# A CLP Based Approach to HPSG

**Johannes Matiasek and Wolfgang Heinz**
**Austrian Research Institute Artificial Intelligence**
**Schottengasse 3,**
**A-1010 Vienna, Austria**
{john,wolfgang}@ai.univie.ac.at

**Abstract**

In this paper we present a CLP based method for the direct implementation of HPSG, a grammar formalism employing strongly typed feature structures and principles to constrain them. We interpret unification of typed feature structures under the restrictions of principled constraints as constraint solving in the CLP paradigm. The aim of our implementation method is to operationalize the declarative grammar specification without having to account for processing aspects, i.e. to clearly separate grammar and processing model. To achieve this goal we employ a lazy instantiation technique which maintains well-typedness of feature structures at every instantiation stage. This method is complemented with a delay mechanism enabling the constraints arising from grammar principles to cope with uninstantiated structures. Applicability conditions of grammar principles may be specified conditionally, viewing them as licensing conditions on every node of a feature structure. This also allows for the reformulation of disjunctive constraints into a conjunction of conditional constraints, thereby reducing the search space.

## 1 Introduction

The development of linguistic theories during the past decade exhibits a distinctive trend towards declarativeness of the grammar. The view of a language as the strings that are generated by a set of rules is replaced by a view that defines a language by giving general principles that constrain the set of linguistic structures. The standard example for this change is the advent of the Principles and Parameters Approach to syntax (Chomsky 1981) replacing earlier Transformational Grammar.

This shift in linguistics has also influenced the development of computational linguistics. Principle-based grammar formalisms lend themselves better to process-neutral formulation, thus the use of the same grammar for parsing and generation was encouraged (cf. e.g. Strzalkowski 1991). But the abandonment of a rule-based backbone of the grammar (e.g. a context-free skeleton) also means the abandonment of a host of well-studies techniques for implementing processing modules for the grammars, which poses a problem for the implementation of a principle-based theory using the grammatical principles

directly. Generate-and-test strategies suffer from the huge search space of possible structures. Therefore many parsers for principle-based grammars reintroduce a covering phrase structure grammar to restrict the search space (cf. Berwick 1991).

This reintroduction clearly contradicts the intentions and theoretical considerations underlying those types of grammar. An implementation in accord with the grammatical theory implemented surely would be preferable, much more so as grammar engineering for increasingly large grammars becomes an issue (cf. Erbach and Uszkoreit 1990). Here, a theoretical gap between the grammar constructed by the grammar writer and the grammar used by the grammar implementor represents a source of serious problems.

A solution to these problems lies in the extension of grammars (and grammar implementations) with an operational core that handles the basic operations in concordance with the grammatical specification, which amounts to an object-oriented approach in that the grammar object also provides methods for its manipulation.

To show that this is a viable approach we present a grammar processing system emphasizing techniques that form part of the grammar engine core. With this system an implementation of Head Driven Phrase Structure Grammar (HPSG, Pollard and Sag 1987, Pollard and Sag in press) is given. HPSG as a grammar formalism embodies the type of grammar based on general principles or constraints. Furthermore, it exhibits another predominant approach in computational linguistics not quite independent from the principle-based approach, namely the use of feature structures to describe linguistic entities and the use of unification as the primary operation of combination.

In our system we have tried to relieve the parser and generator modules as much as possible from general grammar specific operations. As suggested by declarativeness, unification, and constraints, we use Constraint Logic Programming (CLP) as a basic implementation technique viewing the principles of grammar as constraints on feature structures. To shift the workload away from the implementation of parsers and generators (which can then be viewed as query solvers), and to maintain the declarative spirit of the grammar, the operational processes have to be constructed in a direction-neutral way. Together with the nessecity to deal with a vast search space, this leads us to the key technique which we employ in different facettes: elaborated delay mechanisms which enable declarative statements of grammatical principles directly being used by the processing components without incurring the penalty of, for example, infinite looping when encountering hitherto uninstantiated variables.

The delay mechanisms we present here postpone decisions until enough information is available for a choice instead of immediately opening up different paths for all possible choices: lazy instantiation in combination with type inference to efficiently unify typed feature structures (which form the core of the HPSG formalism); and a coroutining technique to enable the use of complex constraints on feature structures to directly implement the grammatical principles of HPSG. The approach taken is not restricted to HPSG but pertains to the whole family of grammar formalisms that employ general constraints over typed feature structures.

