# Typed Unification Grammars

Martin C. Emele, Rémi Zajac
Project Polygloss*
University of Stuttgart
IMS-CL/Ifl-AIS, Keplerstraße 17,
D – 7000 Stuttgart 1, Federal Republic of Germany
{emele,zajac}@is.informatik.uni-stuttgart.dbp.de

**Abstract**

We introduce TFS, a computer formalism in the class of logic formalisms which integrates a powerful type system. Its basic data structures are typed feature structures. The type system encourages an object-oriented approach to linguistic description by providing a multiple inheritance mechanism and an inference mechanism which allows the specification of relations between levels of linguistic description defined as classes of objects. We illustrate this approach starting from a very simple DCG, and show how to make use of the typing system to enforce general constraints and modularize linguistic descriptions, and how further abstraction leads to a HPSG-like grammar.

## 1 Introduction

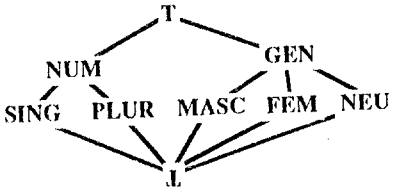
Various proposals have been made for the integration of type information in unification-based grammar formalisms to enforce constraints described in a hierarchical way where types are partially ordered with a subtype relation. Authors describe these extensions as "inheritance grammars", "inheritance networks", "feature sorts", "typed feature structures",... [1, 3, 5, 13, 17, 15, 9, 11, 7, 8].
These formalisms exhibit, to various degrees, one or several of the following properties, characteristic of the so-called object-oriented paradigm: a high level of abstraction, a capacity of inference, modularity and distributed control. Abstraction and modularity are needed when the linguist wants to describe a hierarchy of concepts (like a lexical hierarchy or the hierarchy of phrasal categories), and to describe linguistic data at different levels (e.g. morphology, syntax, semantics). At first glance it seems rather natural to develop separate modules for different linguistic levels, and to describe separately their interactions; however, great difficulties are encountered when these modules have to be integrated. Usually, there are two choices. Either everything is described in a single place using a deeply intricate data structure, like packing both syntactic and semantic equations in CF rules in some LFG extensions (e.g. [10]); the price is a loss in understandability and generality. Or descriptions are kept separate and the processing is done accordingly: first, a morphological phase, then a syntactic analysis, and then a semantic analysis, without any communication between these different steps [4]. The price is that interdependent constraints between these levels are lost, resulting in inadequate linguistic description or very complex control strategies at the implementation level.
In this paper, we argue that typed unification grammars give the linguist a formal framework which has the desirable properties. We will give an introduction to such a formalism, called TFS (Typed Feature Structure), which integrates disjunctions, conjunctions and conditional expressions of typed feature structures. This introduction will start from a very simple DCG, and will show how one can write a DCG-like grammar in TFS, making use of the typing system to enforce general constraints valid for classes of objects and to modularize linguistic descriptions. We then show that further abstraction leads to a HPSG-like grammar. It is not our goal to give here a formal account of the formalism (the interested reader should refer to [2] where a very clear formal semantics on which TFS is based is given), and we will use an informal approach wherever possible.

## 2 Typed feature structures and unification

The basic data structure of the language is a typed feature structure: a feature structure (FS in the following) with which a type can be associated. Compared to untyped FSs (as presented in [16] for example), the TFS system offers the possibility to name complex FSs, and to associate constraints with these names, thus defining a type.
We write feature names in small caps letters (F, G, H), type symbols in upper case letters (A, B), and we use symbols inside a box [1], called tags, for denoting shared values. For example, the typed FS, written in a linear form A[F: [1]B[H: C], G: [1]], is an FS of type A with two features F and G, F having as a value the typed FS B[H: A] and G having the same shared value as F.
In the system, one can specify type definitions which can, as a first approximation, be seen as a kind of template definition like in e.g. PATR-II. There is, however, a major difference. The system uses a type inference mechanism to derive new types dynamically during computation whereas templates in PATR-II are expanded statically at compile time.
A type that encodes agreement features can be written: AGR = [num: NUM,gender: GEN] and types NUM and GEN being themselves defined as NUM = SING ∨ PLUR (where the symbol "∨" denotes the logical OR) and GEN = MASC ∨ FEM ∨ NEU. The types NUM, SG,... do not have definitions: they are called atomic types. AGR, NUM and GEN are called complex types.
From a set of type definitions, one can extract a partial order on type symbols. For example, from the

