A Bag of Useful Techniques for Efficient and Robust Parsing

Bernd Kiefer[†], **Hans-Ulrich Krieger**[†], **John Carroll**[‡], and **Rob Malouf**[*]
[†]German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken
[‡]Cognitive and Computing Sciences, University of Sussex
Falmer, Brighton BN1 9QH, UK
[*]Center for the Study of Language and Information, Stanford University
Ventura Hall, Stanford, CA 94305-4115, USA
{kiefer,krieger}@dfki.de, johnca@cogs.susx.ac.uk, malouf@csli.stanford.edu

Abstract

This paper describes new and improved techniques which help a unification-based parser to process input efficiently and robustly. In combination these methods result in a speed-up in parsing time of more than an order of magnitude. The methods are correct in the sense that none of them rule out legal rule applications.

1 Introduction

This paper describes several generally-applicable techniques which help a unification-based parser to process input efficiently and robustly. As well as presenting a number of new methods, we also report significant improvements we have made to existing techniques. The methods preserve correctness in the sense they do not rule out legal rule applications. In particular, none of the techniques involve statistical or approximate processing. We also claim that these methods are independent of the concrete parser and neutral with respect to a given unification-based grammar theory/formalism.

How can we gain reasonable efficiency in parsing when using large integrated grammars with several thousands of huge lexicon entries? Our belief is that there is no single method which achieves this goal alone. Instead, we have to develop and use a set of "cheap" filters which are correct in the above sense. As we indicate in section 10, combining these methods leads to a speed-up in parsing time (and reduction of space consumption) of more than an order of magnitude when applied to a mature, well engineered unification-based parsing system.

We have implemented our methods as extensions to a HPSG grammar development environment (Uszkoreit et al., 1994) which employs a sophisticated typed feature formalism (Krieger and Schäfer, 1994; Krieger and Schäfer, 1995) and an advanced agenda-based bottom-up chart parser (Kiefer and Scherf, 1996). A specialized runtime version of this system is currently used in VERBMOBIL as the primary deep analysis component.[1]

In the next three sections, we report on transformations we have applied to the knowledge base (grammar/lexicon) and on modifications in the core formalism (unifier, type system). In Section 5–8, we describe how a given parser can be extended to filter out possible rule applications efficiently before performing "expensive" unification. Section 9 shows how to compute best partial analyses in order to gain a certain level of robustness. Finally, we present empirical results to demonstrate the efficiency gains, and speculate on extensions we intend to work on in the near future. Within the different sections, we refer to three corpora we have used to measure the effects of our methods. The reference corpora for English, German, and Japanese consist of 1200–5000 samples.

2 Precompiling the Lexicon

Lexicon entries in the development system are small templates that are loaded and expanded on demand by the typed feature structure system. Thereafter, all lexical rules are applied to the expanded feature structures. The results of these two computations form the input of the analysis stage.

[1]VERBMOBIL (Wahlster, 1993) deals with the translation of spontaneously spoken dialogues, where only a minor part consists of "sentences" in a linguistic sense. Current languages are English, German, and Japanese. Some of the methods were originally developed in the context of another HPSG environment, the LKB (Copestake, 1998). This lends support to our claims of their independence from a particular parser or grammar engine.

In order to save space and time in the run-time system, the expansion and the application of lexical rules is now done off-line. In addition, certain parts of the feature structure are deleted, since they are only needed to restrict the application of lexical rules (see also section 7 for a similar approach). For each stem, all results are stored in compact form as one compiled LISP file, which allows to access and load a requested entry rapidly with almost no restriction on the size of the lexicon. Although load time is small (see figure 1), the most frequently used entries are cached in main memory, reducing effort in the lexicon stage to a minimum.

We continue to compute morphological information online, due to the significant increase of entries (a factor of 10 to 20 for German), which is not justifiable considering the minimal computation time for this operation.

	German	English	Japanese
# stems	4269	3754	1875
space	10.3 KB	10.8 KB	5.4 KB
entries	6	2.2	2.1
load time	25.8 msec	29.5 msec	7.5 msec

Figure 1: Space and time requirements. *space*, *entries* and *load time* values are per stem

3 Improvements in unification

Unification is the single most expensive operation performed in the course of parsing. Up to 90% of the CPU time expended in parsing a sentence using a large-scale unification based grammar can go into feature structure and type unification. Therefore, any improvements in the efficiency of unification would have direct consequences for the overall performance of the system.