In section 2 we give a short overview of HPSG, section 3 describes typed feature structures from a CLP view and the lazy instantiation technique, in section 4 the principles of

grammar are recast as constraints, and the coroutining technique for principle application is introduced, and section 5 compares our approach to other implementations.

# 2 Head Driven Phrase Structure Grammar

Head Driven Phrase Structure Grammar (Pollard and Sag 1987, Pollard and Sag in press) 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 parametrized principles of grammatical wellformedness, 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.

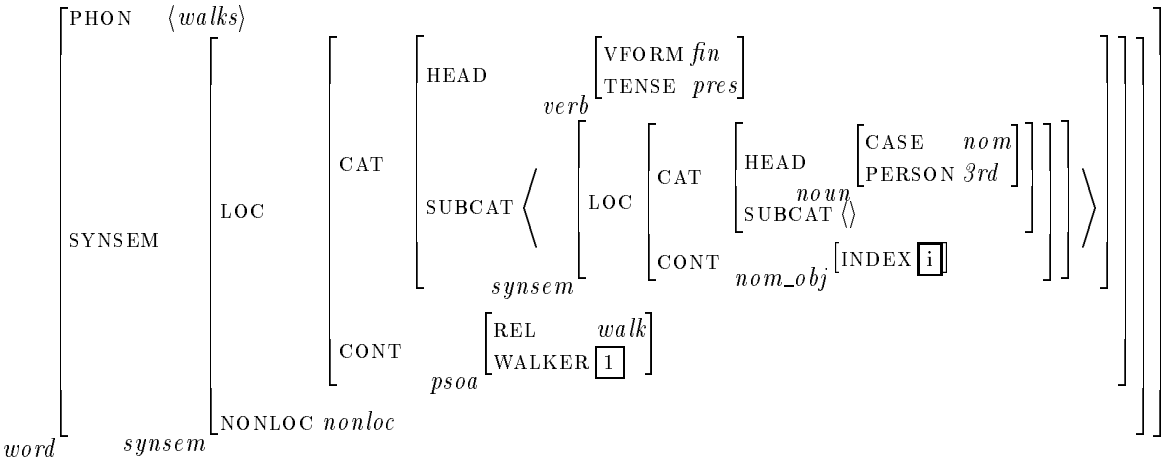
The theory has undergone some revisions and extensions since Pollard and Sag 1987. In our exposition we will follow the version of HPSG set up in Pollard and Sag in press.

## 2.1 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*.[1]

[1]The example is taken from Pollard and Sag in press




In the attribute-value matrices we use here the angle brackets 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 (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 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*—we will consider 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 type of phrases built up by saturating the valencies 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 in press 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:

$$\mathit{phrase}\,[\textsc{dtrs}\ \mathit{head\text{-}struc}] \implies \mathit{phrase}\begin{bmatrix}\textsc{synsem|loc|cat|head} & \boxed{1}\\ \textsc{dtrs|head-dtr|synsem|loc|cat|head} & \boxed{1}\end{bmatrix}$$

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.

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 we can see the effects of the interaction of the principles described so far with lexical entries by inspecting the (simplified, ignoring content) feature structure representing the phrase *Kim walks*:

