TR-I-0071

RETIF

A Rewriting System for Typed Feature Structures

Martin EMELE and Rémi ZAJAC

1989 . 03

Abstract

This report describes a prototype of a rewriting system for typed feature structures which was implemented primarily in order to develop a transfer and a generation model for Machine Translation of dialogues. The formalism is based on the semantics of typed feature structures as described in [Ait-Kaci 84].

In section 1, we present the extension of unification of feature structures with a type system; in section2, the syntax of the formalism is presented, together with the type system derived from the set of definitions; the interpreter is described in the next section, and some necessary developments are outlined in the last section. A short example of transfer and generation is given in the appendix.

ATR Interpreting Telephony Research Laboratories
ATR 自動翻訳電話研究所

RETIF: A Rewriting System for Typed Feature Structures

Martin Emele and Rémi Zajac

ATR Interpreting Telephony Research Laboratories
Sanpeidani, Inuidani, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

Abstract We describe a prototype of a rewriting system for typed feature structures which was implemented primarily in order to develop a transfer and a generation model for Machine Translation of dialogues. The formalism is based on the semantics of typed feature structures as described in [Aït-Kaci 84]. Compared to the one developed by Aït-Kaci, the originality of this first prototype lies mainly in the search for efficiency: (1) reducing the cost of the unification (instead of undoing coreference merging, we use a non-destructive unification algorithm) and (2) avoiding unecessary expansions of disjunctions.

In section 1, we present the extension of the unification of feature structures with a type system. In section 2, the syntax of the formalism is then presented together with the type system derived from a set of definitions. The intepreter is described in the next section. Some necessary developments are outlined in the last section. A short example of transfer and generation is given in the appendix.

Key-Words rewriting system, unification-based formalisms, feature structures, type inheritance, transfer, generation.

TABLE

Introduction

This report describes a calculus of typed feature structures suitable for natural language processing. The implementation of such a system has been motivated by the need for a formalism which can describe (and compute!) mappings between different linguistic representations. More precisely, in our case, we wanted to be able to write grammars (and dictionaries) for transfer and generation modules of a machine translation system prototype.

The goal of this report is to give a rather informal account of the calculus, and is adressed to readers who are interested in unification-based formalism and their applications, rather than just in the theoretical foundations. However, this calculus has a precise formal foundation, and we ask the reader to refer to the original work of [Aït-Kaci 84].

The initial constraints on the choice of a formalism can be characterized as follows.

1) the input for transfer and generation should be a feature structure;
2) the output for transfer should be a feature structure, and for generation a string;
3) it should be able to incorporate current theories in the framework of unification-based grammars;
4) it should enhance modularity in grammar development
5) if possible, it should be inherently reversible

Some formalisms such as FUG [Kay 84] or CIL [Mukai 88] used in computational linguistics meet some of these requirements. However, none of these formalisms facilitate a very important property for formalisms which are intended to be used for reasonable size grammars: modularity. For most of them, this failure also includes the consequence that the computational cost is very high at each step of computation, all rules are possible candidates for application. In fact, some external control mechanism is often added, such as context-free rules for parsing.

If we look at recent developments in programing languages, we find two interesting paradigms: the so-called object-oriented paradigm, and the rewriting system paradigm used for theorem-proving or for implementing specification languages. A recent work by Aït-Kaci [Aït-Kaci 84] presents a unified view of semantic networks, inheritance hierarchies, first-order term rewriting, and partially ordered types in programming. We found that the formalism proposed by Aït-Kaci and which incorporates this synthetic approach met all our requirements, and was perfectly suited for our problem. In particular, the partially ordered type system with inheritance meets the modularity requirement.

The data structures used by the formalism are typed feature structures. The feature structures in this formalism are general graphs. The formalism allows defining partially ordered types, and disjunctions of type definition. The type definitions specify a set of constraints on well-formed typed feature

structures. Given as input a typed feature structure, the interpreter uses a rewriting mechanism to apply this set of constraints on the input, deriving the set of structures compatible with the input and with the set of type definitions. Reversibility can be achieved if some care is taken in the writing of definitions. Depending on the form of the definitions, the system can also be used to generate a set of structures described by some grammar. In that respect, it can be compared favourably with PROLOG (see appendix 2).

1. A Type System for Feature Structures

1.1 A Lattice of Types

The type system is defined on a set of type symbols P which always contain type symbols $\top$ («top») and $\bot$ («bottom»), and which are partially ordered. The partially ordered set of type symbols is called Σ, the signature of the type system.

Take for example P = {$\top$, $\bot$, PERSON, STUDENT, EMPLOYEE, STAFF, FACULTY, PETER, MARY, WORKSTUDY, PAUL, JEAN, BILL, JOAN, SIMON, ROGER}. We define a hierarchy between types, as graphically depicted in Figure 1.

