Dissertation

Principle-based Processing of Natural Language using CLP Techniques

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

eingereicht an der Technisch-Naturwissenschaftlichen Fakultät der Technischen Universität Wien

von

Dipl.-Ing. Johannes Matiasek

Spargelfeldstraße 192/2, 1220 Wien
Matrikelnummer: 7025121
geboren am 23. Mai 1952 in Wien

Wien, im Oktober 1994 ....

Kurzfassung

In dieser Dissertation wird eine Methode zur Implementation prinzipienbasierter Grammatikformalismen, insbesondere *Head Driven Phrase Structure Grammar* (HPSG) vorgestellt. Der Beitrag dieser Arbeit ist die Entwicklung eines Systems, das die deklarative *und* ausführbare Spezifikation prinzipienbasierter Grammatiken, die typisierte Merkmalsstrukturen verwenden, erlaubt. Dadurch kann die Grammatik als ein von der Verarbeitungsrichtung unabhängiger, operationaler Modul im Bereich der Verarbeitung natürlicher Sprache etabliert werden.

Die Motivation für diese Arbeit geht auf zwei Trends, die derzeit in der Computerlinguistik festzustellen sind, zurück: erstens, die Ersetzung von regelbasierten Formalismen durch prinzipienbasierte Ansätze, und zweitens die weitverbreitete Verwendung von Merkmalsstrukturen zur Beschreibung linguistischer Strukturen.

Das Vorhandensein wohlbekannter Techniken zur Verarbeitung von Phrasenstrukturgrammatiken und das gleichzeitige Fehlen entsprechender und gleichermaßen gutverstandener Methoden für prinzipienbasierte Ansätze führte bisher zu dem Effekt, daß prinzipienbasierte Theorien zum Zwecke der Implementierung in der Form von Phrasenstrukturgrammatiken reformuliert wurden. Daher wurden in dieser Dissertation Methoden entwickelt, die eine direkte Implementierung prinzipienbasierter Formalismen erlauben.

In dieser Arbeit wird die Unifikation typisierter Merkmalsstrukturen unter Restriktion durch Prinzipen als *constraint solving* im CLP Paradigma interpretiert. Die Prinzipien der Grammatik werden als Lizensierungsbedingungen auf jedem Knoten einer Merkmalsstruktur gesehen und werden als konditionale Constraints formuliert. Der *constraint solving* Mechanismus, der in DMCAI-Clp—einer Erweiterung von SICStus Prolog, die es erlaubt, Unifikationserweiterungen zu spezifizieren—implementiert ist, verzögert die Anwendung konditionaler Constraints auf unzureichend instantiierte Merkmalsstrukturen. Daher wird eine deklarative Spezifikation der Grammatik ermöglicht, die nichtsdestotrotz operational ist, und zwar unabhängig von der Verarbeitungsrichtung (d.h. Parsing oder Generierung).

Die Praktikabilität dieses Ansatzes zur Implementation typisierter Merkmalsstrukturen wird anhand zweier linguistisch motivierter Anwendungsbeispiele demonstriert, deren Integration in bestehende HPSG Implementierungen nicht unproblematisch ist. Die Integration einer Zweiebenen-Morphologiekomponente in eine kombinierte Satz- und Wortebenengrammatik im HPSG Formalismus als eine spezielle Relation zeigt die Flexibilität des Systems. Die zweite Anwendung behandelt Kasuszuweisung und Argumentreduktionsphänomene im Deutschen. Sie zeigt, wie lexikalische Regeln von HPSG, die nichtmonotone Effekte hervorrufen und daher nicht allzugut in einen Formalismus, dessen Ziel Deklarativität ist, passen, durch einen konstruktiven Mechanismus innerhalb der Wortebenengrammatik ersetzt werden können.

Abschließend werden Kompilationstechniken und verschiedene Methoden der Disjunktionsbehandlung als effizienzsteigernde Erweiterungen des Basissystems diskutiert.

Abstract

This thesis presents a framework for the implementation of principle based grammar formalisms, in particular Head Driven Phrase Structure grammar (HPSG). The achievement of this work is the development of a framework allowing for a declarative *and* executable specification of principle based grammars employing typed feature structures. Thus it establishes grammar as an operational module independent of processing direction within natural language processing.

The motivation for this work stems from two current trends in computational linguistics: firstly, a shift from phrase structure rule based to principle based grammar formalisms and secondly from the predominant use of typed feature structures to represent linguistic structures.

The availability of well-known techniques for processing phrase structure grammars and the absence of equally well-studied methods in the principle based field have lead to the effect that principle based theories are often recast in a phrase structure based style in order to implement them. In this thesis methods allowing for a direct implementation of principle based formalisms have been developed.

In this work, unification of typed feature structures under the restrictions of principled constraints is interpreted as constraint solving in the CLP paradigm. Principles of grammar, viewed as licensing conditions applying to every node of a feature structure, take the form of conditional constraints. The constraint solving mechanism, implemented in DMCAI-Clp—a variant of SICStus Prolog allowing for unification extensions— takes care of delaying the enforcement of conditional constraints on insufficiently instantiated feature structures. Thus a declarative specification of the grammar is made possible, which is nevertheless operational independently of the processing direction (i.e. parsing or generation).

The usefulness of this approach to implementing typed feature systems is demonstrated by two applications which are linguistically motivated, but whose integration into existing HPSG implementations is not unproblematic. The direct integration of two-level morphology into an combined sentence-level and word-level HPSG grammar as a relation demonstrates the flexibility of the system. The second application deals with case assignment and argument reduction phenomena in German. It demonstrates how lexical rules, which provoke nonmonotonic effects and thus don't square well with a formalism aiming at declarativeness, can be replaced by a constructive mechanism within the word-level grammar.

Finally compilation techniques and various methods of disjunction handling are discussed as efficiency enhancing extensions to the basic system.

Acknowledgements

The production of this thesis and the scientific work I have done in the past years would not have been possible in a vacuum and so I am grateful to all the ones who had enriched my working environment.

First of all, I would like to thank Prof. Robert Trappl for giving me the opportunity to work at his institute and for providing a stimulating research atmosphere and continuing support.

My old friend Harald Trost has introduced me to the field of natural language processing and has been my thesis advisor providing much help and support. I am very grateful for that.

I have much benefited from my long term collaboration with Wolfgang Heinz. I thank him for acquainting me with linguistics and for many enlightening discussions.

Furthermore I would like to thank my colleagues at the Austrian Research Institute for Artificial Intelligence, in particular Liz Garner for helping me with my English, Christian Holzbaur for providing DMCAI Clp, Paolo Petta for technological and moral support and Bernhard Pfahringer for fruitful discussions.

Finally, my greatest depth of gratitude is to my wife Liselotte and to my kids Stefanie, Teresa and Bernhard, who have borne my coming home late at night without complaining and who have provided the moral support and encouragement that I needed.

Contents

List of Figures

Chapter 1

Introduction

Computational methods for the processing of natural language are closely connected to the theories and formalisms of linguistics. Therefore, most natural language processing systems in the seventies and early eighties were based on phrase structure models, similar to the ones used for describing artificial formal languages (such as, e.g., programming languages). A wide-spread technique to implement context free phrase structure grammars is the use of push down automata, which are equivalent in expressivity and thus provide the adequate processing model. However, pure phrase structure rules are not sufficient to describe the variety of phenomena occurring in natural language.

As a consequence, augmentations to bare phrase structure rules are necessary. On the processing side simple push down automata are not capable to deal with these extensions. Thus augmented transition networks (ATNs, Woods (1970)) entered the stage. This technology was successful for quite a long time and was used in a number of implemented systems. ATNs offer a powerful process model and can be constructed from an annotated phrase structure grammar in a straightforward way. On the other hand, they provide only capabilities for processing language in one direction, i.e. parsing. A further disadvantage of ATNs is the inherently procedural way, in which the annotations of the grammar have to be implemented within the ATN network, i.e. by means of register operations. A declarative specification of a grammar is hardly ever obtainable, and maintainance or debugging of an ATN-encoded grammar is not an easy undertaking.

In order to provide a declarative framework for the specification of phrase structure rule annotations, and to organize the information cluttered in ATN registers, new formalisms based on unification of feature structures evolved. Since these new unification based formalisms are not implementable by ATNs in a reasonable way, new processing paradigms were developed. The predominant processing model for these unification grammars with a phrase structure backbone was a (mostly chart based) parser augmented by a feature unification component. A well-known system exhibiting that architecture is, e.g., the PATR system (Karttunen, 1986a).

This shift in the representational as well as in the computational means also has led to attempts to use the same grammar for parsing and for generation (Shieber, 1988). This desirable goal could, in principle, be achieved by replacing the parser by a generator and leaving unification component and grammar unchanged. However, in practical systems only approximations to this goal could be accomplished, since the way in which the grammar had to be encoded in the system most often included hidden procedural aspects which favored one particular direction (mostly parsing).

In parallel to the development of unification based formalisms a new view of grammar originating with GB theory (Chomsky, 1981) evolved. These theories aim at describing grammar (and language *competence*) by means of interacting principles instead of using a bunch of alternatively applying phrase structure rules. A formalism in this realm also employing feature structures to model linguistic structures is Head Driven Phrase Structure Grammar (HPSG, Pollard and Sag (1987), Pollard and Sag (1994)), which is the most popular theory in computational linguistics nowadays.

The abandoning of phrase structure rules, which served as the basis of all language processing systems so far, created some difficulties for practical implementations. Whereas well-known techniques for processing phrase structure grammars exist, no equally well-studied methods were available for principle based grammars.

In order to overcome these difficulties, some natural language processing systems simply neglect them and support principle based formalisms only, if they have been recast in a phrase structure style. Although this is possible in most cases, such an undertaking contradicts the motivations which have led to the development of principle based grammars.

A further challenge posed to natural language processing systems by these formalisms is the inventory of constructs employed by the constraint language, which is used to describe the grammar. HPSG, e.g., uses conditional and relational constraints over feature structures. These devices extend the capabilities of most existing feature unification engines. Thus many implementations have to relegate the treatment of such constraints to mechanisms outside the unification engine. A consequence of such an approach is the need to specify these constraints in a way necessarily different from the way feature constraints are specified, a fact which clearly does not enhance the perspicuity and declarativity of the grammar specification. Furthermore, having different modules for processing grammatical constraints immediately leads to the need for interfacing these modules with the usual associated problems. Most importantly, such an interface has to respect the operational characteristics of the interaction and thus cannot abstract away from procedural consideration. Thus the whole grammar implementation is rendered processing direction dependent, which is not what one may wish of a module designed to model language competence.

From the above discussion it is obvious, which requirements a formalism has to fulfill in order to be a useful basis for the implementation of a principle based grammar.

1. It should allow for a specification of the grammar as close as possible to its original formulation. This requirement includes that all constructs of the linguistic formalism

must have a corresponding counterpart on the implementation side. In particular it must not require a reformulation of the grammar in a rule-based style.

2. It must not rely on procedural considerations in order to be independent of the direction of processing, in other words, it must be fully declarative. This is the only way to achieve a clear distinction between grammar and processing model.

3. Nevertheless the formalism should provide an efficient basis for processing.

The outline of this dissertation is as follows: Firstly, a more detailed look at the linguistic motivations for unification grammars and principle based grammar formalisms will be taken and the characteristics of HPSG as a main representative of such a formalism will be reviewed. Then, a short overview of existing systems and their abilities to cope with principle based unification grammars will be presented and the requirements for dealing with such grammars in an adequate way will be outlined.

Chapter 4 describes the realization of a system able to deal with HPSG, implemented in a CLP-based framework. It will be shown, how a declarative grammar specification is made operational and serves as the only module responsible for ensuring well-formedness of linguistic structures in the sense of the grammar. This operationalization is achieved without recurring to procedural considerations and is completely independent of processing direction. The feature formalism implemented is powerful enough to represent all grammatical constraints in a declarative way and handle them without depending on a particular processing or even instantiation sequence. To guarantee well-formedness of feature structures, principles of grammar are viewed as licensing conditions on feature structures and are applied to every node at instantiation time. A delay mechanism allows principles to operate on uninstantiated feature structures without eagerly instantiating them. Thus every structure (maybe only partially) instantiated within the formalism can be regarded as well-formed in the sense, that it is only extensible in a way compliant with all principles of grammar, intrinsically carrying all constraints of grammar.

Applications which extend standard HPSG are described in chapter 5. These applications are linguistically well motivated but their integration into traditional processing models is not unproblematic. How this integration can be accomplished in the framework presented here demonstrates the flexibility of this approach to implementing a principle based unification grammar. The first application presented is an approach to integrate two-level morphology into a combined sentence level and word level grammar of German. Morphology, seen as a relation between lexical and surface string is integrated into the grammar as a relational constraint at the boundary between these two strata. The second application shows, how within a principled treatment of case assignment, argument reduction phenomena can be accounted for without lexical rules. The replacement of lexical rules which provoke nonmonotonic effects by a constructive mechanism within the word level grammar is an achievement, which enhances declarativity of the grammar.

Extensions to the basic system which are expected to further increase the efficiency and the expressive power of the feature unification machinery are described in chapter 6. Most of them already have been implemented. Compilation techniques reduce the need of interpretation at run-time, in particular, feature unification can be implemented as simple table lookup without sacrificing any of the properties of the system. The second extension deals with methods to represent disjunctive information in the system, since ambiguities are inherently present with natural language and disjunctions are the only means to express them. Various methods of disjunction handling which reduce the need for backtracking will be discussed and their integration into the system will be described.

Chapter 2

Linguistic Motivations

2.1 Unification Grammars

In the beginning of the eighties the focus of attention in computational linguistics shifted towards more declarative approaches (for a survey see also Haugeneder and Trost (1993)). These grammar formalisms have in common that they specify the annotations of phrase structure rules not procedurally (such as by means of ATN register operations), but in a declarative way. The combination of these annotations which is necessary to percolate the relevant information in the phrase structure tree is achieved via the operation of *unification*. The precise meaning "unification" depends on the kind of structures used to annotate the phrase structure rules.

One unification based approach, Definite Clause Grammar (DCG, Pereira and Warren (1980)) employing Prolog terms to annotate phrase structure rules utilizes the term unification and resolution engine built into Prolog. The other approach is the use of *feature structures* and *feature unification* as the respective operation. The first grammar formalism introducing unification of feature structures as a basic operation was Functional Unification Grammar (FUG, Kay (1979), Kay (1985)). Other linguistic theories employing phrase structure rules combined with unification of feature structures are Lexical Functional Grammar (LFG, Kaplan and Bresnan (1982)), Generalized Phrase Structure Grammar (GPSG, Gazdar et al. (1985)) and Categorial Unification Grammar (Karttunen, 1986b). A excellent introduction into the field of unification grammar is given in Shieber (1986).

The "unification view" of grammar, emphasizing the aspect of unification as the primary operation on feature structures suggests that processing of language can be seen simply as a information gathering process, which is an attractive option. However, it soon was complemented by a more logically oriented view, since various aspects of modelling language, especially disjunctive and negative descriptions, could not easily be captured within a framework employing only conjunctively specified feature *structures*. The notion of disjunctive feature structures (as, e.g., proposed by Karttunen (1984)) proved to be problematic. Thus, instead of extending the notion of a feature structure, logically oriented theories of feature *descriptions*, which can be thought of to denote sets of feature structures, have been developed (e.g. Kasper and Rounds (1986), Kasper (1987), Johnson (1988)), as well as algebraic approaches (e.g. Aït-Kaci and Nasr (1986)) and various feature logics (e.g. Smolka (1988)). However, we will first sketch how feature structures can be used to encode linguistic information and postpone the discussion of logical extensions.

2.1.1 Features and Values

The notion of a *feature* has been around in linguistics for a long of time. Features are used to ascribe properties to linguistic entities. Kay's Functional Unification Grammar (Kay, 1979) was the first attempt to clarify this notion and give a computational realization to it. The need to employ features to characterize phrases of natural language arises from the inability of the labels of context-free grammars to offer the rich category system needed to describe certain constructs common in natural languages. One such example is the subject-verb agreement in languages such as English or German, where the *number* feature (with the possible values *singular* and *plural*) and the *person* feature (with the possible values *first, second* and *third*) of both the subject and the verb of a sentence must exhibit the same value.

Feature structures are usually notated using so-called Attribute-Value Matrices (AVMs). A feature structure representing a third person singular noun phrase could then (according to some linguistic theory) be represented by the AVM shown in Fig. 2.1 below. The

$$\begin{bmatrix} \textsc{cat} & N \\ \textsc{bar} & \mathit{2} \\ \textsc{agr} & \begin{bmatrix} \textsc{number} & \mathit{singular} \\ \textsc{person} & \mathit{third} \end{bmatrix} \end{bmatrix}$$

Figure 2.1: AVM for a Third Person Singular Noun Phrase

CATegory feature of this structure is N for noun, the BAR level is *2*, and the AGReement feature has a complex value, which is a feature structure itself, with NUMBER = *singular* and PERSON = *third*.

An alternative way to think of feature structures is by representing them as directed graphs. The labelled edges of such a graph correspond to the features and its leaves to atomic values. Thus another way to depict the feature structure of Fig. 2.1 is the Feature Graph shown in Fig. 2.2. Most commonly, feature structures are viewed as partial descriptions of linguistic objects. That means, a description may be made more informative by adding more feature-value pairs to it. This modularity of description is an advantage over other encodings found, e.g., in DCGs, where this encoding is based on fixed arity terms. But,

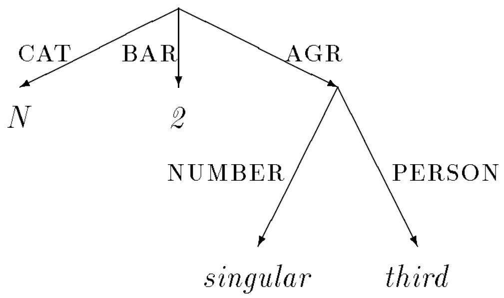

Figure 2.2: Feature Graph for a Third Person Singular Noun Phrase

as for Prolog terms, also for feature structures unification serves as a basic operation, to which we will turn below.

2.1.2 Structure Sharing

A further important concept in unification based grammar formalisms is the notion of structure sharing. Recall the above mentioned subject-verb agreement, requiring that subject and verb must exhibit the same values for their NUMBER and PERSON features. The natural way to enforce this constraint within a feature structure is to require the *identity* of these values. Thus, structure sharing simply means, that features in different parts of a feature structure share their value. The tree shaped feature graphs such as the one in Fig. 2.2 now possibly become *reentrant* graphs, so called directed acyclic graphs (DAGs). How the agreement constraint is stated using reentrancy is shown in Fig. 2.3, both in AVM and graph notation. In AVM notation structure sharing is indicated by boxed integers.

$$\begin{bmatrix} \text{SUBJ} & [\text{AGR } \boxed{1}\ ] \\ \text{PRED} & [\text{VERB } [\text{AGR } \boxed{1}\ ]] \end{bmatrix}$$

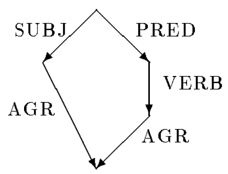

Figure 2.3: Subject-Verb Agreement

2.1.3 Feature Unification

Since feature structures are seen as partial descriptions, they can be ordered in a natural way by the information content they convey. The empty feature structure, containing no features at all, is the most general one and is written $\top$. The *Subsumption* relation $\sqsubseteq$ is defined in way that a feature structure D subsumes a feature structure D' (written $D \sqsubseteq D'$) if D contains a subset of the information contained in D'.

More formally, let $dom(D)$ denote the set of all features defined at the root of D and let $D(p)$ be the value obtained by following the path p in D.[1] Then $D \sqsubseteq D'$ iff $D(l) \sqsubseteq D'(l)$ for all $l \in dom(D)$ and $D'(p) = D'(q)$ for all paths p and q such that $D(p) = D(q)$.

To complete the lattice structure we add the bottom element $\perp$ representing inconsistent information, which is subsumed by every consistent feature structure. Then unification of two feature structures D_1 and D_2 can be defined as the most general feature structure D which is subsumed be both D_1 and D_2. Formally, $D_1 \sqcup D_2 = D$ iff $D_1 \sqsubseteq D$ and $D_2 \sqsubseteq D$ and there is no feature structure D_3 such that $D_1 \sqsubseteq D_3$ and $D_2 \sqsubseteq D_3$ and $D_3 \sqsubseteq D$.

An example, how unification of two feature structures is performed is shown in Fig.2.4.

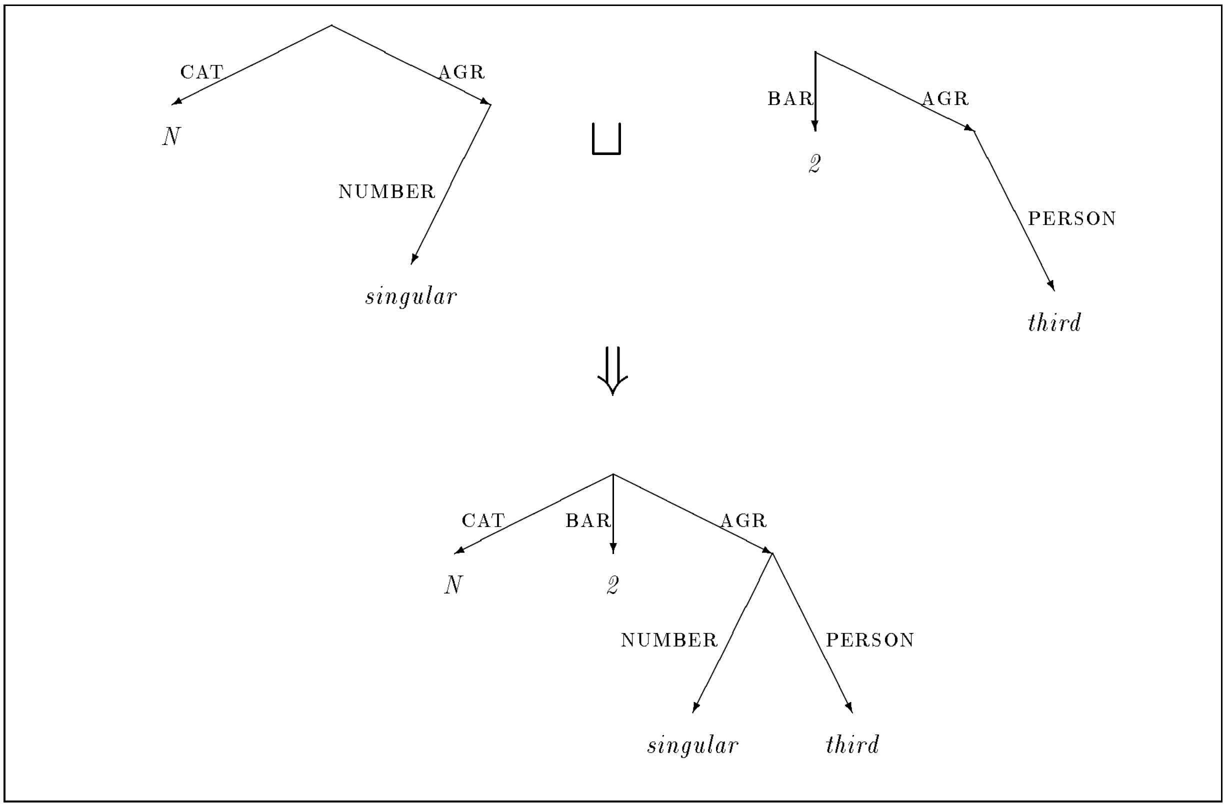

Figure 2.4: Feature Graph Unification

[1]e.g., if D is the structure shown in Fig. 2.2, then $D(agr \mid number)$ is the value *singular*).

2.2 Rule-based vs. Principle-based Formalisms

Besides the trend to use feature structures to represent linguistic information a second, independent tendency towards complementing or even replacing phrase structure rules by principles acting as global well-formedness conditions could be observed. The first linguistic theory that strictly replaced rules by a small set of interacting principles was Government and Binding (GB) Theory (Chomsky (1981),Chomsky (1986)), often also called the "Principles and Parameters" approach to language.

One main advantage of a principle based approach is the compactness of the representation of linguistic knowledge which can be achieved. To obtain the effect of k principles with n_j degrees of freedom, which can be stated in $O(n_1 + n_2 \ldots + n_k)$ lines of code, in a rule based approach one has to use $O(n_1 \times n_2 \ldots \times n_k)$ rules (cf. Berwick (1991)).

However, GB theory does not make use of feature structures (at least not explicit) and is, in general, more a theory of language competence and not of language performance. Thus it does not lend itself very well to practical implementations, at least not directly. Principle based parsers built upon GB theory all employ covering phrase structure grammars and use the principles for filtering (Berwick, 1991).

Nevertheless, the attractive features of principle-based approaches have been recognized also by other, unification based, formalisms. GPSG (Gazdar et al., 1985) augments the phrase structure backbone by various principles (e.g. the Head Feature Principle). HPSG (Pollard and Sag, 1987) abandons phrase structure rules at all and relies solely on well-formedness constraints given by principles.

Besides elegance and compactness of representation, a further reason promoting the shift towards principle based approaches has been the trend in linguistics, to associate part of the phrase structure information with the lexical entries themselves. The reasons for adopting *lexical subcategorization* in practically all modern linguistic formalisms lie in the fact, that every grammar has to account for the ungrammaticality of sentences like "*Ich trete dir zu schweigen", which are correct in other wordings (just substitute "befehle" for "trete"). An approach, which regards such strings as syntactically correct and relegates deciding on grammaticality to semantic filtering is also bound to fail, since there exist lots of examples, where semantically equivalent words exhibit different syntactic argument structure (see, e.g. Gazdar et al. (1985, p32)). Therefore, phrase structure is most appropriately defined by the lexical head of a phrase, i.e. the words in the lexicon explicitly specify the syntactic and semantic restrictions on their arguments.

The encoding of phrase structure in the lexicon leaves little to do for general phrase structure rules. On the other hand, there need to be mechanisms that put the lexical subcategorization encoding to work. It will be sketched below (using the approach and the notation of HPSG), how phrase structure can be generalized and how the restrictions on phrase structure resulting from the lexicon can be defined in a principled way, before the HPSG formalism is discussed in more detail in Section 2.3.

