Inheritance and Constraint-Based Grammar Formalisms

Rémi Zajac*†
Project POLYGLOSS

We describe an approach to unification grammars that integrates two paradigms: the object-oriented approach, which offers multiple inheritance, complex objects with role-value restrictions and role-values equality, querying as subsumption; the relational programming approach, which offers declarativity, logical variables, nondeterminism with backtracking, and existential queries. This approach is embodied in a constraint-based object-oriented formalism. The interpreter of the formalism is described as a term rewriting system based on unification of typed feature structures.

The grammar writer organizes unification grammars as inheritance networks of typed feature structures. Complex linguistic structures are described by means of recursive type constraints. We illustrate the use of inheritance networks with two examples: an HPSG example where implication (as used for "principles") is modeled using inheritance and an example of bilingual transfer where the minimal amount of information needed for the translation is specified at different levels of generalization.

1. Introduction

Ideally, a linguistic formalism combining the best of the object-oriented approach and the unification-based approach would be realized in a constraint-based architecture for an object-oriented language based on inheritance, feature structures, and unification.

The Typed Feature Structure language (TFS) is an attempt to provide a synthesis of several key concepts stemming from unification-based grammar formalisms (feature structure: Kay 1984) knowledge representation languages (inheritance), and logic programming (narrowing). The formalism supports an object-oriented style based on abstraction and generalization through inheritance; it is a fully declarative formalism based on unification of typed feature structures. It is flexible and has enough expressive power to support various kinds of linguistic theories, not necessarily based on constituency[1].

The use of an object-oriented methodology for natural language processing is very attractive, and the use of inheritance offers a number of advantages such as abstraction and generalization, information sharing and default reasoning, and modularity and reusability (Daelemans 1990). Inheritance-based descriptions are already used in computational linguistics: linguistic theories such as Systemic Functional Grammar (Halliday 1985), Word Grammar (Fraser and Hudson 1990), or HPSG (Pollard

* IMS-CL/IfI-AIS, University of Stuttgart, Azenbergstraße 12, D-W-7000 Stuttgart 1. E-mail: zajac@informatik.uni-stuttgart.de.

† 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 author and should not be interpreted as representing official policies.

1 In particular, it allows a direct implementation of HPSG-style grammars (Emele 1988): HPSG is so far the only linguistic theory based on both inheritance *and* feature structures.

and Sag 1987) make use of inheritance to describe linguistic structures at the lexical, morphological, syntactic, or semantic (conceptual) levels. These theories are usually directly implemented in object-oriented programming languages (e.g., LOOM in the case of the PENMAN system [Mann and Matthiessen 1985]), but there is a growing number of linguistic formalisms used for specific purposes, e.g., DATR (Evans and Gazdar 1989) for the lexicon.

On the other hand, current linguistic theories such as LFG, UCG, HPSG, and some formalisms for linguistic description such as FUG or PATR-II are based on the notion of partial information: linguistic structures are described using feature structures that give partial information about the object being modeled, a linguistic structure being described by a set of feature structures that mutually constrain the description. Feature structures are partially ordered according to a subsumption ordering interpreted as an ordering on the amount of conveyed information; the combination of information is defined as the unification of feature structures. Formalisms based on feature structure and unification are declarative, and they can be given a sound formal semantics.

Combining object-oriented approaches to linguistic description with unification-based grammar formalisms, as in HPSG, is very attractive. On one hand, we gain the advantages of the object-oriented approach: abstraction and generalization through the use of inheritance. On the other hand, we gain a fully declarative framework, with all the advantages of logical formalisms: expressive power, simplicity, and sound formal semantics. To arrive at such a result, we have to enrich the formalism of feature structures with the notion of inheritance and abandon some of the procedural features of object-oriented languages in order to gain referential transparency.

Referential transparency is one of the characteristic properties of declarative languages (Stoy 1977), where the meaning of each language construct is given by a few simple and general rules. For example, the value of a variable should be independent from its position within the scope of its declaration. This is true for PROLOG variables inside a clause, but not for PASCAL or LISP variables that make use of assignment. A higher level example is the meaning of a procedure: it is not transparent if the procedure makes use of global variables that are set by some other procedure. Similarly, the meaning of a PROLOG predicate should be transparent because there is no global variable, but a predicate definition might be modified during execution by imperative predicates such as `assert` and `retract`, thus destroying the referential transparency of pure PROLOG.

Clearly, most of the object-oriented languages lack referential transparency in several ways, using for example procedural attachments for object methods. Another example is the use of nonmonotonic inheritance, which is advocated in computational linguistics by, for example, Evans and Gazdar 1989; Bouma 1990; De Smedt and de Graaf 1990; and Fraser and Hudson 1990. Nonmonotonic inheritance is seen as a practical device designed to deal with exceptions, but such a feature goes against generality and referential transparency. Furthermore, as expressed by Etherington et al. 1989, a still unresolved issue in nonmonotonic reasoning is the issue of

> ...scaling formal non-monotonic theories up to real problems (merely a formality?). Most existant theories are intractable—some don't have even a proof theory—and it is often difficult to tell how large bodies of information will (or even *should*) interact.

Given the complexity of the state of the art in nonmonotonic reasoning and the lack of

a basic commonly agreed formalization,[2] the issue of nonmonotonicity is not addressed in the work described in this article.

Knowledge representation languages are evolving toward more declarativity, as exemplified by the evolution from KL-ONE (Brachman and Schmolze 1985) to languages such as CLASSIC (Borgida et al. 1989) or LOOM (MacGregor 1988, 1990). The *terminological* component describing the objects (the data model of object-oriented database systems) has always been more declarative than the *assertional* component (procedural attachment or methods), and the current trend is to integrate those two components more closely, where the assertional component is some kind of rule-based system, as in LOOM (Yen, Neches, and MacGregor 1988).

Typed feature structures are very similar to structured objects of object-oriented languages and to conceptual structures of knowledge representation languages. Thus, typed feature structures have the potential to act as a lingua franca for both computational linguistics and artificial intelligence, and this should ease the communication between those two worlds. Since conceptual structures are used for example in text generation (Bourbeau et al. 1990) or knowledge-based machine translation (Nirenburg et al. 1992), typed feature structures provide an attractive alternative to current procedural implementations.

In Section 2, we present a language that combines the notions of partial information and inheritance in a fully declarative framework. It is based on feature structures augmented with the notion of types, which are organized into an inheritance network. Using types, it is possible to define structured domains of feature structures and to classify feature structures. Logical conditions are attached to types, akin to method attachment, but in a fully declarative framework. Recursivity is an integral part of the language, giving the necessary expressive power for describing complex recursive linguistic structures. We end the section by an overview of the TFS abstract rewrite machine used for computing descriptions of the meaning of typed feature structures.[3]

Section 3 describes the use of inheritance in two examples of unification grammars using the TFS formalism: an HPSG grammar for a fragment of English and an LFG-style transfer grammar for a small machine translation problem between English and French.

2. Inheritance Networks of Typed Feature Structures

Assume the existence of an (abstract) informational domain $\mathcal{U}$, for example, the set of linguistic objects. *Feature structures* describe objects of this universe by specifying values for attributes of objects and equality constraints between some values. More precisely, as feature structures can provide only partial information about the objects they describe, a feature structure denotes a *set* of objects in this universe. This set could be a singleton set, for example, in the case of atomic feature structures. Feature structures are ordered by a subsumption relation: a feature structure f_1 subsumes another feature structure f_2 iff f_1 provides *the same or less information* than f_2: $f_1 \geq f_2$. In our universe, this means that the set described by f_1 is a *superset* of the set described by f_2. Note that there can be feature structures that cannot consistently describe the

2 The KR logic (Rounds and Kasper 1986), from which most of other logic formalizations of feature structures are derived, plays this role in formal accounts of feature structures.

3 This formalism is fully implemented. An interpreter for the TFS rewrite machine has been implemented at the University of Stuttgart by Martin Emele and the author and has been used to test several linguistic models such as DCG, LFG, HPSG, and SFG [Emele and Zajac 1990b; Emele et al. 1990; Zajac 1990a; Bateman and Momma 1991].

same objects. For example, a feature structure describing verb phrases and a feature structure describing noun phrases are not consistent: the intersection of the sets they denote is usually empty.

As different sets of attribute-value pairs make sense for different kinds of objects, we also divide our feature structures into different types. These types are ordered by a subtype relation: a type t_2 is a *subtype* of another type t_1 if t_2 provides *at least as much information* as t_1. For example, assuming that a verb phrase is a phrase, then the set of verb phrases is included in the set of phrases. Using types to model this taxonomic hierarchy, the type symbol **VP** denotes the set of verb phrases, the symbol **PH** denotes the set of phrases, and we define **VP** as a subtype of **PH**.

