneralizedSolutionsForGoal):-
    findall(Goal, top_down_interpret(Goal), SolutionList),
    generalize_all_solutions(SolutionList,GeneralizedSolutionsForGoal).
```

FIGURE 9 Generalizing all solutions for goal

Figure 10 provides the definition of `top_down_interpret/1`, a top-down interpreter provided by Pereira and Shieber (1987, 160ff).[7]

```
top_down_interpret(true).
top_down_interpret(Goal):-
    clause((Goal :- Body)),
    top_down_interpret(Body).
top_down_interpret((Body1, Body2)):-
    top_down_interpret(Body1),
    top_down_interpret(Body2).
```

FIGURE 10 A simple top-down interpreter

This interpreter falls prey to nontermination. For example, in the case of the recursive CELR of Figure 6 it is possible to remove elements from a

[6] Notice that in case there exists only one solution to a goal, the effect of performing constraint propagation on that goal is similar to its *partial evaluation*, see, for example, Pereira and Shieber 1987.

[7] The predicate is renamed here for expository reasons. The nonunit and unit clauses representing an HPSG theory are represented as `clause(( Head :- Body ))`. and `clause(( Head :- true ))`., respectively.

(subcategorization) list that is underspecified as in the extended lexical entry of Figure 5 over and over again.

Motivated by efficiency considerations, le Provost and Wallace (1993) propose the *branch-and-bound* optimization. This optimization also improves termination behavior. However, there exist linguistically motivated types of recursion for which branch-and-bound does not terminate either. Minnen et al. (1996) introduce the notion of a *building series*. Intuitively understood, a building series "builds up" a structure recursively until it matches a "base" case.[8] This type of recursion is problematic for top-down processing as this building can go on forever. Branch-and-bound does not ensure termination in the light of this type of recursion.

These termination problems necessitate an alternative implementation that avoids infinite loops. One possibility is to extend the interpreter in Figure 10 with a depth-bound as shown in Figure 11.[9]

```
db_top_down_interpret(true, Depth, Max):-
    Depth < Max.
db_top_down_interpret(Goal, Depth, Max):-
    Depth < Max,
    clause((Goal :- Body)),
    NewDepth is Depth + 1,
    db_top_down_interpret(Body, NewDepth, Max).
db_top_down_interpret((Body1, Body2), Depth, Max):-
    Depth < Max,
    db_top_down_interpret(Body1, Depth, Max),
    db_top_down_interpret(Body2, Depth, Max).
db_top_down_interpret(_Goal, Depth, Max):-
    Depth >= Max.
```

FIGURE 11 A depth-bounded top-down interpreter

Notice that the use of this highly incomplete interpreter for constraint propagation can only lead to a common factor that is too general. Intuitively understood, the depth-bound can only cut off branches of the search space which will eventually fail or lead to a solution more specific than the partial solution that has been computed. When the depth-bound hits clause 4 of `db_top_down_interpret/3` in Figure 11, the result returned in the first argument does not become further instantiated. As a result the `MoreSpecificGoal` computed can never become too specific and correctness is guaranteed.

[8] An example of a lexical rule that exhibits this type of recursion on structural information is the Add Adjuncts Lexical Rule proposed in van Noord and Bouma 1994.

[9] The call to `top_down_interpret/3` in `generalize_solutions_for_goal/2` shown in Figure 9 has to be changed accordingly.

While the depth-bounded interpreter can be employed in general, it is far from optimal to use it for constraint propagation of the covariation encoding of the lexicon. This is due to the fact that lexical rule application is encoded as forward chaining using *accumulator passing* (O'Keefe 1990): The `OUT` argument of an interaction predicate gets instantiated upon hitting a base case, i.e., a unit interaction clause. It serves only to "return" the lexical entry eventually derived. When the depth-bound cuts off a particular branch of the search space that corresponds to a recursively defined interaction predicate, the `OUT` argument remains completely uninstantiated. Consequently, generalizing over all possible (partial) solutions does not lead to a common factor that is more specific than the original goal selected by `get_goal/1`. In the next section, we show that it is possible to overcome this problem with a specialized interpreter.

19.4.2 Specialized Constraint Propagation

We employ a specialized top-down interpreter that allows us to extract an informative common factor using constraint propagation even in cases of a covariation encoding of an infinite lexicon. The specialized interpreter makes the use of a depth-bound to ensure termination of the interpretation of the interaction predicates superfluous.[10] Intuitively understood, the specialized interpreter exploits the fact that automatic property transfer is not influenced by recursion. I.e., the specifications that are left unchanged by a recursive lexical rule are independent of the number of times the rule is applied. This is a general, i.e., theory independent, property of the covariation encoding of lexical rule application and interaction, and therefore the improvement of the covariation encoding using specialized constraint propagation can be accomplished completely automatically.

We discuss a possible extension of the simple top-down interpreter given in Figure 10. For expository reasons the interpreter given in Figure 12 is simplified in the sense that it deals only with directly recursive interaction predicates such as the one given in Figure 6. Indirectly recursive interaction predicates necessitate a further extension of the interpreter with a tabulation technique as indirect recursion can not be

[10] As nontermination can not only result from recursive interaction predicates, a depth-bound might still be needed for the other predicates. We ignore this complication in the remainder of this section for expository reasons.

identified locally, i.e., as a property of the interaction clause under consideration. The original top-down interpreter is extended with an extra clause, i.e., the second clause of spec_top_down_interpret/1, which is specialized to deal with recursive interaction predicates which are identified by means of a call to recursive_interaction_clause/1. By eliminating the call to the lexical rule predicate (corresponding to the application of the recursive lexical rule) the interpreter abstracts over the information that is changed by the recursive lexical rule. As a result, only unchanged information remains. Subsequently, spec_top_down_interpret/2 is called to ensure that the same recursive interaction predicate is not called (over and over) again.[11]