2.2.1 Generalizing Phrase Structure

As an example how phrase structure rules can be generalized consider the phrases in Fig. 2.5 together with the corresponding phrase structure rules. Although these phrases have

VP	$\rightarrow$ NP V	[eine Förderung]$_{NP_{acc}}$ [beantragen]$_V$
PP	$\rightarrow$ P NP	[wegen]$_P$ [der Förderung]$_{NP_{gen}}$
AP	$\rightarrow$ PP A	[für Förderungen]$_{PP_{für}}$ [zuständige]$_A$
		...
HeadP	$\rightarrow$ {Head, Comps}	

Figure 2.5: Generalizing Phrase Structure Rules

different syntactic categories, there are a lot of similarities to be observed. Every phrase contains one element, which determines the category of the phrase, and is therefore called the *head* of the phrase. Secondly, the other element contained in the phrase can be regarded as argument of the head element. It is syntactically and semantically constrained by the head and called the *complement*. Thus, a generalization of these three phrase structure rules is the immediate dominance rule, that a headed phrase dominates a head and a complement. The ordering information given by the phrase structure rules is factored out and has to be given independently in the form of linear precedence rules. These could be stated, e.g., in a principled way if one adopts the widely accepted decomposition of categories given in (Chomsky, 1970) shown in Fig. 2.6. Then it could be stated, that

	[+ N]	[- N]
[+ V]	**A**	**V**
[- V]	**N**	**P**

Figure 2.6: Decomposition of Syntactic Categories

heads marked [+ V] are right-peripheral in all their projections, whereas heads marked [- V] are left-peripheral. This generalization holds for German and exhibits nicely, how a lot of rules can be replaced by the interaction of simpler, more general constraints.

So the first step in generalizing phrase structure has been made and the different phrase structure trees can all be coerced into one single structure, as can be seen in 2.7.

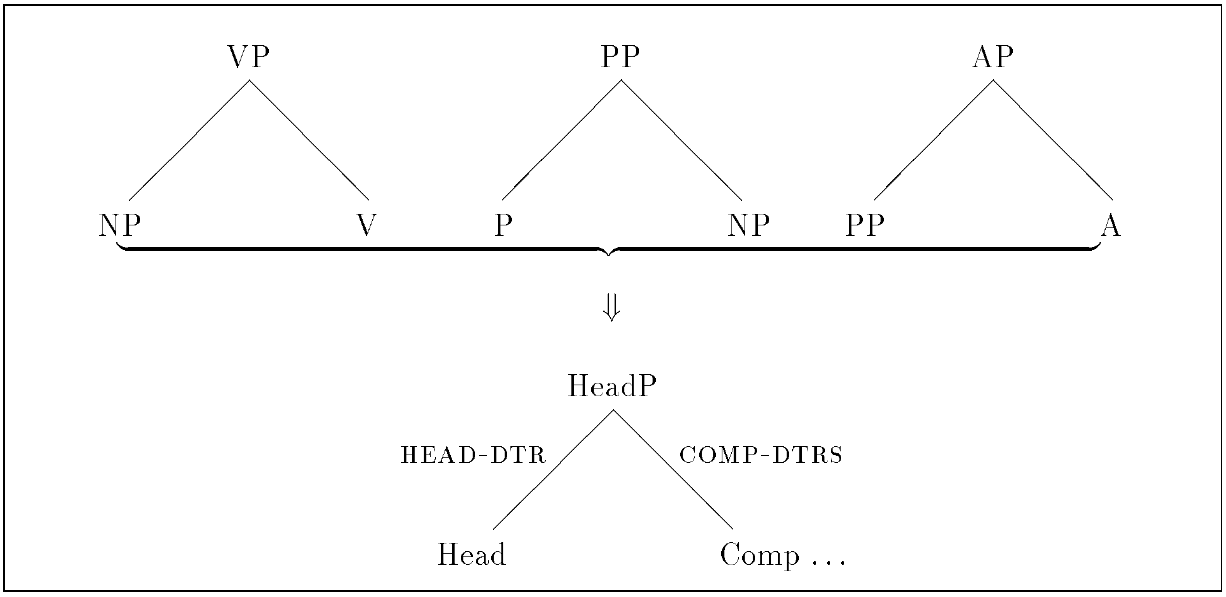

Figure 2.7: Generalizing Phrase Structure Trees

However, to make use of these generalization within a unification based formalism, one must be able to represent phrase structure trees like the one above by means of feature structures. The way this is done in HPSG is depicted in Fig. 2.8.

$$\left[\begin{array}{ll} \textsc{phon} & \textit{phon} \\ \textsc{synsem} & \textit{synsem} \\ \textsc{dtrs} & \left[\begin{array}{ll} \textsc{head-dtr} & \textit{sign} \\ \textsc{comp-dtrs} & \langle \ \ldots \textit{sign} \ldots \rangle \end{array}\right] \end{array}\right]_{\textit{phrase}}$$

Figure 2.8: AVM of a General Phrase

The structure with the type *phrase* has three attributes, PHON for the phonological (or graphemic) string of the phrase, SYNSEM containing the syntactic and semantic attributes and DTRS for the encoding of phrase structure, consisting of the HEAD-DTR and the list of COMP-DTRS[2]. Thus a feature structure representation of phrase structure can be accomplished, but further constraints are needed to ensure that this encoding really works, as it is intuitively expected to do.

[2]The angle bracket notation is an abbreviation for the feature structure representation of nonempty lists (cf. Shieber (1986)):

$$\left[\begin{array}{ll} \textsc{first} & \top \\ \textsc{rest} & \textit{list} \end{array}\right]_{\textit{nelist}}$$

2.2.2 Putting Principles to Work

Intuitively, a feature structure of type *phrase* should have (among others) the following properties, as far as phrase structure information is concerned.

- A feature structure representing a phrase should always exhibit the shape shown in Fig. 2.8 above, that means, no other features should occur in such a structure, and no features must be left out.
- The values for the head daughter and for the complement daughters (DTRS|HEAD-DTR and all elements of the DTRS|COMP-DTRS list) must either be phrases themselves or lexical entries (i.e. *words*).
- Since the shift of phrase structure information to the lexicon has to be accounted for, the lexical entry of the head of the phrase has to provide this information in a suitable form.
- The subcategorization requirements of the lexical head have to be reflected in the structure of the phrase, that means:
  - a correspondence between the subcategorization list of the head and the list of complement daughters has to be established and
  - provisions have to be made, that each argument of the head gets filled exactly once.

To fulfill the structural requirements, HPSG utilizes a type hierarchy along with type definitions, which are inherited along the type hierarchy. The definition of the *sign* type, subsuming *word* and *phrase* is shown in Fig. 2.9. The features specified higher up in the lattice are inherited by all subtypes. Also the type of the value a feature may take has to be specified (and may be further restricted by subtypes inheriting that feature).

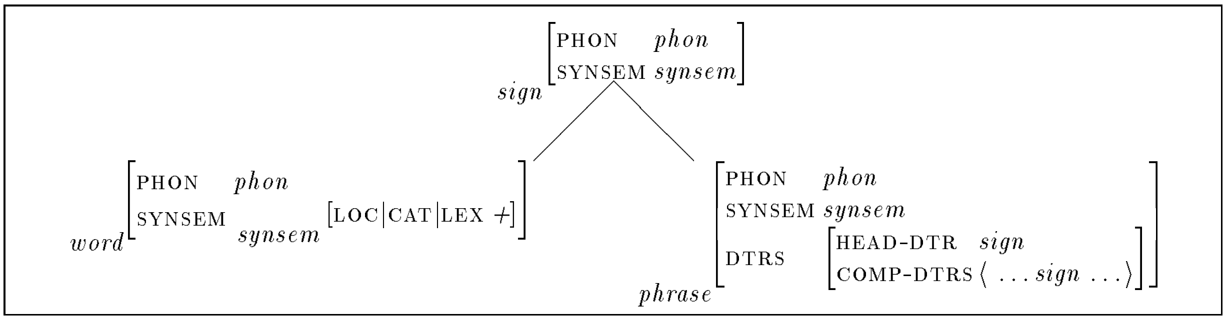

Figure 2.9: Type Definition

To account for the encoding of argument structure in the lexicon, the SUBCATegorization list is provided within the CATegory feature complex. The SUBCAT list contains the syntactic-semantic (*synsem*) values of the head's arguments (for a detailed discussion of the motivations underlying this particular structuring see Pollard and Sag (1994)). The fact, that the German verb "beantragt" subcategorizes for two arguments, one a nominative, the other an accusative noun phrase, is encoded in the SUBCAT list as shown in Fig. 2.10 (only the syntactically relevant features are shown).

$$
\left[\begin{array}{ll}
\text{PHON} & \textit{"beantragt"} \\
\text{SYNSEM}|\text{LOC}|\text{CAT} & \left[\begin{array}{ll}
\text{HEAD} & \left[\begin{array}{ll} \text{NUM} & \textit{sing} \\ \text{PERS} & \textit{3} \end{array}\right]_{\textit{verb}} \\
\text{SUBCAT} & \langle \text{NP}[\textit{nom}], \text{NP}[\textit{acc}] \rangle
\end{array}\right]
\end{array}\right]_{\textit{word}}
$$

where NP[*nom*] and NP[*acc*] are abbreviations for feature structures

$$
\text{NP}[\textit{nom}] = \left[\begin{array}{ll}
\text{LOC}|\text{CAT} & \left[\begin{array}{ll}
\text{HEAD} & \left[\text{CASE } \textit{nom}\right]_{\textit{noun}} \\
\text{SUBCAT} & \langle\rangle
\end{array}\right]
\end{array}\right]_{\textit{synsem}}
$$

Figure 2.10: Subcategorization Encoding in Lexical Entries

In order to account for a principled, lexicon driven treatment of phrase structure in a unification based formalism, one thing remains to be done: a proper correspondence between the subcategorization list of the lexical head and the list of complement daughters of its projection has to be ensured. In HPSG this correspondence is established via the *Subcategorization Principle*, which states, that

> in a headed phrase [...] the SUBCAT value of the head daughter is the concatenation of the phrase's SUBCAT list with the list [...] of SYNSEM values of the complement daughters.

This principle ensures, that

1. in a saturated sign (i.e. a phrase with all argument positions filled) *every argument* that the head subcategorizes for *is realized* and
2. appears *exactly once* in the tree shaped feature structure representing the phrase.

The Subcategorization Principle, as it is stated above, establishes a well-formedness condition on feature structures. It applies only to certain structures (i.e. phrases of a certain type), thus it must be formulated conditionally. Furthermore, since it involves relational dependencies between certain subparts of the constrained feature structures, it clearly goes beyond the capabilities of feature structures as they have been discussed so far. How it can be stated formally and how such a specification can be implemented will be discussed in the following sections.

2.3 HPSG — A Principled Linguistic Theory

Head Driven Phrase Structure Grammar (Pollard and Sag, 1987; Pollard and Sag, 1994) is a unification based grammar formalism modeling linguistic objects by means of typed feature structures. It differs from other unification based theories (such as LFG, cf. Kaplan and Bresnan (1982)) in that it does not use feature structures as an additional device augmenting traditional phrase structure rules. Instead, HPSG uses (universal and language-specific) *Principles* to constrain the feature structure admissible for a particular grammar. This principle based approach to grammar is something HPSG has in common with Government and Binding Theory. Language is described by the interaction of parameterized principles of grammatical well-formedness, and highly specified lexical entries. However, HPSG differs in various important aspects from GB. The most prominent difference is that HPSG does not rely on configurational notions defined in terms of tree structure and explicitly denies derivational notions such as movement. Unbounded dependency phenomena are handled using *structure sharing* instead.

The theory has undergone some revisions and extensions since Pollard and Sag (1987). The exposition below will follow the version of HPSG set up in Pollard and Sag (1994).

2.3.1 Linguistic Aspects

In HPSG, the fundamental objects of linguistic analysis are signs modeled by typed feature structures, which in turn are represented as attribute-value matrices. The basic attributes for signs include PHON for phonological information and SYNSEM for syntactic and semantic information. SYNSEM in turn is highly structured including LOCal and NONLOCal values. LOCal features comprise CONTent containing semantic information, and the CATegory complex including the HEAD features and the SUBCAT list to model subcategorization information NONLOC features are used to model nonlocal dependency constructions like topicalization, questions and relative clauses.

As an example how a lexical sign (a word) is represented in HPSG we show the (somewhat simplified) attribute value matrix of the verb *walks* in Fig. 2.11.[3]

In the attribute-value matrices here the angle bracket notation is used to indicate lists. This notational convention abbreviates the actual representation for lists. In fact, nonempty lists are represented as feature structures with attributes FIRST containing the head of the list and REST containing the tail of the list (cf. Shieber (1986)). The *type* of a structure is indicated at the bottom of the opening square bracket (some types being omitted for space reasons). Structure sharing is indicated by boxed integers.

walks subcategorizes for a nominative NP, the subject, which is encoded by the single member of the SUBCAT list. This member has to be a saturated sign, i.e. a sign whose

[3]The example is taken from Pollard and Sag (1994)

$$
{}_{\mathit{word}}\begin{bmatrix}
\textsc{phon} & \langle \mathit{walks} \rangle \\
\textsc{synsem} & {}_{\mathit{synsem}}\begin{bmatrix}
\textsc{loc} & \begin{bmatrix}
\textsc{cat} & \begin{bmatrix}
\textsc{head} & {}_{\mathit{verb}}\begin{bmatrix} \textsc{vform} & \mathit{fin} \\ \textsc{tense} & \mathit{pres} \end{bmatrix} \\
\textsc{subcat} & \left\langle {}_{\mathit{synsem}}\begin{bmatrix}
\textsc{loc} & \begin{bmatrix}
\textsc{cat} & \begin{bmatrix}
\textsc{head} & {}_{\mathit{noun}}\begin{bmatrix} \textsc{case} & \mathit{nom} \\ \textsc{person} & \mathit{3rd} \end{bmatrix} \\
\textsc{subcat} & \langle\rangle
\end{bmatrix} \\
\textsc{cont} & {}_{\mathit{nom\_obj}}\begin{bmatrix} \textsc{index} & \boxed{1} \end{bmatrix}
\end{bmatrix}
\end{bmatrix} \right\rangle
\end{bmatrix} \\
\textsc{cont} & {}_{\mathit{psoa}}\begin{bmatrix} \textsc{rel} & \mathit{walk} \\ \textsc{walker} & \boxed{1} \end{bmatrix}
\end{bmatrix} \\
\textsc{nonloc} & \mathit{nonloc}
\end{bmatrix}
\end{bmatrix}
$$

Figure 2.11: Lexical Entry for *"walks"*

SUBCAT value is the empty list, with its HEAD feature of type *noun*). The semantic index of the subject fills the WALKER role of the RELation *walk* in the semantic CONTents.

Phrases, also being subtypes of *sign*, inherit the features described above and additionally introduce a daughters (DTRS) attribute taking values of type *constituent-structure*, one of its subtypes—*head-complement-structure*—will be considered in more detail. Via the DTRS attribute the type *sign* becomes recursive in that some values inside this attribute are constrained to be *signs* themselves. This is the way phrase structure is represented in HPSG. What kind of phrases are admissible is stated in terms of principles constraining the feature structures of type *phrase*.

The kind of phrases built up by saturating the valences or argument positions of a head has a DTRS attribute of type *head-complement-structure* with two slots: HEAD-DTR, containing the sign being the head of the phrase, and COMP-DTRS, a list of signs being the arguments. What head-complement phrases are grammatical is determined by the head, specifying what arguments are admissible, interacting with the general principles (including the Immediate Dominance Principle, which serves the purpose phrase structure rules have been designed for, but which is stated just the same way as the other principles of grammar).

To sketch how principles and lexical entries interact in HPSG, we will describe two essential principles, the Head Feature Principle and the Subcategorization Principle and show the derivation of a simple sentence using these two principles and the lexical entry above.

The **Head Feature Principle** states that in every headed phrase the HEAD value of the phrase is token identical with the HEAD value of the head daughter. In Pollard and Sag (1994) this principle is stated as follows:

Head Feature Principle The HEAD value of any headed phrase is structure-shared with the HEAD value of the head daughter.

This principle on the admissibility of a certain subtype of phrase (a headed phrase) was formulated as an implicational constraint in Pollard and Sag (1987):

$$\left[\text{DTRS}\ \textit{head-struc}\right]_{\textit{phrase}} \Longrightarrow \begin{bmatrix} \text{SYNSEM|LOC|CAT|HEAD} & \boxed{1} \\ \text{DTRS|HEAD-DTR|SYNSEM|LOC|CAT|HEAD} & \boxed{1} \end{bmatrix}_{\textit{phrase}}$$

Figure 2.12: Head Feature Principle

The **Subcategorization Principle** ensures that exactly those arguments the head subcategorizes for appear as complement daughters.

Subcategorization Principle In a headed phrase (i.e. a phrasal sign whose DTRS value is of sort *head-struc*), the SUBCAT value of the head daughter is the concatenation of the phrase's SUBCAT list with the list (in order of increasing obliqueness) of SYNSEM values of the complement daughters.

$$\left[\text{DTRS}\ \textit{head-struc}\right]_{\textit{phrase}} \Longrightarrow$$
$$\begin{bmatrix} \text{SYNSEM} & \left[\text{LOC|CAT|SUBCAT}\ \boxed{1}\right] \\ \text{DTRS} & \left[\text{HEAD-DTR}\ \left[\text{SYNSEM|LOC|CAT|SUBCAT}\ \textbf{append(}\boxed{1}\textbf{, synsem-values(}\boxed{2}\textbf{))}\right]\right] \\ \text{COMP-DTRS} & \boxed{2} \end{bmatrix}_{\textit{phrase}}$$

Figure 2.13: Subcategorization Principle

This principle exhibits complex functional constraints (list-append and an operator collecting the values reachable along a path from a list of feature structures), which extend the usual feature structure unification formalisms, but must also be dealt with when implementing HPSG.

Now the effects of the interaction of the principles described so far with lexical entries can be seen by inspecting the (simplified, ignoring content) feature structure in Fig. 2.14 representing the phrase *Kim walks*:

The coindexing tag $\boxed{1}$ results from the Head Feature Principle, the empty SUBCAT list on the phrase level and the coindexing of the first element of the SUBCAT list of the head daughter and the SYNSEM value of the complement daughter is accomplished by the Subcat Principle.

$$
{}_{phrase}\begin{bmatrix}
\text{PHON} & \langle \textit{Kim, walks} \rangle \\
\text{SYNSEM} & \begin{bmatrix} \text{LOC}|\text{CAT} & \begin{bmatrix} \text{HEAD} & \boxed{1}\ {}_{verb}\begin{bmatrix}\text{VFORM} & \textit{fin}\\ \text{TENSE} & \textit{pres}\end{bmatrix} \\ \text{SUBCAT} & \langle\rangle \end{bmatrix}\end{bmatrix} \\
\text{DTRS} & \begin{bmatrix} \text{HEAD-DTR} & {}_{word}\begin{bmatrix} \text{PHON} & \langle \textit{walks}\rangle \\ \text{SYNSEM} & \begin{bmatrix}\text{LOC}|\text{CAT} & \begin{bmatrix}\text{HEAD} & \boxed{1}\\ \text{SUBCAT} & \langle \boxed{2} \rangle\end{bmatrix}\end{bmatrix}\end{bmatrix} \\ \text{COMP-DTRS} & \left\langle {}_{word}\begin{bmatrix}\text{PHON} & \langle\textit{Kim}\rangle \\ \text{SYNSEM}\ \boxed{2} & \begin{bmatrix}\text{LOC}|\text{CAT} & \begin{bmatrix}\text{HEAD} & {}_{noun}\begin{bmatrix}\text{CASE} & \textit{nom}\\ \text{PERSON} & \textit{3rd}\end{bmatrix}\\ \text{SUBCAT} & \langle\rangle\end{bmatrix}\end{bmatrix}\end{bmatrix}\right\rangle \end{bmatrix}
\end{bmatrix}
$$

Figure 2.14: Attribute Value Matrix for *Kim walks*

2.3.2 Formal Aspects

As has become apparent from the discussion above, there are formal requirements associated with the feature structures representing linguistic objects. First of all, feature structures are *strongly typed*, that means, every feature structure must be associated with a type. Secondly, every type restricts its associated feature structure in that only certain features are allowed and the values of these features must have a certain type. Finally, these appropriateness and value restrictions are inherited along the type hierarchy thus leading to an inheritance-based polymorphism which is also well-known in object oriented programming.

The advantages of using typed instead of plain feature structures are obvious. Unification failure can often be detected early inspecting only the types, without creating expensive copies during structural unification. Furthermore, types in many cases allow a more natural and efficient encoding of linguistic information as it would be the case with untyped structures.

The formal characteristics of the feature structures used in HPSG have been sketched in (Pollard and Sag, 1994) and formally defined in (Carpenter, Pollard, and Franz, 1991) and (Carpenter, 1992b). They will be briefly reviewed here for ease of reference.

Types and Feature Structures

A **Type Schema** is defined as a finite consistently complete partial order of the *types* T by subsumption together with a finite set of features F and a partial function $Approp$: $F \times T \longrightarrow T$ such that

1. for every feature f there is a most general type σ such that $Approp(f, \sigma)$ is defined and
2. if $Approp(f, \sigma)$ is defined and σ subsumes τ then f is also appropriate for τ and $Approp(f, \sigma)$ subsumes $Approp(f, \tau)$.

The practical importance of this definition is, that it allows to specify which slots are permissible for each type and to restrict the type of their values. Features must be uniquely introduced and are inherited by the subtypes of the introducing type. Inherited features may be further restricted, but only monotonously.

A **Feature Structure** is defined in the usual way as a rooted, connected, directed graph with vertices labeled by types and arcs labeled by features. A feature structure F **subsumes** a feature structure F', written $F \sqsubseteq F'$, if there exists a total mapping h from the nodes of F onto the nodes of F' such that the root of F is mapped onto the root of F', for every node $q \in F$ the type of q subsumes the type of $h(q)$, and if the edge $q_1 \xrightarrow{f} q_2$ is in F then the edge $h(q_1) \xrightarrow{f} h(q_2)$ is in F'.

A **Unifier** for a pair of feature structures F and F' is any feature structure F'' such that $F \sqsubseteq G$ and $F' \sqsubseteq G$ iff $F'' \sqsubseteq G$. Unique unifiers exist for pairs of consistent feature structures up to alphabetic variance (see Carpenter, Pollard, and Franz (1991)).

Well-typed Unification

A notion which distinguishes the feature structures proposed for HPSG from other formalizations of typed feature structures (e.g. Smolka (1988)) is the notion of *Well-Typing*. A feature structure is **well-typed**, if every feature that appears in it is appropriate and takes an appropriate value (according to the definition of appropriateness above). A feature structure F is said to be *typeable* if there is a feature structure F' such that $F \sqsubseteq F'$ and F' is well-typed.

HPSG requires feature structures to be well-typed, so there is a need to compute well-typed feature structures from arbitrary ones (excluding those which are not typeable). This **Type Inference** procedure (*TypeInf*) can be defined, relying on the unique introduction of features and the monotonicity properties of *Approp*. The steps to be performed iteratively until a closure is reached are to

- unify the type of a node with the most general type appropriate for one of the features present at that node, and to
- unify the value of a feature with the appropriate type restriction.

An important property of *TypeInf* is the monotonous behavior with respect to unification. Unification and Type Inference can be arbitrary interleaved due to the fact that

$$TypeInf(TypeInf(F_1) \sqcup TypeInf(F_2)) = TypeInf(F_1 \sqcup F_2)$$

Thus **Well-Typed Unification** of a pair of well-typed feature structures F_1 and F_2 can be shown to be $TypeInf(F_1 \sqcup F_2)$ (see Carpenter, Pollard, and Franz (1991)).

Principles of Grammar as Constraints on Feature Structures

What form the principles of a grammar take is crucial for their implementation. (Pollard and Sag, 1987) allow general implicative and negative constraints in the form of conditional feature structures, in (Pollard and Sag, 1994) principles are given only in verbal form. Recent work on formalizing the basis of HPSG models them as constraints attached to types (Pollard and Moshier, 1990; Carpenter, Pollard, and Franz, 1991).

Constraints take the form of feature logical formulas which must be satisfied by the feature structure they are applied to. The well-formed formulas of that feature logic in the style of (Rounds and Kaspar, 1986) as defined in (Carpenter, Pollard, and Franz, 1991) along with their satisfaction conditions are given in Fig. 2.15 below:

$F \models \sigma$	if the root node of F is assigned a type at least as specific as σ
$F \models \pi : \phi$	if the value of F at path π is defined and satisfies ϕ
$F \models \pi \doteq \pi'$	if paths π and π' lead to the same node in F
$F \models \phi \wedge \psi$	if $F \models \phi$ and $F \models \psi$
$F \models \phi \vee \psi$	if $F \models \phi$ or $F \models \psi$

Figure 2.15: Satisfaction Conditions for Feature Logic Formulas

Minimal satisfiers exist for all feature logic formulas ϕ (see Carpenter, Pollard, and Franz (1991)).

A **Constraint System** associates each type with a feature logical formula, whereby each type inherits all the constraints imposed on its supertypes. The **Solution** of a formula ψ with respect to a constraint system Φ is a feature structure F satisfying ψ and each substructure of F satisfies all the constraints of Φ inherited by the type of its root node.

Every **Minimal Solution** of ψ can be obtained by applying a rewriting process to a minimal well-typed satisfier of ψ. **Rewriting** is done by arbitrarily choosing a node within the feature structure, unifying that node with a nondeterministically chosen minimal satisfier of the constraints inherited by the type of that node, and well-typing the resulting feature structure by an interleaved type inferencing step. (for a proof of the Solution Theorem see Carpenter, Pollard, and Franz (1991)).

The significance of this result is, that from arbitrary feature logic formulas (which may be seen as partially instantiated feature structures) all fully instantiated, well-typed feature-structures obeying the principles of grammar can be found by a constructive rewriting process. That means, if one specifies only the PHON value of a *sign*, this rewriting produces a **parse** for that string, or, if only the semantic CONTent is specified, rewriting behaves like a **generator**. However, a breadth first enumeration of the rewriting search space is an inefficient method to obtain every solution. Fortunately, the operations of unification, type inference and rewriting are order-independent and may be arbitrarily interleaved. This fact allows for including the constraint rewriting operation into a lazy instantiation scheme, facilitating the implementation of other search strategies.

Chapter 3

Processing Aspects and Desiderata

Having motivated the shift from rule-based to principle-based formalisms and having described the formal aspects of feature formalisms, in particular HPSG, it now has to be investigated, how theories using feature structures constrained by principles can be put to work in order to parse or to generate natural language. Pollard and Sag (1987) state, that the whole range of grammatical utterances are described by the conjunction of all universal and language specific principles with the disjunction of all lexical entries. While this is clearly an elegant and very general view of how the grammar works, as a process model this perspective is too unspecific to overcome the practical problems of efficient processing.