This description implies that, if we know that a linguistic object is a verb phrase, we can deduce that it is a phrase. This deduction mechanism is expressed in our type system as *type inheritance*. Furthermore, with each type we associate *constraints* expressed as feature structures, thereby defining an inheritance network of typed feature structures: if a feature structure is of type t and there exist supertypes of t, then t inherits all the attribute-value pairs and equality constraints of the feature structures associated with all the supertypes of t.

Computation is performed by a typed feature structure machine capable of checking a set of type constraints defined as an inheritance network of typed feature structures. Given a typed feature structure inheritance network, we query the machine by asking if some feature structure satisfies the constraints defined by the network. To produce the answer, the system proceeds by gradually adding the constraints that should be satisfied by the query: an answer will be a set of feature structures where each feature structure is subsumed by the query and where all the type constraints of the network hold on all substructures of the elements of the answer. The answer is the empty set when the query does not satisfy the constraints defined by the network.

Related work.
The basic approach described in this section is based on original work by Aït-Kaci (1984, 1986) on the KBL language and has also been influenced by the work on HPSG by Pollard and Sag (1987) and Pollard and Moshier (1990). Among the growing literature on the semantics of feature structures, many relevant results and techniques have been published by Smolka (1988, 1989), Smolka and Aït-Kaci (1988), and Aït-Kaci and Podelski (1991). Based on Pollard and Sag (1987), Pollard (1990), and Pollard and Moshier (1990), a computational formalism, very close to the TFS formalism, is currently under design at CMU for implementing HPSG (Carpenter 1990, Franz 1990).

The early work presented in Emele and Zajac (1989a) was an attempt to directly implement the ideas presented in Aït-Kaci (1984, 1986); Emele and Zajac (1989b) was already a departure from Aït-Kaci's KBL language, implementing a different evaluation strategy that allowed the use of cyclic feature structures. Compared with the KBL language, the current TFS language

- allows the use of cyclic feature structures,
- has a simple operational semantics implementing an inheritance rule and a specialization rule,
- uses a lazy evaluation scheme for the evaluation of constraints,
- implements static coherence checks,
- has a simple syntax distinguishing the definition of the partial order on type symbols, and the definition of constraints associated to types.

2.1 Types

In the following presentation, we adopt an algebraic approach based on lattice theory (Birkoff 1984; Aït-Kaci 1984). Alternative presentations could be developed as well; for example, a proof-theoretical approach using an adaptation of a feature logic (Rounds and Kasper 1986; Smolka 1988). It is possible to prove formally the equivalence of these different models, as this is done for the LIFE language in Aït-Kaci and Podelski (1991). The presentation is nevertheless rather informal, and a more technical account can be found in Emele and Zajac (1990a) and Zajac (1990b).

The universe of feature structures is structured in an inheritance network that defines a partial ordering on kinds of available information. The backbone of the network is defined by a finite set of type symbols $\mathcal{T}$ together with a partial ordering $\leq$ on $\mathcal{T}$: the partially ordered set (poset) $\langle \mathcal{T}, \leq \rangle$. The ordering $\leq$ defines the subtype relation: for $\mathbf{A}, \mathbf{B} \in \mathcal{T}$ we read $\mathbf{A} \leq \mathbf{B}$ as "**A** is a subtype of **B**." We call the smallest types of $\mathcal{T}$ the minimal types.

To have a well-behaved type hierarchy, we require that $\langle \mathcal{T}, \leq \rangle$ be such that:

- $\mathcal{T}$ contains the symbols $\top$ and $\bot$, where $\top$ is the greatest element and $\bot$ is the least element of $\mathcal{T}$.[4]
- any two type symbols **A** and **B** of $\mathcal{T}$ have a greatest common lower bound written $glb\{\mathbf{A}, \mathbf{B}\}$. A poset where greatest common lower bounds exist is a meet semi-lattice: we introduce a new operation $\mathbf{A} \wedge \mathbf{B} = glb\{\mathbf{A}, \mathbf{B}\}$, where $\mathbf{A} \wedge \mathbf{B}$ is called the *meet* of **A** and **B**.

Since we allow the user to specify any finite poset, a technicality arises when two types do not have a unique greatest common lower bound: in that case, the set of greatest common lower bounds is interpreted disjunctively using the following powerlattice construction, which preserves the ordering and the existing meets.

The poset $\langle \mathcal{T}, \leq \rangle$ is embedded in $\langle crowns(\mathcal{T}), \sqsubseteq_H \rangle$. The set $crowns(\mathcal{T})$ is the set of all nonempty subsets of incomparable elements of $\mathcal{T}$ (the "crowns" of $\mathcal{T}$). These subsets are partially ordered by the Hoare ordering $\sqsubseteq_H$: $\forall X, Y \in crowns(\mathcal{T})$, $X \sqsubseteq_H Y$ iff $\forall x \in X$, $\exists y \in Y$ such that $x \leq y$.

The canonical injection of $\mathcal{T}$ in $crowns(\mathcal{T})$, which takes any element x of $\mathcal{T}$ into the singleton $\{x\}$ trivially preserves the ordering: $\{x\} \sqsubseteq_H \{y\}$ iff $x \leq y$.

The meet between two elements X and Y of $crowns(\mathcal{T})$, $X \sqcap Y$, is defined as the union of the intersection of each of the principal ideals generated by the elements of X with each of the principal ideals generated by the elements of Y and then extracting the maximal elements. It is easy to see that existing meets are preserved: $\{z\} = \{x\} \sqcap \{y\}$ iff $z = x \wedge y$.

Some meets are added, as in the following example. Let $\langle \{a, b, c, d\}, \leq \rangle$ be a poset where $d < b$, $d < a$, $c < b$, and $c < a$ (this poset is represented using a Hasse diagram in Figure 1). The meet $a \wedge b$ does not exist, but $\{a\} \sqcap \{b\} = \{c, d\}$, and this meet is interpreted disjunctively.

The join between two elements X and Y of $crowns(\mathcal{T})$, $X \sqcup Y$, is defined as the set of maximal elements of the union of X and Y. It can be shown that, equipped with $\sqcap$ and $\sqcup$, $\langle crowns(\mathcal{T}), \sqsubseteq_H \rangle$ is a distributive lattice. This construction is carried over to typed feature structures, with the property that this definition of the join does not lose information, contrary to the strict generalization of feature structures: this definition of

4 $\top$ represents underspecified information, and $\bot$ represents inconsistent information.

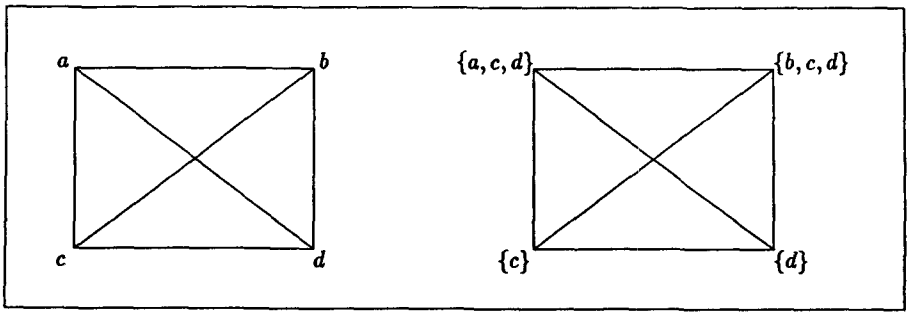

Figure 1
A poset and the set of the principal ideals generated by the elements of the poset ordered by set inclusion.

the join is appropriate to represent disjunctive information as, for example, generated by a nondeterministic computation (see Section 2.4).

This powerlattice construction is completely transparent to the user, and to simplify the presentation, we will assume in the following that $\langle \mathcal{T}, \leq \rangle$ is a meet semilattice.

2.2 Feature Structures

We use the attribute-value matrix (AVM) notation for feature structures, and we write the type symbol for each feature structure in front of the opening square bracket of the AVM. In the remainder of this section, we shall implicitly refer to some given signature $\langle \mathcal{T}, \leq, \mathcal{F} \rangle$ where $\langle \mathcal{T}, \leq, \rangle$ is a type hierarchy and $\mathcal{F}$ is a set of feature symbols, and we shall also assume a set of variables V.

A typed feature structure t is then an expression of the form

$$\boxed{X}\,\mathbf{A}\begin{bmatrix} f_1\colon\ t_1 \\ \dots \\ f_n\colon\ t_n \end{bmatrix}$$

where $\boxed{X}$ is a variable in a set of variables V, $\mathbf{A}$ is a type symbol in $\mathcal{T}$, $f_1, \dots, f_n$ (with $n \geq 0$) are features in $\mathcal{F}$, and $t_1, \dots, t_n$ are typed feature structures.