```
spec_top_down_interpret(true).
spec_top_down_interpret(Goal):-
    clause((Goal :- Body)),
    recursive_interaction_clause((Goal :- Body)),
            % True if the retrieved clause is a directly recursive
            % interaction clause.
    make_body_more_general(Body, AdaptedBody),
            % Removes the call to the recursive lexical rule predicate from
            % Body in order to abstract over changed information.
    spec_top_down_interpret(AdaptedBody,(Goal :- Body)).
spec_top_down_interpret(Goal):-
    clause((Goal :- Body)),
    \+ recursive_interaction_clause((Goal :- Body)),
    spec_top_down_interpret(Body).
spec_top_down_interpret((Body1, Body2)):-
    spec_top_down_interpret(Body1),
    spec_top_down_interpret(Body2).

spec_top_down_interpret(Goal, RecursiveInteractionClause):-
    clause((Goal :- Body)),
    \+ (Goal :- Body) = RecursiveInteractionClause,
            % Avoid repeated application of RecursiveInteractionClause.
    spec_top_down_interpret(Body).
```

FIGURE 12 A top-down interpreter specialized for constraint propagation on (calls to) interaction predicates in a covariation lexicon

[11] We exploit the fact that two interaction clauses can never stand in the subsumption relation. Otherwise, a more elaborate equality test is needed in spec_top_down_interpret/2 to avoid repeated application.

Since we abstract over the information changed by a recursive lexical rule, the common factor that is extracted by means of performing constraint propagation with the specialized top-down interpreter might be too general: In case we are dealing with an infinite lexicon not all (possible infinite) applications of a recursive lexical rules are performed and there might be cases in which the application of a lexical rule after the n-th cycle is impossible even though we are taking it into account during constraint propagation. It is important to note though that such a situation can only lead to a common factor that is too general since generalizing over too large a set of solutions can only lead to a less specific generalization, not a more specific one. Therefore constraint propagation does not influence the soundness and completeness of the encoding. At run-time the additional lexical rule applications not ruled out by constraint propagation will simply fail.

Reconsider the definite clause encoding in Figure 6. As a result of the fact that repeated recursive application of interaction_0/2 is avoided, much relevant information can be lifted into the extended lexical entry. Figure 7 given in the previous section shows the result of performing specialized constraint propagation to the lexical entry for *können* (Figure 5).

19.4.3 Constant time lexical lookup

As Figure 7 shows, optimizing the extended lexical entries by means of specialized constraint propagation can also lift up phonological information in case of infinite lexicons.[12] In the case of parsing, this information can be used to index the lexicon so that constant time lexical lookup can be achieved. For this purpose, the extended lexical entry is split up as shown in Figure 13.