*Research reported in this paper is partly supported by the German Ministry of Research and Technology (BMFT, Bundesminister für Forschung und Technologie), under grant No. 08 B3116 3. The views and conclusions contained herein are those of the authors and should not be interpreted as representing official policies.

set of definitions above, we can derive the following partial order on type symbols (Fig.1) where $\top$ represents the greatest element (no information) and $\bot$ the smallest element (inconsistent information, leading to failure in unification). This partial order is in turn used to derive a lattice of type symbols, which is then extended to typed FSs ordered by (typed) subsumption, forming a lattice on which the interpreter works (see a formal account in [2]).




For example, the FS *f1* AGR[num: NUM] subsumes the FS *f2* AGR[num: PLUR, gender: FEM] because *f2* has more specific information than *f1* : no gender is specified in *f1*, and the number value of *f2* PLUR is more specific than the number value of *f1*, NUM.
Typed unification proceeds as ordinary unification for FSs, recursively combining substructures at the same paths. When two (typed) FSs are unified, first the type symbols are unified, and if this unification succeeds, the FSs are unified. Unification of two types X and Y is defined as the (set of) most general type(s) which is smaller than both X and Y: it is the greatest lower bound (GLB) of these two symbols in the lattice of type symbols. If two types are directly comparable, the smallest is the result of the unification: NUM $\sqcap$ PLUR = PLUR. This extension is consistent with the definition of the unifier of two FSs as the GLB of these structures (see, for example, [16]).

## 3 Feature types as data types and feature types as relations

### 3.1 The LIST type as a data type

A list of words will be defined in a LISP-like fashion as either the END of a list or a CONS with two attributes *first* and *rest*:
LIST = END $\vee$ CONS[FIRST:WORD, REST:LIST].
WORD denotes the set of word forms, and the list of words "John likes Mary" will be encoded as

$$\text{CONS}\left[\begin{array}{l}\text{FIRST: JOHN}\\ \text{REST: CONS}\left[\begin{array}{l}\text{FIRST: LIKES}\\ \text{REST: CONS}\left[\begin{array}{l}\text{FIRST: MARY}\\ \text{REST: END}\end{array}\right]\end{array}\right]\end{array}\right]$$

which is a well-formed list with respect to the LIST definition. (We shall use in the following a more concise syntax for lists: END will be written as $\langle\rangle$; CONS[FIRST:WORD, REST:LIST] will be written as $\langle$WORD . LIST$\rangle$; lists will be written using the usual abbreviation for lists: the list of words "John likes Mary" will then be written as $\langle$JOHN LIKES MARY$\rangle$).

### 3.2 The APPEND type as a relation

One can also understand feature types as relations much like those in PROLOG. Let us recall the classical PROLOG definition of append:

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

In PROLOG, the arguments of a term are identified by their positions in the term, and the presence of all arguments is mandatory. In an FS, arguments (feature values) are not identified by their position but by a label, the feature, and the absence of an attribute-value pair will denote any kind of value for this attribute (type $\top$). Using the TFS syntax, where the symbol ':-' after an FS introduces a condition, a definition for append can be as follows:
APPEND = [F: $\langle\rangle$, B: [L] LIST, W: [L]] $\vee$
[F: $\langle$[X] . [L1]$\rangle$, B: [L2] LIST, W:$\langle$[X] . [L3]$\rangle$]
: – APPEND[F: [L1], B: [L2], W: [L3]].
Note that the tagging syntax allows to specification of identity between structures and a partial instance of the structure. This possibility (together with the fact that typing is enforced by the system) allows the writing of a typed version of append, in contrast to the untyped PROLOG version.