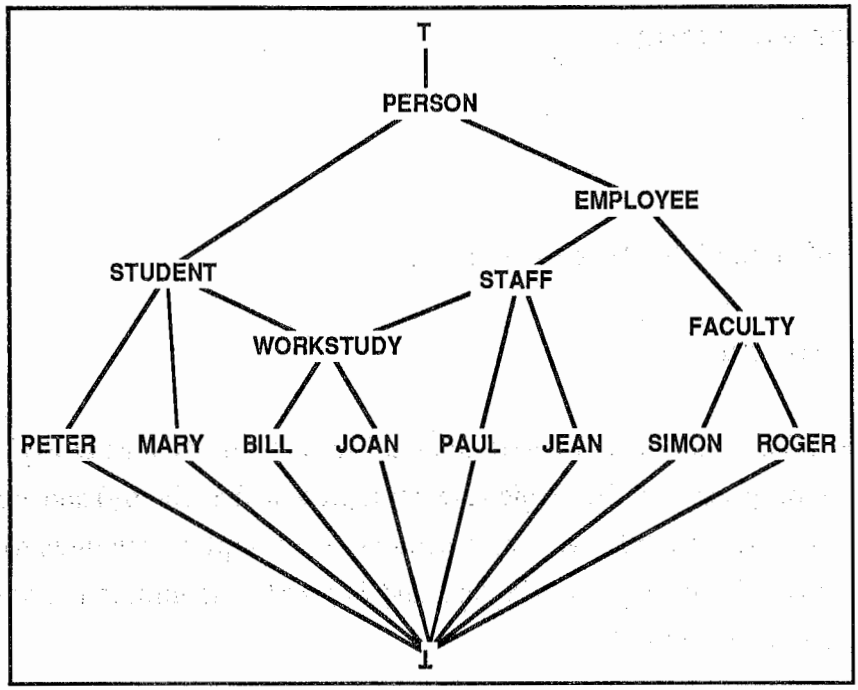

Figure 1: The partial ordering Σ on a set of type symbols.

This hierarchy defines implicitly a partial order $\leq$ on the set of type symbols P:

- STUDENT and JOAN are comparable : JOAN $\leq$ WORKSTUDY $\leq$ STUDENT,
- STUDENT and EMPLOYEE are not comparable.

This partial order defines (in this case) a lattice structure on P: it is both a meet semi-lattice and a join semi-lattice.

Meet semi-lattice:

For any pair of symbols (x, y) of P, there exists a unique symbol z in P such that $x \geq z$ and $y \geq z$, and for all u in P, if $x \geq u$ and $y \geq u$ then $z \geq u$: z is called the Greatest Lower Bound of x and y, and we write $z = x \wedge y$. The operation $\wedge$ is called the *meet* operation.

Join semi-lattice:

For any pair of symbols (x, y) of P, there exists a unique symbol z in P such that $x \leq z$ and $y \leq z$, and for all u in P, if $x \leq u$ and $y \leq u$ then $z \leq u$: z is called the Least Upper Bound of x and y, and we write $z = x \vee y$. The operation $\vee$ is called the *join* operation.

For example:

$\text{STUDENT} \wedge \text{JOAN} = \text{JOAN}$

$\text{STUDENT} \wedge \text{EMPLOYEE} = \text{WORKSTUDY}$

$\text{STUDENT} \wedge \text{SIMON} = \bot$

1.2 Unification of Typed Feature Structures

A. Typed Feature Structures

Feature structures used in unification-based grammar formalisms such as D-PATR [Shieber 86] can have only two kind of types: complex (which are not represented explicitly) and atomic (which are represented by a symbol). A straightforward extension allows complex structures to explicitly bear type symbols, as proposed for example in [Pollard and Sag 86]. The unification algorithm is then extended using a calculus on type symbols.

In the following examples, type symbols are in upper-case letters, and feature symbols in lower-case. Symbols beginning with # are tag symbols, which represent co-reference («sharing») in the structure.

```
#x=STUDENT[advisor:FACULTY[secretary:#y=STAFF,
                           assistant:#x],
           roommate:EMPLOYEE[representative:#y]]

#u=EMPLOYEE[advisor:SIMON[secretary:EMPLOYEE,
                          assistant:#v=PERSON],
           roommate:#w=STUDENT[representative:#w],
           helper:BILL[spouse:#u]]
```