The most promising view of how to process language in a principle-based feature formalism seems to be a deductive one. Starting out with a under-specified feature structure (only the PHON string being given in the case of parsing, only semantic and contextual features being specified in the case of generation) processing could be done by simply extending the feature structure according to the principles of grammar until a fully specified solution is obtained. However, the problem of the search space remains and techniques for intelligent traversal of the search space are still required.

Approaches within the framework of *deductive parsing* originating with Pereira and Warren (1983) and generalized in later work such as Shieber, Schabes, and Pereira (1994) do not really tackle the problem sketched above. They all rely on a phrase structure grammar which is processed using chart parsing techniques. This is of course a viable approach if one does not care too much of the original formulation of the grammar, since also in HPSG a certain amount of constraints can be recast in the form of phrase structure rules.

Similarly, approaches to *principle-based parsing* in the realm of GB, such as Fong (1991) or Johnson (1991) use principles of grammar solely as filters and therefore have to build up structure by other means. All principle-based parsers in Berwick, Abney, and Tenny (1991) thus employ a covering phrase structure grammar and use the principles to rule out invalid derivations.

A further aspect when considering the approaches mentioned above is their bias towards one particular processing direction, i.e. parsing. Although also attempts have been made to develop uniform architectures for parsing and generation employing the same grammar (such as, e.g. Shieber (1988)), these architectures mostly rely also on phrase structure rules and the associated processing techniques.

3.1 Existing Approaches

The following section gives an overview of the characteristics of existing systems suitable for implementing unification and/or principle based grammars. It will be indicated, which grammar formalism they are intended for and to what extent the special requirements of HPSG are being accounted for. There are specialized systems tailored for a particular grammar formalism as well as general workbenches intended to be usable for a range of different grammar formalisms, each design philosophy having its merits.

3.1.1 Early Unification-Based Grammar Environments

PATR

The first workbench for implementing unification grammars has been the PATR system (Karttunen, 1986a). PATR is not tailored for one specific grammar formalism but intended to be used as programming language for grammars. The system employs a chart parser to process the phrase structure backbone of the grammar. The phrase structure rules can be annotated with feature structures specifying how the resulting structure of a phrase has to be built out of the daughters, and to force structure sharing constraints. There is no type system, but a template mechanism is provided, which serves as a kind of macro utility, facilitating the specification of feature structures in the grammar.

3.1.2 Recent Developments

ALE

ALE ("Attribute Logic Engine", Carpenter (1992a) is a attribute logic formalism written in Prolog based on strongly typed feature structures with inheritance. These structures can be manipulated by means of a definite logic programming language built into ALE, which is closely related to both Prolog and LOGIN (Aït-Kaci and Nasr, 1986). Principles of grammar have to be stated using this constraint language, being processed as in Prolog depth-first, left to right. The principles related to phrase structure have to be given a special rule status (such as in HPSG-PL) to make them usable for the built-in chart parser. Application of the other grammar principles is specified by these rules and triggered by their invocation.

Although ALE can be used for implementing other grammar formalism such as, e.g., categorial grammar, it seems to be primarily designed for handling HPSG, especially regarding the type system and the appropriateness constraints.

ALEP

The ALEP system, originating from the EUROTRA ET6/1 project (Pulman, 1991) aims at supporting a very broad range of grammar formalisms and at offering multi-linguality as well as multi-directionality. Furthermore it provides a framework for machine translation systems. Therefore lots of ways to specify a grammar are provided. Features and categories are typed, categories being special types, which are needed by the phrase structure part of the system, since also ALEP relies on the use of phrase structure rules.

CUF

CUF (Dörre and Eisele, 1991; Dörre and Dorna, 1993) offers a type system, whose expressivity is similar to the one provided by ALE. There are no phrase structure rules, processing is done by expanding feature structures. Principles of grammar and other constraints are stated via a definite clause component. Expansion of feature structures is performed by applying the goals of the definite clauses. The selection of goals, however, does not follow the Prolog strategy but, instead, is based on deterministic closure. Furthermore the user may employ delay statements to control runtime behavior.

CLG

A formalism including a general delay mechanism is the Constraint Logic Grammar (CLG) formalism (Balari et al., 1990; Damas, Moreira, and Varile, 1991; Damas and Varile, 1992). It employs strongly typed feature structures and a constraint language operating on them, associating partial specified feature structures with constraints. Constraint resolution is carried out by rewriting, delaying constraints which cannot be decided at the moment.

Alex Franz's HPSG Parser

A system developed especially for parsing of HPSG—written in LISP—is the one by (Franz, 1990). It implements the notion of well-typedness and attaches (possibly relational) constraints to types. The system highly relies on disjunctive specifications leading to performance problems. Parsing is performed by expanding a partially specified feature structure to one that satisfies all constraints of grammar. Satisfaction of disjunctive constraints involves a choice between alternatives, all possible choices at a given state are collected in an agenda. The next step is chosen by heuristics.

HPSG-PL

HPSG-PL (Popowich and Vogel, 1991; Kodrič, Popowich, and Vogel, 1992) is a workbench written in Prolog for developing and parsing HPSG grammars in the style of (Pollard and Sag, 1987). Types are specified using a template-like mechanism, thereby providing means to explicitly specify inheritance. Grammar principles are attached to types the same way as other type specifications and may refer to a set of predefined functional (vs. relational) constraints. These predefined functions also include a delay mechanism to cope with uninstantiated arguments, a behavior not provided for other devices of the system (such as path expressions which produce an error if their destination is uninstantiated). The system contains a chart parser operating on ID-rules (which thus have to be specified differently than the other grammar principles).

TFS

The TFS system (Emele and Zajac, 1990; Emele, 1991; Zajac, 1992) is a LISP implementation of a typed feature unification formalism with inheritance. Types can be defined by formulas comprising types, feature structures and the logical connectives $\wedge$, $\vee$ and `:-`, the last one used for (potentially recursive) definite clause style definitions of types. Partially specified typed feature structures are expanded by a deductive type checking mechanism in a top down fashion. Type checking unifies a feature structure with its type definition, recursively proceeding with embedded substructures. Disjunctive type definitions are explored via backtracking.

3.2 Shortcomings of Existing Processing Models

Some aspects are immediately apparent from the list of systems presented in section 3.1:

- Some systems are biased towards *parsing* (PATR, ALE, HPSG-PL and the Alex Franz parser).
- With the exception of PATR, all other systems offer a *type system*, its expressivity, however, being quite different between the various systems.
- TFS offers the perhaps most powerful type system, even allowing to define relational constraints as types, but not being able to express *appropriateness constraints*, which, on the other hand, can be easily defined in ALE, HPSG-PL, CUF and Alex Franz' parser. The type system of ALEP is not capable of expressing appropriateness constraints.

Besides existence and expressivity of a type system also another point is important to judge the ability of a system to implement HPSG—the way, in which the principles of grammar can be expressed.

PATR offers only the means to specify principles via templates and encode them with the phrase structure rules, also in ALEP, at least relational constraints have to be treated via procedural attachment to phrase structure rules. ALE, althogh designed primarily for HPSG, has no direct means to state principles, they must be stated via the definite clause mechanism and have to be triggered by rules.

Thus, as candidates for a system to bidirectionally implement HPSG only CUF, TFS and CLG remain.

CLG suffers from inefficiency due to its constraint rewriting strategy: it just augments predicates operating on feature structures with two arguments for the list of constraints at clause entry and exit, and "applies a rewriting process to the whole list from time to time".

TFS and CUF offer both all the functionality needed. However, they cannot totally avoid, that the grammar writer must keep procedural considerations in mind. Although the resolution strategy employed delays goals depending on uninstantiated variable, that mechanism cannot guarantee total independance of the instantiation sequence, since both systems fall back to a Prolog like resolution strategy if all goals are blocked.

3.3 Desiderata

An implementation of an HPSG grammar has to fulfill at least the following criteria:

- All constructs used in specifying the grammar must be available within the formalism, as there are:
  - The type system (including appropriateness constraints);
  - A means to state principles without having to recur to extra-grammatical devices;
  - Definable relational constraints.
- The grammar should be specifiable in a declarative fashion, without taking procedural aspect into account; nevertheless it should be operational for efficiency reasons.

In order to achieve the goal of independence of a particular processing direction (i.e. parsing or generation) a clear distinction between grammar and processing model is required to avoid (hidden) encoding of procedural aspects into the grammar.

Instantiation of feature structures should work in a lazy incremental way in order to prevent eager instantiation which at least results in inefficiency but may also lead to nontermination. Nevertheless, all structures built must be regarded as *well-typed* according to the type scheme and *well-formed* in the sense of obeying all principles of grammar.

Given the proposed lazy nature of feature structure instantiation, principles as well as relational constraints need to be able to cope with uninstantiated arguments in a reasonable way (i.e. avoiding nontermination whilst ensuring well-formedness).

Thus the ultimate goal is to operationalize principle based grammars without depending on procedural aspects. That means that every structure (maybe only partially) instantiated within the formalism can be regarded as well-formed in the sense, that it is only extensible in a way compliant with all principles of grammar, intrinsically carrying all constraints of grammar.

Chapter 4

Realization

4.1 The Representation of Typed Feature Structures

4.1.1 Feature Structure Unification

Most existing implementations, either LISP-based or Prolog-based, adopt the DAG-view of feature structures and represent them as directed acyclic graphs. In such an implementation, each node of a feature structure is represented by a structure (usually a list) containing feature/value pairs. Unification of two DAGs is then performed by merging the DAGs in the following way, starting at the root nodes:

1. merge the lists of edges of both nodes
2. for every pair of edges with the same label, remove one edge from the result, and unify its value (i.e. the target node of that edge) with the value of the edge remaining.

This procedure is most efficiently performed in a destructive manner, which raises the need of copying the input structures before starting unification, if the input DAGs may be needed also in their original version in later stages of processing (e.g. in a chart parser).

When considering the requirements HPSG poses onto a feature formalism, some problems with these traditional approaches immediately come to mind:

- Type definitions and well-typing and appropriateness constraints cannot be integrated in such a model easily.
- HPSG builds up very deep and rich structures even for small utterances, so the need to copy whole structures before every unification may become very costly.
- Relational dependencies and principled constraints clearly go beyond the capabilities of a formalism based solely on DAGs.

A further argument which is not directly related to HPSG concerns the implementation of feature structure unification within the logic programming paradigm, e.g. in Prolog. Although there are quite elegant implementations of DAG unification in Prolog (Gazdar and Mellish, 1989), all these implementations have to treat feature structures as some kind of data structure, which needs to be interpreted. Although unification is the primary operation in Prolog, feature structure unification must be treated by some kind of interpreter. This may not only cause less efficient operation but is also conceptually not very elegant.

All these considerations plus the need to have a self contained feature formalism able to cope with all the various types of constraints used in HPSG make it promising, to look at feature structures from a different point of view.

4.1.2 A CLP Perspective

Constraint logic programming (Jaffar and Lassez, 1987) has been developed during the past few years as a generalization of Logic Programming, replacing unification by the more general notion of constraint solving over a suitable algebra. A realization of an instance of a CLP scheme within a logic programming language has to cope with the problem that constraint solving—being a generalization of unification—cannot be handled by syntactic unification alone. The difficulties arise from from the fact that unification in logic programming languages such as Prolog is defined syntactically over Herbrand-terms. This prevents for example syntactically distinct but (in the sense of the underlying theory) semantically equivalent terms to be unified (as for, e.g., '1+3' and '4' in the domain of real numbers). The situation when unifying typed feature structures is quite similar. Consider two types (say A and B) having a greatest lower bound C. Although these two types are unifiable, yielding C, syntactic unification might fail or produce incorrect results. Thus constraint solving mechanisms are the adequate means, when the application domain contains relations other than syntactic equality. Constraint solvers may be embedded into a logic programming language by either writing a meta-interpreter or making use of a system which allows for the implementation of unification extensions, the approach adopted here.

Semantic unification as a means to extend logic programming languages can be realized in different ways. One approach to semantic unification, the one adopted by the the CLP system being used as the basis of the HPSG implementation described here, will be sketched here.

DMCAI CLP[1] (Holzbaur, 1990; Holzbaur, 1992) is a Prolog system, whose unification mechanism is extended in such a way that the user may introduce interpreted terms and specify their meaning with regard to unification through Prolog predicates. The basic mechanism to achieve this behavior is the use of *attributed variables* (Huitouze, 1990). These variables are an additional data-type allowing variables to be qualified by arbitrary

[1]DMCAI CLP is an enhanced version of SICStus Prolog (Carlsson and Widen, 1990) and available by anonymous ftp from `ftp.ai.univie.ac.at`. Patch-level 10 of SICStus Prolog 2.1 will provide the unification extensions described here directly as part of the distribution.

user-defined attributes. Attributed variables behave like ordinary Prolog variables with two notable exceptions: when an attributed variable is to be unified with a non-variable term or another attributed variable the unification extensions come into play. For either case the user has to supply a predicate, which explicitly specifies how the attributes interact and how they have to be interpreted with respect to the semantics of the application domain. Unification succeeds only if this combination—or verification—of the attributes involved is successful.

The following predicates (which the user has to supply) are called by the Prolog kernel when attributed variables are about to be unified:

`combine_attributes(C1,C2)`
: is called, when two attributed variables are to be unified and should compute the resulting attribute according to the underlying theory. The attribute of a variable may be updated, thus achieving the functionality of destructive updates required by many CLP instances. However, changes in the attribute of a variable are backtrackable in case of failure later on.

`verify_attribute(C,T)`
: is called, when an attributed variable is to be unified with a non-variable term.

These constraint solving clauses provide a means to integrate constraint solvers into logic programming languages in a declarative and efficient way. Declarativeness is achieved, because the equality theory can stated simply by predicates. Efficiency is preserved, because the interaction between syntactic and extended unification is coded in the Prolog kernel, dispensing with the need of having a meta-interpreter managing this interaction.

4.1.3 Typed Feature Structures as Interpreted Terms

The representation of typed feature structures in the system described here makes use of the CLP facilities provided by the enhanced Prolog system sketched above. A feature structure rooted in `X` is implemented by the attribute `fs(X,Type,Dag)` of the variable `X`, where `Dag` is either

- a list of feature-value pairs, the features being appropriate for `Type` and their values (attributed variables themselves) appropriately type-restricted, or
- an empty list in case `Type` is atomic, or
- the atom `uninstantiated` if the structure rooted in `X` is uninstantiated.

Unification of two feature structures is triggered by the unification of the attributed variables representing them. It amounts to rewriting the attributes of their root variables and unifying the root-variables. This operation is implemented via the constraint solving clause `combine-attributes`[2] shown in Fig. 4.1. The actual feature unification is done by the

```
combine_attributes( fs(X1,T1,Dag1), fs(X2,T2,Dag2)) :-
    merge_fs(T1, Dag1, T2, Dag2, T3, Dag3),      % unify well-typed
    detach_attribute(X1),                        % remove attribute
    X1=X2,                                       % unify Prolog vars
    update_attribute(X1, fs(X1, T3, Dag3)).      % rewrite attribute
```

Figure 4.1: Feature Constraint Rewriting (preliminary)

predicate `merge_fs/6`, which computes the greatest lower bound `T3` of `T1` and `T2` in the type lattice and merges the attribute/value lists `Dag1` and `Dag2` yielding `Dag3`. Recursion is handled automatically by the same constraint solving mechanism when the variables serving as values of a feature are unified.

By the interpretation of feature unification as constraint rewriting in a CLP system some of the goals mentioned in 4.1.1 have already been achieved. The integration of feature unification and term unification has been accomplished, feature structures are represented by attributed variables, whose unification amounts to the rewriting of their attributes. Secondly, a step in the direction of structure sharing has been made, thereby dispensing with the need of explicit copying. The copying efforts are transferred to the Prolog/CLP system itself, which has to perform the proper copying operation to be able to restore previous states on backtracking, but only for the variables affected during unification. All other parts of a structure remain untouched and need not be copied.

But simple feature unification is not the whole story. Still missing are the integration of well-typing constraints and of the constraints induced by the principles of grammar. However, it will be seen, that the advantages gained so far still carry over when integrating further mechanisms.

4.2 Well-typing Constraints

4.2.1 The Type Hierarchy

Before any typed feature structures can be used the type hierarchy has to be defined. The lattice of types, `top` being the most general type, is defined via the operator `..>/2`. Part of the type hierarchy is shown below.

[2] `detach_attribute/1` and `update_attribute/2` are builtin predicates in DMCAI CLP for manipulating attributed variables

```
top ..> sign.
        sign ..> word.
        sign ..> phrase.
                 phrase ..> headed_phrase.
```

Figure 4.2: The Type Hierarchy

However, traversing the lattice defined that way every time two types need to be unified would be expensive. Therefore, the most often needed predicates on types, such as determining the greatest lower bound of two types, are compiled away and reduced to simple table lookup at run time.

4.2.2 Appropriateness Constraints and Inheritance

Instead of defining *Approp* directly, the implementation described here takes the approach to specify only the introduction (or further restriction) of features. The operator `==>/2` is defined for that purpose, associating a type with a list of *feature:value* pairs, the *value* being the most general type that feature is permitted to have. These appropriateness restrictions on a type are inherited by all its subtypes, which can add new features or value restrict inherited features. Types not introducing any new feature must be distinguished from atomic types, which are not allowed to take any features. Therefore they introduce the empty list.

As an example the appropriateness declarations for the part of the type lattice shown above in Fig. 4.2 are given below in Fig. 4.3.

```
top           ==> [].
sign          ==> [phon:   phon,           % introduce features
                    synsem: synsem].
word          ==> [].                      % inherit only
phrase        ==> [dtrs:   const_struc].   % add feature dtrs
headed_phrase ==> [dtrs:   headed_struc].  % value restrict dtrs
```

Figure 4.3: Appropriateness Constraints

Again, as with the lattice traversing predicates, this rather static information does not warrant expensive traversal at run time. Therefore the often used predicates necessary for ensuring well-typedness are compiled out of the appropriateness declaration:

`introduced_in(Feature,Type)` maps every feature to the type introducing it `type_def(Type,Features)` maps every non-atomic type onto a list of *feature:type* pairs, such that every feature appropriate for that type appears in this list with the type its value is restricted to.

Thus at run time there is no need to check for inherited features or search the type lattice for the most general type allowing for a particular feature.

4.3 Lazy Instantiation and Incremental Well-typing

Since only well-typed feature structures are admissible in HPSG well-typedness of features structures should be assured as soon as possible. Therefore any time a node of a feature structure (represented as an attributed Prolog variable) is instantiated or affected by unification, a type inference step is interleaved to maintain well-typedness. This interleaving is possible due to the monotonicity properties of *TypeInf* (cf. section 2.3).

However, type inference never instantiates feature structures, it only restricts the type of a structure or the type of a value for some feature. Instantiation of feature structures occurs only when explicitly forced by the use of an instantiation expression (such as a path equation). These instantiation expressions are issued by processes such as parsers or generators, lexicon lookup routines, and by the principles of grammar in case their applicability has been established (see below). Uninstantiated nodes of a feature structure are represented also via attributed variables but instead of having a list of *feature:value* pairs they are marked with the atom `uninstantiated`.

This lazy instantiation scheme offers several advantages:

- Recursive types (such as the *phrase* type above which is recursive via its DTRS attribute) **must** be instantiated lazily to avoid infinite loops. If a lazy instantiation scheme is adopted as the general strategy, such recursive types do not require any special treatment.
- Efficiency gains may result when postponing "default"-initializations of feature structures according to the type scheme. In many cases type information alone is sufficient to eliminate branches in the search space. Instantiating the whole structure would be wasted effort in such cases.

Since type inference monotonously restricts feature structures, an incremental version of *TypeInf* can be defined. Such a version of *TypeInf* is implemented in accordance with the lazy instantiation scheme. In its incremental version, type inference is composed of 3 sub-functions, according to the cases that may arise during instantiation (and unification) of feature structures:

a. If a *type* is added to (i.e. unified into) an *uninstantiated* node no type inferencing is performed at the moment (since no features having to be checked are present).

b. If a *type* is added to (i.e. unified into) an *instantiated* node, and unification of the types yields a type other than the one already present at the node in question, then the arcs of that node have to be updated to reflect the appropriateness conditions of the restricted type. This is done by determining the admissible features via `type_def/2` and unifying the list of *feature:type* pairs obtained with the node in question (possibly leading to further type inference steps if the value of some feature is restricted further).

c. If a *feature* is added to a node, then firstly the type introducing that feature is determined (via `introduced_in/2`) and unified with the type of that node. If this unification succeeds, step b. above is performed.

Thus, if a node is instantiated, then (1) every feature appropriate for the nodes type is present and (2) the value of every feature is restricted to the appropriate type (although the values may be uninstantiated). Performing this sort of type inference during instantiation, instantiation of nodes always yields well-typed feature structures according to the definition of incremental well-typedness given below:

Incremental Well-typedness:
A node of a feature structure is *incrementally well-typed* iff:

- its type is *atomic* (i.e. a type which has no supertypes and no subtypes introducing any features)— such a node is represented by the attribute `fs(X,Type,[])`, i.e. the `Dag` is the empty list.;
- or it is *uninstantiated*—this can only be the case with types having some features appropriate for them or having subtypes introducing some feature; such a node is represented by the attribute `fs(X,Type,uninstantiated)`;
- or it is an *instantiated* node, having *all* features appropriate for its type represented in the list of its outgoing edges, the values of these features being incrementally well-typed nodes themselves, of a type at least as specific as required by the appropriateness declaration for its type.

Instantiation Operators

To achieve the instantiation behavior described above and to provide an interface to attributed variables, instantiation operators have been defined (along with the necessary clauses).

`X::=Type`
: creates `X` as an attributed variable representing an uninstantiated feature structure of type `Type`, or, if `X` is already an attributed variable, unifies the feature structure represented by `X` with `Type`.

`X::Path===Y`
: instantiates all values (if not already present) along `Path` (where `Path` is `F1:F2:...:Fn`) within the feature structure `X` unifying the value found at the end of `Path` with `Y`. `Y` may be either a type or a Prolog variable serving as coreference tag for structure sharing. This latter variant is used to state constraints such as the Head Feature Principle.

The interaction between instantiation and type inference is illustrated in the following example. The path expression

```
X::synsem:loc:cat:head===verb
```

gives rise to the feature structure:

$$\begin{bmatrix} \text{PHON} & \textit{phon} \\ \text{SYNSEM} & \begin{bmatrix} \text{LOC} & \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & \textit{verb} \\ \text{SUBCAT} & \textit{synsemlist} \end{bmatrix}_{\textit{cat}} \\ \text{CONT} & \textit{cont} \end{bmatrix}_{\textit{loc}} \\ \text{NONLOC} & \textit{nonloc} \end{bmatrix}_{\textit{synsem}} \end{bmatrix}_{\textit{sign}}$$

First, the attributed variable `X` is created with type *top*. Type inference then restricts the type of `X` to *sign*, since the feature SYNSEM is introduced by that type and inserts all features appropriate for type *sign*. Instantiation proceeds then along the specified path, restricting the node at the end of the path with type *verb*. At each level the appropriate features are inserted together with appropriately restricted, but otherwise uninstantiated values.

4.4 Well-typed Unification

In order to account for the well-typing requirements and for uninstantiated structures the attribute rewriting clauses given in Fig. 4.1 have to be modified. In fact, these additions have to be accounted for within `merge_fs/6`. Since instantiation produces only well-typed nodes and unification is supposed to do so, the input structures to `merge_fs` can safely be assumed to be well-typed. The following unification procedure preserves well-typedness and consists of the following steps:

- The result type is the unification of the input types.
- If both input feature structures are uninstantiated, leave the output feature structure also uninstantiated.

- If the result type equals one of the input types just merge the instantiated input lists of *feature:value* pairs yielding the result feature structure.
- Otherwise instantiate a feature structure containing the appropriate features for the result type with appropriately restricted values (type inference step b.). Then merge the instantiated *feature:value* pairs of the input structures into the instantiated template yielding the result.

That well-typedness is preserved by the unification steps described above can be easily verified. If both nodes are uninstantiated no type inferencing has to be performed due to the lazy incremental nature of the type inferencing scheme employed. If type unification yields a type already present, then unification of the *feature:value* pairs suffices, since all well-typing restrictions must already be present having well-typed input structures. Only when unification restricts both input types the change in the appropriateness conditions has to be accounted for by instantiating the most general feature structure appropriate for the result type (i.e. the list of appropriate *feature:value* pairs) and unifying both input structures into that list.

By the implementation described above an operational basis for representing feature structures has been achieved. A description language for feature structures close to HPSG is available via the instantiation operators. Feature structures are instantiated lazily but their well-typedness as far as they are instantiated is guaranteed at every instantiation stage. A simple and perspicuous interface between processes operating on feature structures and feature structure unification is provided by the implementation of feature structures as constraints on Prolog variables. Feature structure unification is simply triggered by unifying the Prolog variables representing them.

But well-typedness is not the whole story when implementing HPSG. Well-formed (in the sense of the grammar being implemented) structures are far more constrained, having also to obey all principles of grammar. To incorporate them into the framework described, additional provisions have to be made, especially to support the interaction of lazy instantiation and principles constraining (possibly uninstantiated) substructures.

4.5 Principles as Licensing Conditions

Feature logical formulas as the device to state constraints on feature structures are implemented by means of the instantiation expressions introduced in section 4.3, as far path expressions and type restrictions are concerned. Path equations have to be expressed by means of Prolog variables serving as coreference tags, the role of the logical connectives $\wedge$ and $\vee$ is taken over by the Prolog operators "," and ";". For example, let ϕ be the feature logic formula

$$phrase \wedge synsem : loc : cat : head \doteq dtrs : head\_dtr : synsem : loc : cat : head$$

and F the minimal feature structure such that $F \models \phi$, then these facts are written as:

```
F::=phrase,
F::synsem:loc:cat:head===H,
F::dtrs:head_dtr:synsem:loc:cat:head===H
```

In fact, being implemented as Prolog clauses, the path expressions incrementally instantiate the minimal satisfier of the formula they stand for. Complex constraints employing path equations and logical connectives are interpreted as a Prolog program triggering the unification of the feature structures made up by the simple path expressions. This simple interface between feature logical formulas and well-typed unification of feature structures is made possible by the implementation of feature structures as an attribute of Prolog variables together with extensible (term) unification provided by DMCAI CLP.