### 3.3 Type checking as deduction

Contrary to PROLOG, there is no distinction in TFS between top-level types (which could be interpreted as predicates) and inner types (which could be interpreted as arguments): they are all typed FSs, and the same deduction mechanism applies for the top-level structure as well as for all substructures. A (typed) FS is consistent with respect to a set of type definitions if it unifies with the definition of its type, and if each of its substructures is also consistent. Conditions like in the definition of append above introduce additional constraints which are erased after having been successfully evaluated. When a type is defined as a disjunction, a structure has to be consistent with at least one element of the disjunction (but all possibilities are explored, creating as many possible solutions as there are disjuncts). When a type is defined as a conjunction (using the AND operator noted "$\wedge$"), a structure has to be consistent with every single element of the conjunction. The order used for type checking (roughly top-down) guarantees that the solution the system finds is the GLB of the set of definitions augmented by the initial structure [2].
For example, the (typed) FS AGR[num:PLUR] is consistent with regard to the set of definitions above (Sect.1). The interpreter will apply the definition of AGR at the root of the FS : AGR[num:PLUR] $\sqcap$ [num:NUM,gender:GEN] = AGR[num:PLUR,gender:GEN]
AGR[num:MASC] is an inconsistent (typed) FS : AGR[num:MASC] $\sqcap$ [num:NUM,gender:GEN] = $\bot$ because the types MASC and NUM have only $\bot$, the bottom of the lattice, as a common subtype representing inconsistent information. Note that this type checking process may introduce new type symbols also used for checking, thus defining a type inheritance mechanism.
A full evaluation of APPEND[w:$\langle$A B$\rangle$] produces a set of three FSs:
[F: $\langle\rangle$, B: [L]$\langle$A B$\rangle$, W: [L]] $\vee$
[F: $\langle$[0]A . $\langle\rangle\rangle$, B: [2]$\langle$B$\rangle$, W: $\langle$[0] . [2]$\rangle$] $\vee$
[F: $\langle$[0]A . $\langle$[1]B$\rangle\rangle$, B: [3]$\langle\rangle$, W: $\langle$[0] . $\langle$[1] . [3]$\rangle\rangle$]

## 4 Typed unification grammars

### 4.1 DCGs

In this section, we describe how one can (but should not) write grammars using this formalism. To make comparisons easier, we will start from the small example of DCG presented in [Pereira and Warren 80] and show how this grammar (Fig.2) can be written in TFS.

```
sentence(s(NP,VP)) → noun_phrase(Num,NP), verb_phrase(Num,VP).
noun_phrase(Num, np(Det,Noun)) → determiner(Num,Det), noun(Num,Noun).
noun_phrase(singular,np(Name)) → name(Name).
verb_phrase(Num,vp(TV,NP)) → trans_verb(Num,TV),noun_phrase(N1,NP).
determiner(Num,det(W)) → [W], is_determiner(W,Num).
noun(Num,n(Root)) → [W], is_noun(W,Num,Root).
name(name(W)) → [W], is_name(W).
trans_verb(Num,tv(Root)) → [W], is_trans(W,Num,Root).
```

(Figure 2)

```
is_determiner(all, plural).
is_noun(man, singular, ma
is_noun(men, plural, man).
is_name(mary).
is_trans(likes,singular, like)
is_trans(like, plural, like).
```

In a specification like this, there are three different kinds of information mixed together. Take for example the rule "noun_phrase(Num, np(Det, Noun)) → determiner(Num, Det), noun(Num, Noun)". In this rule we find:

1. a specification of a set of well-formed substrings using the CF skeleton: noun_phrase → determiner, noun;
2. a specification of well-formed (partial) syntactic structures: the structure np(Det, Noun) is well-formed if Det and Noun are a well-formed structure and if its agreement value (variable Num) is the same for the Det, the Noun, and the noun_phrase;
3. a specification of a relation between well-formed (partial) syntactic structures and well-formed substrings by augmenting the CF skeleton with annotations representing those structures.