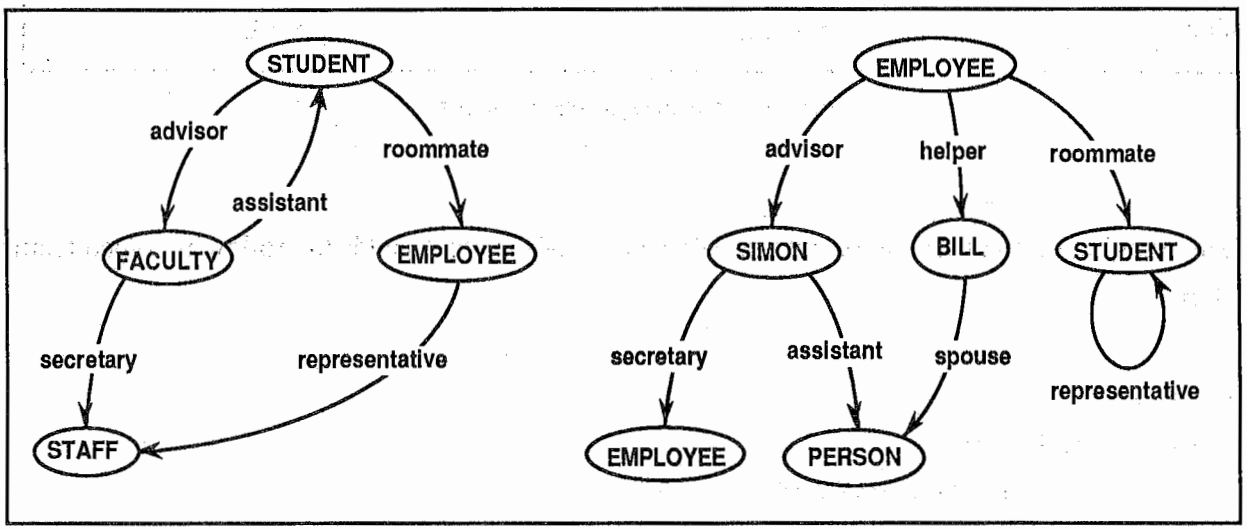

Figure 2: Graphical representation of two typed feature structures with type symbols inside nodes. Note that these are general directed graphs, not only DAGs.

B . The Unification Algorithm

The standard unification algorithm for feature structures is modified to use type information (see [Aït-Kaci 84 pp 102-111]). The unification on ordinary feature structures is already defined as a meet operation on the set of feature structures partially ordered by the subsumption ordering (see for example [Shieber 86]). This is extended in a straightforward way also using the meet operation on the lattice of types to compute the new type associated with the result of the unification of two feature structures. For atomic types, we get exactly the same interpretation as for ordinary feature structures, and we can simplify the internal representation of the lattice by not explicitly including atomic types. Instead, we consider every type symbol which is not explictly defined in the lattice to be atomic.

The unification of the two feature structures above can be done essentially like ordinary unification. The only extension we need is to compute the meet of two type symbols which are associated with the feature structures.

merged paths	*associated types*	*new type symbol*
ε advisor.assistant helper.spouse	STUDENT ∧ EMPLOYEE ∧ PERSON	WORKSTUDY
advisor	FACULTY ∧ SIMON	SIMON
advisor.secretary roommate roommate.representative	STAFF ∧ EMPLOYEE ∧ STUDENT	WORKSTUDY
helper	T ∧ BILL	BILL

Figure 3: the meet for merged paths.

The merging of common paths and the computation of the meet yields the following typed feature structure:

```
#x=WORKSTUDY
   [advisor:SIMON
            [secretary:#y=WORKSTUDY
                         [representative:#y],
                          assistant:#x],
    roommate:#y,
    helper:BILL[spouse:#x]]
```

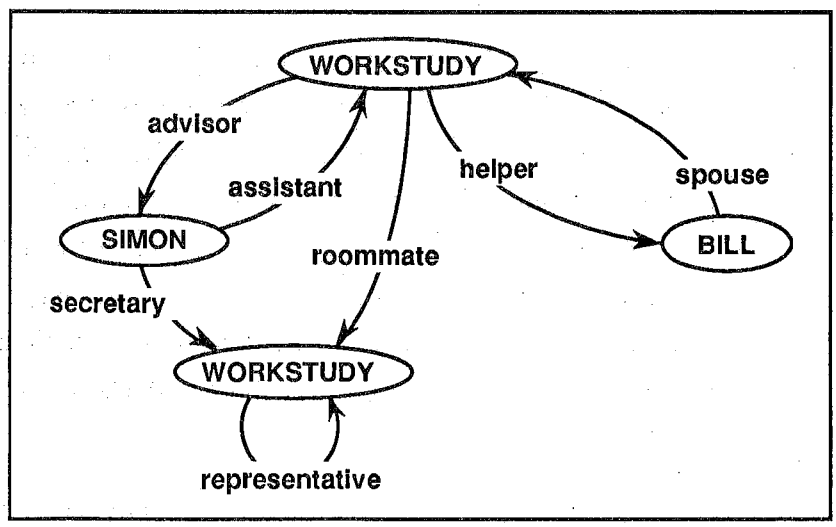

Figure 4: Graphical representation of the typed feature structure.

This kind of unification algorithm could be used in a unification-based parser, such as the D-PATR system, but we use it as the basic operation of a type rewriting system that fully exploit the possibilities of the type system, namely to inherit type definitions according to the subsumption ordering introduced by the KB definitions. In addition, disjunctions of types can be used to express indeterminacy in the feature descriptions.

2. The RETIF Formalism and the Type System

2.1 RETIF Definitions

A type is defined as a disjunction of typed feature structures (a disjunction can be reduced to one element):

<type definition> ::= <type-symbol> '=' <typed feature structure> { '|' <typed feature structure> } .