We have to add some restrictions that capture properties commonly associated with feature structures:

1. A feature is a selector that gives access to a substructure: it has to be unique for a given feature structure.
2. $\bot$ represents inconsistent information: it is not allowed in a feature structure.
3. A variable is used to capture equality constraints ("reentrancy") in the feature structure, and the shared value is represented only once: there is at most one occurrence of a variable $\boxed{X}$ that is the root of a structure different from $\boxed{X}\top$.

Given a signature $\langle \mathcal{T}, \leq, \mathcal{F} \rangle$, feature structures are partially ordered by a subsumption relation. This captures the intuitive notion that a feature structure t' containing

more information than a feature structure t is more specific than t. A feature structure t subsumes a structure t', $t \geq t'$ iff:

1. all paths in t are in t';
2. all equality constraints in t hold in t'.
3. for a given path in t, its type is greater or equal than the corresponding type in t'.

Since we have a partial order on feature structures, the meet operation between two feature structures t and t' is defined in the usual way as the greatest common lower bound of t and t'. It is computed using a typed unification algorithm. A feature structure is represented as a graph where each node has a type, an equivalence class used to represent equational constraints ("co-references"), and a set of outgoing arcs. The unification algorithm uses the union/find procedure on an inverted set representation of the equivalence classes adapted by Aït-Kaci (1984) after Huet (1976). The actual algorithm used in the system is optimized using several different techniques to minimize copying and to behave as efficiently as a pattern-matcher in cases when one of the feature structures subsumes the other (Emele 1991).

2.3 Inheritance Network of Feature Structures

The template mechanism (as, for example, in PATR-II [Shieber 1986]) already provides a simple inheritance mechanism used to organize lexical descriptions. In comparison, networks of typed feature structures are more expressive and provide a more general and more powerful inheritance mechanism, which allows the use of recursive type definitions, whereas recursivity is forbidden in templates since they are expanded statically using a macro-expansion mechanism. Furthermore, typing provides a notion of well-formedness that is used to implement a type-discipline and consistency checks, giving the user the means of checking statically the coherence of a set of type definitions.

2.3.1 Type Discipline. As different combinations of attribute value pairs make sense for different kinds of objects, we divide our feature structures into different classes by associating with a type a certain class of feature structures. Each type defines a specific collection of features that are appropriate for it, restrictions on their possible values, and equality constraints between values. The definition of a type **A** is *def*(**A**), expressed as a feature structure (of type **A**). A type symbol that does not have any definition is called an atomic type. A type that has a definition is called a complex type.

The association of types and feature structures allows the definition of well-formedness conditions for feature structures using the following two typing rules:

Typing Rule 1:

A feature appearing in a feature structure has always to be declared as appropriate for some type. The user cannot introduce arbitrary features: one must declare all the features that one will use. All features that are not explicitly declared as being appropriate for some type are by default defined as being appropriate for $\perp$.

Typing rule 2:

A feature structure cannot have a feature that is not appropriate for its

type or for one of the supertypes. Thus, any feature structure with a feature f has to belong to some type for which f is appropriate.

These well-formedness conditions are enforced at compile-time using a type inference procedure which infers for each feature structure its possible minimal types. If the inferred type is $\perp$, an error is reported indicating that the respective feature structure does not obey the typing rules. The internal representation built by the compiler uses these inferred minimal types to ensure that it is not possible to add an arbitrary feature to a feature structure during computation, but only those declared for the type of the structure, thus preserving well-formedness.

2.3.2 Inheritance and Generalization. Since types are organized in an inheritance network, a type inherits all the features, value restrictions, and equality constraints from all its super-types monotonically: the constraints expressed as feature structures are conjoined using typed unification. The compiler makes sure that the user has specified an inheritance network, building an internal representation where for every two types such that $\mathbf{A} \leq \mathbf{B}$ we have $def(\mathbf{A}) \leq def(\mathbf{B})$.[5] If there is a type **A** such that $\mathbf{A} \neq \perp$ and $def(\mathbf{A}) = \perp$, the network is inconsistent and an error is reported. The compiler also has a generalization step where all constraints common to all subtypes of a given type are also defined for that type.

2.3.3 Interpreting an Inheritance Network. The constraints expressed as an inheritance network are interpreted as follows. For a given typed feature structure $t = \boxed{\mathbf{X}}\mathbf{A}[\ldots]$, the feature structure t belongs to the domain of **A** (i.e., it satisfies the constraints associated with **A**) if and only if:

Inheritance Rule:
t satisfies the constraints specified by the definition of **A** and by the definitions of all the supertypes of **A**;

Specialization Rule:
t satisfies the constraints specified by the definitions of at least one of the subtypes of **A**.

The inheritance rule states the necessary conditions for a feature structure of type **A** to satisfy the constraints associated with **A**. The specialization rule states the sufficient conditions and implements a kind of closed-world assumption: a type is exhaustively covered by its subtypes. For example, a feature structure of type **LIST** can be an empty list (type **NIL**) or a nonempty list (**CONS**), but nothing else. These two rules are implemented by the TFS interpreter described in Section 2.4.

Example. A simple example of an inheritance network of feature structures is displayed in Figure 2 using Hasse diagrams. The subnetwork on the right defines a domain of lists expressed as feature structures: the set of all possible lists is defined by the type **LIST**, which has no associated constraints. This type has two subtypes:

- **NIL** is an atomic type and represents the empty list;
- **CONS** is a complex type and represents the set of all possible nonempty lists and defines the following constraints. A feature structure of type

5 Inheritance is pre-computed statically: $\mathbf{A} \leq \mathbf{B} \Rightarrow def(\mathbf{A}) = def(\mathbf{A}) \wedge def(\mathbf{B})$.

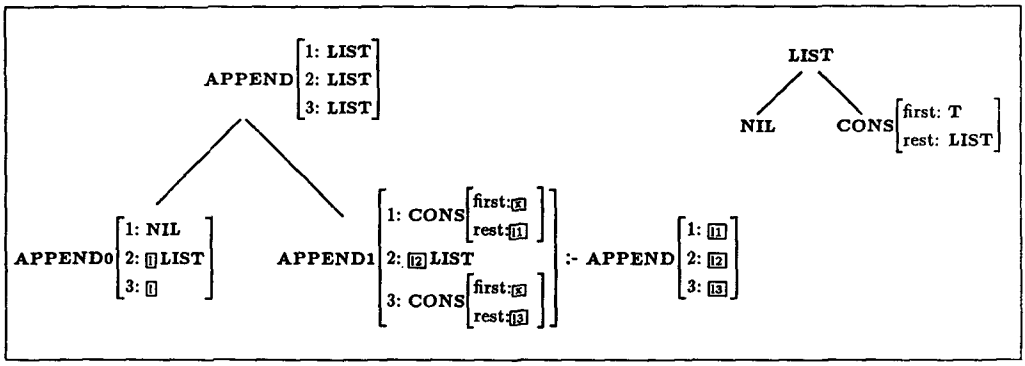

Figure 2
Type hierarchy for LIST and APPEND ($\top$ and $\bot$ omitted).

> **CONS** has only two features (first typing rule): *first*, whose value is the first element of the list and can be anything, and *rest*, whose value is constrained to be a list.[6] Note that this latter constraint is recursive.

The subnetwork on the left defines the domain of the **APPEND** relation, encoded using feature structures. The constraints associated with the supertype **APPEND** say that it has three arguments, identified by the features 1, 2, and 3, and that the values of all arguments should be in the **LIST** domain. The subtypes **APPEND0** and **APPEND1** encode the two cases where the first argument is the empty list (**APPEND0**), and the nonempty list (**APPEND1**), in a way similar to the classical PROLOG encoding. As shown for the type **APPEND1**, it is possible to have additional constraints that are not represented in the feature structure proper: they are introduced by the ':-' sign.[7] These conditions can be inherited and are conjoined using the logical *and* operation. For **APPEND1**, the condition states the recursive constraint on the concatenation of the lists, which is expressed as a feature structure of type **APPEND**.

2.4 The TFS Abstract Rewrite Machine

The meaning (denotation) of a typed feature structure t in a universe $\mathcal{U}$ defined by an inheritance network is represented by the largest set of feature structures $\mathcal{S}_t = \{t_1, \ldots, t_n\}$ such that, for all t_i

1. $t_i \leq t$, and
2. for all substructures $u = \boxed{X}\mathbf{A}[\ldots]$ of t_i, type **A** is a minimal type and $u \leq def(\mathbf{A})$.

The first condition says that all the elements of $\mathcal{S}_t$ satisfy the constraints expressed by t. The second condition says that all the elements of $\mathcal{S}_t$ satisfy the constraints defined

6 Conversely, using the second typing rule, we can deduce that **CONS** is a possible type for [*first* : **Mary**, *rest* : $\top$], since the combination of *first* and *rest* is defined as appropriate for **CONS**.

7 This construction provides room for future evolution of the formalism by adding new kinds of constraints that cannot be directly expressed in the AVM format e.g., negation. A definition "$X :- Y, Z.$" is read "X such that Y and Z."