Grammar principles are now implemented as constraints on feature structures specified by means of feature logical formulas. But instead of relying solely on types to define the applicability of constraints, my system takes a slightly different approach. Although types play the most important role in stating the conditions when rewriting has to take place, additional devices to narrow these conditions are provided. This does not mean, that the implementation fully resembles the conditional feature structures of (Pollard and Sag, 1987). The only difference to the formalization of principles by means of constraint systems attached to types is the possibility to require a certain typing of embedded structures as a prerequisite of applying the constraint. The same effect could be obtained by a more fine grained type hierarchy together with suitable appropriateness conditions, so the theoretical results of (Carpenter, Pollard, and Franz, 1991) carry over to the approach described here. However, it may be convenient for the grammar writer not to overload the type hierarchy and to rely on this explicit subtyping. A more important aspect of these additional conditions is their use in specifying the wait conditions for the delay mechanism described below.

Because type information alone is not sufficient to determine which constraints have to be applied to a particular node due to the more general format of the constraints allowed in the implementation described here , a *licensing* view of the principles is adopted:

Every node of a feature structure has to be licensed by all principles of grammar.

A node is **licensed** by a principle if

- either the feature structure F rooted in that node *satisfies* the applicability conditions of the principle and the constraints expressed by the principle are instantiated and unified into F.
- or the feature structure F rooted in that node is *incompatible* with the applicability conditions of the principle.

The interesting case is, when a feature structure does not satisfy the applicability conditions of the principle but is compatible with them. Thus applicability of the principle can be decided only later, when further instantiation or unification steps have restricted the (sub)structure rooted at that node. In precisely this case the application (or the abandoning) of the constraint has to be delayed. The mechanisms in which way this delaying is implemented will be described in section 4.5.1.

This licensing view also affects the strategy how these constraints are processed in a way that complies well with the lazy incremental instantiation of feature structures. Having to look up the constraints associated with a node's type any time the type of the node is restricted during unification involves either to have to rewrite with all inherited constraints or to do some bookkeeping on which constraints have already been applied. In contrast, in my implementation the set of constraints is applied to a node at the time of its instantiation, i.e. only *once*. Applicable constraints are applied immediately, irrelevant constraints are dropped, and possibly relevant constraints are delayed. Inheritance of constraints along the type hierarchy is achieved automatically by this method.

As an example, how principles of grammar are stated in the form of a conditional constraint the Head Feature Principle of HPSG is shown in Fig. 4.4.

```
head_feature_principle(X) :-
    X::=headed_phrase
===>
    X::synsem:loc:cat:head            ===Head,
    X::dtrs:head_dtr:synsem:loc:cat:head===Head.
```

Figure 4.4: Implementation of the Head Feature Principle

4.5.1 Conditional Constraints and Coroutining

Conditional constraints are written as Prolog clauses where the body of the clause consists of *preconditions* and *constraints*, separated by the operator `===>`. The preconditions may contain only conjunctively connected path expressions; the constraint part may contain arbitrary Prolog code. To achieve the licensing and delaying behavior of the principles stated as conditionals, these clauses have to be given a special interpretation at load time (calling the path expressions of the antecedent in the usual way would just instantiate them). The procedural interpretation of such conditional constraints reflects their use as rules of inference and is achieved via `term_expansion/2` in the following way:

- The preconditions (which are required to be type restricting path expressions) are wrapped into a call to `preconditions/2`, which interprets these conditions at run time.

- Path expressions are interpreted by going down the path in the feature structure and at the end checking the type of the node reached against the type required. The path expression is

  satisfied by the feature structure if the path is fully instantiated and the type of the node the path is leading to is at least as specific as the type required;

  satisfiable if the path is not fully instantiated but the feature continuing the path is appropriate for the type of the uninstantiated node discontinuing the path or if the type of the target node is more general than the type required;

  incompatible in case of inappropriate features in the path or incompatible type of the target node.

- `preconditions/2` encodes the result of this subsumption check in its second argument, a list. If the list is empty, all preconditions are satisfied, in case of satisfiable conditions, the uninstantiated or too general nodes (i.e. the attributed variables representing them) are collected in this result list.[3] In case of conditions incompatible with the feature structure `preconditions/2` fails.

Precisely, these conditional clauses are translated in the following way:

```
term_expansion((G :- PreConds ===> Actions), Z) :-
 Z=(G :- ( preconditions(PreConds, L) ->  % check conditions
           ( L == [] -> Actions           % satisfied    -> apply
           ; block_goal(L,G) )            % satisfiable  -> delay
         ; true)).                        % incompatible -> succeed
```

Figure 4.5: Load Time Expansion of Conditional Constraints

The actual delaying mechanism is implemented via `block_goal(VarList,Goal)`, which stores `Goal` with each variable in `VarList`. The idea behind implementing this kind of principle application delaying is to annotate the variables that are "responsible" for the unability to decide on the antecedent of the principle that has to be applied.[4]

To allow for the annotation of feature structures with goals the `fs`-attribute has also to account for such delayed constraints. Thus feature structures are finally represented by the attribute `fs(X,Type,Dag,DelayedGoals)`.[5] The thawing of a delayed goal is attempted

[3] In fact only the first variable of a conjunction needs to be collected, for disjunctive preconditions only the first one of each alternative.

[4] This mechanism bears some similarity with the `block` declaration of SICStus Prolog, which allows to specify delay conditions via argument patterns. But while with `block` only a distinction between `var`*s* and `nonvar`*s* is possible, the method presented here allows for subsumption checks along arbitrary paths of feature structures.

[5] The variable `X` is a reference to the Prolog variable the attribute is attached to.

every time unification affects the node this goal is attached to. Because the node might now have been restricted enough to satisfy the applicability conditions or to contradict them, the delayed goal has to be called again. If the delayed goal is unable to decide on its applicability again, it reinserts itself at an appropriate node in the feature structure, waiting to be triggered again.

Since a new attempt to satisfy a delayed constraint has to be triggered by unification of feature structures, the mechanism handling the attribute rewriting corresponding to feature structure unification is also responsible for thawing the delayed goals. Thus the final version of `combine_attributes/2`:[6] can now be given in Fig. 4.6.

```
combine_attributes( fs(X1,T1,Dag1,P1), fs(X2,T2,Dag2,P2)) :-
  merge_fs(T1, Dag1, T2, Dag2, T3, Dag3),
  (X1 == X2 -> true                           % already unified
  ; detach_attribute(X1),                     % remove attribute
    X1=X2,                                    % unify Prolog vars
    update_attribute(X1, fs(X1, T3, Dag3,[])), % rewrite attribute
    start_goals(P1), start_goals(P2)          % restart constraints
  ).
```

Figure 4.6: Feature Constraint Rewriting (final version)

After the unification result (`T3` and `Dag3`) has been computed via `merge_fs`, the attribute is rewritten with an empty goals list. `start_goals/1` successively calls the delayed goals of both input structures. Principles that cannot be applied due to insufficient specification of the resulting feature structure themselves take care of inserting themselves at the appropriate place using the mechanism described above for interpreting their preconditions. Note, that before actually starting the rewriting of the attribute, it is checked whether the to variables have not been already unified (this could have been the effect of a delayed constraint triggered by the unification of some substructures during `merge_fs`).

The licensing view of grammar principles and the coroutining technique developed enable these conditional constraints to cope with situations where no immediate decision on their applicability can be made. This allows for applying the grammar principles at instantiation time without having to expand the feature structure early, thus conforming well with the lazy incremental instantiation scheme. Thus feature structures may always be regarded as well-formed, since their well-typedness and their compliance with the grammar principles is guaranteed in the sense that they are not unifiable with or expandable into an ill-formed feature structure. This incremental conception of well-formedness is independent from the degree of instantiation of the feature structure and makes it possible to exploit the benefits

[6] `detach_attribute/1` and `update_attribute/2` are builtin predicates in DMCAI CLP for manipulating attributed variables

of *vaguely* specified (e.g. only partially instantiated) feature structures instead of forcing instantiation or subtyping leading to *ambiguity*, thus helping in reducing backtracking when processing such feature structures.

4.5.2 An Instantiation Example

In order to show the interaction of instantiation, well-typing constraints and principles as licensing conditions, different stages of instantiated-ness of a feature structure of type *phrase* are given in the following examples. The only principle applying is assumed to be the head feature principles (see Fig. 4.4), the appropriateness declarations are according to Pollard and Sag (1994). `pp/1` is a predicate for pretty-printing a typed feature structure in an AVM-like notation. Numbers surrounded by square brackets stand for coreference tags, pending principled constraints are shown at the type, where they are attached at the moment.

The first example (Fig. 4.7) shows the "instantiation" of a phrase. In fact, only an

```
| ?- X::=phrase, pp(X).

phrase <--head_feature_principle
```

Figure 4.7: Instantiation (1)

uninstantiated node with type *phrase* is created. Since the preconditions of the head feature principle are neither satisfied nor incompatible with the node, it is attached to the nodes goals list and waits, to be triggered at future unification steps.

The second example (Fig. 4.8) shows the effects of type inference and well-typing con-

```
| ?- X::dtrs===Y, pp(X).

phrase
phon:   phon
synsem: synsem
dtrs:   const_struc <--head_feature_principle
```

Figure 4.8: Instantiation (2)

straints. Instantiating a otherwise unspecified structure with a DTRS attribute, whose value is some variable results in inferring the maximal type carrying such a attribute, i.e.

phrase, and, according to the appropriateness definitions, instantiating all slots appropriate for *phrase*, since one of them has been referenced and must be instantiated. Whether the head feature principle is applicable or not depends on the restriction which may be imposed on *const_struc*, the value of the DTRS attribute. Therefore that node is annotated with the delayed principle.

Finally, an example where the head feature principle applies is given in Fig. 4.9. By

```
| ?- X::dtrs:head_dtr===word,pp(X).

phrase
phon:   phon
synsem: loc: cat: head: [1]
dtrs:   headed_struc
        head_dtr: word
                  synsem: loc: cat: head: [1]head
```

Figure 4.9: Instantiation (3)

forcing the HEAD_DTR of the DTRS attribute to be a *word*, type inference can conclude, that the type of the value of the DTRS attribute is actually a *headed_struc*, which triggers instantiation of the constraint induced by the head feature principle.

4.5.3 Relational Dependencies

Up to now, only constraints involving equality and type restriction have been discussed, but some principles of HPSG involve relational constraints (such as *append*) expressible only by recursive definitions. These relational constraints can be integrated straightforwardly into the framework presented here without introducing any further mechanisms. The reasons for the ease this integration may be accomplished are:

- Prolog variables representing feature structures can be used by programs like ordinary variables without loosing their special properties (as, e.g., maintaining well-typedness etc.). Thus the interface between feature structures and programs manipulating them is as simple as possible.
- The syntax to state conditional constraints as described in section 4.5.1 is only restricted for the antecedent part of the conditional, the constraint itself may be an arbitrary Prolog program. Thus also relational dependencies can be stated.
- The delay mechanism is not confined to unary grammar principles. Also predicates with more than one argument may make use of the delay mechanism using the conditional constraint syntax, as long as

1. their arguments (or at least the arguments involved in the preconditions) are feature structures, and
2. the predicate is defined by a single clause.

A crucial requirement when integrating relational dependencies into the lazy instantiating feature formalism is the ability of those relations to cope with insufficiently instantiated arguments. Consider for example the standard definition for `append/3`.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

When called with uninstantiated first and third argument, `append/3` serves as a generator for arbitrary long lists and traps the program that did such an unfortunate call in an infinite loop. Such a behavior is unacceptable in a framework that aims on independence of a particular processing sequence and therefore provides no means to avoid such traps by relying on procedural considerations. As a consequence, also the execution of relations has to be delayed until their arguments are sufficiently instantiated to decide on the result (at least partially).

The conditional syntax used in defining the grammar principles provides the means to specify the delay conditions also for these recursive relations, as in the following example, defining a typed version of *append* for feature structure lists: [7]

```
fs_append(X,Y,Z) :-
       X::=list,Y::=list,Z::=list,       % argument typing
       fs_empty_append(X,Y,Z),
       fs_nonempty_append(X,Y,Z).
fs_empty_append(X,Y,Z) :-
       X::=elist
  ===> Y = Z.
fs_nonempty_append(X,Y,Z) :-
       X::=nelist
  ===> X::first===F, Z::first===F,
       X::rest===XRest, Z::rest===ZRest,
       fs_append(XRest,Y,ZRest).
```

Figure 4.10: Append for Feature Structure Lists

The argument typing causes `fs_append/3` to fail on improper arguments. Once this test is passed, the two specialized clauses are applied in parallel and wait until it can be determined which of the two constraints has to be enforced. Thus the advantages of the

[7]The type *list* subsumes *elist* (the empty list) and *nelist* (the type of nonempty lists, with the appropriate attributes FIRST and REST)

conditional definition are twofold. First, the disjunctive relation *append* can now be written as conjunction of two specialized cases applying conditionally. Second, infinite loops due to uninstantiated variables can never occur.

This scheme to specify relational constraints on feature structures as exemplified above allows to cope with only partial instantiated structures. Moreover it allows for replacing disjunctive constraints by a conjunction of specialized conditional constraints. Whereas the delay mechanism is inevitable to avoid infinite recursion, it helps also—together with the reformulation of disjunctive into conjunctive specifications—to reduce the search space by avoiding to introduce choice points and waiting instead until the choice becomes deterministic.

4.6 Principled Specification of Phrase Structure

Now the apparatus needed to pursue the goals formulated in section 3.3 is available. A process-model independent formulation of the principles of HPSG relating to phrase structure can be given in terms of conditional and relational constraints. In particular, the Subcategorization Principle (see Fig. 2.13) and the relations involved in that principles can be implemented using the constructs introduced above. The `fs_append/3` relation is already given in Fig. 4.10, the relation extracting the SYNSEM values of a list of *signs* is shown in Fig. 4.11. The Subcategorization Principles itself is given in Fig. 4.12, and

```
synsem_values(SignList,SynsemList) :-
       synsem_values_empty(SignList,SynsemList),
       synsem_values_nonempty(SignList,SynsemList).

synsem_values_empty(SignList,SynsemList) :-
       SignList::=elist
  ===> SynsemList::=elist.

synsem_values_nonempty(SignList,SynsemList) :-
       SignList::=nelist
  ===> SignList::first:synsem ===FirstSynsem,
       SynsemList::first      ===FirstSynsem,
       SignList::rest         ===SignRest,
       SynsemList::rest       ===SynsemRest
       synsem_values(SignRest,SynsemRest).
```

Figure 4.11: Extraction of the SYNSEM Values

strongly resembles the AVM notation used in stating this principle in Fig. 2.13. An al-

```
subcat_principle(X) :-
       X::=phrase, X::dtrs===head_struc
  ===> X::synsem:loc:cat:subcat===Subcat,
       X::dtrs:head_dtr:synsem:loc:cat:Subcat===HeadDtrSubcat,
       X::dtrs:comp_dtrs===CompDtrs,
       synsem_values(CompDtrs,CompDtrsSynsemValues),
       fs_append(HeadDtrSubcat,CompDtrsSynsemValues,Subcat).
```

Figure 4.12: Implementation of the Subcat Principle

ternative way to state this principle would be to collapse the two relations `fs_append` and `synsem_values`, yielding a relation between two synsem-lists and one sign-list, the SYNSEM values of which are extracted (cf. Franz (1990)).

The encoding of phrase structure in the form of generalized immediate dominance schemata (as outlined in section 2.2.1) is encoded within the feature formalism by means of the type definition (cf. Fig. 4.3). The argument structure encoding of lexical item is made effective by the Subcategorization Principle as stated above in Fig. 4.12.

The constraints expressed by the Subcategorization Principle require the proper "discharging" of the SUBCAT list of the phrase's head. The complement daughters SYNSEM values must unify with the respective elements of the head's SUBCAT list, only argument positions not already saturated may be filled at higher projection levels (due to the `fs_append` constraints).

As can be easily verified, these constraints work independently of the order in which the structure is built up. In a top down processing model, the phrasal skeleton is constrained in a lazy fashion, as long as no lexical head is instantiated. But, even if the head is not known already, complements filled into the phrase constrain the possibilities for the lexical head by constraining its SUBCAT list via the `fs_append` constraint of the Subcategorization Principle. In a bottom-up processing model, the subcategorization requirements of the lexical head are known, and the `fs_append` constraint fires as soon as the instantiation of a complement is attempted.

Thus the encoding of the grammar in terms of feature constraints is really self contained and does not need any additional stipulations outside the feature formalism. In particular, it does not need to rely on a certain processing or instantiation sequence.

Combining the operational grammar specification with a particular processing model is therefore as easy as could be and can be done in a number of ways.

4.7 Summary

The method for the implementation of a feature formalism developed in this chapter aims at the support of principle based approaches to grammar, especially at covering all details needed for implementing HPSG. The goal of achieving independence of a particular processing direction (i.e. parsing or generation) requires a clear distinction between grammar and processing model to avoid (hidden) encoding of procedural aspects into the grammar. To be able to express all aspects of a principle based grammar without recurring to procedural considerations requires the feature formalism to be powerful enough to represent them in a declarative way and handle them without depending on a particular processing or even instantiation sequence.

However, the declarative specification of a grammar is not static data, the connection of feature logic formulas to feature structures via the instantiation operators makes it an operational basis for processing. This instantiation of feature structures works in a lazy incremental way, structures are only built on demand. Nevertheless, all structures built can be regarded as well-typed according to the type scheme and well-formed in the sense of obeying all principles of grammar. To achieve this behavior, the notion of well-typedness is interpreted in an incremental way, and type inference steps are interleaved with unification and instantiation.

To guarantee well-formedness of feature structures, principles of grammar are viewed as licensing conditions on feature structures and applied to every node at instantiation time. A delay mechanism allowing principles to operate on uninstantiated feature structures without eagerly instantiating them has been implemented. A conditional syntax for specifying principles has been introduced which allows to state their applicability conditions. Delay occurs, if a node fails to meet the preconditions at the moment but has the potential to meet them later. Relational constraints may also make use of the delay mechanism using the same conditional syntax.

The implementation was done on top of a Prolog system with extensible unification (DMCAI CLP) allowing to specify the rewrite operations on feature structures in a clear and declarative way. Feature structures are implemented as attributes of Prolog variables. The rewrite operations to achieve well-typed unification and principled constraint application are performed via the extended unification mechanism when two variables representing feature structures are unified. The delay mechanism has been integrated into this attribute rewriting clauses. Thus the use of a CLP system dispenses with the need of having a meta-interpreter managing this interaction and also leads to efficiency gains, because the interaction between syntactic and extended unification is coded in the Prolog kernel. A further advantage of the use of attributed variables to represent feature structures is the simple interface they provide to processing modules. Unifying two feature structures can be performed simply by unifying the Prolog variables representing them, success of this unification yields also a rewritten feature structure, its well-typedness and well-formedness being guaranteed.

Thus the goal of operationalizing principle based grammars has been achieved without depending on procedural aspects. Every structure (maybe only partially) instantiated within the formalism can be regarded as well-formed in the sense, that it is only extensible in a way compliant with all principles of grammar, intrinsically carrying all constraints of grammar.

How conditional and relational constraints can be utilized further will be demonstrated in the next chapter, which deals with applications extending standard HPSG.

Chapter 5

Applications

To illustrate the usefulness of the feature formalism described in Chapter 4, I will describe two application that go beyond the original setting of developing a formalism for a principled implementation of HPSG.

The first application exploits the possibility of adding arbitrary Prolog goals to the consequent part of a conditional constraint and the definability of relational dependencies without risking infinite loops due to uninstantiated arguments. It deals with the integration of word-level processing, in particular computational morphology and sentence-level processing in the style of the HPSG (see also Trost and Matiasek (1994)). Such an integration conforms to the spirit of HPSG, which views grammar as a relation between the phonological (or graphemic) form of an utterance and its syntactic/semantic representation. Moreover, this way the treatment of phenomena transcending the boundary between morphology and syntax is also made possible.

Despite the obvious advantages of such an integration, most (if not all) existing implementations of HPSG do not tackle morphological issues at all and rely on full-form lexicons containing all inflected and derived forms of the words. One reason for the lack of attempts in this direction may be found in the conceptual differences between mostly finite-state morphological processors and unification based grammar formalisms, which present problems to a tight coupling of these components. A loose coupling of syntax and morphology, on the other hand, makes it rather tedious, if not impossible, to take advantage of an interaction between syntax and morphology.

The feature formalism presented here now offers precisely the necessary machinery (i.e. the definability of relations and the use of these relations from within the feature formalism) needed to achieve a tight integration of these two modules.

The second application shows how specific problems of German related to case alternation in passive constructions, which are not covered by "standard" HPSG can elegantly be solved within the proposed framework. A simple-minded lexicalist approach to these phenomena immediately runs into problems since the correct case assignment for so-called "structural" arguments cannot be determined on the lexical level alone, but depends on the syntactic context. Furthermore, this alternation of case goes along with a change in the availability of argument positions.

A principled solution—following the lines of Heinz and Matiasek (1994)—to these two related phenomena will be presented, which introduces the notion of structural case and syntactic principles realizing structural case according to the syntactic context. Argument reduction phenomena, which occur, e.g., in the context of past participle formation will be given a treatment in the context of the syntax/morphology interface presented in Section 5.1, which has been originally described in Trost and Matiasek (1994).

5.1 Interfacing HPSG and Two Level Morphology

A widely used approach in word-level processing is finite-state morphology, most notably two-level morphology (for an introduction, see Sproat (1992)). Morphological components are successfully used for a wide range of stand-alone applications like spelling correction and hyphenation, but their use in NLP systems for analysis and generation of text has been not so frequent as one might have expected, mostly due to the problem of interfacing morphology with syntax.

However, a tight integration between word and sentence level processing has not only technical but also linguistic advantages. The boundary between morphology and syntax is fuzzy. When processing written text the units morphology has to deal with are—in a technical sense—not words but character strings separated by delimiters. While these strings roughly correspond to the words of a sentence there are problematic cases. In German, e.g., *zu*-infinitive or verbs with separable prefixes are written as a single unit in some instances and separately in others.

One step to reduce interface problems has already been taken by Trost (1990; 1991) by describing word formation with the same means, i.e. a unification-based grammar, as sentence level phenomena. However, the problem of integrating morphophonology into that framework remains. I will show, how this integration in fact can be accomplished within the framework described in chapter 4 by adopting a relational view of morphophonology and integrating word-level and sentence-level processing within the same grammar.

5.1.1 Two-Level Morphology

The "two-level model" of computational morphology, developed by Koskenniemi (1983) is the predominant computational paradigm for handling morphophonological phenomena. The basic idea of two-level morphology is, that two strings, the *surface* string corresponding to the graphemic representation at the word level, and the *lexical* string, built up by concatenating the lexical strings of the single morphs composing the word, are related by a set of rules. These two-level rules license the transitions between lexical and surface characters depending on the context surrounding each such transition pair. Special diacritic characters may be used on the lexical level for, e.g., marking morph-boundaries. At the surface level, special *null* characters are used to indicate transitions, where lexical characters "disappear" on the surface. These null characters roughly correspond to ε-productions of phrase structure grammars.

Fig. 5.1 shows an example of how the two-level model works. The surface form *"rät"* of the third person singular present tense form of the German verb *raten* (to guess) is composed of the stem *"rAt"* and the suffix *"+t"*, similarly *"sagt"* (= "tells") is combined from the stem *"sag"* and the same suffix. As can be seen, in the case of *raten* the stem-final t is

surface level	0	r	ä	t	0	0	0	= *rät*	0	s	a	g	0	t	0	= *sagt*
	↕	↕	⇕	⇕	↕	↕	↕		↕	↕	↕	↕	↕	↕	↕	
lexical level	$	r	A	t	+	t	$		$	s	a	g	+	t	$	

Alphabet of Transitions: $:0, +:0, a:a, A:a, **A:ä**, g:g, r:r, s:s, t:t, **t:0**, ...
Rules: **t:0** $\Longleftrightarrow$ _ +:0 t
...

Figure 5.1: The Two-Level Model

elided whereas in the case of *sagen* only default transitions (marked ↕ in Fig. 5.1) take place.

The set of all possible transitions is given in an alphabet of pairs, some of them having to be licensed by rules[1]. The diacritics *"$"* and *"+"* mark word and morph boundaries respectively, *"A"* may translate to *"a"* or *"ä"* on the surface. Rules license a certain transition within a context, which may be given by means of regular expressions. The *t-elision*-rule in Fig. 5.1, e.g., states, that a lexical *"t"* may be elided (i.e. the transition *t:0* may be taken) in a context, where it is followed by a morph boundary which translates to ε (i.e. the transition *+:0*, and then by a *"t"* (which is a shorthand for the pair *t:t*). In fact, elision *must* take place in that context, since this is an equivalence rule (as indicated by $\Longleftrightarrow$, there are also weaker rules employing $\Rightarrow$ or $\Leftarrow$ with the obvious interpretation).

The fact that context is given by means of regular expressions has led to the approach (pioneered by Karttunen (1983)) to model two level rules by means of finite state transducers. Due to the linear complexity of finite state automata this approach has gained much attention and led to the implementation of a number of efficient systems.

[1] Pairs that have to be licensed by rules are marked boldface in Fig. 5.1

How morphs may be combined to form valid words (*morphotactics*) is controlled by continuation classes associated with the morphs and a continuation class automaton controlling morph combination.

There are, however, some non-concatenative phenomena, which cannot be captured by two-level rules alone. One of them is umlaut, which does not depend on lexical realizations but on intrinsic properties of the morphs being combined. Attempts to overcome these problems by excessively using diacritics were not very successful and do not contribute to the clarity of the formalism. Therefore it seems more appropriate to extend the two-level paradigm.

5.1.2 X2MorF

X2MorF (Trost, 1990; Trost, 1991; Trost, 1993a) differs from standard two-level morphology in two important respects. Continuation classes are replaced by a feature-based word grammar. This allows for a more fine-grained description of morphs. It is also a prerequisite for a tight integration with a unification-based grammar. X2MorF uses a morph lexicon where each morph has one or more feature structures assigned. The word grammar itself is simple. Morphs have a functor-argument structure along the lines of (di Sciullo and Williams, 1987). Affixes are unary functors while stems are arguments without any further structure, resulting in a binary tree structure.

The other extension concerns the two-level rules, which are supplemented with a morphological filter consisting of a feature structure. This is important because in morphophonology only some rules are purely phonologically motivated. Others are triggered by a mixture of phonological and morphological facts. Such rules cannot be properly represented in the standard approach.