There are two constraints imposed on definitions:

1) Definitions must not lead to a cycle in the type system, otherwise, the type system cannot be ordered.
2) Tags are local to one disjunct.

During the evaluation process, any typed feature structure of type A must verify the constraints stated in the definition of A: for a disjunction, it must verify one of the constraints stated as disjuncts; for one disjunct (a typed feature structure), it must be compatible with the feature structure, and it inherits the constraints of the type of this feature structure (see section 3).

The three cubes example

```
:KB threecubes

COLOR = GREEN | NON-GREEN.
NON-GREEN = BLUE | OTHERS.

THREECUBES = _ THREECUBES is-a STACK which-has top, middle and bottom slots.
             STACK[top:GREEN,
                   middle:COLOR,
                   bottom:BLUE].
```

```
ON = _ ON is either a THREECUBES where top = above and middle = below
     _ or a THREECUBES with middle = above and bottom = below.
      THREECUBES[top:#x,
                 middle:#y,
                 above:#x,
                 below:#y]
    | THREECUBES[middle:#z,
                 bottom:#t,
                 above:#z,
                 below:#t].

QUERY = _ Is there a green cube on top of a non green cube?
        ON[above:GREEN, below:NON-GREEN].
```

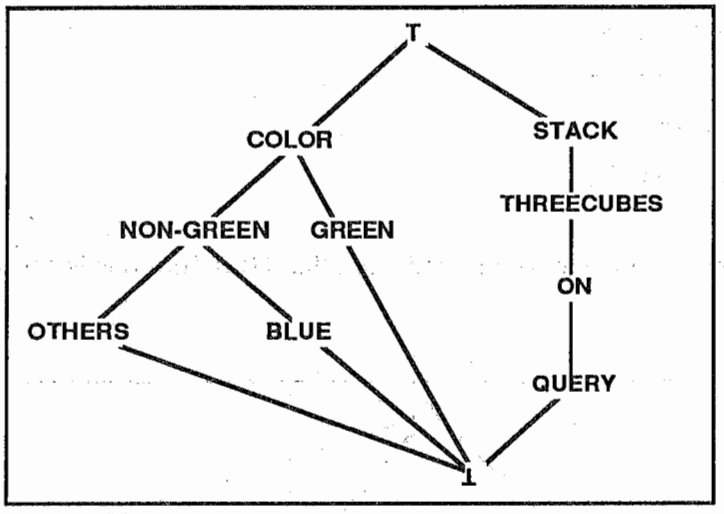

Figure 5: The partial ordering on type symbols extracted from the KB definitions

2.2 Embedding the Partial Ordering in a Meet Semi-Lattice

A. The embedding

The Normal Form of the Definitions

In order to be interpreted, the definitions of the KB are put in Normal Form where a disjunction of typed feature structures is replaced with a disjunction of new symbols, one for each disjunct. For example, the definition of ON above is replaced with:

```
ON = ON1 | ON2.
ON1 = THREECUBES[top:#x,
                 middle:#y,
                 above:#x,
                 below:#y].
ON2 = THREECUBES[middle:#z,
                 bottom:#t,
                 above:#z,
                 below:#t].
```

The new partial order on type symbols extracted from the KB definitions is shown in Figure 6.

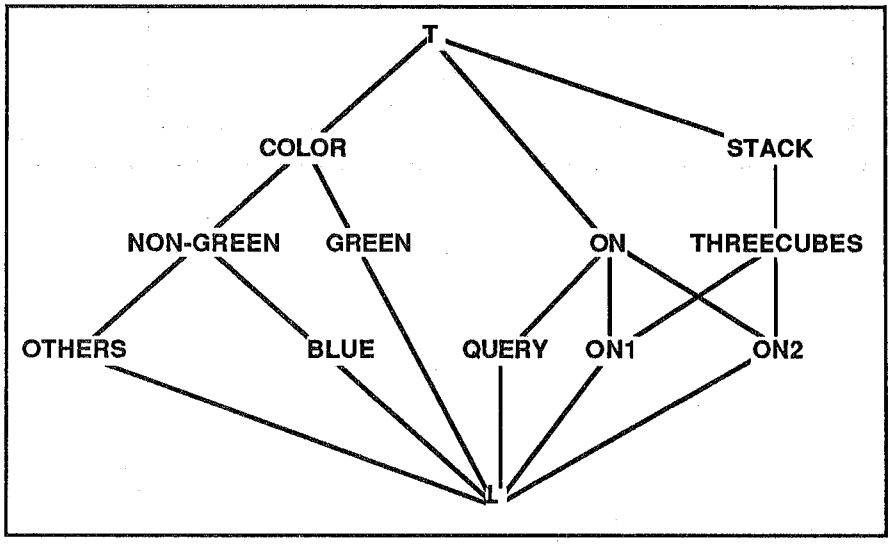

Figure 6: The partial ordering on type symbols extracted from the KB definitions

The partial ordering on type symbols can be extended to a partial ordering on typed feature structures, as shown in Figure 7.