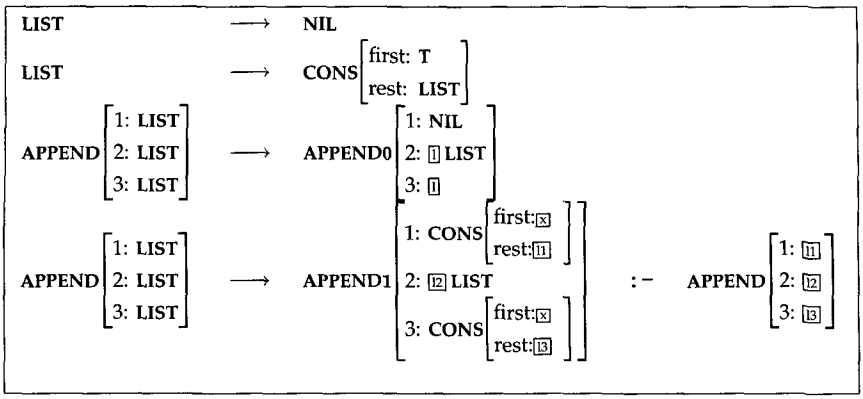

Figure 3
Rewrite rules for LIST and APPEND.

by the network. If S_t is empty, the feature structure t is inconsistent (modulo the constraints of the inheritance network). S_t can be finite, e.g. in the case of a dictionary, but it can also be infinite in the case of recursive types: for example, the set of feature structures subsumed by **LIST** is the (infinite) set of all possible lists represented as feature structures.[8]

In this section, we describe an abstract rewrite machine for computing the representation of the denotation of typed feature structures given an inheritance network. The rewrite mechanism is based on a variant of narrowing[9] adapted to feature structures.

An inheritance network of feature structures is compiled into a rewriting system as follows: each direct link between a type **A** and a subtype **B** generates a rewrite rule of the form $\mathbf{A}[a] \rightarrow \mathbf{B}[b]$ where $\mathbf{A}[a]$ and $\mathbf{B}[b]$ are the definitions of **A** and **B**, respectively. Figure 3 shows the rewrite rules corresponding to the network of Figure 2.

The interpreter is given a "query" (formulated as a typed feature structure) to evaluate. The first step is to check that the feature structure respects the two typing rules (Section 2.3.1). The idea is then to try to satisfy all the constraints defined by the inheritance network by incrementally adding more constraints to the query using the rewrite rules (nondeterministically) to get closer to the solution step by step. The rewriting process stops when conditions 1 and 2 described above hold.

A rewrite step for a structure t is defined as follows: if u is a substructure of t at path p and u is of type **A**, and there exists a rewrite rule $\mathbf{A}[a] \rightarrow \mathbf{B}[b]$ such that

- $\mathbf{A}[a] \wedge u \neq \bot$, and
- $\mathbf{A}[a] \wedge u < \mathbf{A}[a]$

then the right-hand side $\mathbf{B}[b]$ is unified with the substructure u at path p, giving a new structure t' that is more specific than t (Figure 4).

8 See Aït-Kaci (1984), Pollard and Moshier (1990), and Emele and Zajac (1990a) for fixed-point characterizations of the denotation of typed feature structures.

9 Narrowing uses unification instead of pattern-matching for checking the applicability of the l.h.s. of a rule. Narrowing is used in the logic programming paradigm (e.g., as an alternative to resolution for implementing PROLOG interpreters). Pattern-matching is used in the functional programming paradigm.

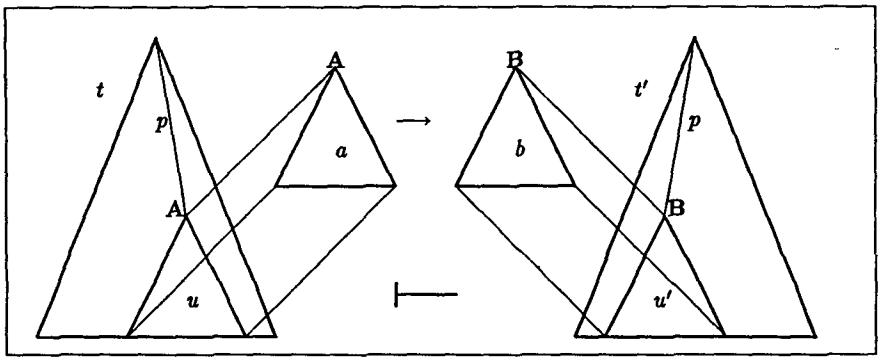

Figure 4
A rewrite step.

The first condition checks that the rule is applicable: the l.h.s. has to be consistent with the substructure. The implementation factorizes common l.h.s., avoiding the DNF expansion: if a l.h.s. l of a rule $l \rightarrow r$ is not consistent with the substructure, this computation branch is a failure branch, and all rules $u \rightarrow v$ where $u \leq l$ are discarded in one step without further computation.

The second condition implements a *lazy rewriting* strategy: if $\mathbf{A}[a] \wedge u$ is equal to $\mathbf{A}[a]$, all rules $\mathbf{A}[a] \rightarrow \mathbf{B}[b]$ could be applied with success, and failure could come only from the rewriting of some other substructures after the exploration of all choices for u. To avoid the exploration of failure branches as much as possible, the evaluation of the substructure u is suspended until the evaluation of some other substructure having some part in common with u makes u more specific, narrowing the set of potential choices for the subtypes of **A** for u. Thus, the search space is explored "intelligently," postponing the evaluation of branches of computation that would correspond for example to uninstantiated PROLOG goals (see for example van Hentenryck and Dincbas [1987], van Hentenryck [1989] on evaluation techniques in constraint logic programming).

Rewrite steps are applied nondeterministically everywhere in the structure until no further rule is applicable.[10]

The choice of which substructure to rewrite is only partly determined by the availability of information (using the lazy rewriting rule). When there are several substructures that could be rewritten, the computation rule is to choose one of the outermost ones, i.e., one closest to the root of the feature structure (innermost strategies are usually nonterminating). This outermost rewriting strategy is similar to hyper-resolution in logic programming. In comparison, PROLOG uses a leftmost computation rule.

For a given substructure, the choice of which rule to apply is done nondeterministically, and the search space is explored depth-first using a backtracking scheme. Although this strategy is not complete (a complete breadth-first search strategy could be used for debugging purposes), the use of the outermost rule has favorable termination

10 Conditions do not change this general scheme (they are evaluated using the same rewriting mechanism) and are omitted from the presentation here for the sake of simplicity. See for example Dershowitz and Plaisted (1988) and Klop (1990) for a survey on rewriting.

properties when compared to PROLOG's leftmost rule: there are problems where a TFS computation will terminate when the corresponding logic program implemented in PROLOG will not; for example, for left-recursive rules in naive PROLOG implementations of DCGs.

3. Inheritance and Constraint-Based Grammars

3.1 Related Approaches

In the constraint-based framework, a grammar is regarded as a set of constraints to be satisfied by a given linguistic object: parsing and generation differ only in the nature of the "input," and use the same constraint evaluation mechanism. The properties of a computational framework for implementing constraint-based grammars are:

- A unique general constraint solving mechanism is used: grammars define constraints on the set of acceptable linguistic structures.
- As a consequence, there is no formal distinction between "input" and "output." For example, the same kind of data structure could be used to encode both the string and the structural description, and, as for the HPSG sign (Pollard and Sag 1987), they could be embedded into a single data structure that represents the relation between the string and the associated linguistic structure.
- Specific mapping properties, based on constituency, linear precedence, or functional composition, are not part of the formalism itself, but can be encoded explicitly in the formalism.

An approach that uses a unique deductive mechanism for parsing and generation is described in Dymetman and Isabelle (1988). Within this approach, a lazy evaluation mechanism based on the specification of input/output arguments is implemented (in PROLOG), and the evaluation is completely data-driven: the same program parses or generates, depending only on the form of the input structure.

A constraint-based grammar does not need a context-free mechanism to build up constituent structures for parsing or generation: Dymetman, Isabelle, and Perrault (1990) describe a class of reversible grammars ("Lexical Grammars") based on a few composition rules that are very reminiscent of categorial grammars. Other kinds of approaches have been proposed, e.g., using a dependency structure and linear precedence relations (Reape 1990; see also Pollard and Sag [1987]). In Saint-Dizier (1991), linear precedence rules are defined as constraints in a language based on typed feature structures and SLD-resolution, which is used to experiment with GB theory.

In the following sections, we describe two examples of constraint-based grammars: an HPSG grammar for a fragment of English, and an LFG-style transfer grammar for a small machine translation problem between English and French.