### 4.2 A TFS specification

All this information mixed together can be separated out and specified in a more modular way.

1. The set of well-formed strings of words is defined as in Sect.2.1, where WORD = allVmen...
2. The set of well-formed partial syntactic structures, i.e. every syntactic constraint like agreement or subcategorisation, should be expressed in this part of the specification.

PHRASAL_CATEGORY = S ∨ NP ∨ VP.
S = [NP: NP[AGR: ⓐNUM], VP: VP[AGR: ⓐ]].

$$\text{NP} = \begin{bmatrix} \text{DET}: & \text{DET}[\text{AGR}: \boxed{a}\text{NUM}] \\ \text{NOUN}: & \text{N}[\text{AGR}: \boxed{a}] \\ \text{AGR}: & \boxed{a} \end{bmatrix} \vee \begin{bmatrix} \text{NAME}: & \text{PN} \\ \text{AGR}: & \text{SG} \end{bmatrix}.$$

VP = [V: TV[AGR: ⓐNUM], NP: NP, AGR: ⓐ].

LEXICAL_CATEGORY = DET ∨ N ∨ PN ∨ V.
DET = ALL ∨ EVERY ∨ A ∨ THE.
ALL = [WORD: all, AGR: PL].
N = MAN ∨ WOMAN.
MAN = [WORD:man, AGR:SG] ∨ [WORD:men, AGR:PL].
PN = JOHN ∨ MARY.
MARY = [WORD: Mary].
V = IV ∨ TV.
TV = LIKE ∨ LOVE.
LIKE = [WORD:likes, AGR:SG] ∨ [WORD:like, AGR:PL].

3. The relation between strings and structures should be stated independently of well-formedness conditions on syntactic structures. It is expressed here in CF manner by using the APPEND relation on strings. (However, we do not advocate the exclusive use of CF-like relations; more complex ones can be specified to gain expressive power, e.g. by incorporating linear precedence rules).

SENTENCE =
[STRING: ⓢ-string, C-STR: S[NP: ⓝp, VP: ⓥp]] :–
NOUN_PHRASE[STRING: ⓝp-string LIST, C-STR: ⓝp]
VERB_PHRASE[STRING: ⓥp-string, C-STR: ⓥp]
APPEND[F: ⓝp-string, B: ⓥp-string, W: ⓢ-string]

NOUN_PHRASE =
[STRING: ⓝp-string, C-STR: NP[DET: ⓓ, NOUN: ⓝ]] :–
DETERMINER[STRING: ⓓ-string, C-STR: ⓓ]
NOUN[STRING: ⓝ-string, C-STR: ⓝ]
APPEND[F: ⓓ-string, B: ⓝ-string, W: ⓝp-string]
∨
[STRING: ⓝp-string1, C-STR: PN[NAME: ⓝ1]] :–
NAME[STRING: ⓝp-string1, C-STR: ⓝ1]

VERB_PHRASE =
[STRING: ⓥp-string, C-STR: VP[V: ⓥTV, NP: ⓝp]] :–
TRANS_VERB[STRING: ⓥ-string, C-STR: ⓥ]
NOUN_PHRASE[STRING: ⓝp-string, C-STR: ⓝp],
APPEND[F: ⓥ-string, B: ⓝp-string, W: ⓥp-string]

LEXICAL_RULE = [STRING: ⟨ⓦ⟩, C-STR: [WORD: ⓦ]].
DETERMINER = LEXICAL_RULE[C-STR: DET].
NOUN = LEXICAL_RULE[C-STR: N].
NAME = LEXICAL_RULE[C-STR: PN].
TRANS_VERB = LEXICAL_RULE[C-STR: TV].

### 4.3 Parsing and generation

Both parsing and generation in the system amount to type inference. Either (1) for parsing or (2) generation yield the same result (3).