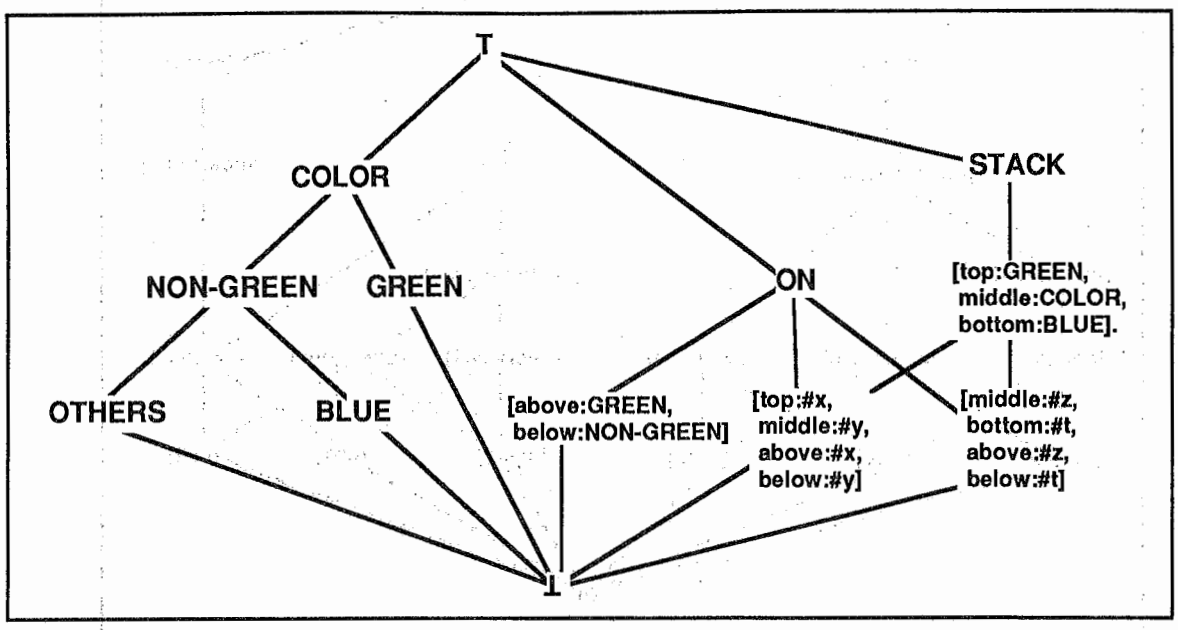

Figure 7: The partial ordering on typed feature structures extracted from the KB definitions.

B . The Meet Operation

As we allow any kind of definition, the partial order extracted from the type definitions is not guaranted to be a lattice, as in the example above: there are two solutions for the meet of ON and THREECUBES, ON1 and ON2. The solution is to embed the partial ordering P extracted form the KB definitions in the (restricted) power set $2^{(P)}$ (the set of all non-empty finite subsets of pairwise not comparable elements of P).

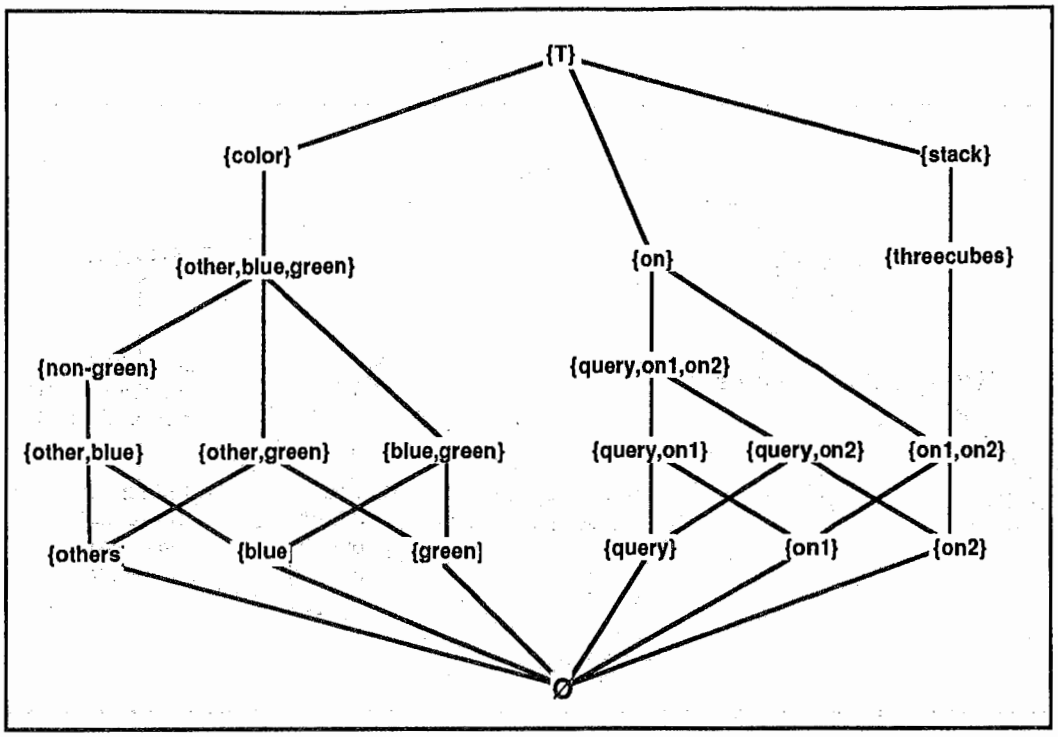

Figure 8: Part of the restricted power set

The meet operation is then defined using the inclusion ordering in $2^{(P)}$: the meet of two elements X and Y of P is then defined as the maximal restriction of the intersection of the sets of sub-type symbols for each pair of symbols x and y of X and Y.