3.2 Head-Driven Phrase Structure Grammar

In general, a grammar describes the relation between strings of words and linguistic structures. To implement a constraint-based grammar in TFS, we have to encode both kinds of structures using the same data structure provided by the TFS language: typed feature structures. A linguistic structure will be encoded using features and values. Conditions that constrain the set of valid linguistic structures have to be declared explicitly. A string of words will be encoded as a list of word forms, using the same kind

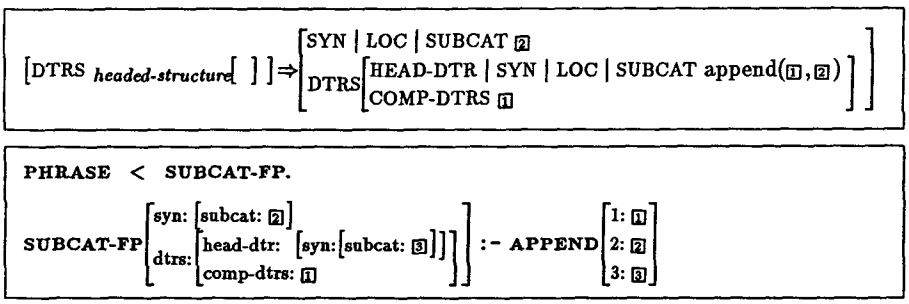

Figure 5
The HSPG subcategorization principle and its TFS encoding.

of definitions as in Figure 2.[11] HPSG is so far the only formal linguistic theory based on the notion of typed feature structures (Pollard 1990), and is thus a good candidate to illustrate the possibilities of the TFS formalism. The following presentation is based on Emele (1988) and Emele and Zajac (1990b).

The basic linguistic object in HPSG (Pollard and Sag 1987) is a complex linguistic structure, the "sign," with four levels of description: phonology, constituent structure, syntax, and semantics. In HPSG, there is no distinction between "input" and "output:" the relation between a string and a linguistic structure is encoded as a single feature structure representing the "sign."

HPSG "principles" are encoded using inheritance: a feature structure of type **PHRASE** inherits the constraints associated with types **SUBCAT-FP**, **HEAD-FP**, and **SEM-FP**. For example, the HPSG subcategorization principle is encoded in TFS using inheritance to model implication, and the TFS **APPEND** relation to encode the functional constraint on concatenation (Figure 5). In a similar way, **HEAD-FP** encodes the HPSG Head Feature Principle and **SEM-FP** encodes the Semantics Principle.

The type **SIGN** is divided into several subtypes corresponding to different mappings between a string and a linguistic structure. We first have the basic distinction between phrases and words. The definition of a phrase recursively relates subphrases and substrings, and defines the phrase as a composition of subphrases and the string as the concatenation of substrings. Since the formalism itself does not impose any constraints on how the relations between phrases and strings are defined, the grammar writer has to define them explicitly. In HPSG (Pollard and Sag 1987), the ordering of phrases is defined using linear precedence relations: the order in which the substrings associated with subphrases are concatenated to give the string associated with a phrase are guided by these linear precedence relations (Reape 1990).

In the example given below (Figure 6), we make simplifying assumptions: the *LOCAL* feature is not used and there are two possible orderings for complements. The type **IDP1** encodes Grammar Rule 1 (Pollard and Sag 1987, pp. 149–155), which says that a "saturated phrasal sign," i.e., a feature structure with $[syn : [subcat : \langle\rangle]]$, is the combination of an unsaturated phrasal head with one phrasal complement on the left. For example, for structures like $S \rightarrow NP\ VP$, S is the "saturated phrasal sign," NP is

11 We will use a more condensed notation for lists with angle brackets provided by the TFS language: a list
`CONS[first: Mary, rest: CONS[first: sings, rest: NIL]]` is written as `<Mary sings>`.

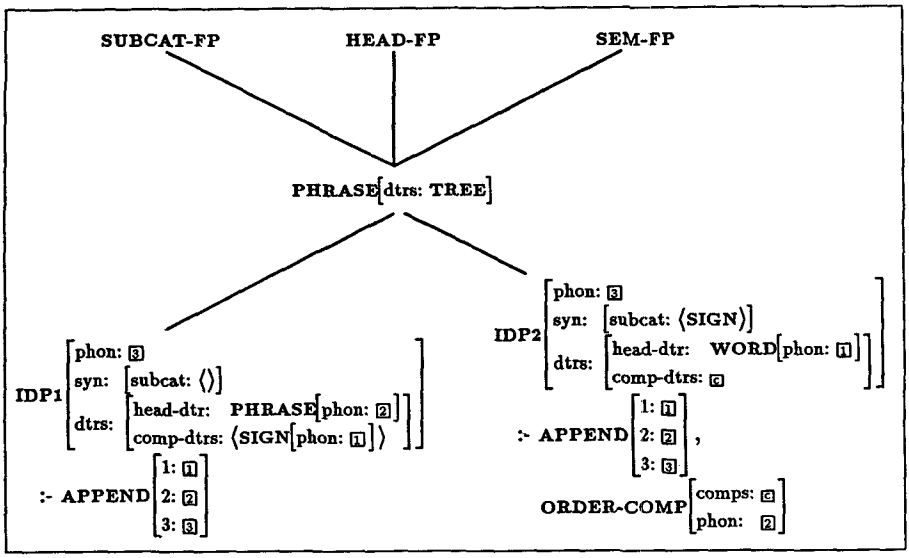

Figure 6
Part of the HPSG PHRASE hierarchy: PHRASE inherits from "principles" (given here without their definitions) and is subdivided into two subtypes corresponding to different complement orderings.

the left phrasal complement and *VP* is the unsaturated phrasal head. Furthermore, the string (the value of the *phon* feature) of the **IDP1** phrase is the concatenation of the string of the complement with the string of the head.

The type **IDP2** encodes Grammar Rule 2 and states that an "unsaturated phrasal sign," i.e., a feature structure with [*syn* : [*subcat* : ⟨**SIGN**⟩]], is the combination of a lexical head with any number of complements on the right (e.g., for $VP \rightarrow V\ XP^*$): the string associated with **IDP2** is the concatenation of the string of the head with the concatenation of the strings of the complements, where the relation **ORDER-COMP** defines in which order the complements strings are concatenated.

The difference between the parsing and the generation problem is then only in the form of the structure given to the interpreter for evaluation. A query for the parsing problem is an underspecified structure where only the string is given; conversely, a query for the generation problem is an underspecified structure where only the semantic form is given (Figure 7).

In both cases, the interpreter uses the same set of rewrite rules to fill in "missing information" according to the type definitions. The result in both cases is exactly the same: a fully specified structure containing the string, the full semantic form, and also all other syntactic information such as the constituent structure (Figure 8).

3.3 Bi-directional Transfer in Machine Translation

We have sketched above a very general framework for specifying mappings between a linguistic structure, encoded as a feature structure, and a string, also encoded as a feature structure. We apply a similar technique for specifying transfer rules for machine translation, which we prefer to call "contrastive rules" since there is no directionality involved (Zajac 1989; 1990a).

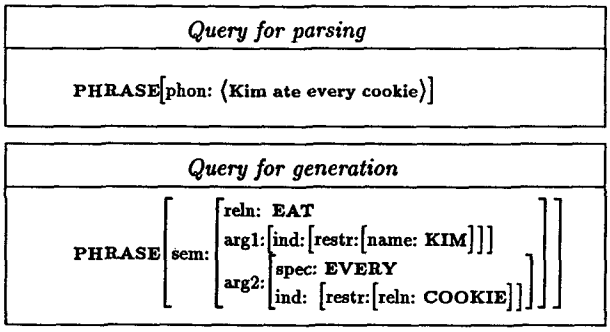

Figure 7
Queries for parsing and generation.

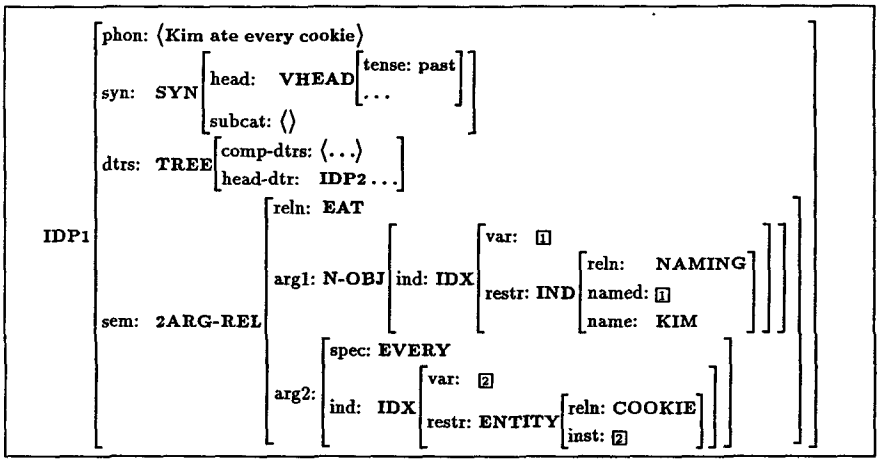

Figure 8
The common solution to the parsing and generation problems.