$$\mathit{phrase}\begin{bmatrix}\textsc{phon} & \langle \mathit{Kim},\ \mathit{walks}\rangle & \\ \textsc{synsem} & \begin{bmatrix}\textsc{loc|cat} & \begin{bmatrix}\textsc{head} & \boxed{1}\\ \textsc{subcat} & \langle\rangle\end{bmatrix}\end{bmatrix} & \mathit{verb}\begin{bmatrix}\textsc{vform} & \mathit{fin}\\ \textsc{tense} & \mathit{pres}\end{bmatrix}\\ \textsc{dtrs} & \begin{bmatrix}\textsc{head-dtr} & \mathit{word}\begin{bmatrix}\textsc{phon} & \langle \mathit{walks}\rangle\\ \textsc{synsem} & \begin{bmatrix}\textsc{loc|cat} & \begin{bmatrix}\textsc{head} & \boxed{1}\\ \textsc{subcat} & \langle \boxed{2}\rangle\end{bmatrix}\end{bmatrix}\end{bmatrix} & \\ \textsc{comp-dtrs} & \left\langle \mathit{word}\begin{bmatrix}\textsc{phon} & \langle \mathit{Kim}\rangle\\ \textsc{synsem} & \boxed{2}\\ \textsc{loc|cat} & \begin{bmatrix}\textsc{head} & \mathit{noun}\begin{bmatrix}\textsc{case} & \mathit{nom}\\ \textsc{person} & \mathit{3rd}\end{bmatrix}\\ \textsc{subcat} & \langle\rangle\end{bmatrix}\end{bmatrix} \right\rangle & \end{bmatrix}\end{bmatrix}$$

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.

## 2.2 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 inheritence-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 and 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 in press and formally defined in Carpenter et al. 1991 and Carpenter 1992b. We will briefly review them 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 $\sigma$ such that $Approp(f, \sigma)$ is defined and
2. if $Approp(f, \sigma)$ is defined and $\sigma$ subsumes $\tau$ then $f$ is also appropriate for $\tau$ 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 et al. 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 typable). 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 et al. 1991).

# 3 A CLP View of Typed Feature Structures

Constraint logic programming (Jaffar and Lassez 1987) has been developed during the past few years as a generalization of Logic Programming by 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'). 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, yieding $C$, 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 metainterpreter or making use of a system which allows for the implementation of unification extensions, the approach adopted here.

## 3.1 A Prolog System with Extensible Unification

Semantic unification as a means to extend logic programming languages can be realized in different ways. We will shortly sketch the approach to semantic unification underlying the CLP system being used as the basis for our HPSG implementation.

DMCAI CLP[2] (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 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 metainterpreter managing this interaction.

## 3.2 Feature Structures as Constraints on Variables

The implementation of typed feature structures in our system makes use of the CLP facilities provided by the enhanced Prolog system described above. Feature structures are implemented by the attribute `fs(Type,Dag)`, where `Dag` is a list of feature-value pairs.

[2]DMCAI CLP is an enhanced version of SICStus Prolog and available by anonymous ftp from `ftp.ai.univie.ac.at`

Well-typed unification of two feature structures is implemented via the constraint solving clause `combine-attributes` described above, taking the Type Hierarchy and feature appropriateness into account, whose implementation we will turn to below.

### 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.

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

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.

### Appropriateness and Inheritance

Instead of defining *Approp* directly, our implementation 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 we give the appropriateness declarations for the part of the type lattice shown above:

```
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
```

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_features(Type,Features)` maps every nonatomic 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.

### Lazy Instantiation and Type Inference

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 2.2).

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 subfunctions, 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 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_features/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).

### 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 uninstatiated 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:

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

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.

### Well-typed Unification

Unification of feature structures now takes place when Prolog unifies two variables with an `fs` attribute. Well-typed unification as described above serves as the "equality theory" for this CLP instance. The attribute of the (Prolog) unified varible is rewritten with the result of unifying and well-typing the feature structures represented by the two "input" attributes. This attribute rewriting is coded in the clauses for `combine_attributes/2` and performs essentially the following operations:

- 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.

Recursion is handled automatically by the same constraint solving mechanism, when the Prolog variables serving as values of a feature are unified.

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 we have achieved an operational basis for representing feature structures. 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 perspicious 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. Wellformed (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 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 in press 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 et al. 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 et al. 1991 along with their satisfaction conditions are given below:

| | |
|---|---|
| $F \models \sigma$ | if the root node of $F$ is assigned a type at least as specific as $\sigma$ |
| $F \models \pi : \phi$ | if the value of $F$ at path $\pi$ is defined and satisfies $\phi$ |
| $F \models \pi \doteq \pi'$ | if paths $\pi$ and $\pi'$ 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 $F \models \psi$ |

Minimal satisfiers exist for all feature logic formulas $\phi$ (see Carpenter et al. 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 $\psi$ with respect to a constraint system $\Phi$ is a feature structure $F$ satisfying $\psi$ and each substructure of $F$ satisfies all the constraints of $\Phi$ inherited by the type of its root node.