The maximal restriction $\lceil P \rceil$ takes the set of maximal elements of P: when 2 elements are comparable, the smaller is removed: $\lceil P \rceil = \{x \in P \mid y \leq x \Rightarrow x = y\}$. The set of sub-types symbols of x is called the ideal principal of P generated by x : $I_x = \{y \in P \mid y \leq x\}$.

If X and Y are elements of $2^{(P)}$, the meet of X and Y is defined as

$$X \wedge Y = \lceil \cup_{x \in X, y \in Y} (I_x \cap I_y) \rceil.$$ (Note that this construction does not preserve LUBs)

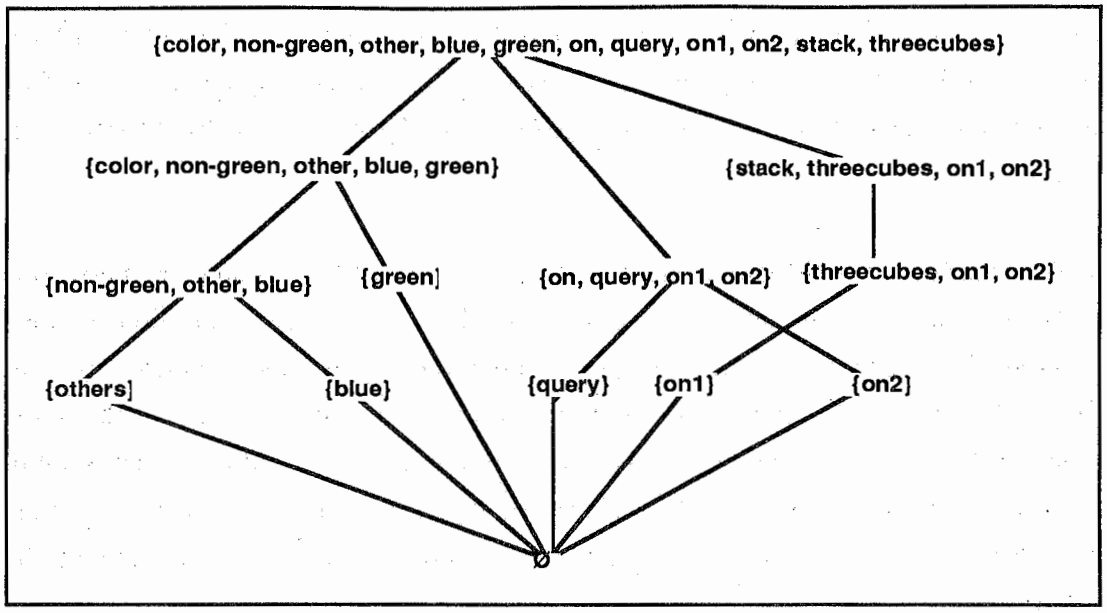

Figure 9: ***The embedding in the set of all principal ideals preserve the GLBs: it is a meet semi-lattice.***

Example:

$$\text{ON} : \{\text{ON}, \text{QUERY}, \text{ON1}, \text{ON2}\},\ \text{THREECUBES} : \{\text{THREECUBES}, \text{ON1}, \text{ON2}\}\ .$$

$$\text{ON} \wedge \text{THREECUBES} : \{\text{ON}, \text{QUERY}, \text{ON1}, \text{ON2}\} \wedge \{\text{THREECUBES}, \text{ON1}, \text{ON2}\} \rightarrow \{\text{ON1}, \text{ON2}\}$$

The RETIF interpreter directly uses this meet operation on sets of type symbols, eliminating the need for the so-called psi-expansion, as suggested in [Aït-Kaci 84, pp155-156]: the interpreter prunes all non-maximal elements in one step of computation during unification before expanding a disjunction.

Implementation note

The efficiency of the meet operation is crucial for the overall performance of the interpreter. The evaluation of the cost of the unification algorithm (which is almost linear with the number of nodes) does not take into account the cost of the meet operation.