The idea is rather simple: assume we are working with linguistic structures similar to LFG's functional structures for English and French as proposed in Kaplan et al. (1989). We define a translation relation as a type **TAU-LEX** with two features, *eng* for the English structure and *fr* for the French structure. This "bilingual sign" is defined on the lexical structure: each subtype of **TAU-LEX** defines a lexical correspondence between a partial English structure and a partial French structure for a given lexical equivalence. Such a lexical contrastive definition for a verb also has to pair the arguments recursively. This is expressed in the condition part of the definition (Figure 9) by a recursive condition **TAU-LEX** on the arguments. The translation of syntactic and semantic features, like tense or determination, is also specified in the condition part, and these contrastive definitions can and should be defined separately from the lexical definitions as different subnetworks.

Inheritance is used here to capture generalizations over *relations*: all the information in the example of Figure 9 can be unpacked and redefined as follows. We have

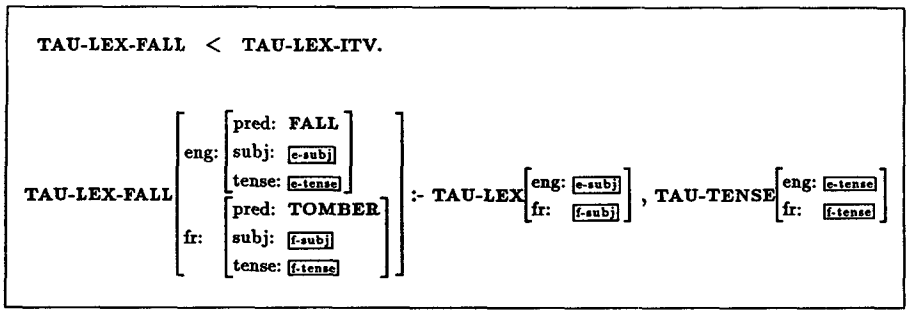

Figure 9
A transfer rule.

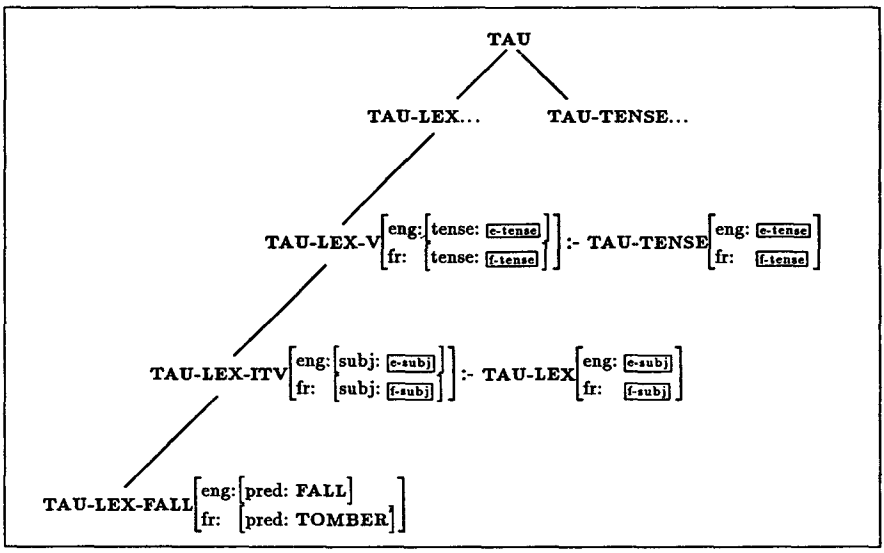

Figure 10
Part of an inheritance network of transfer relations for verbs.

three levels of generalization for the translation of verbs: TAU-LEX-V is subdivided into several relations for translating, for example, transitive verbs (TAU-LEX-TV), intransitive verbs (TAU-LEX-ITV), etc. The condition that specifies the translation of tenses is defined for the whole class of verbs TAU-LEX-V. The condition that specifies the translation relation between subjects is defined for intransitive verbs (TAU-LEX-ITV), but cannot be specified for all verbs since it does not apply to, e.g., impersonal verbs. This leaves only the translation of predicates to be defined at the leaves of the hierarchy (Figure 10). Thus, at each level of generalization, we specify the minimal amount of information needed for translation. The same kind of organization can be used for nouns, where syntactic features such as determination are associated with the higher types, and where the minimal types define the equivalences between lexical forms of nouns themselves.

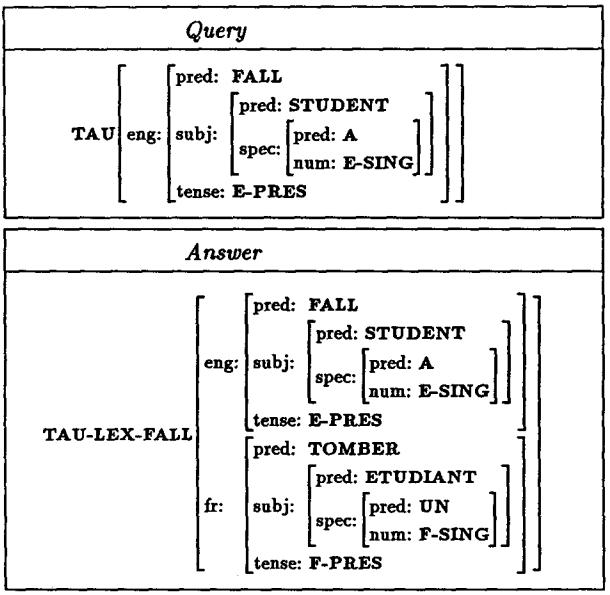

Figure 11
Query and answer for the translation of "A student falls."

The transfer problem for each direction is then stated in the same way as for parsing or generation: the input structure is an underspecified "bilingual sign" where only the structure for the source language is given. Using the contrastive grammar, the interpreter fills in missing information and builds a bilingual sign[12] (Figure 11).

It is not necessary to specify in the contrastive definitions all monolingual constraints that have to be satisfied by the English structure and by the French structure. We can assume that we have monolingual grammars that define the appropriate mappings between the set of English sentences and the set of associated English structures, and similarily for French. Using these monolingual constraints in addition to the contrastive grammar, the TFS interpreter would build the fully specified monolingual structures, implementing a constraint-based translation system.

3.4 Termination Problems

For parsing and generation, since no constraints are imposed on the kind of mapping between the string and the semantic form, termination has to be proved for each class of grammar and for the particular mechanism used for either parsing or generation with this grammar. If we restrict ourselves to classes of grammars for which terminating evaluation algorithms are known, we can implement those directly in TFS. However, the TFS evaluation strategy allows more naive implementations of grammars, and the outermost rewriting of "sub-goals" terminates on a strictly larger class of programs than for corresponding logic programs implemented in PROLOG. Furthermore, the grammar writer does not need, and actually should not, be aware of the control that follows the shape of the input rather than a fixed strategy, thanks to the lazy evaluation mechanism.

12 See also Reape (1990) for another approach to MT using feature structures and based on Whitelock (1990).

For HPSG-style grammars, completeness and coherence as defined for LFG, and extended to the general case by Wedekind (1988), are implemented in HPSG using the "subcategorization feature principle" (Johnson 1987): for the TFS implementation of HPSG, termination is guaranteed, at least for the simplified version containing only head-complement structures, described in Section 3.2. Termination conditions for parsing are well understood in the framework of context-free grammars. For generation using feature structures, one of the problems is that the input could be "extended" during processing, i.e., arbitrary feature structures could be introduced in the semantic part of the input by unification with the semantic part of a rule. However, if the semantic part of the input is fully specified according to a set of type definitions describing the set of well-formed semantic structures (and this condition is easy to check), this cannot arise in a type-based system since it is not possible to add arbitrary features to a typed feature structure.

A more general approach is described in Dymetman, Isabelle, and Perrault (1990), who define sufficient termination properties for parsing and generation for the class of "Lexical Grammars." These termination properties are conditions on the existence of "conservative guides" for parsing and generation and seem generalizable to other classes of grammars as well, and are also applicable to TFS implementations. Since Lexical Grammars are implemented in PROLOG, left-recursion must be eliminated for parsing and for generation, but this does not apply to TFS implementations. The idea of conservative guides is relatively simple and says that for parsing, each rule must consume a nonempty part of the string, and for generation, each rule must consume a nonempty part of the semantic form. These conditions seem to be equivalent as to require the existence of a well-founded relation on strings (for parsing) and of a well-founded relation on semantic forms (for generation). The existence of such well-founded relations is actually a necessary condition for proving the termination of parsing and generation (see Deville [1990] for a more general discussion on well-founded relations in the context of logic programming).