One key to reducing the cost of unification is to find the simplest set of operations that meet the needs of grammar writers but still can be efficiently implemented. The unifier which was part of the original HPSG grammar development system mentioned in the introduction (described by (Backofen and Krieger, 1993)) provided a number of advanced features, including distributed (or named) disjunctions (Dörre and Eisele, 1990) and support for full backtracking. While these operations were sometimes useful, they also made the unifier much more complex than was really necessary.

The unification algorithm used by the current system is a modification of Tomabechi's (Tomabechi, 1991) "quasi-destructive" unification algorithm. Tomabechi's algorithm is based on the insight that unification often fails, and copying should only be performed when the unification is going to succeed. This makes it particularly well suited to chart-based parsing.

During parsing, each edge must be built without modifying the edges that contribute to it. With a non-backtracking unifier, one option is to copy the daughter feature structures before performing a destructive unification operation, while the other is to use a non-destructive algorithm that produces a copy of the result up to the point a failure occurs. Either approach will result in some structures being built in the course of an unsuccessful unification, wasting space and reducing the overall throughput of the system. Tomabechi avoids these problems by simulating non-destructiveness without incurring the overhead necessary to support backtracking. First, it performs a destructive (but reversible) check that the two structures are compatible, and only when that succeeds does it produce an output structure. Thus, no output structures are built until it is certain that the unification will ultimately succeed.

While an improvement over simple destructive unification, Tomabechi's approach still suffers from what Kogure (Kogure, 1990) calls *redundant copying*. The new feature structures produced in the second phase of unification include copies of all the substructures of the input graphs, even when these structures are unchanged. This can be avoided by reusing parts of the input structures in the output structure (Carroll and Malouf, 1999) without introducing significant bookkeeping overhead.

To keep things as simple and efficient as possible, the improved unifier also only supports conjunctive feature structures. While disjunctions can be a convenient descriptive tool for writing grammars, they are not absolutely necessary. When using a typed grammar formalism, most disjunctions can be easily put into the type hierarchy. Any disjunctions which cannot be removed by introducing new supertypes can be eliminated by translating the grammar into

disjunctive normal form (DNF). Of course, the ratio of the number of rules and lexical entries in the original grammar and the DNFed grammar depends on the 'style' of the grammar writer, the particular grammatical theory used, the number of disjunction alternatives, and so on. However, context management for distributed disjunctions requires enormous overhead when compared to simple conjunctive unification, so the benefits of using a simplified unifier outweigh the cost of moving to DNF. For the German and Japanese VERBMOBIL grammars, we got 1.4–3× more rules and lexical entries, but by moving to a sophisticated conjunctive unifier we obtained an overall speed-up of 2–5.

4 Precompiling Type Unification

After changing the unification engine, type unification now became a big factor in processing: nearly 50% of the overall unification and copying time was taken up by the computation of the greatest lower bounds (GLBs). Although we have in the past computed GLBs online efficiently with bit vectors, off-line computation is of course superior.

The feasibility of the latter method depends on the number of types $\mathcal{T}$ of a grammar. The English grammar employs 6000 types which results in 36,000,000 possible GLBs. Our experiments have shown, however, that only 0.5%–2% of the type unifications were successful and only these GLBs need to be entered into the GLB table. In our implementation, accessing an arbitrary GLB takes less than 0.002 msec, compared to 15 msec of 'expensive' bit vector computation (following (Aït-Kaci et al., 1989)) which also produces a lot of memory garbage. Our method, however, does not consume any memory and works as follows. We first assign a unique code (an integer) to every type $t \in \mathcal{T}$. After that, the GLB of s and t is assigned the following code (again an integer, in fact a fixnum): $code(s) \times |\mathcal{T}| + code(t)$. This array-like encoding guarantees that a specific code is given away to a GLB at most once. Finally, this code together with the GLB is stored in a hash table. Hence, type unification costs are minimized: two symbol table lookups, one addition, one multiplication, and a hash table lookup.

In order to access a unique maximal lower bound (= GLB), we must require that the type hierarchy is a lower semilattice (or bounded complete partial order). This is often not the case, but this deficiency can be overcome either by pre-computing the missing types (an efficient implementation of this takes approximately 25 seconds for the English grammar) or by making the online table lookup more complex.