If $p=|P|$ the number of symbol in the type system P, then the size of X and Y, elements of $2^{(P)}$ is bound by p. The cost of computing an ideal is the cost of a traversal of the graph which represents the partial order, and is linear with the number of nodes p, and there are at most 2p such traversals (one for each element of X and Y): $2p^2$. The size of an ideal, element of $2^{(P)}$ is bounded by p. The cost of an intersection is then p^2 and there are also at most p^2 intersections: p^4. The cost of each union is also

p^2, and there are at most p^2 unions: p^4. The maximal restriction compares each element of the set with each other: p^2 comparison. Each comparison requires a traversal of the graph and costs p. The cost of the maximal restriction is then in p^3. Finally, we get $O(2p^3 + p^4 + p^4 + p^3) = O(p^4)$ for the cost of the meet operation.

During the construction of the lattice out of the KB definitions, all ideals and the intersections of all ideals are pre-computed, and the intersections are cleaned up by maximal restriction. As the meet is most often a meet between two singletons, this is already pre-computed and requires only an access to a hash table, but the cost of the meet is still in $O(p^4)$ in the worst case. The size of the hash table is $O(p^2)$. It could be possible in principle to pre-compute all meets on $2^{(P)}$ and have a constant time, but the space requirement is extravagant: $O(2^{|P|})$. However, this size would probably be small in practical use, if only meets different from Ø are stored, and if we take into account the commutativity of the meet operation.

3. The Interpreter

3.1 The EVAL function

A knowledge base KB consists of a set of definitions $t_i = KB(t_i)$, where t_i is a type symbol and $KB(t_i)$ is either a term (a typed feature structure) $f[l_1: t_1, \ldots, l_n: t_n]$ or a disjunction of type symbols $t_1 | \ldots t_n$. We can define a function EVAL : *term* $\rightarrow$ *term*

(1) $\text{EVAL}(t_1 | \ldots t_n) = \bigvee_{i=1\ldots n} \text{EVAL}(t_i)$

(2) $\text{EVAL}(f[l_1: t_1, \ldots, l_n: t_n]) = \text{EVAL}(KB(f) \wedge T[l_1: \text{EVAL}(t_1), \ldots, l_n: \text{EVAL}(t_n)])$

Equations (1) and (2) define an *operational semantics* which reflects the type-as-set semantics of *terms* in the sense that they compute unions and intersections of sets (cf. [Aït-Kaci 84, pp 117-147]).

3.2 The Rewriting Algorithm

Def: A **Symbol Rewriting System** (SRS) on Σ (signature of type symbols) is a system S of n equations $s_i = E_i$, $i = 1, \ldots, n$ where $s_i \in \Sigma$ and E_i is a term: $S = \{ s_i = E_i \}$.

The set of symbols of Σ which have a definition is $E = \{ s_1, \ldots, s_n \}$, the set of *S-expandable symbols*. The set of symbols of Σ which do not have a definition is $N = \Sigma - E$, the set of *non-S-expandable symbols*. In the three cubes example, E = { COLOR, NON-GREEN, ON, ON1, ON2, QUERY } and N = { GREEN, BLUE, PURPLE, OTHERS, STACK }.

Def: A *one step rewriting relation* $t_1 \rightarrow t_2$ is defined *iff* there exists a symbol $s_i \in E$ at some address (path) u in t_1 such that E_i is «substituted» at address u: E_i is unified with the sub-term at adress u, and the result of unification is inserted at that adress. The new term is called t_2:

$t_2 = t_1[E_i/u] = t_1[u: T] \wedge u.E_i$

3.3 A detailed example

Let's take the three cubes KB:

```
COLOR = GREEN | NON-GREEN.   (= E1)
NON-GREEN = BLUE | PURPLE | OTHERS.   (= E2)
THREECUBES = STACK   [top:GREEN, middle:COLOR, bottom:BLUE].   (= E3)
ON = ON1 | ON2.   (= E4)
ON1 = THREECUBES[top:#x, middle:#y, above:#x, below:#y].   (= E5)
ON2 = THREECUBES[middle:#z, bottom:#t, above:#z, below:#t].   (= E6)
QUERY = ON[above:GREEN, below:NON-GREEN].   (= E7)
```

We shall show the behavior of the interpreter on the evaluation of the term QUERY : is there a green cube on top of a non-green cube?

At first we substitute the type symbol of the query with the corresponding KB value (E_7). In particular this is performed by a substitution of the root symbol QUERY (at each step, expandable symbols are written in bold face).

t_1 = **QUERY**

$t_2 = t_1[E_7/\varepsilon] = t_1[\varepsilon:T] \wedge \varepsilon$.ON[above: GREEN, below:NON-GREEN]
= **ON**[above:GREEN, below:**NON-GREEN**]

The new term t_2 is expandable with symbol ON at address ε and with symbol NON-GREEN at address *below*: Note that the expansion of NON-GREEN is a disjunction of symbols BLUE | PURPLE | OTHERS, which are not further expandable. This disjunction is kept local.

$t_3 = t_2[E_4/\varepsilon]$
= (ON1 | ON2) [above:GREEN, below:**NON-GREEN**]
$t_4 = t_3[E_2$/below]
= (ON1 | ON2) [above:GREEN, below: BLUE | PURPLE | OTHERS]

After these two rewritings of t_2 we get a term with a disjunction of symbols at the root and at address *below*. Notice, that only the disjunction at the top-level has to be further expanded: the disjunction at address *below* need not be further expanded because these symbols are

non-S-expandable symbols. The next step expands the disjunction ON1 | ON2 and create two new terms t_4' and t_4'' with type symbols ON1 and ON2 respectively. These terms are further expanded in two differents branches of computation.