Termination for reversible transfer grammars is discussed in van Noord (1990). One of the problems mentioned there is the extension of the "input," as in generation, and the answer is similar (see above). However, properties similar to the "conservative guides" of Dymetman, Isabelle, and Perrault (1990) have to hold in order to ensure termination.

4. Conclusion

The TFS system has been developed to provide a computational environment for the design and the implementation of formal models of natural language. The TFS formalism is designed as a *specification language* that can be used to design and implement formal linguistic models. It is not a programming language: it does not offer means of defining control information that would make execution more efficient (but less general), as it would be needed if it would be envisaged to use the system in an application-oriented environment (e.g., as a parser in a natural language interface to a database system). From formal linguistic models developed in TFS, it could be envisaged to develop programs, i.e., parsers or generators, that would efficiently implement the declarative knowledge contained in the formal specifications.[13]

13 See for example in Biggerstaff and Perlis (1989) the papers on the development of programs from specifications, a very important issue in software engineering. See also Aït-Kaci and Meyer (1990) for a programming language based on typed feature structures.

The TFS system is implemented using rewriting techniques in a constraint-based architecture adapted to feature structures:

- The language is a logical language directly based on typed feature structures, and supports an object-oriented style based on multiple inheritance.
- Grammars are expressed as inheritance networks of typed feature structures. They define constraints on the set of acceptable linguistic structures. As a consequence, there is no formal distinction between "input" and "output."
- A unique general constraint solving mechanism is used. Specific mapping properties, based on constituency, linear precedence or functional composition, are not part of the formalism itself, but can be encoded explicitly using the formalism.

Although the current implementation is very much at the level of an experimental prototype, and is still evolving, it has validated the basic concepts of the language and of the implementation, and has been used to test different linguistic models and formalisms such as LFG, DCG, HPSG, and SFG on small examples. From these various experimentations, we have defined extensions and improvements, both on the language and on the implementation, that are needed for scaling up the system.

On the language side, more expressivity is needed. For example, disjunctions over feature structures, various kinds of negation (Aït-Kaci 1986), and sets of feature structures (Pollard and Moshier 1990) are necessary to formalize, e.g., nontrivial semantic structures. Some types together with a specific syntax and associated operations could be conveniently added to the system as libraries of built-in types, e.g., characters, strings, and trees.

On the implementation side, the use of implementation techniques adapted from PROLOG implementations, constraint satisfaction languages, and object-oriented languages can be beneficial to the implementation of typed feature structure–based systems and have to be more thoroughly explored.[14] One of the major efficiency issues in the current implementation is the lack of an efficient indexing scheme for typed feature structures. For example, since the dictionaries are accessed using unification only, each entry is tried one after the other, leading to an extremely inefficient behavior with large dictionaries. Thus, the use of a general indexing scheme based on a combination of methods used in PROLOG implementations and in object-oriented database systems is necessary and will be implemented in a future version of the system.

Acknowledgments

The design and the implementation of the TFS system have been carried out in cooperation with Martin Emele. I would like to thank Stefan Momma and Ulrich Heid; their numerous comments and advices helped to make this article more readable. I would also like to thank anonymous referees for their many detailed and helpful comments. All remaining errors are of course the sole responsibility of the author.

14 For example, the use in the current implementation of several techniques adapted from PROLOG implementations such as structure sharing, chronological dereferencing (Emele 1991) and last call optimization, have improved efficiency by several orders of magnitude over previous "naive" implementations.

References

Aït-Kaci, Hassan (1984). "A lattice theoretic approach to computation based on a calculus of partially ordered type structures." Doctoral dissertation, University of Pennsylvania, Philadelphia, PA.

Aït-Kaci, Hassan (1986). "An algebraic semantics approach to the effective resolution of type equations." *Theoretical Computer Science*, 45, 293–351.

Aït-Kaci, Hassan, and Meyer, Richard (1990). Wild_LIFE, a user manual. DEC-PRL Technical Note PRL-TN-1, Rueil-Malmaison, France.

Aït-Kaci, Hassan, and Podelski, Andreas (1991). Towards a meaning of LIFE. DEC-PRL Research Report PRL-RR-11, Rueil-Malmaison, France.

Bateman, John A., and Momma, Stefan (1991). The nondirectional representation of Systemic Functional Grammars and Semantics as Typed Feature Structures. IMS Technical Report, University of Stuttgart, Germany.

Biggerstaff, Ted J., and Perlis, Alan J. (eds). (1989). *Software Reusability*. ACM Press–Addison-Wesley.

Birkoff, Garrett (1940). *Lattice Theory*. American Mathematical Society, third edition, 1973.

Borgida, Alexander; Brachman, Ronald J.; McGuinness, Deborah L.; and Resnick, Lori Alperin (1989). "CLASSIC: a structural data model for objects." In *Proceedings, 1989 ACM SIGMOD International Conference on Management of Data*, Portland, Oregon.

Bouma, Gosse (1990). "Non-monotonic inheritance and unification." In *Proceedings, Workshop on Inheritance in Natural Language Processing*, Institute for Language Technology and AI, Tilburg University, The Netherlands.

Bourbeau, L.; Carcagno, D.; Goldberg, E.; Kittredge, R.; and Polguère, A. (1990). "Bilingual generation of weather forecasts in an operation environment." In *Proceedings, 13th International Conference on Computational Linguistics* (COLING'90). Helsinki.

Brachman, Ronald J., and Schmolze, James G. (1985). "An overview of the KL-ONE knowledge representation language." *Cognitive Science* 9, 171–216.

Carpenter, Bob (1990). "Typed feature structures: inheritance, (in)equality and extensionality." In *Proceedings, Workshop on Inheritance in Natural Language Processing*, Institute for Language Technology and AI, Tilburg University, The Netherlands.

Daelemans, Walter (1990). "Inheritance and object-oriented natural language processing." In *Proceedings, Workshop on Inheritance in Natural Language Processing*, Institute for Language Technology and AI, Tilburg University, The Netherlands.

Dershowitz, N., and Plaisted, D. A. (1988). "Equational programming." In *Machine Intelligence 11*, edited by Hayes, Michie, and Richards. Oxford: Clarendon Press.

De Smedt, Koenraad, and de Graaf, Josje (1990). "Structured inheritance in frame-based representation of linguistic categories." In *Proceedings, Workshop on Inheritance in Natural Language Processing*. Institute for Language Technology and AI, Tilburg University, The Netherlands.

Deville, Yves (1990). *Logic programming. Systematic Program Development*. Reading, MA: Addison-Wesley.

Dymetman, Marc, and Isabelle, Pierre (1988). "Reversible logic grammars for machine translation." In *Proceedings, 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language*. Pittsburgh, PA.

Dymetman, Marc; Isabelle, Pierre; and Perrault, François (1990). "A symmetrical approach to parsing and generation." In *Proceedings, 13th International Conference on Computational Linguistics* (COLING'90). Helsinki.

Emele, Martin (1988). "A typed feature structure unification-based approach to generation." In *Proceedings, WGNLC of the IECE*. Oiso University, Japan.

Emele, Martin (1991). "Unification with lazy non-redundant copying." In *Proceedings, 29th Annual Meeting of the ACL*. Berkeley, CA.

Emele, Martin, and Zajac, Rémi (1989). "RETIF: A rewriting system for typed feature structures." ATR Technical report TR-I-0071, ATR, Kyoto.

Emele, Martin, and Zajac, Rémi (1989). "Multiple inheritance in RETIF." ATR Technical report TR-I-0114, ATR, Kyoto.

Emele, Martin, and Zajac, Rémi (1990a). "A fixed-point semantics for feature type systems." In *Proceedings, 2nd Workshop on Conditional and Typed Rewriting Systems (CTRS'90)*. Montréal, Québec.

Emele, Martin, and Zajac, Rémi (1990b). "Typed unification grammars." In *Proceedings, 13th International Conference on Computational Linguistics* (COLING'90). Helsinki.

Emele, Martin; Heid, Ulrich; Momma, Stefan; and Zajac, Rémi (1990). "Organizing linguistic knowledge for multilingual generation." In *Proceedings,*

13th International Conference on Computational Linguistics (COLING'90). Helsinki.

Etherington, David W.; Forbus, Kenneth D.; Ginsberg, Matthew L.; Israel, David; and Lifschitz, Vladimir (1989). "Critical issues in nonmonotonic reasoning." In *Proceedings, 1st International Conference on Principles of Knowledge Representation and Reasoning*. Toronto, Ontario.

Evans, Roger, and Gazdar, Gerald (1989). "Inference in DATR." In *Proceedings, 4th European ACL Conference*. Manchester, U.K.

Franz, Alex (1990). "A parser for HPSG." CMU report CMU-LCL-90-3, Laboratory for Computational Linguistics, Carnegie Mellon University.