Every **Minimal Solution** of $\psi$ can be obtained by applying a rewriting process to a minimal well-typed satisfier of $\psi$. **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 et al. 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 the lazy instantiation scheme, facilitating the implementation of other search strategies.

## 4.1 Implementing the Principles

Feature logical formulas as the device to state constraints on feature structures are implemented by means of the instantiation expressions introduced in section 3.2, 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 locical connectives $\wedge$ and $\vee$ is taken over by the Prolog operators "," and ";". For example, let $\phi$ 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 in our system 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 (Prolog) 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, our 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 our 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 prerequiste to 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 et al. 1991 carry over to our approach. 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 specifing 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 in our implementation, we adopt a *licensing* view of the principles. 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 4.2.

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 our 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 our system we give the Head Feature Principle of HPSG

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

Principles 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, which is executed when the antecedent of the conditional is satisfied by the feature structure `X`, dropped, if the preconditions are incompatible with `X`, and delayed otherwise. This behavior is achieved by using `term_expansion/2` (see below). Failure of a principle stated in this form occurs only if feature structure `X` satisfies the preconditions but the constraint part fails, or (trivially) if `X` is not a feature structure. Inapplicable or delayed principles succeed always (at the time they are applied—of course, delayed constraints may fail later on when inappropriate instantiations are attempted).

The set of principles that a grammar requires to license each node is defined by the grammar writer via `apply_principles/1` which is called with every node during instantiation.[3] Thus every node of a feature structure may safely be assumed to be appropriately constrained. Not only well-typedness (in the incremental sense defined above) of nodes (i.e. of the feature structures rooted therein) is guaranteed automatically but also their well-formedness (in the sense of being appropriatey constrained by the principles of grammar, dissallowing instantiation of substructures within their scope not compliant with them).

## 4.2 A Coroutining Technique for Principle Application

To achieve the licensing and delaying behavior of the principles stated as conditionals as it is described above, clauses employing the `===>` operator 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). This interpretation is achieved via `term_expansion/2` and does the following:

- 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 inapproprate 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.[4] In case of conditions incompatible with the feature structure `preconditions/2` fails.

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

[3] A hierarchical grouping of the principles according to the type of the nodes they apply to may be used to massively reduce the amount of preconditions checking required during node instantiation.

[4] In fact only the first variable needs to be collected, since at the moment only conjunctive preconditions are supported

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

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 unabiliy to decide on the antecedent of the principle that has to be applied.[5]

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)`.[6] The thawing of a delayed goal is attempted 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 we can give now the final version of `combine_attributes/2`:[7]

```
combine_attributes( fs(X1,T1,Dag1,P1), fs(X2,T2,Dag2,P2)) :-
  well_typed_unify_fs(T1, Dag1, T2, Dag2, T3, Dag3),
  (X1 == X2 -> true                              % beware of circularity
  ; 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)             % apply pending constraints
  ).
```

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.

The licensing view of grammar principles and the coroutining technique developed enable these conditional constraints to cope with situations where no immediate decision

[5] 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, our method allows for subsumption checks along arbitrary paths of feature structures.

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

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

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 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 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.

## 4.3 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 4.1 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: [8]

```
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).
```

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 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.

# 5 Comparison to other approaches

Recently some other systems employing typed feature formalisms—and thus well-suited to process HPSG—have been developed.

The TFS system (Emele and Zajac 1990, Emele 1991) is a LISP implementation of a typed feature unification formalism with inheritance. Types can be defined by formulas

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

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.

A system developed especially for parsing of HPSG—also 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 (Kodrič et al. 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).

ALE (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 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.