(1) SENTENCE[STRING: ⟨Mary likes all men⟩]

(2) SENTENCE

$$\begin{bmatrix} \text{C-STR}: & \text{S} \\ & \begin{bmatrix} \text{NP}: & \text{NP}[\text{NAME}: \text{MARY}] \\ \text{VP}: & \text{VP} \begin{bmatrix} \text{V}: & \text{LIKE} \\ \text{NP}: & [\text{DET}: \text{ALL}, \text{NOUN}: \text{MAN}] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(3) SENTENCE

$$\begin{bmatrix} \text{STRING}: & \langle \boxed{1}\text{Mary } \boxed{2}\text{likes } \boxed{3}\text{all } \boxed{4}\text{men} \rangle \\ \text{C-STR}: & \text{S} \\ & \begin{bmatrix} \text{NP}: & \text{NP}[\text{NAME}: \text{MARY}[\text{WORD}: \boxed{1}], \text{AGR}: \boxed{a}\text{SG}] \\ \text{VP}: & \text{VP} \\ & \begin{bmatrix} \text{V}: & \text{LIKE}[\text{WORD}: \boxed{2}, \text{AGR}: \boxed{a}] \\ \text{NP}: & \text{NP} \\ & \begin{bmatrix} \text{DET}: & \text{ALL}[\text{WORD}: \boxed{3}, \text{AGR}: \boxed{b}\text{PL}] \\ \text{NOUN}: & \text{MAN}[\text{WORD}: \boxed{4}, \text{AGR}: \boxed{b}] \\ \text{AGR}: & \boxed{b} \end{bmatrix} \\ \text{AGR}: & \boxed{a} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This shows that the formalism has the same power as PROLOG to synthesize unspecified arguments, and the same evaluation mechanism can be used for both generation and parsing, depending on the input.

### 4.4 From DCG to HPSG

In the following, we explain how one can generalize the principles used for describing a DCG grammar in TFS to write an HPSG-like grammar. HPSG linguistic objects of all kinds, be they syntactic, phrase-structural, or semantic, are modeled by feature structures [14]. In addition, HPSG relies heavily on the notion of type. Hence, TFS is perfectly suited for an implementation of HPSG. The grammar itself is purely declarative in the sense that it characterizes what constraints should hold on linguistic objects independently of the order in which these constraints are actually applied.

We first generalize the description of linguistic structures: instead of defining explicit types for sentences, noun phrases, etc., we define a generic constituent structure for any kind of phrase. According to the specification of HPSG linguistic objects, we define `SIGN`s as being either of type `PHRASAL_SIGN` or of type `LEXICAL_SIGN` [15]. A `SIGN` has a phonological value, represented as a list of words, and syntactic and semantic information (omitted for this comparison). The subtypes `PHRASAL_SIGN` and `LEXICAL_SIGN` inherit all the attributes and type restrictions of `SIGN`.

(4) SIGN = (PHRASAL_SIGN ∨ LEXICAL_SIGN) ∧
$$\begin{bmatrix} \text{PHON: LIST\_OF\_STRINGS} \\ \text{SYN: CATEGORY} \\ \text{SEM: SEMANTIC\_OBJECT} \end{bmatrix}$$

`PHRASAL_SIGN`s (5) differ from `LEXICAL_SIGN`s (6) by having an additional dtrs ("daughters") attribute that gives information about the (lexical or phrasal) signs which are their immediate constituents. This attribute encodes the kind of information about constituency conventionally described as constituent structures. In addition, the various daughters are distinguished according to what kinds of information they contribute to the sign as a whole. Thus, daughters are classified as heads and complements as in the standard X-bar theory. In order to be a well formed object of type PHRASAL_SIGN, a linguistic object has to obey some general principles such as the "Head Feature Principle" and the "Subcategorization Feature Principle".