```
ind_lex_entry(können, [OUT][PHON < können >]):- extended_lex_entry([OUT]).
ind_lex_entry(kann, [OUT][PHON < kann >]):- extended_lex_entry([OUT]).
...
```

FIGURE 13 The result of splitting up the optimized lexical entry in Figure 7

On the basis of the input string it is now possible to access the lexicon in constant time. Without specialized constraint propagation this is impossible as the possible values of the phonology feature are hidden

[12] If there are recursive phonology changing rules the phonological information cannot be lifted by the constraint propagation using the specialized interpreter presented.

away deep in the covariation encoding of the lexical entries that can be derived from the base lexical entry.

19.5 Implementation Results

The depth-bounded constraint propagation method was implemented for the ConTroll system (Gerdemann and King 1994, Götz and Meurers 1997a) under Prolog. Test results on a complex grammar implementing an analysis of partial VP topicalization in German (Hinrichs et al. 1994) show that constraint propagation significantly improves parsing with a covariation encoding of lexical rules. For the lexicons produced by the covariation compiler, the implementation revealed that the most specific generalization which is propagated contains much valuable information. This is the case because usually the lexical entries resulting from lexical rule application only differ in few specifications compared to the number of specifications in a base lexical entry. The relation[13] between parsing times with the expanded (EXP), the covariation (COV) and the constraint propagated covariation (OPT) lexicon for the above grammar can be represented as OPT : EXP : COV = 1 : 1.3 : 14.

Acknowledgments

The research reported here was supported by Teilprojekt B4 'From Constraints to Rules: Efficient Compilation of HPSG Grammars' of SFB 340 'Sprachtheoretische Grundlagen für die Computerlinguistik' of the Deutsche Forschungsgemeinschaft. The authors wish to thank Thilo Götz, Dale Gerdemann and the anonymous reviewers for comments and discussion.

References

Frank, Anette. 1994. Verb Second by Underspecification. In *KONVENS '94*, ed. Harald Trost, 121–130. Berlin. Springer-Verlag.

Gerdemann, Dale, and Paul King. 1994. The Correct and Efficient Implementation of Appropriateness Specifications for Typed Feature Structures. In *Proceedings of the 15th Conference on Computational Linguistics.* Kyoto, Japan.

Götz, Thilo, and Detmar Meurers. 1995. Compiling HPSG Type Constraints into Definite Clause Programs. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics.* Boston, USA.

Götz, Thilo, and Detmar Meurers. 1997a. The ConTroll System as Large Grammar Development Platform. In *Proceedings of the ACL/EACL post-conference workshop on Computational Environments for Grammar Development and Linguistic Engineering.* Madrid, Spain.

[13]The comparison was done without indexing the lexicon by the word form, since such indexing is not possible for the covariation lexicon without constraint propagation.

Götz, Thilo, and Detmar Meurers. 1997b. Interleaving Universal Principles and Relational Constraints over Typed Feature Logic. In *Proceedings of the 35th Annual Meeting of the ACL and the 8th Conference of the EACL.* Madrid, Spain.

Griffith, John. 1996. Modularizing Contexted Constraints. In *Proceedings of the 16th Conference on Computational Linguistics.* Copenhagen, Denmark.

Hinrichs, Erhard, Detmar Meurers, and Tsuneko Nakazawa. 1994. Partial-VP and Split-NP Topicalization in German—An HPSG Analysis and its Implementation. Arbeitspapiere des SFB 340 no. 58. University of Tübingen, Germany.

Hinrichs, Erhard, and Tsuneko Nakazawa. 1989. Flipped Out: AUX in German. In *Papers from the 25th Regional Meeting of the Chicago Linguistic Society.* Chicago, Illinois. Chicago Linguistic Society.

Kathol, Andreas. 1994. Passives without Lexical Rules. In *German in Head-Driven Phrase Structure Grammar*, ed. John Nerbonne, Klaus Netter, and Carl Pollard. 237–272. Lecture Notes 46. CSLI Publications.

Krieger, Hans-Ulrich, and John Nerbonne. 1992. Feature-Based Inheritance Networks for Computational Lexicons. In *Default Inheritance Within Unification-Based Approaches to the Lexicon*, ed. Ted Briscoe, Ann Copestake, and V. de Paiva. Cambridge: Cambridge University Press.

le Provost, Thierry, and Mark Wallace. 1993. Generalised Constraint Propagation over the CLP Scheme. *Journal of Logic Programming 10.*

Marriott, Kim, Lee Naish, and Jean-Louis Lassez. 1988. Most Specific Logic Programs. In *Proceedings of 5th Int. Conference and Symposium on Logic Programming.*

Meurers, Detmar, and Guido Minnen. 1997. A Computational Treatment of Lexical Rules in HPSG as Covariation in Lexical Entries. *Computational Linguistics 23(4).*

Minnen, Guido. 1996. Magic for Filter Optimization in Dynamic Bottom-up Processing. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics.* Santa Cruz, USA.

Minnen, Guido, Dale Gerdemann, and Erhard Hinrichs. 1996. Direct Automated Inversion of Logic Grammars. *New Generation Computing 14(2).*

O'Keefe, Richard. 1990. *The Craft of Prolog.* Cambridge, USA: MIT Press.

Pereira, Fernando, and Stuart Shieber. 1987. *Prolog and Natural Language Analysis.* CSLI Lecture Notes, No. 10. Chicago, USA: Chicago University Press.

Pollard, Carl, and Ivan A. Sag. 1987. *An Information-based Approach to Syntax and Semantics: Volume 1 Fundamentals.* CSLI Lecture Notes, No., No. no. 13. Stanford, USA: Center for the Study of Language and Information.

Pollard, Carl, and Ivan Sag. 1994. *Head-driven Phrase Structure Grammar.* Chicago, USA: University of Chicago Press.

Ramakrishnan, Raghu. 1988. Magic Templates: A Spellbinding Approach to Logic Programs. In *Proceedings of the 5th Int. Conference and Symposium on Logic Programming.*

Riehemann, Susanne. 1993. Word Formation in Lexical Type Hierarchies: A Case Study of *bar*-Adjectives in German. Master's thesis, University of Tübingen.

Tamaki, Hisao, and Taisuke Sato. 1984. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the 2nd Int. Conference on Logic Programming.* Uppsala, Sweden.

van Noord, Gertjan, and Gosse Bouma. 1994. The Scope of Adjuncts and the Processing of Lexical Rules. In *Proceedings of the 15th Conference on Computational Linguistics.* Kyoto, Japan.

Part VI

Semantics and Pragmatics