When contrasting these systems with the method presented here, the following points can be made: All the systems above (with the exception of TFS) are biased in the direction of parsing. Their control strategy is fixed (with the exception of Franz's system, where experimentation with the agenda ordering function is encouraged) and can be only influenced by taking into account the way, in which grammar specifications are processed by the system. This leads to amalgamating declarative grammar specification with procedural aspects and furthermore biases the grammar as a whole in one processing direction. This constrasts with our approach that draws a clear distinction between grammar specification and the processing model. However, grammar specification is *operational* in actively constraining the variables representing feature structures but, using the lazy instantiation scheme and the delay mechanism, does not bias the instantiation process. The delay mechanism is available throughout the system and not restricted to special functions (as in HPSG-PL, the other systems do not offer such a facility). A further aspect of our system

(shared with TMS and Franz's implementation) is that it does not rely on special rules factored out of the principles, but instead allows for a uniform representation of all grammar principles in the spirit of HPSG. Furthermore, by conditional formulation of principles the numbers of disjunctions in the grammar can be drastically reduced (see the `fs_append/3` example above, thus performance problems due to the large number of disjunctions (such as in Franz's system) do not arise.

A formalism including a general delay mechanism is the Constraint Logic Grammar (CLG) formalism (Balari et al. 1990, Damas et al. 1991, and 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. However, when comparing our approach to the implementation of HPSG in CLG(2) (Balari et al. 1990) our approach benefits from what could be called indexing. Every variable is related to exactly those constraints that are relevant for this variable and to no other constraint whatsoever. So rewriting can be done just at the right point in time, namely when the variable is augmented by an additional constraint or instantiated during unification, and rewriting need only consider the relevant, *small* subset of all constraints. CLG(2) just augments predicates operation 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".

# 6 Conclusion

The methods for implementing HPSG developed in this paper aim at a clear distinction between grammar and processing model (e.g. parser, generator). Since different search strategies are required for parsing and generation this distinction avoids (hidden) encoding of procedural aspects into the grammar. 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-structure, 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 constrains may also make use of the delay mechanism using the same conditional syntax.

A further advantage of conditional constraints is the possibility to reformulate disjunctive constraints by a conjunction of conditional constraints as has been exemplified. This reformulation can be done in many cases, thus minimizing the need for disjunctive constraints being a source of inefficiency.

On the implementation side the use of a Prolog system with extensible unification (DMCAI CLP) has helped a lot in specifying 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 achive 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 metainterpreter managing this interaction and also lead 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.

The techniques described have been employed in the implementation of a HPSG grammar for German (Heinz and Matiasek to appear) being used in a natural language consulting system. They are, however, not confined to HPSG and applicable for implementing any strongly typed feature formalism employing principled constraints.

## Acknowledgements

This research has been sponsored by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*, Grant No. P7986-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian *Ministry of Science and Research*. We would like to thank Bernhard Pfahringer for fruitful discussions and comments, Christian Holzbaur for providing DMCAI CLP, Harald Trost for helpful comments, and Prof. R. Trappl for his continuing support.

# References

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 COLING*, 7–12. Helsinki.

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

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.* No. 32 Cambridge Tracts in Theoretical Computer Science. New York: Cambridge University Press.

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*, 143–153. Cancun, Mexico.

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

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*, 173–178.

Damas, L., and G. B. Varile. 1992. On the Satisfiablity of Complex Constraints. In *Proceedings of COLING-92*, 108–112. Nantes.

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

Emele, M. C., and R. Zajac. 1990. Typed Unification Grammars. In *Proceedings of the 13th COLING*, Vol. 2, 293–298. Helsinki.

Erbach, G., and H. Uszkoreit. 1990. Grammar Engineering: Problems and Prospects. A Report on the Saarbrücken Grammar Engineering Workshop. Technical Report CLAUS-Report No. 1, Universität des Saarlandes.

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

Heinz, W., and J. Matiasek. to appear. Argument Structure and Case Assignment in German. In *HPSG for German*, ed. J. Nerbonne, K. Netter, and C. Pollard. Stanford: CSLI Publications.

Holzbaur, C. 1990. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD 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 *Programming Language Implementation and Logic Programming*, ed. M. Bruynooghe and M. Wirsing, 260–268. No. 631 LNCS. Springer.

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

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

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

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.

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

Pollard, C., and I. Sag. in press. *Head-Driven Phrase Structure Grammar*. To be published by University of Chicago Press and CSLI Publications.

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

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.

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

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

Strzalkowski, T. (ed.). 1991. *Reversible Grammar in Natural Language Processing, Proceedings of a Workshop sponsored by the Special Interest Groups on Generation and Parsing of the ACL*. Berkeley, CA. University of California.