A naïve implementation of the off-line computation (*compute the GLBs for* $\mathcal{T} \times \mathcal{T}$) only works for small grammars. Since type unification is a commutative operation ($glb(s,t) = glb(t,s)$; $s,t \in \mathcal{T}$), we can improve the algorithm by computing only $glb(s,t)$. A second improvement is due to the following fact: if the GLB of s and t is bottom, we do not have to compute the GLBs of the subtypes of both s and t, since they guarantee to fail. Even with these improvements, the GLB computation of a specific grammar took more than 50 CPU hours, due to the special 'topology' of the type hierarchy. However, not even the failing GLBs need to be computed (which take much of the time). When starting with the leaves of the type hierarchy, we can compute maximal components w.r.t. the supertype relation: by following the subsumption links upwards, we obtain sets of types, s.t. for a given component C, we can guarantee that $glb(s,t) \neq \bot$, for all $s,t \in C$. This last technique has helped us to drop the off-line computation time to less than one CPU hour.

Overall when using the off-line GLBs, we obtained a parsing speed-up of 1.5, compared to the bit vector computation.[2]

5 Precompiling Rule Filters

The aim of the methods described in this and the next section is to avoid failing unifications by applying cheap 'filters' (i.e., methods that are cheaper than unification). The first filter we want to describe is a rule application filter. We have used this method for quite a while, and it has proven both efficient and easy to employ.

Our rule application filter is a function that

[2] An alternative approach to improving the speed of type unification would be to implement the GLB table as a cache, rather than pre-computing the table's contents exhaustively. Whether this works well in practice or not depends on the efficiency of the primitive $glb(s,t)$ computation; if the latter were relatively slow then the parser itself would run slowly until the cache was sufficiently full that cache hits became predominant.

takes two rules and an argument position and returns a boolean value that specifies if the second rule can be unified into the given argument position of the first rule.

Take for example the binary filler-head rule in the HPSG grammar for German. Since this grammar allows not more than one element on the SLASH list, the left hand side of the rule specifies an empty list as SLASH value. In the second (head) argument of the rule, SLASH has to be a list of length one. Consequently, a passive chart item whose topmost rule is a filler-head rule, and so has an empty SLASH, can not be a valid second argument for another filler-head rule application. The filter function, when called with arguments (*filler-head-rule-nr*, *filler-head-rule-nr*, 2) for mother rule, topmost rule of the daughter and argument position respectively, will return false and no unification attempt will be made.

The conjunctive grammars have between 20 and 120 unary and binary rule schemata. Since all rule schemata in our system bear a unique number, this filter can be realized as a three dimensional boolean array. Thus, access costs are minimized and no additional memory is used at run-time. The filters for the three languages are computed off-line in less than one minute and rule out 50% to 60% of the failing unifications during parsing, saving about 45% of the parsing time.

6 Dynamic Unification Filtering ('Quick Check')

Our second filter (which we have dubbed the 'quick check') exploits the fact that unification fails more often at certain points in feature structures than at others. For example, syntactic features such as CAT(egory) are very frequent points of failure, whereas unification almost never fails on semantic features which are used merely to accumulate pieces of the logical form. Since all substructures are typed, unification failure is manifested by a type clash when attempting a type unification. The quick check is invoked before each unification attempt to check the most frequent failure points, each stored as a feature *path*.

The technique works as follows. First, there is an off-line stage, in which a modified unification engine is used that does not return immediately after a single type unification failure, but instead records in a global data structure the paths at which all such failures occurred. Using this modified system a set of sentences is parsed, and the n paths with the highest failure counts are saved. It is exactly these paths that are used later in filtering.

During parsing, when an active chart item (i.e., a rule schema or a partly instantiated rule schema) and a passive chart item (a lexical entry or previously-built constituent) are combined, the parser has to unify the feature structure of the passive item into the substructure of the active item that corresponds to the argument to be filled. If either of the two structures has not been seen before, the parser associates with it a vector of length n containing the types at the end of the previously determined paths. The first position of the vector contains the type corresponding to the most frequently failing path, the second position the second most frequently failing path, and so on. Otherwise, the existing vectors of types are retrieved. Corresponding elements in the vectors are then type-unified, and full unification of the feature structures is performed only if all the type unifications succeed.

Clearly, when considering the number of paths n used for this technique, there is a tradeoff between the time savings from filtered unifications and the effort required to create the vectors and compare them. The main factors involved are the speed of type unification and the percentage of unification attempts filtered out (the 'filter rate') with a given set of paths. The optimum number of paths cannot be determined analytically. Our English, German and Japanese grammars use between 13 to 22 paths for quick check filtering, the precise number having been established by experimentation. The paths derived for these grammars are somewhat surprising, and in many cases do not fit in with the intuitions of the grammar-writers. In particular, some of the paths are very long (of length ten or more). Optimal sets of paths for grammars of this complexity could not be produced manually.