Now the rule controlling the transition *A:ä* in Fig. 5.1 can be stated, which was not possible using only standard two-level rules: This rule is not sensitive to any context but

$$\textbf{A:ä} \Longleftrightarrow \_ \; ; \; [\textsc{morph}|\textsc{mhead}|\textsc{umlaut} \; +]$$

Figure 5.2: Umlaut Rule

to a UMLAUT feature set by the interaction of morph lexicon and morph grammar.

Thus X2MorF can be seen as a relation between a surface string (the word form), a lexical string, and a feature structure (the interpretation of the word form). Nevertheless, X2MorF in its original implementation encodes this relation in a rather complicated way, still employing finite state automata for handling the two-level part of this relation and controlling the interaction between filter-rules and morph grammar in a procedural fashion. One way to overcome this difficulty could be the approach by Krieger, Pirker, and Nerbonne

(1993), who encode finite state automata directly in a typed feature formalism. Since two-level rules can be compiled into such automata, morphophonology can be straightforwardly integrated into the grammar. While this is formally elegant it seems to be no good solution for practical considerations. First, it is not entirely clear from their paper how the problem of null characters can be handled. Second, encoding large automata will result in a very large and unwieldy type hierarchy. In general, introducing automata into feature structures and encoding morphophonology directly at that level seems to be too low-level.

5.1.3 A Relational Perspective

The approach to integration presented here also takes the morph grammar as the primary component and embeds the relation between surface and lexical strings driven by the two-level rules into the feature formalism. However, this relation is not implemented in terms of finite state automata but is encoded directly as a Prolog relation. Arguments supporting this approach can be found in Abramson (1992), who rediscovered the original, relational spirit of Koskenniemi (1983). But the approach of Abramson (1992) is extended here in a way that explicitly accounts for the insertion of null characters and introduces the filter concept of X2MorF into the relational approach.

The general format of a two-level rule specification in the system described here is

```
LeftContext <=> Transition <=> RightContext [:- Filter]
```

in the case of equivalence rules, optional rules are written using only single arrows (`=>` and `<=`). These rules are compiled into Prolog clauses[2] relating the lexical and surface character streams appropriately (see Fig. 5.3 for an example of the *t-elision* rule for German).

	$t{:}0 \Longleftrightarrow \_ \; +{:}0 \; t$
Input	`_ <=> t:0 <=> ['+':0, t:t].`

Compiled	`morphrule([116,43,116|LS],[Sc,48,116|SS0],SS,LCon,SCon,F) :-`
	`!, Sc=48,`
	`morphology([43,116|LS],[48,116|SS0],SS,[116|LCon],[Sc|SCon],F).`

Figure 5.3: Sample Two-Level Rule

To obtain a correct relationship between surface and lexical string every transition has to be licensed by a morphological rule. Transitions not mentioned by rules are handled by a default rule. Instantiation of contexts may not be done by the rules themselves, since this

[2]Note that left contexts are encoded reversed to account for the left to right traversal of the pair of character streams. Left contexts can be remembered and checked most efficiently this way.

would make it impossible to obtain negation via the cut-operator. Instead, it is handled separately in a backtrackable fashion.

The central relation is the `morphology` predicate, (see Fig. 5.4) mediating between lexical string, surface string (with inserted null elements), the pure (de-nullified) surface string and the feature structure of the morphological sign. Instantiation of pairs is done depending on the possible lexical continuations (the lexicon being represented by a trie-structure). The amount of lookahead is determined by the current pair which is to be licensed by `morphrule`.[3] Synchronization of surface and lexical string by insertion of null characters is also handled at the instantiation level.

```
morphology(LexStream,[SC|SurfStream],WordFs) :-
  X::=msign,
  morphology(LexStream,[SC|_],[SC|SurfStream],[],[],WordFs).

morphology(LexStream,SurfStream0,SurfPlainIn,LexContext,SurfContext,F) :-
  instantiate(LexStream,SurfStream0,SurfPlainIn,SurfPlainOut,F),
  morphrule(LexStream,SurfStream,SurfPlainOut,LexContext,SurfContext,F).

instantiate([LC|LCs],[SC|SCs],SurfPlainIn,SurfPlainOut,F) :-
  valid_alphabet_pair(LC,SC),
  synchronize([SC|SCs],SurfPlainIn,SurfPlainOut),
  lookahead(LC,LCs,SCs,SurfPlainOut).

synchronize([48|_],Stream,Stream) :- !.
synchronize([Char|_],[Char|Stream],Stream).
```

Figure 5.4: The `morphology` Relation

5.1.4 Integrating Word-level and Sentence-level Processing

Originally X2MorF was realized as a separate morphological component interfaced to the sentence analyzer/generator only via sequential data transfer. In the case of analysis, the feature structure representing the word form was transmitted to the parser. For generation, X2MorF expected a feature structure as input reproducing one or more word forms. This purely sequential architecture was not satisfactory because of the problems mentioned above.

[3]This interaction and the lexicon lookup of the feature structure corresponding to the current morph, which takes place when encountering a morph boundary is not shown for the sake of simplicity.

The integration of the two-level relation into the general framework of the feature based sentence-level and word-level grammars is now performed by adding this relation as a principled constraint at the appropriate level.

In a definite clause style AVM notation this could be written as follows in Fig. 5.5 (given that `morphology/3` is a wrapper around the morphology relation `morphology/6` given above in Fig. 5.4, starting with empty left context and hiding the nullified surface stream):

$$\left[\begin{array}{ll} \text{PHON} & \boxed{2}\,\textit{string} \\ \text{SYNSEM} & \textit{synsem} \\ \text{MORPH} & \boxed{3}\left[\begin{array}{ll} \text{MSTRING} & \boxed{1}\,\textit{string} \\ \text{STEM} & \textit{string} \\ \text{MHEAD} & \textit{mhead} \end{array}\right]_{\textit{msign}} \end{array}\right]_{\textit{word}} \longleftarrow \quad \textit{morphology}(\boxed{1},\boxed{2},\boxed{3})$$

Figure 5.5: The Morphology Constraint

The actual implementation as a principled constraint within the feature formalism additionally takes care of delaying the actual enforcement of this relation in case the strings are not sufficiently instantiated.

A second provision has to be made in the word level grammar to assure proper concatenation of the lexical strings of the morphological signs being combined. Given the subtyping of *msign* into *marg* and *mfunctor*, which in turn has the subtypes *leftfunctor* and *rightfunctor*, the principled constraints ensuring concatenation of a left functor with its argument are shown in Fig. 5.6. Concatenation is delayed until the argument's MSTRING is instantiated. Thus, infinite loops when concatenating are avoided.

```
concat_right_functor(X) :-
        X::=rightfunctor,
        X::arg:mstring===subtype_of(string)
  ===>
        X::arg:mstring===Arg,
        X::affix===Suffix,
        X::mstring===Mstring,
        concat(Arg,Suffix,Mstring).
```

Figure 5.6: Concatenation of Lexical Strings

As an example we demonstrate how these constraints interact in forming the third person singular present tense form of the German verb *raten* (to guess). The lexical string is composed of the stem *rAt* and the suffix *+t*. The lexical entries of these two morphs are given in Fig. 5.7.

"rat"

$$\left[\begin{array}{ll} \text{MSTRING} & \textit{"rAt"} \\ \text{STEM} & \textit{"rat"} \\ \text{MHEAD} & {}_{verb\_stem}\left[\begin{array}{ll}\text{EPENTHESE} & - \\ \text{UMLAUT} & + \end{array}\right] \end{array}\right]_{marg}$$

"+t"

$$\left[\begin{array}{ll} \text{STEM} & \boxed{1}\ \textit{string} \\ \text{AFFIX} & \textit{"+t"} \\ \text{MHEAD} & \left[\begin{array}{ll}\text{EPENTHESE} & \boxed{3}\ \textit{boolean} \\ \text{PERSON} & \textit{pers3} \\ \text{TENSE} & \textit{tense\_pres} \\ \text{UMLAUT} & \boxed{2}\ \textit{umlaut} \end{array}\right]_{verb\_form} \\ \text{ARG} & \left[\begin{array}{ll} \text{STEM} & \boxed{1} \\ \text{MHEAD} & \left[\begin{array}{ll}\text{EPENTHESE} & \boxed{3} \\ \text{UMLAUT} & \boxed{2}\end{array}\right]_{verb\_stem} \end{array}\right]_{msign} \end{array}\right]_{rightfunctor}$$

Figure 5.7: Lexical Entries for Morphological Signs

The two-level rules applicable for this example are the *t-elision* rule (Fig.5.3) and two rules with filters licensing *a-umlaut* and *epenthesis*, given in the input notation of my system (Fig.5.8).

A-umlaut	`_ <=> A:"a <=> _ :- filter(X,[X::mhead:umlaut==='+'])`
Epenthesis	`dental <=> '+':e <=> s_or_t :- filter(X,[X::mhead:epenthese==='+'])`

Figure 5.8: Filter Rules

Interaction between syntactic and morphological processes takes place at the word level. The application of the two-level rules relating the surface string (i.e the PHON-value of the *word*) and the lexical-string (i.e. MORPH|MSTRING) is also triggered here. This interaction is completely neutral with respect to the direction of processing due to its relational nature. Parsing is performed by simply instantiating the PHON value. Generation can be achieved when MORPH|MSTRING is present, which in turn is obtained by concatenating the lexical strings of the *msigns* instantiated by the morph grammar.

As a result of this constraint interaction the structure shown in Fig. 5.9 is obtained.

Features relevant at the syntactic level (such as PERSON and TENSE) are percolated from MORPH|MHEAD to SYNSEM|LOC|CAT|HEAD via structure sharing constraints attached to

$$
{}_{\mathit{word}}\left[\begin{array}{ll}
\textsc{phon} & \text{"}\mathit{r\ddot{a}t}\text{"} \\
\textsc{morph} & {}_{\mathit{rightfunctor}}\left[\begin{array}{ll}
\textsc{mstring} & \text{"}\mathit{rAt{+}t}\text{"} \\
\textsc{stem} & \boxed{1}\ \text{"}\mathit{rat}\text{"} \\
\textsc{affix} & \text{"}\mathit{+t}\text{"} \\
\textsc{mhead} & {}_{\mathit{verb\_form}}\left[\begin{array}{ll}
\textsc{epenthese} & \boxed{3}\ - \\
\textsc{person} & \mathit{pers3} \\
\textsc{tense} & \mathit{tense\_pres} \\
\textsc{umlaut} & \boxed{2}\ +
\end{array}\right] \\
\textsc{arg} & {}_{\mathit{marg}}\left[\begin{array}{ll}
\textsc{mstring} & \text{"}\mathit{rAt}\text{"} \\
\textsc{stem} & \boxed{1} \\
\textsc{mhead} & {}_{\mathit{verb\_stem}}\left[\begin{array}{ll}
\textsc{epenthese} & \boxed{3} \\
\textsc{person} & \mathit{pers3} \\
\textsc{umlaut} & \boxed{2}
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 5.9: Result of Constraint Interaction (simplified)

the type *word* (this interaction is not shown in Fig. 5.9). Information on subcategorization and semantic content for the word is obtained from the lexeme lexicon using MORPH|STEM as a key. These constraints complete the interaction between syntactic and morphological processing at the word-level.

5.2 Case Assignment and Argument Reduction Phenomena in German

The second application of the constraint based feature unification machinery presented in chapter 4 deals with phenomena of case alternation in German (such as, e.g., in passive constructions). These case alternations can be tracked down to three different sources influencing them:

1. the syntactic context (especially the syntactic category of the head of the argument),
2. the distinction between *structural* and *inherent* argument positions, and
3. argument reduction phenomena induced by morphological processes.

Following the lines of Heinz and Matiasek (1994) it will be shown that a correct treatment of these phenomena can be obtained by a combination of

- a syntactic case principle determining case assignment in a way sensitive to the syntactic context and

- morphological principles affecting the argument structure of lexical items sensitive to their (morphological) word-form.

However, the treatment of the morphological level differs from the one given in Heinz and Matiasek (1994) in that it does not rely on lexical rules, whose theoretical status is not quite clear, but, instead, makes use of the morphology interface described in 5.1.

First, an outline of the problem, together with some data will be presented, then the solution of Heinz and Matiasek (1994) will be discussed. Finally, argument reduction phenomena will be accounted for by the construction of the SUBCAT list of *word*s by mechanisms incorporated in the morph grammar, dispensing with the need for lexical rules. The implementability of the proposed solution will be exemplified.

5.2.1 Case in German

In HPSG argument structure is encoded in the SUBCAT list which specifies how a phrasal head combines with its complements. Case assignment (including assigning nominative to subjects) is treated as part of subcategorization; "no separate theory of case" is provided (Pollard and Sag, 1994, chap. 1). This approach fits well with arguments exhibiting the same case in all syntactic constructions (usually called *lexical* or *inherent case*), but makes it difficult to cope with complements showing variations of case depending on the syntactic context (usually referred to as *structural case*). This deficiency becomes apparent with languages like German, where case is marked by morphological means.

German makes use of different morphological case forms to express the relation between a predicate and its thematic dependents. The morphological case form of a complement NP is determined by the element representing the predicate:

(1) a. Der Mann sucht den Hund
The man$_{nom}$ looks for the dog$_{acc}$
'The man is looking for the dog'

b. Der Mann ist mir nicht bekannt
The man$_{nom}$ is me$_{dat}$ not known
'The man is not known to me'

c. Der Freund meines Vaters
The friend$_{nom}$ my father$_{gen}$
'My father's friend'

In (1), *nom, gen, dat* and *acc* stand for *nominative, genitive, dative* and *accusative* case, respectively. The elements representing the predicate and determining the case forms are of the category *Verb* in (1a), *Adjective* in (1b) and *Noun* in (1c).

Some morphological case forms alternate with the syntactic context, others do not. If case varies according to the syntactic environment we speak of *structural* case:

(2) a. Der Installateur kommt
The plumber$_{nom}$ comes
'The plumber is coming'

b. Der Mann läßt den Installateur kommen
The man lets the plumber$_{acc}$ come
'The man orders the plumber to come'

c. das Kommen des Installateurs
the coming the plumber$_{gen}$
'the coming of the plumber'

In (2a) the agent of *kommen, der Installateur*, is assigned nominative case, whereas *Installateur* receives accusative case marking in the causative construction of (2b) and genitive in the nominalization environment of (2c).

If case remains constant in different syntactic environments we speak of *lexical* case. Examples are given in (3).

(3) a. Der Mann ist mir nicht bekannt
The man$_{nom}$ is me$_{dat}$ not known
'The man is not known to me'

b. des mir nicht bekannten Mannes
the me$_{dat}$ not known man$_{gen}$
'(of) the man not known to me'

Mir is assigned dative case irrespective of the position of the adjective phrase as part of a noun phrase or a sentence.

In addition to morphological case forms lexical case may also be expressed using a preposition stripped of its inherent meaning. As the examples above show, the four morphological cases of German appear in varying distributions, which are not predictable by lexical means alone.

Another apparent example of case alteration is the passive. Consider for example the sentences in (4) and (5), which involve the semantically similar verbs *helfen* and *unterstützen*.

(4) a. Der Berater unterstützt den Kunden
The consultant$_{nom}$ supports the customer$_{acc}$

b. Der Kunde wird unterstützt
The customer$_{nom}$ is supported

(5) a. Der Berater hilft dem Kunden
The consultant$_{nom}$ helps the customer$_{dat}$

b. Dem Kunden wird geholfen
The customer$_{dat}$ is helped

c. *Der Kunde wird geholfen
The customer$_{nom}$ is helped

The active sentences (4a) and (5a) both exhibit the nominative subject (*der Berater*), in (4a) the patient of the helping action (the customer, *Kunde*) exhibits accusative case, but in (4b) dative case. Passiviziation in both cases results in a reduction of the former "subject" (i.e. the consultant). But while in (4b) the former accusative object (*Kunde*) promotes to subject bearing nominative case, in (5b) no subject appears at all, the patient of the helping action retains dative case; (5c) is clearly ungrammatical.

A unified treatment covering not only correct case assignment according to the syntactic construction, but also accounts for auxiliary selection, raising and control verbs, argument reduction phenomena and "parasitic" verbs such as *bekommen* is given in Heinz and Matiasek (1994). I will first describe the basic ideas of this treatment and then develop a treatment of argument reduction in the context of the morphology component described in section 5.1.

5.2.2 The Structure of Argument Structure

The HPSG Approach to Argument Structure

HPSG encodes the argument structure of lexical items in the SUBCAT list containing partially specified *synsem*-values serving as constraints on the possible role-fillers. These constraints comprise not only syntactic restrictions (such as head category or case) but may hold also semantic restrictions on the arguments. Furthermore, the semantic contribution of the arguments to the semantics of the mother is specified via structure sharing, thus replacing the notion of θ-marking in government-binding theory (Chomsky, 1981).

In HPSG the semantic content of a sign is formulated using the framework of Situation Semantics. The CONT value of a verbal lexical entry is given by a *parameterized state-of-affairs (psoa)*. A *psoa* consist of a relation, arguments labeled by their role names, and a polarity.[4] An example of the CONT value of the lexical entries for the verbs *helfen* and *unterstützen*, both meaning 'to help', is given in (5).

(6) $$\begin{bmatrix} \text{REL} & \textit{help} \\ \text{HELPER} & \boxed{1} \\ \text{HELPED} & \boxed{2} \end{bmatrix}_{\textit{psoa}}$$

The CONT value of nominals (sort *nom-obj*) also contains psoas as elements of the RESTRiction value, in addition to the INDEX value. For example, the CONT value of *Hund* ('dog') is:

[4]The polarity will be left unexpressed in examples where its value is *yes*.

(7)
$$\left[\begin{array}{ll}\text{INDEX} & \boxed{1}\\ \text{RESTR} & \left\{\left[\begin{array}{ll}\text{REL} & \textit{dog}\\ \text{INST} & \boxed{1}\end{array}\right]\right\}\end{array}\right]_{\textit{nom-obj}}$$

The role labels used here are unique to the relation. Following the practice of Pollard and Sag (1994), chap. 7, and Sag and Pollard (1991), developed for the semantic characterization of control relations, relations may be of different types.[5] The relation in (6) would be of the more general type *action* with an AGENT role fulfilled by the HELPER.

The linking of syntactic and semantic argument structure originates at the lexical level. A lexical sign specifies the links by structure sharing of parameter values of NPs in the SUBCAT list with parameter values in its CONT value. Lexical signs for *hilft* and *unterstützt* combining syntactic and the semantic structures given above are shown in Fig. 5.10[6].

$$\textit{hilft:}\quad \left[\begin{array}{ll}\text{PHON} & \langle \textit{hilft}\rangle\\ \text{SYNSEM}|\text{LOC} & \left[\begin{array}{ll}\text{CAT} & \left[\begin{array}{ll}\text{HEAD} & \textit{verb}\,[\text{VFORM}\ \textit{fin}]\\ \text{LEX} & +\\ \text{SUBCAT} & \langle\, \text{NP}[\textit{case}]_{\boxed{1}},\ \text{NP}[\textit{dat}]_{\boxed{2}}\,\rangle\end{array}\right]\\ \text{CONT} & \left[\begin{array}{ll}\text{REL} & \textit{help}\\ \text{HELPER} & \boxed{1}\\ \text{HELPED} & \boxed{2}\end{array}\right]\end{array}\right]\end{array}\right]$$

$$\textit{unterstützt:}\quad \left[\begin{array}{ll}\text{PHON} & \langle \textit{unterstützt}\rangle\\ \text{SYNSEM}|\text{LOC} & \left[\begin{array}{ll}\text{CAT} & \left[\begin{array}{ll}\text{HEAD} & \textit{verb}\,[\text{VFORM}\ \textit{fin}]\\ \text{LEX} & +\\ \text{SUBCAT} & \langle\, \text{NP}[\textit{case}]_{\boxed{1}},\ \text{NP}[\textit{case}]_{\boxed{2}}\,\rangle\end{array}\right]\\ \text{CONT} & \left[\begin{array}{ll}\text{REL} & \textit{help}\\ \text{HELPER} & \boxed{1}\\ \text{HELPED} & \boxed{2}\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 5.10: Two Lexical Entries

Phrasal signs that contain these lexical signs as daughters inherit the links by the principles of grammar (Subcat Principle, Semantics Principle).

[5] Technically, as noted in Pollard and Sag (1994), chap. 8, not the *relations*, but the *psoas* are subtyped. Then the REL attribute is redundant.

[6] $\text{NP}[\textit{case}]_{\boxed{i}}$ is an abbreviation for the structure:

$$\left[\begin{array}{ll}\text{LOC} & \left[\begin{array}{ll}\text{CAT} & \left[\begin{array}{ll}\text{HEAD} & \textit{noun}[\text{CASE}\ \textit{case}]\\ \text{SUBCAT} & \langle\,\rangle\end{array}\right]\\ \text{CONT}|\text{INDEX} & \boxed{i}\end{array}\right]\end{array}\right]_{\textit{synsem}}$$

You may have noticed, that only for the argument corresponding to the HELPED role of *hilft* case (i.e. *dat*) has been explicitly specified, since this argument has undergone no case alternation in (5). Case for all other arguments cannot be specified on the lexical level. But, indeed, *case*—subsuming all cases—is no adequate specification, so something has to be done to arrive at correct restrictions.

More Structure to Argument Structure

An approach within the general GB-framework that aims at an explanation of these phenomena is advocated in Haider (1985; 1986).

Haider "propose[s] a way of handling case that depends crucially on the distinction between case indices, supplied by a lexical element, and their realization in the syntactic structure" (cite[sect. 5]Haider85case. He relates θ-assignment and case by the

(8) *Case Criterion* (Haider, 1985, p. 8)
Case is a function that maps θ-roles onto arguments.

This function is composed of two subfunctions: (1) θ-roles to case indices, (2) case indices to NPs. In addition, there is a *realization function* that is a morphological or structural 'spelling-out' of the case index at the NP.

He also proposes a condition that constrains the realization of case (Haider, 1985, sect. 3.3)—the *Realization Principle*—requiring that, if an element assigns structural indices then one of these indices has to be realized externally.

Haider's distinction between *structural* and *lexical* case indices is made in terms of how rigidly case is assigned with respect to structural context: *structural* cases vary with the context, *lexical* cases do not. Structural indices are realized under government conditions, lexical indices as specific morphological cases. For reasons related to realization (one structural index has to be realized externally), he introduces syntactically internal and external arguments. On the thematic side, he borrows the notion of semantically external argument introduced in (Williams, 1981). To factor out the semantically external argument that has to become the syntactically external argument he uses the notion of *designated* argument. With his rules of case assignment under government (Verbs assign accusative, Infl assigns nominative) and the argument structures sketched above he accounts for a wide range of data.

Although the mechanism proposed by Haider is not directly transferable to an HPSG approach (it relies on structural notions like government, and it uses the empty element PRO), Heinz and Matiasek (1994) take his ideas as a starting point for their treatment of case assignment in the HPSG framework.

Structuring the Case Values

For their treatment of case assignment, Heinz and Matiasek (1994) take up the distinction between syntactic case indices and their morphological realization. The four morphological cases—and therefore possible feature values—of German are *nom(inative), gen(itive), dat(ive)* and *acc(usative)*.

Syntactically, the cases are *str(uctural)* or *lex(ical)*. *Nom, acc* (in verbal environments) and *gen* (in nominal environments) may be *str* as exemplified by (2). Cases which may be *lex(ical)* are *gen, dat* and *acc*[7].

Although HPSG theory opposes very strictly against basic assumptions of GB theory, the distinction between structural and lexical case has been used informally by HPSG-practitioners such as Kiss (1991) and Kathol (1991). In Nerbonne, Netter, and Pollard (1994) even Pollard (1994)

> "[...] concludes with some proposals for incorporating into HPSG theory a distinction analogous to the one drawn in government-binding theory between inherent and structural case."

The twofold distinction—syntactic and morphological case on the one hand, and structural and lexical case on the other hand—leads to a sort lattice of case values shown in Fig. 5.11, that reflects the syntactic and morphological properties of case (thus subsuming Haider's case indices and their morphological realization).

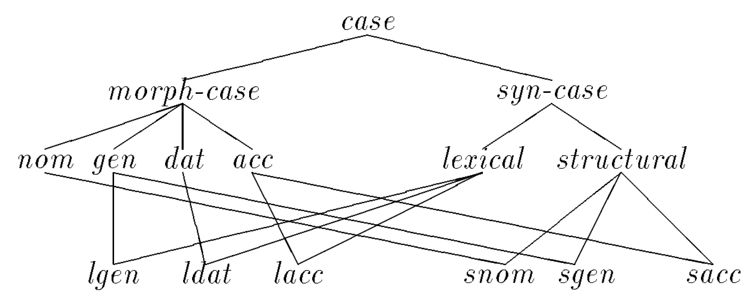

Figure 5.11: Lattice of Case Values

So the possible values of the CASE feature are atoms (types without attributes, cf. Pollard and Sag (1987), p. 197) from the sort lattice in Fig. 5.11.

To achieve a correct case assignment for the sentences given in (4) and (5), the lexical entries for *hilft* and *unterstützt* (see Fig. 5.10) have to be adjusted: the underspecified

[7]For arguments supporting the existence of lexical accusative cf. Heinz and Matiasek (1994).

CASE attribute of the structural arguments has to be given a value of *str*. Case realization then has to be performed depending on the syntactic context, i.e. by means of syntactic principles. This issue will be dealt with below.

The Case Principle

The specification of the CASE value for an NP[*str*] hinges on the syntactic environment, as noted above. We have observed that finite verbs assign nominative case to their 'subject', i.e. the least oblique argument with structural case, and accusative to their 'direct object', i.e. the second least oblique argument with structural case.[8]

Using the standard SUBCAT list definition, Haider's notions of *(syntactically) external* and *(syntactically) internal* argument can be redefined without recourse to configurational notions alien to HPSG.

Syntactically External Argument ('Subject'):
If the first element of the SUBCAT list of a sign is an NP[*str*], it is called the *(syntactically) external argument* of that sign.

Syntactically Internal Argument ('Direct Object')
If the second element of the SUBCAT list of a sign is an NP[*str*], it is called the *(syntactically) internal argument* of that sign.

Although the notions *external* and *internal* argument are not entirely descriptive of the properties of the arguments in that formulation (there is no tree-like structure in the SUBCAT list, both arguments are 'inside' the SUBCAT list), Heinz and Matiasek (1994) nevertheless prefer them to the notion *Subject* (and *Direct Object*) as there is a crucial difference to the SUBJ feature used in newer versions of HPSG.