Fraser, Norman M., and Hudson, Richard A. (1990). "Word grammar: An inheritance-based theory of language." In *Proceedings, Workshop on Inheritance in Natural Language Processing*. Institute for Language Technology and AI, Tilburg University, The Netherlands.

Halliday, M. A. K. (1985). *Introduction to Functional Grammar*. London: Edward Arnold.

Huet, Gérard (1976). *Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω*. Doctoral dissertation, Université de Paris VII.

Johnson, Mark (1987). "Grammatical relations in attribute-value grammars." In *Proceedings, West Coast Conference on Formal Linguistics*, Vol. 6. Stanford, CA.

Kaplan, Ronald M.; Netter, Klaus; Wedekind, Jürgen; and Zaenen, Annie (1989). "Translation by structural correspondences." In *Proceedings, 4th European ACL Conference*. Manchester, U.K.

Kay, Martin (1984). "Functional unification grammar: A formalism for machine translation." In *Proceedings, 10th International Conference on Computational Linguistics (COLING-84)*. Stanford, CA.

Klop, Jan Willem (1990). "Term rewriting systems." To appear in *Handbook of Logic in Computer Science*, Volume 1, edited by S. Abramsky, D. Gabbay and T. Maibaum. Oxford University Press.

MacGregor, Robert M. (1988). "A deductive pattern matcher." In *Proceedings, 7th National Conference on Artificial Intelligence (AAAI'88)*. St. Paul, MN, 403–408.

MacGregor, Robert M. (1990). "LOOM user manual." USC/ISI Technical Report, Marina del Rey, CA.

Mann, William C., and Matthiessen, Christian I. M. I. (1985). "Demonstration of the Nigel text generation computer program." In *Systemic Perspectives on Discourse, Volume 1*, edited by James D. Benson and William S. Greaves. Norwood, NJ: Ablex.

Nirenburg, Sergei, Carbonele, Jaime, Tomita, Masaru, and Goodman, Kenneth (1992). *Machine Translation. A Knowledge-Based Approach*. San Mateo, CA: Morgan Kaufmann.

Pollard, Carl J. (In press). "Sorts in unification-based grammar and what they mean." In *Unification in Natural Language Analysis*, edited by M. Pinkal and B. Gregor. Cambridge, MA: The MIT Press.

Pollard, Carl J., and Moshier, Drew (1989). "Unifying partial descriptions of sets." In *Information, Language and Cognition*, edited by P. Hanson, Vancouver Studies in Cognitive Science 1. Vancouver: University of British Columbia Press.

Pollard, Carl J., and Sag, Ivan A. (1987). *Information-Based Syntax and Semantics. Volume 1: Syntax*. CSLI Lecture Notes 13, Chicago University Press.

Pollard, Carl J., and Sag, Ivan A. (Unpublished research). *Information-Based Syntax and Semantics. Volume 2: Agreement, Binding and Control*.

Reape, Mike (1990). "Parsing semi-free word order and bounded discontinuous constituency and "shake 'n' bake" machine translation (or 'generation as parsing')." *International Workshop on Constraint Based Formalisms for Natural Language Generation*, Bad Teinach, Germany.

Rounds, Williams C., and Kasper, Robert T. (1986). "A Complete Logical Calculus for Record Structures Representing Linguistic Information." *IEEE Symposium on Logic in Computer Science*.

Saint-Dizier, Patrick (1991). "Processing language with logical types and actives constraints." In *Proceedings, 5th Conference of the European Chapter of the ACL*. Berlin, Germany.

Shieber, Stuart (1986). *An Introduction to Unification-based Grammar Formalisms*. CSLI Lectures Notes 4, Chicago University Press.

Smolka, Gert (1988). "A feature logic with subsorts." LILOG Report 33, IBM Deutschland GmbH, Stuttgart.

Smolka, Gert (1989). "Feature constraint logics for unification grammars." IWBS Report 93, IBM Deutschland GmbH, Stuttgart.

Smolka, Gert, and Aït-Kaci, Hassan (1988). "Inheritance hierarchies: Semantics and unification." *J. Symbolic Computation*, 7, 343–370.

Stoy, Joseph E. (1977). *Denotational Semantics:*

The Scott–Strachey Approach to Programming Language Theory. Cambridge, MA: The MIT Press.

van Hentenryck, Pascal (1989). *Constraint Satisfaction in Logic Programming,* Cambridge, MA: The MIT Press.

van Hentenryck, Pascal, and Dincbas, Mehmet (1987). "Forward checking in logic programming." In *Proceedings, 4th International Conference on Logic Programming.* Melbourne, Australia.

van Noord, Gertjan (1990). "Reversible unification-based machine translation." In *Proceedings, 13th International Conference on Computational Linguistics* (COLING'90). Helsinki.

Wedekind, Jürgen (1988). "Generation as structure driven generation." In *Proceedings, 12th International Conference on Computational Linguistics (COLING'88).* Budapest.

Whitelock, Pete (1990). "Shake-and-bake translation." Ms. Sharp Laboratories of Europe, Oxford.

Yen, John; Neches, Robert; and MacGregor, Robert (1988). "Classification-based programming: A deep integration of frames and rules." Technical Report ISI/RR-88-213, USC/Information Science Institute.

Zajac, Rémi (1989). "A transfer model using a typed feature structure rewriting system with inheritance." In *Proceedings, 27th Annual Meeting of the ACL.* Vancouver, B.C.

Zajac, Rémi (1990a). "A relational approach to translation." In *Proceedings, 3rd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language.* Austin, Texas.

Zajac, Rémi (1990). "Semantics of typed feature structures." *International Workshop on Constraint Based Formalisms for Natural Language Generation.* Bad Teinach, Germany.

Appendix: Syntax of the TFS Language

In the TFS language, there are two kinds of comments. In-line comments begin with a semicolon and end with the end of line. These comments can appear anywhere where a white space character is allowed in the syntax. They are skipped during reading. Syntactic comments begin and end with the % character. These comments can appear only where specified in the BNF syntax specification. They are attached to the structure produced by the reader and can be displayed if the appropriate printer option is set. Identifiers are case-sensitive. Macros are expanded statically by the compiler. The operator & is interpreted as the meet (unification).

The following extensions of the BNF notation are used:

- [*X*] denotes the optional element X (zero or one occurrence).
- X^* denotes the free iteration of element X (zero, one or more occurrences).
- X^+ denotes the iteration of element X (one or more occurrences).
- Symbols in `typewriter` font denote symbols of the TFS syntax.

```
<entity> ::= <query> | <definition>
<query> ::= ? <expression> .
<definition> ::= <po-definition> | <macro-definition> | <type-definition>
<po-definition> ::= <type-symbol> < <type-symbol> .
<macro-definition> ::= <identifier> := <expression> .
<type-definition> ::= <type-symbol> <expression> :- <conditions> .
<conditions> ::= <expression> [ , <conditions> ]
<expression> ::= <feature-structure> [ & <expression> ]
<feature-structure> ::= <comments> <tagged-feature-structure>
<tagged-feature-structure> ::= <tag> [ = <typed-feature-structure> ] |
                                        <typed-feature-structure>
<typed-feature-structure> ::= <type-symbol> [ <attribute-value-matrix> ] |
                              <attribute-value-matrix> |
                              <list>
<attribute-value-matrix> ::= [ [ <attribute-value-pairs> ] ]
<attribute-value-pairs> ::= <attribute-value-pair> [ , <attribute-value-pairs> ]
<attribute-value-pair> ::= <attribute> : <expression>
<list> ::= < <expression>* [ . <expression> ] >
<tag> ::= #<identifier>
<type-symbol> ::= <identifier>
<attribute> ::= <identifier>
```

Example 1: the textual definitions for Figure 2.

```
NIL < LIST.

CONS < LIST.
CONS[first: T, rest: LIST].

APPEND[1: LIST, 2: LIST, 3: LIST].
```

```
APPEND0 < APPEND.
APPEND0[1: NIL, 2: #1=LIST, 3: #1].

APPEND1 < APPEND.
APPEND1[1: <#x . #11>, 2: #12, 3: <#x . #13>]
       :- APPEND[1: #11, 2: #12, 3: #13].
```

Example 2: the textual definitions for Figure 10.

```
TAU-LEX < TAU.

TAU-LEX-V < TAU-LEX.
TAU-LEX-V[eng: [tense: #e-tense],
          fr:  [tense: #f-tense]]
         :- TAU-TENSE[eng: #e-tense, fr: #f-tense].

TAU-LEX-ITV < TAU-LEX-V.
TAU-LEX-V[eng: [subj: #e-subj],
          fr:  [subj: #f-subj]]
         :- TAU-LEX[eng: #e-subj, fr: #f-subj].

TAU-LEX-FALL < TAU-LEX-ITV.
TAU-LEX-FALL[eng: [pred: FALL],
             fr:  [pred: TOMBER]]
```