The technique will only be of benefit if type unification is computationally cheap—as indeed it is in our implementation (section 4)—and if the filter rate is high (otherwise the extra work

performed essentially just duplicates work carried out later in unification). There is also overlap between the quick check and the rule filter (previous section) since they are applied at the same point in processing. We have found that (given a reasonable number of paths) the quick check is the more powerful filter of the two because it functions *dynamically*, taking into account feature instantiations that occur during the parsing process, but that the rule filter is still valuable if executed first since it is a single, very fast table lookup. Applying both filters, the filter rate ranges from 95% to over 98%. Thus almost all failing unifications are avoided. Compared to the system with only rule application filtering, parse time is reduced by approximately 75%[3].

7 Reducing Feature Structure Size via Restrictors

The 'category' information that is attached to each chart item of the parser consists of a single feature structure. Thus a rule is implemented by a feature structure where the daughters have to be unified into predetermined substructures. Although this implementation is along the lines of HPSG, it has the drawback that the tree structure that is already present in the chart items is duplicated in the feature structures.

Since HPSG requires all relevant information to be contained in the SYNSEM feature of the mother structure, the unnecessary daughters only increase the size of the overall feature structure without constraining the search space. Due to the *Locality Principle* of HPSG (Pollard and Sag, 1987, p. 145ff), they can therefore be legally removed in fully instantiated items. The situation is different for active chart items since daughters can affect their siblings.

To be independent from a certain grammatical theory or implementation, we use *restrictors* similar to (Shieber, 1985) as a flexible and easy-to-use specification to perform this deletion. A *positive restrictor* is an automaton describing the paths in a feature structure that will remain after *restriction* (the deletion operation), whereas a *negative restrictor* specifies the parts to be deleted. Both kinds of restrictors can be used in our system.

In addition to the removal of the tree structure, the grammar writer can specify the restrictor further to remove features that are only used locally and do not play a role in further derivation. It is worth noting that this method is only correct if the specified restrictor does not remove paths that would lead to future unification failures. The reduction in size results in a speed-up in unification itself, but also in copying and memory management.

As already mentioned in section 2, there exists a second restrictor to get rid of unnecessary parts of the lexical entries after lexicon processing. The speed gain using the restrictors in parsing ranges from 30% for the German system to 45% for English.

8 Limiting the Number of Initial Chart Items

Since the number of lexical entries per stem has a direct impact on the number of parsing hypotheses (in the worst case leads to an exponential increase), it would be a good idea to have a cheap mechanism at hand that helps to limit these initial items. The technique we have implemented is based on the following observation: in order to contribute to a reading, certain items (concrete lexicon entries, but also classes of entries) require the existence of other items such that the non-existence of one allows a safe deletion of the other (and vice versa). In German, for instance, prefix verbs require the right separable prefixes to be present in the chart, but also a potential prefix requires its prefix verb.

Note that such a technique operates in a much larger context (in fact, the whole chart) than a local rule application filter or the quick-check method. The method works as follows. In a preprocessing step, we first separate the chart items which encode prefix verbs from those items which represent separable prefixes. Since both specify the morphological form of the prefix, a set-exclusive-or operation yields exactly the items which can be safely deleted from the chart.

Let us give some examples to see the usefulness of this method. In the sentence *Ich <u>komme</u> morgen* (*I (will) come tomorrow*), *komme* maps

[3]There are refinements of the technique which we have implemented and which in practice produce additional benefits; we will report these in a subsequent paper. Briefly, they involve an improvement to the path collection method, and the storage of other information besides types in the vectors.

onto 97 lexical entries—remember, *komme* might encode prefix verbs such as *ankommen* (*arrive*), *zurückkommen* (*come back*), etc. although here, none of the prefix verb readings are valid, since a prefix is missing. Using the above method, only 8 of 97 lexical entries will remain in the chart. The sentence *Ich komme morgen an* (*I (will) arrive tomorrow*) results in 8+7 entries for *komme* (8 entries for the *come* reading together with 7 entries for the *arrive* reading of *komme*) and 3 prepositional readings plus 1 prefix entry for *an*. However in *Der Mann wartet an der Tür* (*The man is waiting at the door*), only the three prepositional readings for *an* come into play, since no prefix verb *anwartet* exists. Although there are no English prefix verbs, the method also works for verbs requiring certain particles, such as *come, come along, come back, come up*, etc.