With these notions an informal characterization of the *Case Principle* can be given:

Case Principle:
In a *head-complement-structure* of category

verb[fin]	the external argument has a CASE value of *snom*,
verb	the internal argument has a CASE value of *sacc*,
noun	the internal argument has a CASE value of *sgen*.

These are the only saturated or almost saturated *head-complement-structures* with structural arguments.

In terms of feature descriptions, the Case Principle is given by the 3 sub-principles shown in Fig. 5.12[9]. These sub-principles are accompanied by complementary structures

[8] A note on the obliqueness hierarchy: arguments with structural case are less oblique than arguments with lexical case, on obliqueness itself cf. Pollard and Sag (1987).

[9] The ellipsis in the list representation stands for an under-specified value of the REST attribute when the list is fully spelled out using the FIRST/REST notation (cf. Pollard and Sag (1994), chap. 1).

$$\begin{bmatrix} \textsc{synsem} & \begin{bmatrix}\textsc{loc}|\textsc{cat} & \begin{bmatrix}\textsc{head} & \begin{bmatrix}\textsc{vform}\ fin\end{bmatrix}_{verb} \\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\end{bmatrix} \\ \textsc{dtrs} & \begin{bmatrix}\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, \text{NP}[str], \ldots\rangle\end{bmatrix}_{h\text{-}c\text{-}str}\end{bmatrix}$$

$$\Longrightarrow \begin{bmatrix}\textsc{dtrs}|\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, \text{NP}[snom], \ldots\rangle\end{bmatrix}$$

$$\begin{bmatrix} \textsc{synsem} & \begin{bmatrix}\textsc{loc}|\textsc{cat} & \begin{bmatrix}\textsc{head} & verb \\ \textsc{subcat} & \langle\,\rangle \vee \langle\, synsem\,\rangle\end{bmatrix}\end{bmatrix} \\ \textsc{dtrs} & \begin{bmatrix}\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, synsem,\ \text{NP}[str], \ldots\rangle\end{bmatrix}_{h\text{-}c\text{-}str}\end{bmatrix}$$

$$\Longrightarrow \begin{bmatrix}\textsc{dtrs}|\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, synsem,\ \text{NP}[sacc], \ldots\rangle\end{bmatrix}$$

$$\begin{bmatrix} \textsc{synsem} & \begin{bmatrix}\textsc{loc}|\textsc{cat} & \begin{bmatrix}\textsc{head} & noun \\ \textsc{subcat} & \langle\,\rangle \vee \langle\, synsem\,\rangle\end{bmatrix}\end{bmatrix} \\ \textsc{dtrs} & \begin{bmatrix}\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, synsem,\ \text{NP}[str], \ldots\rangle\end{bmatrix}_{h\text{-}c\text{-}str}\end{bmatrix}$$

$$\Longrightarrow \begin{bmatrix}\textsc{dtrs}|\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, synsem,\ \text{NP}[sgen], \ldots\rangle\end{bmatrix}$$

Figure 5.12: Case Principle

excluding reduction of the SUBCAT list in other cases; i.e. for the external argument only finite verbal heads are legal (see. Fig. 5.13).

$$\begin{bmatrix} \textsc{synsem} & \begin{bmatrix}\textsc{loc}|\textsc{cat} & \begin{bmatrix}\textsc{head} & \neg \begin{bmatrix}\textsc{vform}\ fin\end{bmatrix}_{verb} \\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\end{bmatrix} \\ \textsc{dtrs} & \begin{bmatrix}\textsc{head-dtr}|\textsc{synsem}|\textsc{loc}|\textsc{cat}|\textsc{subcat}\ \langle\, \text{NP}[str], \ldots\rangle\end{bmatrix}_{h\text{-}c\text{-}str}\end{bmatrix} \Longrightarrow \bot$$

Figure 5.13: Blocking Improper Assignment

From this formulation of the case principle follow some desirable properties as an immediate consequence. First of all it overcomes the impossibility of correct case assignment by lexical means alone (for a detailed discussion cf. Heinz and Matiasek (1994)). Furthermore, since only finite verbal heads may assign case to their external argument, any other category has either to leave its external argument unexpressed (e.g. subjects of controlled infinitival complements) or let it receive case indirectly by being part of an attracted SUBCAT list (e.g. in raising constructions).

The implementation of the Case Principle parallels the specification given in Fig. 5.12 and is shown below in Fig. 5.14.

```
case_principle(X) :-
   X::=head_comp_phrase
  ===>
   assign_extarg_v(X), assign_intarg_v(X), assign_intarg_n(X),
   block_extarg_not_vfin(X).

assign_extarg_v(X) :-
   X::synsem:loc:cat:head:vform===fin,
   X::synsem:loc:cat:subcat===elist,
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===str
 ===>
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===snom.

assign_intarg_v(X) :-
   X::synsem:loc:cat:head===verb,
   ( X::synsem:loc:cat:subcat===elist ; X::synsem:loc:cat:subcat:rest===elist),
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===str
===>
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===sacc.

assign_intarg_n(X) :-
   X::synsem:loc:cat:head===noun,
   ( X::synsem:loc:cat:subcat===elist ; X::synsem:loc:cat:subcat:rest===elist),
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===str
 ===>
  X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===sgen.

block_extarg_not_vfin(X) :-
   X::synsem:loc:cat:subcat===elist,
   X::dtrs:head_dtr:synsem:loc:cat:subcat:first:loc:cat:head:case===str
 ===>
   X::synsem:loc:cat:head:vform===fin.
```

Figure 5.14: Implementation of the Case Principle

The `case_principle` applies the three sub-principles together with the blocking condition in parallel. However, the blocking condition (`block_extarg_not_vfin`) is formulated somewhat differently, since there is no negation operator available in preconditions. It states, that, whenever an external argument (i.e. the first element of the SUBCAT list if it bears structural case) is filled, the head of the phrase is required to be a finite verb. Note, that the working of the Case Principle has to be considered together with the other principles of grammar, esp. the Head Feature Principle and the Subcat Principle, ensuring the percolation of the head features and the proper relation between the mother's and the head-daughter's SUBCAT lists and the list of complement daughters.

Designated Argument Reduction and the Passive

While it is pretty straightforward to see, how case assignment operates for active verbal forms in simple tenses like present or past, where the main verb itself exhibits the finite form, other forms such as the perfect tense or the passive need further explanations. Both, the passive and the perfect tense involve a combination of an auxiliary verb (*haben* or *sein* for perfect tense, *werden* for the passive) and the past participle of the main verb.

Consider the example sentences given in (9).

(9) a. Der Berater hat den Kunden unterstützt
The consultant$_{nom}$ has the customer$_{acc}$ supported
The consultant has supported the customer

b. Der Kunde wird unterstützt
The customer$_{nom}$ is supported

The past participle *unterstützt* supplies the argument structure, but–according to the Case Principle—the nominative case for the external argument has to be assigned by the finite verb, i.e. the auxiliary. So three questions arise, which have to be accounted for:

1. How does the auxiliary access the argument structure of the participle?
2. How is the argument reduction in the passive case (9b) performed?
3. How can be accounted for the fact, that the apparently different argument structures for the active sentence (9a) and the passive sentence (9b) can arise from the same past participle, given that the past participle in German never exhibits different morphological forms for one and the same root verb. To postulate a systematic ambiguity would clearly contradict the observable facts.

Regarding the structure of the auxiliaries, Heinz and Matiasek (1994) follow the approach of Hinrichs and Nakazawa (1990), which now has become standard practice (cf. Pollard (1991) and Kiss (1991)). Auxiliaries *attract* the SUBCAT list of the verb form they subcategorize for. That means, that their SUBCAT list is defined in terms of the SUBCAT list of one of its members. The basic mechanism is shown in Fig. 5.15[10].

$$\left[_{cat}\ \text{SUBCAT}\ \boxed{1} \oplus \langle [\text{LOC}|\text{CAT}|\text{SUBCAT}\ \boxed{1}] \rangle \right]$$

Figure 5.15: Schema of Argument List Attraction

[10] $\oplus$ stands for *append* of feature structure lists.

To account for the reduction facts in (9), where in (9a)both the external (with nominative case) and the internal argument (with accusative case) are realized, whereas in (9b) the external argument does not appear and the internal argument bears nominative, a structure for the past participle enabling both constructions has to be developed. The key idea is that the verb that subcategorizes for a participle is also responsible for the correct execution of the argument structure. But there is too little information in the SUBCAT list alone to correctly select the argument structure (consider, e.g., non-passivizable or ergative verbs). Therefore additional information has to be provided—singling out an argument and designating it as the 'subject' or external argument. Heinz and Matiasek (1994), taking up Haider's (1985) notion of *designated argument*, show that the use of a single additional feature, DA, is sufficient for that purpose (as opposed to approaches by Pollard (1991) and Kathol (1991), who introduce a SUBJ and a ERG feature for that purpose).

The idea behind the *designated argument* is to identify the NP[*str*] that has the 'subject' properties and not the 'object' properties. Thus lexical entries for basic verb forms have either a single-valued DA feature whose single element is structure shared by the first element of the SUBCAT list (transitives and 'normal' intransitives) or the empty list as a DA value (ergative intransitives and verbs without an NP[*str*] as, e.g. the German verb *grauen*).

The lexical entry for the base form of *unterstützen* (omitting the semantic part, but introducing the DA) now looks as shown in Fig. 5.16.

$$\left[\begin{array}{ll} \text{PHON} & \langle \textit{unterstütz} \rangle \\ \text{SYNSEM}|\text{LOC} & \left[\begin{array}{ll}\text{CAT} & \left[\begin{array}{ll} \text{HEAD} & \underset{verb}{}[\text{VFORM } bse] \\ \text{DA} & \langle \boxed{1} \rangle \\ \text{SUBCAT} & \langle \boxed{1}\text{NP}[str],\ \text{NP}[str] \rangle \end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 5.16: Lexical Entry for *"unterstützen"*

Given that structuring of argument structure and the fact, that only a designated argument may undergo reduction (for further data and explanation cf. Heinz and Matiasek (1994)), now a rule for the derivation of the past participle from the base form of a verb can be stated. Heinz and Matiasek (1994) give this rule in the form of a lexical rule of HPSG, an alternative approach in terms of the morphology module of section 5.1 will be given in section 5.2.3. The basic idea of the reduction rule is, that only a designated argument may be removed from the SUBCAT list. But in that case it still remains in the DA slot and can be recovered by the perfect auxiliary. If, one the other hand, no designated argument exists, then no reduction can take place and the SUBCAT list remains unchanged. This method also accounts nicely for the different behavior of ergative and non-ergative verbs, when their past participles are used as noun modifiers (consider, e.g., *der geschlagene Hund* vs. *der aufgewachte Hund*), given that modification requires identity of the indices of the modified noun and the first structural argument of the SUBCAT list.

The Past Participle Rule as stated by Heinz and Matiasek (1994) is shown in Fig. 5.17.

$$\begin{bmatrix} \text{HEAD} & {}_{verb}\begin{bmatrix}\text{VFORM } bse\end{bmatrix} \\ \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{1} \oplus \boxed{2} \end{bmatrix} \longmapsto \begin{bmatrix} \text{HEAD} & {}_{verb}\begin{bmatrix}\text{VFORM } ppp\end{bmatrix} \\ \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{2} \end{bmatrix}$$

Figure 5.17: Past Participle Lexical Rule

It remains to be shown, how the auxiliaries get along with this structure of the past participle to produce the correct past tense or passive forms. The perfect auxiliaries (*haben* or *sein*) have to "reinsert" the designated argument, which has been removed from the SUBCAT list during past participle formation, the passive auxiliary (*werden*) simply executes the SUBCAT list of the participle (provided that the DA slot is nonempty, since ergative verbs are not passivizable). The respective category values for the auxiliaries are shown in Fig. 5.18.

Perfect *(haben,sein)*

$${}_{cat}\begin{bmatrix} \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{1} \oplus \boxed{2} \oplus \left\langle \begin{bmatrix} \text{LOC}|\text{CAT} & \begin{bmatrix} \text{HEAD} & {}_{verb}\begin{bmatrix}\text{VFORM } ppp\end{bmatrix} \\ \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{2} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

Passive *(werden)*

$${}_{cat}\begin{bmatrix} \text{DA} & \langle\rangle \\ \text{SUBCAT} & \boxed{1} \oplus \left\langle \begin{bmatrix} \text{LOC}|\text{CAT} & \begin{bmatrix} \text{HEAD} & {}_{verb}\begin{bmatrix}\text{VFORM } ppp\end{bmatrix} \\ \text{DA} & \langle\ synsem\ \rangle \\ \text{SUBCAT} & \boxed{1} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix}$$

Figure 5.18: Category Structure of Auxiliaries

Note that perfect tense can be formed regardless of the presence of a designated argument. If a verb does not exhibit one, then nothing could have been removed from the SUBCAT list during past participle formation, and appending the empty list contained in the DA slot to the SUBCAT list of the auxiliary does no harm. Further evidence on the usefulness of this approach to case assignment and argument structure, especially in the context of raising and control, can be found in Heinz and Matiasek (1994).

5.2.3 Argument Reduction in a Morphological Context

One drawback of this approach in its original formulation is the use of lexical rules to capture argument reduction phenomena. Although lexical rules are part of standard HPSG theory, their theoretical status is not fully clear. Especially if one employs a HPSG-inspired grammar for the description and implementation of morphological word-formation processes it is not at all clear, at which level of the grammar these lexical rules should enter the picture. At the level of the morph lexicon there is nothing left to do for these rules. At the boundary between the word level and the sentence level grammar (at the sign type *word*) would be the place where they would apply in a grammar employing a traditional lexicon. But here the application of lexical rules turns out to be rather problematic, since these rule carry out *transformations* on the lexical items. So, if one wants to make use of lexical rules *and* a morphology component based on a word-level grammar, either the smooth integration which has been shown to be obtainable (see section 5.1) has to be given up, or—even worse—destructive operations have to be made possible, thereby sacrificing the monotonicity properties of the formalism.

The most promising solution to this dilemma is to not make use of lexical rules at all and employ other methods to obtain the desired effects. The key idea here is not to rely on the *destructive* effects which are characteristic for lexical rules, but to include *constructive* mechanisms in the word-level grammar, which build up the desired structure depending on the morphs being combined. Similar proposals have been made by Krieger (1991), Krieger and Nerbonne (1991) and by Riehemann (1993).

In the following section I will demonstrate, that this is a feasible approach, and present a reformulation of the lexical rule given in Fig. 5.17 in terms of the word-level grammar sketched in section 5.1.

Argument Reduction vs. Argument List Construction

To start with, recall the structure of the type *word* as proposed in section 5.1, interfacing between the word-level and the sentence-level grammar. The values of PHON (the surface string) and of MORPH|MSTRING (the lexical string) are related via the two-level relation (see the morphology constraint shown in Fig. 5.5). The syntactically relevant features of MORPH|MHEAD need to be percolated to SYNSEM|LOC|CAT|HEAD of the *word* and the values for the SUBCAT list and the DA slot of word have to be derived somehow from the MORPH feature. For that purpose it is necessary to enrich the *msign* type with features holding argument information. The basic argument information has to be stored with the morph-signs of the stems in the lexicon, the affixes (i.e. the functors operating on the stems and constructing the various word forms) have to be enriched with information, how the stems argument list is percolated to the morphological phrase. Most affixes will not change the argument structure (case suffixes, e.g., just pass through the argument list of the stem), but some will alter it, most notable the suffixes *+t* and *+n* in their reading as past participle constructors.

$$
{}_{word}\begin{bmatrix}
\text{PHON} & \textit{string} \\
\text{SYNSEM}|\text{LOC}|\text{CAT} & {}_{cat}\begin{bmatrix} \text{HEAD} & \textit{head} \\ \text{DA} & \textit{synsemlist} \\ \text{SUBCAT} & \textit{synsemlist} \end{bmatrix} \\
\text{MORPH} & {}_{msign}\begin{bmatrix} \text{MSTRING} & \textit{string} \\ \text{STEM} & \textit{string} \\ \text{MHEAD} & \textit{mhead} \end{bmatrix}
\end{bmatrix}
$$

Figure 5.19: AVM for a Sign of Type *word*

While in the lexical rule approach words are treated as undivisible units and simply being transformed from one word-form to another, the approach advocated here allows for a much more fine-grained specification of particular effects. Each morph which acts as a functor may specify which effects it induces on the SUBCAT structure of its argument. Thus the subcategorization frame of a word can be viewed as the result of a constructive process driven by the affixes involved in building the particular word form. Not only reduction phenomena can be captured that way, but also a treatment of derivation and even composition is possible.

In order to be capable of holding the content and subcategorization information, that originates with the stem of a word and is possibly modified via affixation or derivation, morphological signs are enriched with a feature SEM-ARGS[11]. The CONT slot holds the

$$
{}_{sem\text{-}args}\begin{bmatrix} \text{CONT} & \textit{cont} \\ \text{DA} & \textit{synsemlist} \\ \text{SUBCAT} & \textit{synsemlist} \end{bmatrix}
$$

Figure 5.20: Semantics and Argument Structure Specification in *msigns*

contents, which should finally appear in the SYNSEM|LOC|CAT|CONT slot of the *word*, the

[11]The different representation of syntactic and morphosyntactic information, i.e. the different structuring of *signs* of the sentence-level grammar and *msigns* of the word-level grammar is on purpose. Although other authors (such as Krieger (1991), Krieger and Nerbonne (1991) or Riehemann (1993)) use similar representations for the word and for the sentence level, I consider this as more confusing than clarifying. Since the simple functor/argument structure of the word-level grammar and the phrase structure of sentence level phrases with their different types of daughters have only little in common, and furthermore different sets of principles apply at these two strata, this distinction seems warranted. Also the semantic and subcategorization information at the morph level does not contribute to the phrase structure at that level but is rather a data structure modifiable by morphological processes. For all these reasons, this somewhat nonstandard representation is well motivated.

SUBCAT and the DA slot of *sem-args* correspond to the ones to be found in SYNSEM|LOC|CAT of the *word*, but are subject to possible modifications triggered by morphological processes.

The lexical entry of the base-form (i.e. the stem) of the German verb *unterstützen* as used in the examples of section 5.2.3 is shown in Fig. 5.21[12]. The argument structure of the

$$
{}_{\textit{marg}}\left[\begin{array}{ll}
\text{MSTRING} & \textit{"unterstütz"} \\
\text{STEM} & \textit{"unterstütz"} \\
\text{MHEAD} & {}_{\textit{verb\_stem}}[\ldots] \\
\text{SEM-ARGS} & {}_{\textit{sem-args}}\left[\begin{array}{ll}
\text{CONT} & \left[\begin{array}{ll} \text{REL} & \textit{help} \\ \text{HELPER} & \boxed{2} \\ \text{HELPED} & \boxed{3} \end{array}\right] \\
\text{DA} & \langle \boxed{1} \rangle \\
\text{SUBCAT} & \langle \boxed{1}\,\text{NP}[\textit{str}]_{\boxed{2}},\ \text{NP}[\textit{str}]_{\boxed{3}} \rangle
\end{array}\right]
\end{array}\right]
$$

Figure 5.21: Lexical Entry for Verb-Stem *"unterstütz-"*

base form is the same as with the lexical-rule approach, but the base form itself, being only a morphological stem cannot participate in syntactic processes on the sentence-level, a fact accounted for by the different argument encoding. The *word* based on that stem (i.e. a *verb* with a particular *verb-form*) has to be constructed by applying the word-level grammar first. The morphological functors operating on the stem have not only to provide the correct inflectional affixes but also have to account for the correct passing or modifying of the argument structure. The final transition from the morphological word form to the syntactic *word* is governed by a principle properly installing semantic content and argument structure within SYNSEM|LOC, shown in Fig. 5.22

$$
{}_{\textit{word}}[\,] \quad \Longrightarrow \quad {}_{\textit{word}}\left[\begin{array}{ll}
\text{SYNSEM}|\text{LOC} & \left[\begin{array}{ll} \text{CAT} & \left[\begin{array}{ll} \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{2} \end{array}\right] \\ \text{CONT} & \boxed{3} \end{array}\right] \\
\text{MORPH}|\text{SEM-ARGS} & \left[\begin{array}{ll} \text{CONT} & \boxed{3} \\ \text{DA} & \boxed{1} \\ \text{SUBCAT} & \boxed{2} \end{array}\right]
\end{array}\right]
$$

Figure 5.22: Interfacing Syntactic and Morphological Level

[12]For the sake of simplicity, the composition of the basic verb-stem *stütz* and the prefix *unter* is not considered here.

Argument Reduction via Argument Construction

Now it remains to be shown, how the argument reduction triggered by the formation of the past participle can actually be implemented by a constructive process. Each morphological functor determines by its definition in the morph lexicon not only, how and with what morphological signs it combines morphosyntactically, but also, how it affects the syntactic-semantic argument structure of them. The way in which this argument construction is performed is shown in Fig. 5.23[13]. As can be seen, the effect achieved is the same as

$$
{}_{\textit{rightfunctor}}\begin{bmatrix}
\text{AFFIX} & \text{"}{+t}\text{"} \\
\text{MHEAD} & {}_{\textit{verb\_form}}\begin{bmatrix} \text{VFORM} & \textit{ppp} \\ \ldots & \end{bmatrix} \\
\text{SEM-ARGS} & \begin{bmatrix} \text{CONT} & \boxed{1} \\ \text{DA} & \boxed{2} \\ \text{SUBCAT} & \boxed{3} \end{bmatrix} \\
\text{ARG} & {}_{\textit{msign}}\begin{bmatrix} \text{MHEAD} & {}_{\textit{verb\_stem}}[\ldots] \\ \text{SEM-ARGS} & \begin{bmatrix} \text{CONT} & \boxed{1} \\ \text{DA} & \boxed{2} \\ \text{SUBCAT} & \boxed{2} \oplus \boxed{3} \end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

Figure 5.23: Suffix for Past Participle Formation

using the lexical rule of Fig. 5.17, but without destructive replacement of a word feature structure by another one. Instead, the reduction of the designated argument is performed in a constructive way preserving monotonicity of the formalism.

The actual implementation (shown in Fig. 5.24), however, singles out the designated

```
designated_argument_reduction(X) :-
      X::sem_arg:da        ===DA,
      X::sem_arg:subcat    ===REDUCED_SUBCAT,
      X::arg:sem_arg:da    ===DA,
      X::arg:sem_arg:subcat===SUBCAT,
      fs_append(DA,REDUCED_SUBCAT,SUBCAT).
```

Figure 5.24: Implementation of Designated Argument Reduction

[13]The morphosyntactic constraints on the argument are not shown. $\oplus$ stands for *append* of feature structure lists.

argument reduction rule into a Prolog predicate, which may be attached to the morphs performing such a reduction. Again it is made use of the delay mechanism integrated into the feature formalism developed in chapter 4, but in an indirect fashion, since the relation `fs_append` (cf. Fig. 4.10) is defined in a such a way, that it waits for proper instantiation of its arguments. Therefore the designated argument reduction rule can safely be used for lexical specification of morphological functors.

Chapter 6

Extensions

This chapter describes extensions to the system described in chapter 4, which are expected to further increase the efficiency and the expressive power of the feature unification machinery. These have in part been implemented and tested, but up to now not fully integrated into the system.

The first extension describes a compilation technique for typed feature structures. Although some compilations are already done in the basic system, especially with regard to the type lattice, there are further operations, which can be compiled away. The basic idea underlying this compilation technique is, to compile not only the traversal of the type lattice required for determining the greatest lower bound during unification of atomic type symbols into a table lookup predicate, but to treat unification of whole structures that way. A speed up by a factor of 2 can be achieved by using a technique that represents typed feature structures as Prolog terms and avoids the traversal of attribute-value lists during unification.

The second extension deals with methods to represent disjunctive information in the system. While, from a linguistic viewpoint, it is desirable, to have means for expressing disjunctions within feature structures, disjunctions are often the source of computational inefficiency due to the exponential complexity induced by disjunctive descriptions. However, disjunctions are inherently present with natural language, so methods have to be developed, to cope with them. One method to eliminate disjunctions by replacing them by a conjunction of implications has already been discussed in 4.5.3, but this scheme is useful only in specifying disjunctive filtering conditions, since instantiation of the preconditions must be done somewhere else. Furthermore it maybe too cumbersome to handle simple cases such as the disjunction of atomic values, a case occurring quite often with the specification of lexical entries.

In order to cope with such atomic disjunctions, domain variables will be introduced. It will be shown, that these variables representing finite domains of atomic types can be integrated easily into the feature constraint rewriting framework.

6.1 Compilation

The main idea with compilation is to enhance efficiency by avoiding calculations which can be done beforehand (and then need to be done only *once*), whereas their processing at execution time must be carried out *repeatedly*. A typical candidate for compilation is therefore the type hierarchy. Although all relevant relations between types could be deduced at run time by traversing of the type lattice, efficiency is increased enormously, if frequently used relations between types (as, e.g., subsumption or the greatest lower bound of two types) are calculated beforehand and stored in a table. Then it can be decided in constant time if a certain relation holds or not.

Compilation, especially in a setting like the one sketched above, is always a tradeoff between execution speed and storage requirements. It depends on the ratio between the gain in overall speed and the cost (in terms of compilation time and storage needs of compiled relations), whether it pays off to compile a certain relation at load time or it is cheaper to compute it at run time.

The compilation carried out here is not (as in classical compilers for programming languages) a transformation of some high level input language to a machine language, but a transformation of Prolog code to Prolog code, or simply an augmentation of Prolog code with some additional clauses. The reason why this makes sense is, that some representation of a problem may be well suited for specifying the task, whereas another representation my be advantageous for performing the task. Thus compilation amounts to a transformation of input-oriented relations to more operational predicates.

A similar technique has been employed in the ALE system (Carpenter, 1992a), which has been described in (Carpenter, 1993). The implementation here differs mainly in two aspects from the approach used there: Firstly, the implementation described here is based on a Prolog system with extensible unification. Attributed variables serve as "names" for feature structures and may be updated destructively (but in a backtrackable fashion). So considerations on dereferencing need not to be made, these issues are handled by the underlying Prolog system. Secondly, principles in the form of implicational constraints are not part of ALE, but our system needs to cope with such constraints also during compilation.

6.1.1 Compiling Type Definitions

As already noted in section 4.2.1, the type hierarchy – being an obvious candidate for compilation – is compiled during grammar loading. Recall, that the type hierarchy is defined using the `..>` operator (cf. Fig. 4.2). First, the transitive closure of that operator is computed (using Warshall's algorithm, cf. O'Keefe (1990)) yielding the subsumption relation `..>*` on types. This step also allows for checking the correct definition of the type lattice. It must be noted, however, that every change in the definition of the type lattice itself or of the appropriateness conditions require a recompilation.