(5) phrasal_sign = (HEAD_FP ∧ SUBCAT_FP ∧ ... ∧ (CH_CO_FP ∨ HC*_CO_FP ...)) ∧
$$\begin{bmatrix} \text{DTRS: C\_STRUCTURE} \\ \quad \begin{bmatrix} \text{HEAD-DTR: SIGN} \\ \text{COMP-DTRS: LIST\_OF\_SIGNS} \end{bmatrix} \end{bmatrix}$$

(6) lexical_sign = VERB ∨ PNOUN ∨ NOUN ∨ DET ∨ ...

**General principles** The "Head Feature Principle" ensures that the head features of the head-daughter always be shared with their phrasal projections. It generalizes the passing of agreement information from e.g. a verb to the VP for all kind of constituent and for all information related to agreement and subcategorisation.

(7) HEAD_FP =
$$\begin{bmatrix} \text{SYN: } [\text{HEAD: } \boxed{\text{head}}] \\ \text{DTRS: } [\text{HEAD-DTR: } [\text{SYN: } [\text{HEAD: } \boxed{\text{head}}]]] \end{bmatrix}$$

In the DCG example, subcategorization was expressed by introducing different kinds of lexical categories like transitive verb (TV) vs. intransitive verbs (IV). In HPSG, subcategorization is expressed by using a list of signs. This list specifies the number and kind of signs that the head subcategorizes for the formation of a complete sign. Subcategorization information is described in lexical entries. The "Subcat Feature Principle" ensures that in any phrasal sign, the subcat list of the head-daughter is the concatenation of the list of complement daughters and the subcat list of the mother. (The order of the elements in the complements list does not reflect the surface order but rather the more abstract "obliqueness hierarchy" ([14] Chap.7)).

(8) SUBCAT_FP =
$$\begin{bmatrix} \text{SYN: } [\text{SUBCAT: } \boxed{\text{rest-subcat}}] \\ \text{DTRS: } \begin{bmatrix} \text{HEAD-DTR: } [\text{SYN: } [\text{SUBCAT: } \boxed{\text{subcat}}]] \\ \text{COMP-DTRS: } \boxed{\text{comps}} \end{bmatrix} \end{bmatrix} :-$$
$$\text{APPEND} \begin{bmatrix} \text{F: } \boxed{\text{comps}} \\ \text{B: } \boxed{\text{rest-subcat}} \\ \text{W: } \boxed{\text{subcat}} \end{bmatrix}$$

**Grammar rules** Just as we have generalized the notion of constituency, we are also able to generalize the relations between phonological representations and their desired constituent structure representations. The specialized CF-like relations for a sentence, a noun phrase, and so on in the DCG example can be replaced by two more general rules which specify constituent structure configurations according to the X-bar theory.

The "Complement Head Constituent Order Feature Principle" (9) simply states that a "saturated phrasal sign" (i.e. with `[syn:[subcat:⟨⟩]]`) is the combination of an unsaturated phrasal head with one phrasal complement (e.g. S → NP VP).

(9) CH_CO_FP =
$$\begin{bmatrix} \text{PHON: } \boxed{\text{phon}} \\ \text{SYN: } [\text{SUBCAT: } \langle\rangle] \\ \text{DTRS: } \begin{bmatrix} \text{HEAD-DTR: PHRASAL\_SIGN } [\text{PHON: } \boxed{\text{head-phon}}] \\ \text{COMP-DTRS: } \langle \text{SIGN } [\text{PHON: } \boxed{\text{comp-phon}}] \rangle \end{bmatrix} \end{bmatrix}$$
$$:- \text{ APPEND} \begin{bmatrix} \text{F: } \boxed{\text{comp-phon}} \\ \text{B: } \boxed{\text{head-phon}} \\ \text{W: } \boxed{\text{phon}} \end{bmatrix}$$

The "Head Complements Constituent Order Feature Principle" (13) states that an "unsaturated phrasal sign" is the combination of a lexical head and any number of complements (e.g. VP → V XP*). The relation ORDER_COMPL is used for specifying the ordering of the phonological values of all complements. The phonological value of the whole phrase can then be specified as the concatenation of the head phonology value with the complement phonology value.