The parsing time for the second example goes down by a factor of 2.4; overall savings w.r.t. our reference corpus is 17% of the parsing time (i.e., speed-up factor of 1.2).

9 Computing Best Partial Analyses

Given deficient, ungrammatical, or spontaneous input, a traditional parser is not able to deliver a useful result. To overcome this disadvantage, our approach focuses on partial analyses which are combined in a later stage to form total analyses without giving up the correctness of the overall deep grammar. But what can be considered good partial analyses? Obviously a (sub)tree licensed by the grammar which covers a continuous part of the input (i.e., a passive parser edge). But not every passive edge is a good candidate since otherwise we would end up with perhaps thousands of them. Instead, our approach computes an 'optimal' connected sequence of partial analyses which cover the whole input. The idea here is to view the set of passive edges as a directed graph and to compute shortest paths w.r.t. a user-defined estimation function.

Since this graph is acyclic and topologically sorted, we have chosen the DAG-shortest-path algorithm (Cormen et al., 1990) which runs in $\Theta(V + E)$. We have modified this algorithm to cope with the needs we have encountered in speech parsing: (i) one can use several start and end vertices (e.g., in case of n-best chains or word graphs); (ii) all best shortest paths are returned (i.e., we obtain a shortest-path subgraph); (iii) estimation and selection of the best edges is done incrementally when parsing n-best chains (i.e., only new passive edges entered into the chart are estimated and perhaps selected). This approach has one important property: even if certain parts of the input have not undergone at least one rule application, there are still lexical edges which help to form a best path through the passive edges. This means that we can interrupt parsing at any time, but still obtain a useful result.

Let us give an example to see how the estimation function on edges (= trees) might look like (this estimation is actually used in the German grammar):

- n-ary tree ($n > 1$) with utterance status (e.g., NPs, PPs): value 1
- lexical items: value 2
- otherwise: value ∞

This approach does not always favor paths with longest edges as the example in figure 2 shows—instead it prefers paths containing no lexical edges (where this is possible) and there might be several such paths having the same cost. Longest (sub)paths, however, can be obtained by employing an exponential estimation function. Other properties, such as prosodic information or probabilistic scores could also be utilized in the estimation function. A detailed description of the approach can be found in (Kasper et al., 1999).

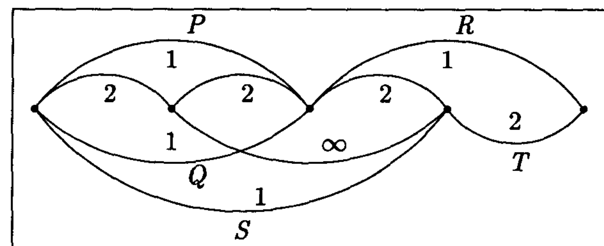

Figure 2: Computing best partial analyses. Note that the paths PR and QR are chosen, but not ST, although S is the longest edge.

10 Conclusions and Further Work

The collection of methods described in this paper has enabled us to unite deep linguistic analysis with speech processing. The overall speedup compared to the original system is about a factor of 10 up to 25. Below we present some absolute timings to give an impression of the current systems' performance.

	German	English	Japanese
# sentences	5106	1261	1917
# words	7	6.7	7.2
# lex. entries	40.9	25.6	69.8
# chart items	1024	234	565
# results	5.8	12.4	53.6
time first	1.46 s	0.24 s	0.9 s
time overall	4.53 s	1.38 s	4.42 s

In the table, the last six rows are average values per sentence. *time first* and *time overall* are the mean CPU times to compute the first result and the whole search space respectively. *# lex. entries* and *# chart items* give an impression of the lexical and syntactic ambiguity of the respective grammars[4]

The German and Japanese corpora and half of the English corpus consist of transliterations of spoken dialogues used in the VERBMOBIL project. These dialogues are real world dialogues about appointment scheduling and vacation planning. They contain a variety of syntactic as well as spontaneous speech phenomena. The remaining half of the English corpus is taken from a manually constructed test suite, which may explain some of the differences in absolute parse time.

Most of the methods are corpus independent, except for the quick check filter, which requires a training corpus, and the use of a purely conjunctive grammar, which will do worse in cases of great amounts of syntactic ambiguity because there is currently no ambiguity packing in the parser. For the quick check, we have observed that a random subset of the corpora with about one to two hundred sentences is enough to obtain a filter with nearly optimal filter rate.