The `glb/3` relation is compiled as a table containing the greatest lower bound for each pair of compatible types (i.e. `glb(X,Y,Z)` iff `X..>*Z` and `Y..>*Z` and there is no type `Z0` such that `Z0..>*Z` and `X..>*Z0` and `Y..>*Z0`). To take advantage of first argument indexing, the `glb` relation is split up into specialized relations incorporating the first argument in their name (see in Fig. 6.1, how a part of the case lattice (Fig.5.11) is compiled).

```
scase ..> str.
          str ..> snom.
          str ..> sgen.
          str ..> sacc.
scase ..> lex.
          lex ..> lgen.
          lex ..> ldat.
          lex ..> lacc.
mcase..>  nom.
          nom ..> snom.
          nom ..> lnom.
  ...
```

$\rightsquigarrow$

```
glb(str,X,Y) :- glb_str(X,Y).
glb(lex,X,Y) :- glb_lex(X,Y).
  ...
glb_str(nom,snom).
glb_str(gen,sgen).
glb_str(acc,sacc).

glb_lex(gen,lgen).
glb_lex(dat,ldat).
glb_lex(acc,lacc).
   ...
```

Figure 6.1: Compiling `glb/3`

To facilitate appropriateness checking and type inferencing, relations indicating the permitted slots for a certain type (`Type permits Slot`) and where a certain slot has been introduced (`Slot introduced_in Type`) are computed from the input relation `Type==>[F:V,...]`. An example is given in Fig. 6.2. These relations facilitate the run time computations a

```
sign   ==> [phon:   phon,
             synsem: synsem].
word   ==> [].
phrase ==> [dtrs:const_struc].
```

$\rightsquigarrow$

```
phon   introduced_in sign.

synsem introduced_in sign.
dtrs   introduced_in phrase.
  ...
sign   permits phon.
sign   permits synsem.
sign   permits dtrs.
word   permits phon.
word   permits synsem.
phrase permits phon.
phrase permits synsem.
phrase permits dtrs.
```

Figure 6.2: Compiling Relations for Type Inference

lot. There are, however, further possibilities to speed up calculations. Consider, e.g., Type

Inference: if an attempt is made to add a feature `dtrs` to a feature structure of type `sign`, it can be simply determined that `dtrs` is a slot appropriate for `sign` via the `permits/2` relation. Type inference then has to check which type introduces `dtrs`, i.e. `phrase`, and has to perform the appropriate narrowing of the type. These tasks may also be performed at load time, the type inferencing efforts can also be compiled away.

6.1.2 An Alternative Representation of Typed Feature Structures

Not only type inference, but also unification of feature structures itself may gain efficiency through compilation. Unification of atomic types is already handled that way via the `glb/3` predicate. The problem with compiling unification of structured types lies in their representation as list of feature-value pairs. Two such lists have to be checked against each other during unification and merged properly. But since HPSG imposes a strict type discipline on feature structures and the appropriateness constraints determine for each type, which features it may take, the representation of typed feature structures by fixed arity Prolog terms may be exploited. Unification of two such terms can then be performed efficiently by the built in mechanisms of Prolog. So the basic idea of this approach is to map the feature-value list of a non-atomic type to a fixed arity term, and to provide the necessary predicates needed for accessing values by their slot name or to recover the slot names when printing a features structure.

Representing Typed Feature Structures by Prolog Terms

Typed feature structures are currently represented by the attribute `fs(X,Type,FVL,Goals)`, where `FVL` is a list of feature-value pairs. Since for every type it is exactly determined which features have to be present, this representation is changed to `fs(X,FeatureTerm,Goals)`. The type of the structure is the functor of `FeatureTerm`, its arguments being the slot-values. Functors representing atomic types have arity 0, uninstantiated feature structures (formerly represented by the value `uninstantiated` for `FVL`) are also represented by their type name only. The distinction between atomic and uninstantiated types can thus only be made by checking for the existence of a type definition. However, an explicit representation of uninstantiatedness is not necessary, as all actions depending on this distinction take place within compiled relations, as will be seen below. Since it is known during compilation, if an atomic value of a certain type means really *atomic* or *uninstantiated*, the different actions depending on this distinction are simply compiled into the relation (such as, e.g., instantiating the corresponding type if a slot access is attempted). The representation of feature structures by feature terms is exemplified in Fig. 6.3 (ignoring delayed goals for the moment).

However, this representation is not sufficient for accessing slot-values by their name or for the purpose of printing a feature structure. Therefore, a type definition more explicit

$$\begin{bmatrix} \text{PHON} & \textit{phon} \\ \text{SYNSEM} & \textit{synsem} \\ \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR} & \textit{sign} \end{bmatrix} \end{bmatrix}_{\textit{headed-phrase}} \rightsquigarrow$$

```
fs(X,headed_phrase(Y,Z,U)).
fs(Y,string).
fs(Z,synsem).           % uninstantiated
fs(U,headed_struc(V)).
fs(V,sign).             % uninstantiated
```

Figure 6.3: A Term Representation of Feature Structures

than the original input predicates has to accompany that term-based representation. This definition is accomplished by the predicate `type_def/4` containing entries for the type name, the typing requirements for the slot-values, a template of the feature term itself and a list representation, including the slot names and linking the slot values with the template via variable coreference. The compiled type definitions of the types involved in Fig. 6.3 is shown in Fig. 6.4. This predicate provides all the information necessary for slot access and

```
type_def(headed_phrase,headed_phrase(string,synsem,headed_struc),
         headed_phrase(A,B,C),[phon:A,synsem:B,dtrs:C]).
type_def(synsem, synsem(loc,nonloc),synsem(A,B),[loc:A,nonloc:B]).
type_def(headed_struc,headed_struc(sign),headed_struc(A),[head_dtr:A]).
```

Figure 6.4: Type Definition for Feature Term Representation

printing of a structure. Accessing slots by name, however, involves searching the attribute-value list and is possible directly only with types exhibiting the slot. Thus type inference needs further considerations. But first the consequences of the changed representation for unification will be discussed.

Unification via Table Lookup

The main advantage of the new representation is a reduction of the size of a structure[1] and the possibility to efficiently employ the built-in Prolog unification for feature structures. This does not mean, however, that feature unification can be handled by term unification, the machinery involving the attributed variables is still necessary. But indirect advantages can be gained from the fact, that arity and order of arguments are now standardized for all types, and that the type is the principal functor of the term representing the feature structure.

Remember that unification of two feature structures is triggered by unification of the two attributed variables representing them. But, instead of looking up only the `glb` of the type *symbols* of the two input structures and then (on success) performing

[1]There is only one functor, instead of the previous list involving several `'.'/2` functors plus the `':'/2` operators for each feature-value pair

1. the usual list merging of the attribute value pairs,
2. looking up the type definition of the result type and
3. applying the well-typing constraints resulting from this type definition,

the new representation allows for efficient table lookup of the whole feature terms, since first argument indexing is performed using the principal functor as key. The result contains not only the name of the result type, but the whole new feature term, its arguments properly unified. Since the `glb` table now serves as a template, showing which arguments of the input feature terms have to be unified, no list merging has to be performed and recursive unification can be left to Prolog unification and the attributed variable mechanism. Well-typing constraints as well as instantiations of slot values are attached as Prolog goals to the respective clause if necessary. Furthermore, uninstantiated types need no special treatment during run time, since all necessary instantiations are likewise compiled into the `glb` table.

As an example, consider a subtyping of *inflectional* heads into *nominal* and *verbal* categories, *adj* being both *nominal* and *verbal* (this way the effects of the classification of categories shown in Fig. 2.6 can be obtained).

```
inflectional
   ..> nominal.
       nominal ..> noun.
       nominal ..> adj.
inflectional
   ..> verbal.
       verbal  ..> adj.
       verbal  ..> verb.
 ...
inflectional
   ==> [num:     num,
        per:     per].
nominal
   ==> [case:    case,
        gend:    gend].
```

$\rightsquigarrow$

```
 ...
glb(nominal,verbal,adj).
glb(nominal,verbal(C,D),adj(A,B,C,D)) :-
      fsvar(case,A),
      fsvar(gend,B).
glb(nominal(A,B,C,D),verbal,adj(A,B,C,D)).
glb(nominal(A,B,C,D),verbal(C,D),adj(A,B,C,D)).
 ...
```

Figure 6.5: Unification Lookup Table

As can be seen, the way how arguments have to be unified is specified completely within the head of the `glb` clauses. The handling of uninstantiated types is integrated and appropriateness is maintained on newly introduced slots (cf. the second `glb` clause in Fig. 6.5). Slot values that have not been present before are instantiated with the appropriate type via `fsvar/2`.

In order to obtain indexing for both input arguments, the `glb` relation is again split into several sub-relations specialized for the first argument of the original one (this is not shown here, but see Fig. 6.1).

Slot Access

In a similar fashion, all type inferencing and appropriateness checking required during slot access or feature structure instantiation triggered by slot access can be compiled. To achieve this the `permits` relation of Fig. 6.2 is replaced by a more informative `slot/3` relation between a feature and a type as input arguments and the result type, which must always be instantiated, together with the value the feature has within that result type. This relation not only indicates which slots are permissible for a type, but incorporates also all the necessary type inference (the result type, e.g., must be the greatest lower bound of the input type and the feature introducing type).

How this relation looks like is exemplified in Fig. 6.6, showing some clauses for accessing the `case` slot (based on the type definitions of Fig. 6.5).

```
 ...
slot(case,inflectional,A,nominal(A,B,C,D)) :-
     fsvar(case,A),fsvar(gend,B),fsvar(num,C),fsvar(per,D).
slot(case,inflectional(C,D),A,nominal(A,B,C,D)) :-
     fsvar(case,A),fsvar(gend,B).
slot(case,nominal,A,nominal(A,B,C,D)) :-
     fsvar(case,A),fsvar(gend,B),fsvar(num,C),fsvar(per,D).
slot(case,nominal(A,B,C,D),A,nominal(A,B,C,D)).
slot(case,verbal,A,adj(A,B,C,D)) :-
     fsvar(case,A),fsvar(gend,B),fsvar(num,C),fsvar(per,D).
slot(case,verbal(C,D),A,adj(A,B,C,D)) :-
     fsvar(case,A),fsvar(gend,B).
 ...
```

Figure 6.6: Compiled Slot Access

Again, to achieve indexing also for the second argument, the `slot` relation is split in the same way as the `glb` predicate before.

6.1.3 Compiling Node Licensing

The compilations done so far all had to do with the access and unification of typed feature structures under well-typing constraints. But further constraints that have to be considered are the licensing conditions induced by the principles of grammar. Every node of a feature structure must be licensed by all principles of grammar. The principles themselves are formulated as conditional constraints, the preconditions indicating to what type of node the principle applies. The explicit type requirements as given in the preconditions, which may involve instantiation or subtyping requirements of subexpressions are too fine-grained to be reflected in the type hierarchy, but the minimal type contained in the type hierarchy subsuming the precondition can be deduced (either it is given explicitly or it may be inferred via type inference from the path expressions in the precondition). But by attaching a principle to the minimal type subsuming its preconditions (similar to the approach of Carpenter, Pollard, and Franz (1991)) not very much would be gained (cf. the argumentation in section 4.5). Thus, instead of annotating types with principles, *type transitions* are annotated with exactly those principles, the minimal precondition subsuming type of which is crossed by the type transition.

The type of a node is changed only on two occasions: either during unification or during type inference (when a permissible slot not contained in the type is accessed). The possible type transitions are already contained in compiled predicates covering exactly these two possibilities. Thus each `glb` clause and each `slot` clause is now annotated with the principles potentially applying when this particular clause is executed. Recall that a precondition of the form `X::=type` is satisfied, if `X` represents an instantiated feature structure of type `type` or has a subtype of `type`. Thus, if it has been established during compilation of a principle P, that it applies to a type T, then every `glb` clause and every `slot` clause corresponding to a transition of a supertype (or an uninstantiated version) of T to an instantiated version or a subtype of T has to be annotated with P. Such a transition is a necessary condition for the applicability of P; if P really applies, the principle has to decide itself on its applicability like before. Nevertheless, a lot of unnecessary preconditions checking can be avoided this way, since it is no longer necessary to apply all principles to every node during instantiation.

6.1.4 Summary

We have described two stages of compilation: the first one, incorporated already in the basic system deals only with the type lattice and appropriateness checking. Thus, at run time no need arises to traverse the type lattice, a potentially costly operation. Furthermore, predicates operationalizing some aspects of maintaining well-typedness are generated. Type inference and well-typed unification still has to be done by interpreting the actual structures.

The second compilation stage involves a change in the representation of typed feature structures from a slot-value list to a fixed arity term, the functor being the type name. By virtue of this representational change the benefits of built-in term unification can be exploited. Well-typed unification of feature structures is now possible simply by table lookup. The same holds for slot access and type inference. Although the licensing view of grammar principles is maintained, applying them to all nodes during instantiation is no longer required, instead they are triggered by the precompiled predicates whenever a type transition that has to be licensed by a principle takes place.

These additional compilations have resulted in a performance gain by a factor of 2 as compared with stage one compilation.

6.2 Handling Disjunction within Feature Structures

Constraints on linguistic objects cannot always be stated in a single feature structure, since feature structures represented by DAGs are inherently conjunctive. In order to be able to specify ambiguous lexical entries (such as, e.g., noun forms ambiguous in their CASE value) or ambiguities at the phrasal level (as in *"I saw the man with the telescope"*) some means for expressing disjunctions have to be provided. Negative descriptions may be useful sometimes also (cf. Karttunen (1984)). Nevertheless, the need to express disjunctions is more urgent, so these will be dealt with in the first place.

Handling of disjunctions appears to be a crucial point for the efficient processing of natural language and has therefore attracted a lot of attention in the last few years (for an overview see Trost (1993b)). Disjunctions result from two different sources, namely the grammar and the lexicon. For ambiguities expressed by nondeterministic grammar rules usually enumeration of the different choices via backtracking leads to an acceptable runtime, as the number of such disjunctions is typically rather small. To handle disjunctions coming from the lexicon the same way, usually leads to the combinatorial explosion of search space and consequently of runtime, as serious lexicons tend to contain an abundance of such disjunctions. One can distinguish two types of disjunction, namely *simple* disjunctions of atomic values and *hard* disjunctions of arbitrary feature structures.

There are several possibilities to deal with disjunctions. The one mechanism built into Prolog is *backtracking*. Alternative goals are explored in a left-to-right, depth first manner (at least when employing the built-in search strategy), alternative solutions can be enumerated this way. Depth first search with backtracking, however, may be an inefficient method when processing natural language. Consider a situation where there is a choice point early in the search space, but the correctness of the decision can be decided only upon finishing the processing of the whole sentence. In such a case all the in between processing must be repeated (although its is supposedly completely independent of the choice). To handle such situations more efficiently, some other means are desirable.

One method to cope with that problem has been already introduced. Replacing a disjunction by a conjunction of implicative constraints solves exactly the problem of too early a commitment to a particular choice. In the case sketched above, the choice would be delayed until the final unifications at the end force a decision. Thus the wasted effort of processing the same constituents over and over again can be avoided.

However, this method may turn out to be somewhat cumbersome to use, if one only has in mind to specify a local lexical ambiguity, say two possible (atomic) values for the CASE feature. Here another possibility springs to mind. One could enrich the type lattice to include also disjunctions of values. While this is viable, if only few types need to be expanded that way, as a general solution this would lead to an exponential explosion of the type hierarchy. Furthermore it is unwise to overload the type hierarchy with intermediate types which are not linguistically motivated. Such a strategy may cause confusion, slow down traversing of the lattice and blow up compiled glb-tables.

A further possibility to express disjunctions of atomic types is the use of *finite domains* (cf. van Hentenryck and Dincbas (1986), van Hentenryck (1989)), which will be explained in more detail below.

6.2.1 Finite Domains of Atomic Types

Finite domains are finite sets defining the set of legal values for a given variable (van Hentenryck and Dincbas, 1986; van Hentenryck, 1989). In the context of typed feature structures finite domains can be used for representing legal alternatives for the value of a certain feature. However, the appropriateness restrictions additionally restrict these values. Only subtypes of the type appropriate for a feature may occur in the domain of its value. But within this range of subtypes disjunctions can be specified via domain variables without having to introduce them into the type lattice as special disjunctive types.

Since the specification of disjunctions of instantiated structured types is beyond the scope of domain variables, discussion will be limited to atomic types (i.e. types not allowing for features). This extension has been implemented and has resulted in a significant speed-up in the processing of lexical ambiguities. A possible extension to disjunctions of *uninstantiated* structured types will be discussed later on.

Unification with Domain Variables

Since domain variables may serve as values inside of feature structures, their behavior with respect to unification has to be specified. Furthermore, since they represent disjunctions of *types* standing in a subsumption relation, under some circumstances these disjunctions can be simplified. The rewrite rules are given in Fig. 6.7.

Types subsumed by another type within the domain can be dropped, singleton sets can be replaced by the type which is the only member of the domain, empty domains correspond to failure. Unification of a type t with a domain amounts to the disjunction of the results of unifying each member of the domain with t. Finally, the unification of two domains can be obtained by taking the union of the unification results of each member of one domain with the other domain. Domains resulting from such unifications are of course subject to the simplification rules above (the implementation described below interleaves the two steps).

Simplification Rules

$$\begin{array}{rcl}
\{\} & \rightarrow & \perp \\
\{t\} & \rightarrow & t \\
\{t_1,\ldots,t_i,\ldots,t_j,\ldots,t_n\} & \rightarrow & \{t_1,\ldots,t_i,\ldots,t_n\} \text{ if } t_i \sqsubseteq t_j
\end{array}$$

Unification Rules

$$\begin{array}{rcl}
t \sqcup \{t_1,\ldots,t_n\} & \rightarrow & \{s_i | s_i = t \sqcup t_i\} \\
\{s_1,\ldots,s_n\} \sqcup \{t_1,\ldots,t_m\} & \rightarrow & \bigcup_i \{d_i | d_i = s_i \sqcup \{t_1,\ldots,t_m\}\}
\end{array}$$

Figure 6.7: Rewrite Rules

Integrating Domain Variables into the Constraint Rewriting Framework

The integration of domain variables and feature structure unification now can be done rather straightforward. Recall that the unification of feature structures is triggered by unifying the attributed variables carrying those feature structures as attributes. Domain variables are represented the same way, i.e. by an attribute `dom(X,ListOfTypes)`. Now attribute rewrite clauses for two new cases have to be added to `combine_attributes/2` (see Fig. 4.6), describing the interaction of two domain variables and of a domain variable with a feature structure. Domain variables may hold only disjunctions of atomic types, therefore only combination with feature structures representing an atomic type are possible. Thus only attributes having the form `fs(X,T,[],P)` have to be considered, unifying a domain variable with a structured type is bound to fail. The added clauses are shown in Fig. 6.8 (note, that a third slot has been added to `dom/3` to hold delayed goals).

The predicate `restrict_dom/3` performs unification of a type with a domain, i.e., it restricts the domain in such a way that the result is subsumed by the type. `dom_intersect/3` performs unification of two domains (which would be a simple set intersection if not types but only atomic values with no possibility to subsume each other were involved).

Both forms of attribute combination may result in either a domain variable or a feature structure variable. The latter case occurs, when a singleton domain is the result of unification. Note that the goals slot of `dom/3` is needed to hold the delayed goals that may be present on a feature structure variable.

Domain Variables and Structured Types

Disjunctions of structured, but uninstantiated types may be treated in a similar way as disjunctions of atomic types. As long as no slot reference is made, these types behave just like atomic types. However, as soon as a domain of structured types is unified with an

```
combine_attributes( dom(X2,Types,P2), fs(X1,T1,[],P1) ) :-
    combine_attributes( fs(X1,T1,[],P1), dom(X2,Types,P2)).
combine_attributes( fs(X1,T1,[],P1), dom(X2,Types,P2)) :-
    restrict_dom(T1,Types,NewDomain),
    ( NewDomain=[]   -> fail
    ; detach_attribute(X1),
      X1 = X2,
      ( NewDomain=[T2] ->
        update_attribute(X1,fs(X1,T2,[],[])),
        start_goals(P1), start_goals(P2)
      ; append(P1,P2,P3),
        update_attribute(X1,dom(X1,NewDomain,P3))
      )
    ).
combine_attributes( dom(X1,Dom1,P1), dom(X2,Dom2,P2) ) :-
    dom_intersect(Dom1,Dom2,NewDomain),
    ( NewDomain==[]   -> fail
    ; detach_attribute(X1),
      X1 = X2,
     ( NewDomain=[T2] ->
       update_attribute(X1,fs(X1,T2,[],[])),
       start_goals(P1), start_goals(P2)
     ; append(P1,P2,P3),
       update_attribute(X1,dom(X1,NewDomain,P3))
     )
   ).
```

Figure 6.8: Attribute Rewriting with Domain Variables

instantiated feature structure and no unique solution results from unifying the type of the feature structure with the domain, the mechanism employed above does not work anymore.

Two amendments are possible: one could either allow domain variables to additionally carry a feature structure (which in fact transforms them into a kind of feature structure variables annotated with a domain further specializing it). The other possibility is to stop propagating the disjunction of types via domain variables if such a situation arises and to explore the different possibilities via backtracking. However, in practical grammars disjunctions of structured types are almost always specified via the type hierarchy. Therefore, neither of these options has been implemented by now.

6.2.2 Boolean Constraints on Atomic Types

After introducing domain variables to offer the possibility to specify disjunctions of atomic values, the question arises, if it would not be useful to have conjunction and negation operators as well, to be able to express the full range of boolean constraints. However, taking into account the type lattice, it turns out that conjunction amounts to the unification of two types, and negation is expressible via disjunction, given the finiteness of the type lattice (if only declared types are allowed). Thus, in a closed type system negation is only syntactic sugar, since the set of possible values is known and negation can be expanded into a finite domain. To make things more interesting, one could consider allowing also undeclared atomic types, which—of course—are not unifiable with any other type than $\top$[2].

Under such an assumption, the introduction of a negation operator makes sense and ways to integrate full boolean constraints on atomic values are worthwhile to be considered.

However, the type lattice induces equivalences in addition to the usual ones of boolean algebra and propositional logic, which lead to the simplification rules stated in Fig. 6.9. One main point is the observation that conjunction of types and their unification are two

Simplification Rules

(1)	$s \wedge t$	$\rightarrow$	$\mathrm{glb}(s,t)$
(2)	$A \wedge \neg B$	$\rightarrow$	$\bigvee_i C_i$ with $A \sqsubseteq C_i \wedge B \not\sqsubseteq C_i$
(3)	$a \wedge \neg B$	$\rightarrow$	a
(4)	$t \wedge \neg t$	$\rightarrow$	$\perp$
(5)	$t \wedge \neg a$	$\rightarrow$	t

Figure 6.9: Simplification Rules for Boolean Constraints on Types

sides of the same coin. Thus conjoining two types (denoted by $s, t, \ldots$) amounts to unify them, i.e., the conjunction of two types may be simplified to their greatest lower bound in the type lattice. Non-declared types (denoted by lowercase letters $a, b, c \ldots$) which are incompatible with every type other than $\top$ cause each conjunction in which they appear to fail[3].

Negation of declared types (denoted by uppercase letters $A, B, C \ldots$) would be enumerable as a finite disjunction, if we hadn't allowed for undeclared types. However, if such a negation appears within a conjunction with a positive type, immediate simplification to a finite disjunction is possible, or it can be dropped in case of incompatibility (see rules (2)

[2] In fact, such an extension of the type lattice has been already made tacitly, since any string may occur as a subtype of *string*, and any number as a subtype of *number*.

[3] $\mathrm{glb}(s,t) = \perp$ if s or t are undeclared types.

and (3) in Fig. 6.9). The negation of a non-declared type within a conjunction can be dropped (provided that not also a positive instance of that type occurs in the conjunction, see rules (4) and (5) of Fig. 6.9).

Given these additional simplification rules, the question arises, if the representation of full boolean constraints on atomic types within feature structures requires full-fledged propositional formulas or if simpler means in the style of the domain variables of section 6.2.1 are sufficient. Recall that appropriateness restrictions restrict the type of the value a feature may take. If we restrict our attention to the case of boolean constraints operating on such values (which is indeed the case they are needed for) and additionally stipulate, that appropriateness constraints at least have to specify whether the value of a feature must have a declared or an undeclared type, then a drastically simplified representation can be obtained. In order to allow for specifying a non-declared type as a value restriction we introduce a dichotomy into the type lattice and change the definition of an undeclared type in such a way, that all undeclared types are subsumed by "*undeclared*". Likewise all declared types have to be subsumed by "*declared*". The rules of Fig. 6.9 have to be interpreted now in the way that the lowercase letter a also ranges over the type "*undeclared*".

Every boolean constraint can be transformed to disjunctive normal form using the standard propositional equivalences. Then the appropriateness restriction defined on the slot the boolean type constraint is applied to can be unified with it. Thus, a type which is known to be a declared type or the type "*undeclared*" subsuming all undeclared types is conjoined with each disjunct. Now each disjunct contains a positive instance of a type. Repeated application of rules (2) and (3) of Fig. 6.9 together with redistributing the resulting (finite!) disjunctions eliminates all occurrences of negated structured types. On the other hand, negated undeclared types can only survive in a conjunction consisting solely of negated types, since rules (4) and (5) eliminate them in case of positive type instances inside a conjunction.

The shape of the conjunctions which might appear in the DNF is thus greatly restricted. If the restricting type A (from the appropriateness declaration) has been a declared type, then every element of the disjunction can be reduced to a declared type (negated types have disappeared due to rules (2) and (5); atomic types fall out of the disjunction since they cannot have been successfully unified with A). Thus the representation of the whole boolean constraint is possible by means of domain variables as introduced in section 6.2.1.

If, on the other hand, the restricting type is "*undeclared*", only undeclared types and conjunctions of negated undeclared types remain in the disjunction (the case with an undeclared type a as value restriction is straightforward, the result is then either a or $\bot$). Declared types cannot survive due to rule (1), negated types are eliminated by rules (3) and (5), the special case of conjunctions containing negated undeclared types has been discussed above. Since a conjunction of negated types is equivalent to the negated disjunction of those types, such a term will be called a *negated domain*.

A further simplification is possible: If more then one negated domain occurs within a disjunction, a simplified negated domain can be computed using de Morgans law by negating the result of conjoining the two domains. Thus, if two domains are disjoint, the disjunction of their negations reduces to $\top$. In any case, a boolean constraint on undeclared types can be reduced to the disjunction of a finite domain and a negated finite domain.