(13) HC*_CO_FP =
$$\begin{bmatrix} \text{PHON: } \boxed{\text{phon}} \\ \text{DTRS: map} \begin{bmatrix} \text{HEAD-DTR: LEXICAL\_SIGN} \\ [\text{PHON: } \boxed{\text{head-phon}}] \\ \text{COMP-DTRS: } \boxed{\text{comps}} \end{bmatrix} \end{bmatrix} :-$$
$$\text{APPEND} \begin{bmatrix} \text{F: } \boxed{\text{head-phon}} \\ \text{B: } \boxed{\text{comp-phon}} \\ \text{W: } \boxed{\text{phon}} \end{bmatrix}$$
$$\text{ORDER\_COMPL} \begin{bmatrix} \text{COMPS: } \boxed{\text{comps}} \\ \text{PHON: } \boxed{\text{comp-phon}} \end{bmatrix}$$

(10) SIGN[PHON:⟨"Mary" "likes" "all" "men"⟩].

(11) SIGN [ DTRS: [ HEAD-DTR: [ DTRS: [ HEAD-DTR: LIKE
COMP-DTRS: ⟨ [ DTRS: [ HEAD-DTR: [DTRS:[HEAD-DTR: MAN]]
COMP-DTRS: ⟨ALL⟩ ] ] ⟩ ]
COMP-DTRS: ⟨MARY⟩ ] ] ] ] ] .

(12)

PHRASAL_SIGN [ PHON: ⟨[1]"Mary" . [2]⟨[3]"likes" . [4]⟨"all" "men"⟩⟩⟩
SYN: [HEAD:[5][PERSON:3, NUM:sg, CASE:nom], SUBCAT:⟨⟩]
DTRS: [ HEAD-DTR: PHRASAL_SIGN [ PHON: [2]
SYN: [HEAD:[5], SUBCAT: ⟨[6]⟩]
DTRS: [ HEAD-DTR: LEXICAL_SIGN [ PHON: ⟨[3]⟩
SYN: [HEAD: [5]
SUBCAT: ⟨[6] [7]⟩ ] ]
COMP-DTRS: ⟨[7]PHRASAL_SIGN[PHON:[4] ...]⟩ ] ]
COMP-DTRS: ⟨ [6]PHRASAL_SIGN [ PHON: ⟨[1]⟩
SYN: [ HEAD: [LEX: [1]
NUM: sg ]
SUBCAT: ⟨⟩ ] ] ⟩ ] ]

**Lexical entries**

ALL = DET[SYN: [HEAD: [LEX:"all", NUM:pl]]] .
MAN = NOUN[SYN:[HEAD: [LEX:"man", NUM:sg]∨ ]].
[LEX:"men", NUM:pl]
MARY = PNOUN[SYN:[HEAD: [LEX:"mary", NUM:sg]]].
LIKE = TRANS ∧ (3RD_SG[SYN:[HEAD:[LEX:"likes"]]]∨).
PL[SYN:[HEAD:[LEX:"like"]]]
3RD_SG = [SYN:[HEAD:[PERSON:3, NUM:sg]]].
TRANS = [SYN:[SUBCAT:⟨[SYN:[HEAD:[CASE:acc]]] ⟩]].
[SYN:[HEAD:[CASE:nom]]]

## 5 Parsing and generation

Either (10) for parsing or (11) generation, the evaluation yields the same fully specified sign (12).

## 6 Conclusion

The main characteristics of the formalism we presented are (1) type inheritance which provides a clean way of defining classes and subclasses of objects, and (2) an evaluation mechanism based on typed unification which provides a very powerful and semantically clear means of specifying and computing relations between classes of objects.
The possibility of defining types as (conditional) expressions of typed FSs encourages a very different approach to grammar specification than integrated CF based approaches like DCG or LFG: the grammar writer has to define the set of linguistic objects relevant for the problem, define the possible relations between these objects, and specify explicitly the constraints between objects and relations.
The TFS system has been implemented in Common-Lisp and has been tested on Symbolics, TI Explorer, VAX and Allegro Common-Lisp. Sample grammars have been developed([6], [18]) in order to demonstrate the feasibility of the approach.