Although the actual efficiency gain will vary for differently implemented grammars, we are certain that these techniques will lead to substantial improvements in almost every unification based system. It is, for example, quite unlikely that unification failures are equally distributed over the different nodes of the grammar's feature structure, which is the most important prerequisite for the quick check filter to work. Avoiding disjunctions usually requires a reworking of the grammar which will pay off in the end.

We have shown that the combination of algorithmic methods together with some discipline in grammar writing can lead to a practical high performance analysis system even with large general grammars for different languages.

There is, however, room for further improvements. We intend to generalize to other cases the technique for removing unnecessary lexical items. A detailed investigation of the quick-check method and its interaction with the rule application filter is planned for the near future. Since almost all failing unifications are avoided through the use of filtering techniques, we will now focus on methods to reduce the number of chart items that do not contribute to any analysis; for instance, by computing context-free or regular approximations of the HPSG grammars (e.g., (Nederhof, 1997)).

[4]The computations were made using a 300MHz SUN Ultrasparc 2 with Solaris 2.5. The whole system is programmed in Franz Allegro Common Lisp.

Acknowledgments

The research described in this paper has greatly benefited from a very fruitful collaboration with the HPSG group of CSLI at Stanford University. This cooperation is part of the deep linguistic processing effort within the BMBF project VERBMOBIL. Special thanks are due to Stefan Müller for discussing the topic of German prefix verbs. Thanks to Dan Flickinger who provided us with several English phenomena. We also want to thank Nicolas Nicolov for reading a version of this paper. Stephan Oepen's and Mark-Jan Nederhof's fruitful comments have helped us a lot. Finally, we want to thank the anonymous ACL reviewers for their comments. This research was supported by the German Federal Ministry for Education, Science, Research and Technology under grant no. 01 IV 701 V0, and by a UK EPSRC Advanced Fellowship to the third author, and also is in part based upon work supported by the National Science Foundation under grant number IRI-9612682.

References

Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. 1989. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January.

Rolf Backofen and Hans-Ulrich Krieger. 1993. The *TDL/UDiNe* system. In R. Backofen, H.-U. Krieger, S.P. Spackman, and H. Uszkoreit, editors, *Report of the EAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken*, pages 67–74. DFKI Research Report D-93-27.

John Carroll and Robert Malouf. 1999. Efficient graph unification for parsing feature-based grammars. University of Sussex and Stanford University.

Ann Copestake. 1998. The (new) LKB system. Ms, Stanford University, `http://www-csli.stanford.edu/~aac/newdoc.pdf`.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Jochen Dörre and Andreas Eisele. 1990. Feature logic with disjunctive unification. In *Proceedings of the 13th International Conference on Computational Linguistics, COLING-90*, pages Vol. 3, 100–105.

Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, C.J. Rupp, and Karsten L. Worm. 1999. Charting the depths of robust speech parsing. In *Proceedings of the ACL-99 Thematic Session on* Robust Sentence-Level Interpretation.

Bernd Kiefer and Oliver Scherf. 1996. Gimme more HQ parsers. The generic parser class of DISCO. Unpublished draft. German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Kiyoshi Kogure. 1990. Strategic lazy incremental copy graph unification. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING '90)*, pages 223–228, Helsinki.

Hans-Ulrich Krieger and Ulrich Schäfer. 1994. *TDL*—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, pages 893–899. An enlarged version of this paper is available as DFKI Research Report RR-94-37.

Hans-Ulrich Krieger and Ulrich Schäfer. 1995. Efficient parameterizable type expansion for typed feature formalisms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI-95*, pages 1428–1434. DFKI Research Report RR-95-18.

Mark Jan Nederhof. 1997. Regular approximations of cfls: A grammatical view. In *Proceedings of the 5th International Workshop on Parsing Technologies, IWPT'97*, pages 159–170.

Carl Pollard and Ivan A. Sag. 1987. *Information-Based Syntax and Semantics. Vol. I: Fundamentals.* CSLI Lecture Notes, Number 13. Center for the Study of Language and Information, Stanford.

Stuart M. Shieber. 1985. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, ACL-85*, pages 145–152.

Hideto Tomabechi. 1991. Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, volume 29, pages 315–322.

Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. 1994. DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of COLING-94*, pages 436–440. DFKI Research Report RR-94-38.

Wolfgang Wahlster. 1993. VERBMOBIL—translation of face-to-face dialogs. Research Report RR-93-34, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany. Also in Proc. MT Summit IV, 127–135, Kobe, Japan, July 1993.