These constraints can easily be implemented. We have to introduce one new attribute which represents such an *extended domain* $\text{xdom}(\mathcal{P},\mathcal{N})$ representing the disjunction of a finite domain of possible values $\mathcal{P}$ with the negation of a finite domain $\mathcal{N}$ holding the excluded types.

Since boolean constraints of declared types can be stated as before using ordinary domain variables as introduced in section 6.2.1, these variables are now restricted to carry constraints on declared types only. Constraints on undeclared types are generally handled via the `xdom(Pos,Neg)` attribute, `Pos` or `Neg` possibly being empty. By introducing this distinction, the specification of the unification behavior is considerably simplified.

How *extended domains* interact with feature structures and ordinary domains during unification under the restrictions discussed above is shown in Fig. 6.10 using the same notational conventions as above. Interpreting unification as conjunction, they follow straightforwardly from the considerations about simplification of boolean constraints made above.

Unification Rules

(1)	$\text{xdom}(\mathcal{N},\mathcal{N}) \sqcup A$	$\rightarrow$	A if $\mathcal{N}$ is not empty
(2)	$\text{xdom}(\mathcal{P},\mathcal{N}) \sqcup a$	$\rightarrow$	a if $a \in \mathcal{P}$ or $a \notin \mathcal{N}$
(3)	$\text{xdom}(\mathcal{P},\mathcal{N}) \sqcup \text{dom}(\mathcal{D})$	$\rightarrow$	$\text{dom}(\mathcal{D})$ if $\mathcal{N}$ is not empty
(4)	$\text{xdom}(\mathcal{P}_1,\{\}) \sqcup \text{xdom}(\mathcal{P}_2,\{\})$	$\rightarrow$	$\text{xdom}(\mathcal{P}_1 \cap \mathcal{P}_2,\{\})$ if $\mathcal{P}_1 \cap \mathcal{P}_2 \neq \{\}$
(5)	$\text{xdom}(\{\},\mathcal{N}_1) \sqcup \text{xdom}(\{\},\mathcal{N}_2)$	$\rightarrow$	$\text{xdom}(\{\},\mathcal{N}_1 \cup \mathcal{N}_2)$
(6)	$\text{xdom}(\mathcal{P}_1,\{\}) \sqcup \text{xdom}(\{\},\mathcal{N}_2)$	$\rightarrow$	$\text{xdom}(\mathcal{P}_1 - \mathcal{N}_2,\{\})$ if $\mathcal{P}_1 - \mathcal{N}_2 \neq \{\}$
(7)	$\text{xdom}(\mathcal{P}_1,\mathcal{N}_1) \sqcup \text{xdom}(\mathcal{P}_2,\{\})$	$\rightarrow$	$\text{xdom}((\mathcal{P}_1 \cap \mathcal{P}_2) \cup (\mathcal{P}_2 - \mathcal{N}_1),\{\})$ if $(\mathcal{P}_1 \cap \mathcal{P}_2) \cup (\mathcal{P}_2 - \mathcal{N}_1) \neq \{\}$
(8)	$\text{xdom}(\mathcal{P}_1,\mathcal{N}_1) \sqcup \text{xdom}(\mathcal{P}_2,\mathcal{N}_2)$	$\rightarrow$	$\text{xdom}(\mathcal{P}_3,\mathcal{N}_1 \cup \mathcal{N}_2)$ where $\mathcal{P}_3 = (\mathcal{P}_1 \cap \mathcal{P}_2) \cup (\mathcal{P}_1 - \mathcal{N}_2) \cup (\mathcal{P}_2 - \mathcal{N}_1)$ if $\mathcal{P}_3 \neq \{\}$

all other combinations yield $\bot$

Figure 6.10: Unification Rules for Extended Domains

These unification rules can be added to `combine_attributes/2`—the constraint rewriting predicate handling extended unification—in the same way as this has been done for the integration of domain variables in Fig. 6.8. This integration is straightforward the rules being given.

6.2.3 Further Extension Possibilities

The methods for handling disjunctions discussed so far have only dealt with disjunctions of atomic values. While most ambiguities arising in natural language processing (primarily from the lexicon) can be handled efficiently by these means and general disjunction of feature structures in practical grammars occurs only in the type lattice, there still remains one class of phenomena which cannot efficiently be dealt with. An example for such a case is the lexicon entry of the German noun-form *Koffer* (suitcase). The CASE feature of that noun is *nom, dat* or *acc* if the NUMber feature is *singular*, but if NUMber is specified as *plural*, the possible CASE values are *nom, gen* or *acc*.

Expressing such a kind of ambiguity by means of a general disjunction of feature structures is expensive, since allowing general disjunctions may in the worst case involve expanding to DNF. To avoid this behavior, methods of keeping these disjunctions as local as possible have been developed (Eisele and Dörre, 1988; Kasper, 1987). However, both methods involve the need for copying substructures, if a disjunction cannot be kept local during unification and has to be lifted.

Distributed disjunctions, as developed by Eisele and Dörre (1990)[4] and independently by Maxwell and Kaplan (1991), provide a means to overcome these copying problems by coindexing of the alternatives that vary in parallel. Disjunctions are named, if one branch in such a named disjunction is taken, in all other disjunctions with the same name the respective branch has to be taken as well.

A representation of the lexical entry for *Koffer* employing distributed disjunctions is shown in Fig. 6.11. The named disjunction ($\vee_1$) forces, that in all disjunctions with the same index the same branch (either the left or the right one) is taken. As can be seen, employing

$$\left[\begin{array}{ll} \text{PHON} & \textit{“Koffer”} \\ \text{SYNSEM}|\text{LOC}|\text{CAT}|\text{HEAD} & \left[\begin{array}{ll} \text{NUM} & sing \vee_1 plur \\ \text{CASE} & (nom \vee dat \vee acc) \vee_1 \ (nom \vee gen \vee acc) \end{array}\right]_{noun} \end{array}\right]_{word}$$

Figure 6.11: Lexical Entry with Distributed Disjunction

distributed disjunctions—even if only for atomic types— may provide a useful technique to reduce the need for backtracking and thus enhance efficiency.

Existing implementations of distributed disjunctions (Eisele and Dörre, 1990; Matiasek, 1993) require a context maintainance component to keep track upon which alternative currently is explored and to record failing alternatives. Given the mechanism of conditional constraints developed in this thesis, an alternative approach could be exploited[5]. The key

[4] A method for a structure sharing implementation of distributed disjunctions has been put forth in Matiasek (1993).

[5] I would like to thank Martin Emele for helpful discussion on this subject.

idea of this approach is to use dynamically generated conditional constraints that have to ensure the proper restriction of all value slots affected by a distributed disjunction according to the branch taken.

The details of this approach have not yet been worked out In particular, the interaction with domain variables that is needed in the above example is yet clarified. Further development into this direction, however, seems promising and further efficiency gains are to be expected.

Chapter 7

Conclusion

The method for the implementation of a feature formalism developed in this thesis aims at the support of principle-based approaches to grammar, in particular at covering all details needed for implementing HPSG. The goal of achieving independence from a particular processing direction (e.g. parsing or generation) requires a clear distinction between grammar and processing model to avoid (hidden) encoding of procedural aspects into the grammar. In order to be able to express all aspects of a principle-based grammar without recurring to procedural considerations, the feature formalism must be powerful enough to represent them in a declarative way and handle them without depending on a particular processing or even instantiation sequence.

These goals have been achieved by interpreting the declarative specification of a grammar not as static data, but as the operational basis for processing. The operators employed in feature logic formulas to express some constraint simultaneously serve as instantiation operators, interpreting these formulas as partially instantiated feature structures. This instantiation of feature structures works in a lazy incremental way, structures are built only on demand.

Nevertheless, all structures built can be regarded as *well-typed* according to the type scheme and *well-formed* in the sense of obeying all principles of grammar. To achieve this behavior, the notion of well-typedness is interpreted in an incremental way, and type inference steps are interleaved with unification as well as with instantiation.

To guarantee that feature structures are not only always well-typed, but also *well-formed* in the sense of the grammar, principles of grammar are viewed as licensing conditions on feature structures and applied to every node at instantiation time. A delay mechanism allowing principles to operate on uninstantiated feature structures without eagerly instantiating them has been implemented. A conditional syntax for specifying principles has been introduced which allows to state their applicability conditions. Delay occurs, if a node fails to meet the preconditions at the moment but has the potential to meet them later.

Relational constraints which are indispensable for specifying grammatical principles of HPSG have been implemented likewise. To enable them to cope with uninstantiated arguments, a situation which must be expected in a lazy instantiating feature formalism, they also make use of the delay mechanism used in the implementation of the principles, using the same conditional syntax.

The implementation was done on top of a Prolog system with extensible unification (DMCAI Clp) allowing to specify the rewrite operations on feature structures in a clear and declarative way. Feature structures are implemented as attributes of Prolog variables. The rewrite operations to achieve well-typed unification and principled constraint application are performed via the extended unification mechanism when two variables representing feature structures are unified. The delay mechanism has been integrated into this attribute rewriting clauses. Thus the use of a CLP system dispenses with the need of having a meta-interpreter managing this interaction and also leads to efficiency gains, because the interaction between syntactic and extended unification is coded in the Prolog kernel. A further advantage of the use of attributed variables to represent feature structures is the simple interface they provide to processing modules. Unifying two feature structures can be performed simply by unifying the Prolog variables representing them, success of this unification yields also a rewritten feature structure, its well-typedness and well-formedness being guaranteed.

Thus the goal of operationalizing principle based grammars has been achieved without relying on procedural aspects. Every structure (maybe only partially) instantiated within the formalism can be regarded as well-formed in the sense, that it is only extensible in a way compliant with all principles of grammar, intrinsically carrying all constraints of grammar.

The feasibility and usefulness of this approach to typed feature structures has been demonstrated in two applications, both extending standard HPSG.

The first one presents a framework for the tight integration of word level and sentence level processing (Trost and Matiasek, 1994) and demonstrates the capabilities of relational constraints, as they are implemented here. Within this framework a reimplementation of X2MorF, an extended two-level morphology system for German (Trost, 1991) was integrated into the grammar as a specialized relation at the word level, being the boundary between the morphological and syntactic level.

This architecture has computational as well as linguistic advantages. Integrating morphology and morphophonology directly into the grammar is in the spirit of HPSG, which views grammar as a relation between the phonological (or graphemic) form of an utterance and its syntactic/semantic representation. This way the treatment of phenomena transcending the boundary between morphology and syntax is also made possible, as is demonstrated in the second application dealing with syntactic argument reduction phenomena triggered by morphological processes.

On the implementation side, the practical problems of interfacing two inherently different modules are eliminated. For applications this means that using a morphological component is made easy. Nevertheless, this tight integration still leaves morphology and syntax/semantics as autonomous components, enabling direct use of existing data sets describing morphophonology in terms of the two-level paradigm.

The second application accounts for argument reduction phenomena and case assignment in German along the lines of Heinz and Matiasek (1994). The HPSG framework is extended in a conservative manner, maintaining the structure of the SUBCAT list and the formulation of the Subcat Principle. By structuring the CASE-values a distinction between structural and lexical case as well as between syntactic case and its morphological realization is made possible. Case assignment of structural case is achieved via the Case Principle. In this way case assignment may be constrained to operate locally on the SUBCAT list.

Argument reduction and ergativity is accounted for by introducing a list-valued *cat* feature DA holding the designated argument. Argument reduction phenomena such as past participle formation, which are originally given a treatment via lexical rules by Heinz and Matiasek (1994), are handled now in a *constructive* manner. Instead of relying on nonmonotonic effects induced by the use of lexical rules, morphosyntactic constraints formulated within the framework of the first application yield the same results as the lexical rule approach (e.g. only one *ppp* entry in several constructions—passives and perfect tenses) without destructive operations.

In order to increase efficiency some extensions to the basic feature formalism have been implemented. *Compilation* of feature grammars has been implemented in two stages. The first one, incorporated already in the basic system has dealt only with the type lattice and appropriateness checking. Thus at run time no need arises to traverse the type lattice, which may be a costly operation. Furthermore, predicates operationalizing some aspects of maintaining well-typedness have been generated.

The second compilation stage involved a change in the representation of typed feature structures from a slot-value list to a fixed arity term, allowing to exploit the benefits of built-in term unification. Well-typed unification of feature structures is now possible simply by table lookup. The same holds for slot access and type inference. Although the licensing view of grammar principles is maintained, applying them to all nodes during instantiation is no longer required, instead they are triggered by the precompiled predicates at the appropriate places, i.e. whenever a type transition has to be licensed by a principle. These additional compilations have resulted in a performance gain by a factor of 2 as compared with stage one compilation.

A further extension to the basic formalism concerns the treatment of disjunctions. One method to cope with disjunctions is already built into the basic formalism and amounts to replace a disjunction by a conjunction of implicative constraints. While this method is well-suited for the specification of complex disjunctions such as, e.g., relational constraints, it is somewhat cumbersome to use if only ambiguities in an atomic value have to be expressed. In order to handle such cases efficiently, domain variables have been introduced and integrated into the feature unification framework.

Summing up, the feature unification formalism developed here provides the means for implementing principle based grammar formalisms by simply specifying the grammar in a declarative fashion. This has been accomplished by means of a constraint rewriting system, which is able to operate on structures only partially instantiated, whilst maintaining all restrictions needed to preserve their well-formedness in the sense of the grammar. By virtue of this system the user is enabled to keep procedural considerations completely out of the grammar specification. Nevertheless the grammar is operationalized, since the specification results in executable code. Thus the declarative grammar specification constitutes an operational module, admitting only well-formed structures.

References

Abramson, H. 1992. A Logic Programming View of Relational Morphology. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92), Nantes*, pages 850–854, August 23-28.

Aït-Kaci, H. and R. Nasr. 1986. LOGIN: A logical programming language with built-in inheritance. *Journal of Logic Programming*, 3:187–215.

Balari, S., G. B. Varile, L. Damas, and N. Moreira. 1990. CLG(n): Constraint Logic Grammars. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 7–12, Helsinki.

Berwick, R., S. Abney, and C. Tenny, editors. 1991. *Principle-Based Parsing*. Kluwer, Dordrecht.

Berwick, R. 1991. Principles of Principle-Based Parsing. In R. Berwick, S. Abney, and C. Tenny, editors, *Principle-Based Parsing*. Kluwer, Dordrecht.

Carlsson, M. and J. Widen. 1990. Sicstus Prolog Users Manual. Technical Report SICS/R-88/88007C, Swedish Institute of Computer Science.

Carpenter, B. 1992a. ALE—The Attribute Logic Engine. Technical Report CMU-LCL-92-1, Carnegie Mellon University, Pittsburgh, PA.

Carpenter, B. 1992b. *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science, number 32. Cambridge University Press, New York.

Carpenter, B. 1993. Compiling Typed Attribute-Value Logic Grammars. In *Proceedings of the Third International Workshop on Parsing Technologies (IWPT'93)*, pages 39–48, Tilburg, The Netherlands.

Carpenter, B., C. Pollard, and A. Franz. 1991. The Specification and Implementation of Constraint-Based Unification Grammars. In *Proceedings of the Second International Workshop on Parsing Technology*, pages 143–153, Cancun, Mexico.

Chomsky, N. 1970. Remarks on Nominalization. In R. A. Jacobs and P. S. Rosenbaum, editors, *Readings in English Transformational Grammar*. Ginn and Company, Waltham, MA, pages 184–221.

Chomsky, N. 1981. *Lectures on Government and Binding.* Foris, Dordrecht.

Chomsky, N. 1986. *Barriers.* MIT Press, Cambridge, Mass.

Damas, L., N. Moreira, and G. B. Varile. 1991. The Formal and Processing Models of CLG. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, pages 173–178.

Damas, L. and G. B. Varile. 1992. On the Satisfiablity of Complex Constraints. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92)*, pages 108–112, Nantes.

Dawar, A. and K. Vijay-Shanker. 1990. An Interpretation of Negation in Feature Structure Descriptions. *Computational Linguistics*, 16(1):11–21.

di Sciullo, A.-M. and E. Williams. 1987. *On the Definition of Word.* MIT Press, Cambridge, MA.

Dörre, J. and M. Dorna. 1993. CUF - A Formalism for Linguistic Knowledge Representation. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description*, , number DYANA-2, Deliverable R1.2.A. Esprit Basic Research Project 6852, pages 1–22.

Dörre, J. and A. Eisele. 1991. A Comprehensive Unification-Based Grammar Formalism. Technical Report DYANA Deliverable R3.1.B, ESPRIT Basic Research Action BR3175.

Eisele, A. and J. Dörre. 1988. Unification of Disjunktive Feature Descriptions. In *Proceedings of 26th Annual Meeting of the Association for Computational Linguistics*, pages 286–294, Buffalo, NY. State University of New York.

Eisele, A. and J. Dörre. 1990. Feature Logic with Disjunctive Unification. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 100–105, Helsinki.

Emele, M. C. 1991. Unification with Lazy Non-Redundant Copying. In *Proceedings of 29th Annual Meeting of the Association for Computational Linguistics*, pages 323–330, Berkeley, CA. University of California.

Emele, M. C. and R. Zajac. 1990. Typed Unification Grammars. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 293–298, Helsinki.

Fong, S. 1991. The Computational Implementation of Principle-Based Parsers. In R. Berwick, S. Abney, and C. Tenny, editors, *Principle-Based Parsing.* Kluwer, Dordrecht.

Franz, A. 1990. A Parser for HPSG. Technical Report CMU-LCL-90-3, Carnegie Mellon University, Pittsburg, PA.

Gazdar, G., E. Klein, G.K. Pullum, and I.Sag. 1985. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, Mass.

Gazdar, G. and C. Mellish. 1989. *Natural Language Processing in Prolog.* Addison-Wesley, Reading, MA.

Haider, H. 1985. The Case of German. In J. Toman, editor, *Studies on German Grammar.* Foris, Dordrecht.

Haider, H. 1986. Fehlende Argumente: vom Passiv zu kohärenten Infinitiven. *Linguistische Berichte*, 101:3–33.

Haugeneder, H. and H. Trost. 1993. Beschreibungsformalismen sprachlichen Wissens. In G. Görz, editor, *Einführung in die Künstliche Intelligenz,.* Addison-Wesley.

Heinz, W. and J. Matiasek. 1994. Argument Structure and Case Assignment in German. In J Nerbonne, K. Netter, and C. Pollard, editors, *German in Head-Driven Phrase Structure Grammar.* CSLI Publications, Stanford, pages 199–236.

Hinrichs, E. and T. Nakazawa. 1990. Subcategorization and VP structure. In S. Hughes, editor, *Proceedings of the Third Symposium on Germanic Linguistics*, Amsterdam. Benjamins.

Holzbaur, C. 1990. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* Ph.D. thesis, Dept. of Medical Cybernetics and Artificial Intelligence, University of Vienna, Vienna.

Holzbaur, C. 1992. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, LNCS 631. Springer, pages 260–268.

Huitouze, S. le. 1990. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Maluszunski, editors, *Programming Language Implementation and Logic Programming.* Springer, Heidelberg, pages 136–150.

Jaffar, J. and J. L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the* 14^{th} *ACM POPL Conference*, Munich.

Johnson, M. 1988. *Attribute–Value Logic and the Theory of Grammar.* CSLI Lecture Notes 16. CSLI, Stanford, CA.

Johnson, M. 1991. Deductive Parsing: The Use of Knowlede of Language. In R. Berwick, S. Abney, and C. Tenny, editors, *Principle-Based Parsing.* Kluwer, Dordrecht.

Kaplan, R. and J. Bresnan. 1982. Lexical-Functional Grammar: A Formal System for Grammatical Representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Mass.

Karttunen, L. 1983. KIMMO: A General Morphological Processor. In *Texas Linguistic Forum 22*, pages 163–186.

Karttunen, L. 1984. Features and Values. In *Proceedings of the 10th International Conference on Computational Linguistics (COLING-84)*, Stanford, CA.

Karttunen, L. 1986. D-PATR: A Development Environment for Unification-Based Grammars. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING-86)*, pages 74–80, Institut für Angewandte Kommunikations- und Sprachforschung, Bonn, BRD.

Karttunen, L. 1986. Radical Lexicalism. Technical Report CSLI-86-63, Center for the Study of Language and Information, Stanford University, Stanford, CA.

Kasper, R. 1987. A Unification Method for Disjunktive Feature Descriptions. In *Proceedings of 25th Annual Meeting of the Association for Computational Linguistics*, pages 235–242, Stanford, CA. Stanford University.

Kasper, R. 1988. Conditional Descriptions in Functional Unification Grammar. In *Proceedings of 26th Annual Meeting of the Association for Computational Linguistics*, pages 233–240, Buffalo, NY. State University of New York.

Kasper, R. and W. Rounds. 1986. A Logical Semantics for Feature Structures. In *Proceedings of 24th Annual Meeting of the Association for Computational Linguistics*, pages 257–266, New York. Columbia University.

Kathol, A. 1991. Participles in German. Paper presented at the Workshop "German Grammar in HPSG", Saarbrücken, August 1991.

Kay, M. 1979. Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*. Berkeley Linguistics Society, Berkeley, CA.

Kay, M. 1985. Parsing in Functional Unification Grammar. In D. R. Dowty, L. Karttunen, and A. M. Zwicky, editors, *Natural Language Parsing*. Cambridge University Press, Cambridge, England, chapter 7, pages 251–278.

Kiss, T. 1991. Variable Subkategorisierung: Eine Theorie unpersönlicher Einbettungen im Deutschen. Technical Report IWBS Report 161, IBM TR 80.91-015, IBM Germany, Stuttgart.

Kodrič, S., F. Popowich, and C. Vogel. 1992. The HPSG-PL System, Version 1.2. Technical Report CSS-IS TR 02-05, Simon Fraser University, Canada.

Koskenniemi, K. 1983. Two-Level Model for Morphological Analysis. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann.

Krieger, H.-U. 1991. Lexicon-Driven Derivation in HPSG. Technical Report Research Report RR-91-36, DFKI, Saarbrücken.

Krieger, H.-U. and J. Nerbonne. 1991. Feature-Based Inheritance Networks for Computational Lexicons. Technical Report Research Report RR-91-31, DFKI.

Krieger, H.-U., H. Pirker, and J. Nerbonne. 1993. Feature-based Allomorphy. In *Proceedings of 31st Annual Meeting of the Association for Computational Linguistics*, pages 140–147, Columbus, Ohio.

Matiasek, J. 1993. Structure Sharing Unification of Disjunctive Feature Descriptions. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*. Ellis Horwood, Chichester, UK.

Matiasek, J. 1994. Conditional Constraints in a CLP-based HPSG Implementation. In H. Trost, editor, *Konvens 94*, pages 230–239. Springer.

Maxwell, J. T. and R. M. Kaplan. 1991. A Method for Disjunctive Constraint Satisfaction. In Masaru Tomita, editor, *Current Issues in Parsing Technology*. Kluwer, Dordrecht.

Nerbonne, J., K. Netter, and C. Pollard, editors. 1994. *German in Head-Driven Phrase Structure Grammar*. Lecture Note Series 46, edited by John Nerbonne and Klaus Netter and Carl Pollard. CSLI, Stanford.

O'Keefe, R. 1990. *The Craft of Prolog*. MIT Press.

Pereira, F.C.N. and D.H.D. Warren. 1980. Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278.

Pereira, F.C.N. and D.H.D. Warren. 1983. Parsing as Deduction. In *Proceedings of 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, MA.

Pollard, C. 1991. Case Assignment and a Unitary Rule of Passive. Paper presented at the Workshop "German Grammar in HPSG", Saarbrücken, August 1991.

Pollard, C. 1994. Toward a Unified Account of Passive in German. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*. CSLI Publications, Stanford.

Pollard, C. and I. Sag. 1987. *Information-Based Syntax and Semantics, Vol. 1: Fundamentals*. CSLI Lecture Notes 13. CSLI, Stanford, CA.

Pollard, C. and I. Sag. 1994. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago and CSLI Publications, Stanford, CA.

Pollard, C. J. and M. D. Moshier. 1990. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language and Cognition*, Vancouver Studies in Cognitive Science 1. University of British Columbia Press, Vancouver.

Popowich, F. and C. Vogel. 1991. A Logic Based Implementation of Head-Driven Phrase Structure Grammar. In Brown and Koch, editors, *Natural Language Understanding and Logic Programming, III.* Elsevier.

Pulman, S. G. 1991. Eurotra ET6/1: Rule formalism and virtual machine design study. Final Report of Eurotra ET6/1. Technical report, Cambridge, United Kingdom.

Riehemann, S. 1993. Word Formation in Lexical Type Hierarchies: A Case Study of *bar*-Adjectives in German. Technical Report SfS-Report-02-93, Seminar für Sprachwissenschaft, University of Tübingen.

Rounds, W. C. and R. T. Kaspar. 1986. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, Cambridge, Mass.

Sag, I. and C. Pollard. 1991. An Integrated Theory of Complement Control. *Language*, 67:63–113.

Shieber, S. 1986. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes 4. CSLI, Stanford, CA.

Shieber, S. 1988. A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING-88)*, pages 614–619, Budapest.

Shieber, S.M., Y. Schabes, and F.C.N. Pereira. 1994. Principles and Implementation of Deductive Parsing,. Technical Report TR-11-94, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University.

Smolka, G. 1988. A Feature Logic with Subsorts. Technical Report LILOG-Report 33, IBM-Germany, Stuttgart.

Sproat, R. 1992. *Morphology and Computation.* ACL-MIT Series in NLP. MIT Press, Cambridge, MA.

Trost, H. 1990. The application of two-level morphology to non-concatenative German morphology. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 371–376, Helsinki.

Trost, H. 1991. X2MORF: A Morphological Component Based on Augmented Two-Level Morphology. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia.

Trost, H. 1993a. Coping with Derivation in a Morphological Component. In *6th Conference of the European Chapter of the ACL*, pages 368–376, Utrecht.

Trost, H., editor. 1993b. *Feature Formalisms and Linguistic Ambiguity*. Ellis Horwood, Chichester, UK.

Trost, H. and J. Matiasek. 1994. Morphology with a Null-Interface. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-94)*, Kyoto, Japan, August 5-9.

van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA.

van Hentenryck, P. and M. Dincbas. 1986. Domains in Logic Programming. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Los Altos, CA. Morgan Kaufmann.

Williams, E. 1981. Argument Structure and Morphology. *The Linguistic Review*, 1:81–114.

Woods, W.A. 1970. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, 13:591–606.

Zajac, R. 1992. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182.