**Acknowledgments** The current system is based on a previous implementation carried out by the authors at ATR, Kyoto, as a part of a visiting research program. We would like to thank Dr. Akira Kurematsu, president of ATR Interpreting Telephony Research Laboratories for making our stay possible, and Mr. Teruaki Aizawa, head of the Natural Language Understanding Department for his constant support. We owe many clarifications to Sondra Ahlen with whom we had many lively discussions. This paper has benefited from many comments from our collegues at the IMS of the University of Stuttgart.

## References

[1] Hassan Aït-Kaci: *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, Ph.D. Thesis, University of Pennsylvania. 1983

[2] Hassan Aït-Kaci: "An Algebraic Semantics Approach to the effective Resolution of Type Equations." in: *Theoretical Computer Science*, Vol. 45, p. 293-351. 1986

[3] Hassan Aït-Kaci, Patrick Lincoln: *LIFE: a natural language for natural language*, MCC Technical Report ACA-ST-074-88.

[4] D.J. Arnold, S. Krauwer, M. Rosner, L. des Tombes, G.B. Varile: "The <C,A>,T framework in Eurotra: a theoretically committed notation for MT", *11th International Conference on Computational Linguistics (COLING-86)*, Bonn. 1986.

[5] Hélène Bestougeff, Gérard Ligozat: "Parameterized abstract objects for linguistic information processing", *2nd European ACL Conference*, Geneva. 1985.

[6] Martin C. Emele: "A Typed Feature Structure Unification-based Approach to Generation" in: *Proceedings of the WGNLC of the IECE 1988*, (Japan: Oiso University) 1989.

[7] Martin Emele, Rémi Zajac: "RETIF: A Rewriting System for Typed Feature Structures", (Kyoto) 1989, [ATR Technical Report TR-I-0071]

[8] Martin Emele, Rémi Zajac: "Multiple Inheritance in RETIF", (Kyoto) 1989, [ATR Technical Report TR-I-0114]

[9] Roger Evans , Gerald Gazdar: "Inference in DATR", in: *4th European ACL Conference*, Manchester. 1989.

[10] Jens E. Fenstad, Per-Kristian Halvorsen, Tore Langholm, Johan van Benthem: *Situation, language, and logic*, 1987,(Dordrecht: Reidel)

[11] Marc Moens, Jo Calder, Ewan Klein, Mike Reape, Henk Zeevat: "Expressing generalizations in unification-based formalisms", in: *4th European ACL Conference*, 1989, (Manchester)

[12] Fernando C.N. Pereira, David H.D. Warren: "Definite Clause Grammars for Language Analysis-A Survey of the Formalism and a Comparison with Augmented Transition Networks", in: *Artificial Intelligence 13*: 231-278. 1988.

[13] Harry H. Porter: "Incorporating Inheritance and Feature Structures into a Logic Grammar Formalism", in: *25th Annual Meeting of the ACL*, 1987, (Stanford)

[14] Carl Pollard, Ivan A. Sag: *Information-based Syntax and Semantics*. CSLI, Lectures Notes Number 13, Chicago University Press, 1987

[15] Carl Pollard: "Sorts in unification-based grammar and what they mean", To appear in M. Pinkal and B. Gregor (eds.), *Unification in natural language analysis*, 1988.

[16] Stuart M. Shieber: *An Introduction to Unification-based Approaches to Grammar*, CSLI, Lecture Notes Number 4, Chicago University Press, 1986.

[17] Gert Smolka: *A feature logic with subsorts*, LILOG report 33, IBM Deutschland, Stuttgart, 1987.

[18] Rémi Zajac: "A Transfer Model Using a Typed Feature Structure Rewriting System with Inheritance.", in: *Proceedings of the 27th Annual Meeting of the ACL-89* (Vancouver, Canada) 1989.