Branch 1 of the disjunction: expand ON1.

t_4' = **ON1**[above: GREEN, below: BLUE | PURPLE | OTHERS]

$t_5 = t_4'[E_5/\varepsilon]$ = **THREECUBES**[top: #x=GREEN,
middle: #y= BLUE | PURPLE | OTHERS,
above: #x, below: #y]

We yield the first solution by finally expanding THREECUBES. At this time the rewriting process stops because there are no more S-expandable symbols in t_6.

First solution: $t_6 = t_5[E_3/\varepsilon]$ = STACK[top:#x=GREEN,
middle:#y=BLUE | PURPLE | OTHERS,
bottom:BLUE,
above:#x, below:#y]

Branch 2 of the disjunction: expand ON2.

t_4'' = ON2[above: GREEN, below: BLUE | PURPLE | OTHERS]

$t_7 = t_4''[E_5/\varepsilon]$ = **THREECUBES**[middle: #z= GREEN,
bottom: #t = BLUE | PURPLE | OTHERS,
above: #z,
below: #t]

Second solution: $t_8 = t_7[E_5/\varepsilon]$ = STACK[top: GREEN,
middle: #z = GREEN,
bottom: #t = BLUE,
above: #z,
below: #t]

4. Developments

4.1 Keeping Disjunctions Local

In the present implementation, all disjunctions are expanded to yield a set of terms where no disjunctions occur inside a term, except for disjunctions of atomic type symbols which cannot be rewritten.

First, this systematic expansion is computationally very costly: the interpreter must create as many copies of a term as there are disjuncts, where some will probably be rewritten to ⊥.

Second, in some cases, it is preferable to describe alternatives locally inside a term, factoring common sub-parts, rather than expanding the whole expression. This could be done with the introduction of a dummy type symbol for a local disjunction, and then this symbol would be defined as a disjunction. However, it would be much simpler to allow disjunctions to appear inside a term. For grammar writing for example, it would be preferable to keep the description of possible ambiguities local: this would help to keep track of the sources of ambiguities, and to make possible a finer analysis of these phenomena.

For implementation, the algorithm described in [Eisele and Dörre 88] would be a good candidate, and could be incorporated in the present unification algorithm. The syntax of the formalism needs to be extended to allow disjunctions of terms inside a term instead of having disjunctions at the top level of a definition only. There would then be three different possible computational behaviors for handling disjunctions: (1) all disjunctions are expanded to the top level of terms, as it is done presently, (2) internal disjunctions are expanded locally and disjunctions at the top level of a definition are only expanded to the top level, (3) all disjunctions are expanded locally and the result is a single term. Note that it is possible in any case to factorize a set of typed feature structures, and build a compact typed feature description (with local disjunctions) that represent this set (as the «pack» function described in [Eisele and Dörre 88]).

4.2 Multiple Inheritance

The interpreter described in [Aït-Kaci 84] allows only *single* inheritance. For grammar description, we need *multiple* inheritance, especially for lexical descriptions: the verb *like* could be described as a BASE verb and a MAIN verb and a STRICT-TRANSITIVE verb [Pollard and Sag 87].

We shall augment knowledge base definitions to handle disjunction and give the definition of the evaluation function for disjunctions. A knowledge base KB consists of a set of definitions $t_i = KB(t_i)$,

and $KB(t_i)$ is then allowed to be a conjunction of type symbols t_1 & ... t_n. The EVAL function is augmented with the following definition:

(3) $\quad EVAL(\ t_1\ \&\ \ldots\ t_n\ ) = \&_{i\,=\,1\ldots n}\ EVAL(t_i)$

As for local disjunctions, multiple inheritance is syntactically nothing other than a conjunction of terms, and the formalism can be extended to accomodate local conjunctions.

4.3 Negation

Another extension we need is negation. This was introduced by [Aït-Kaci 84] as complemented types. Negation could always be written as it is in the three cubes example: the set of colors is divided into two sub-sets: GREEN and NON-GREEN. NON-GREEN can in turn be decomposed in the same way. But this makes the description of knowledge very cumbersome. [Aït-Kaci 84] proposed writing COLOR\GREEN and write COLOR as a simple set of colors. This is equivalent saying COLOR $\wedge \neg$GREEN, and if we do not want to overspecify, we can simply state $\neg$GREEN.

Let's take for example the term STUDENT[sport: SURFING]. The set of students who practice surfing is a sub-set of the set of students who practice a sport, which is a sub-set of the set of students, which is a sub-set of the set of persons, as depicted in Figure 10. With the type-as-set interpretation in mind, it is not difficult to see that $\neg$STUDENT[sport: SURFING] should be interpreted as the set of persons who are not students or if they are students, then we take the set of students who do not practice any sport, or if a sport is practiced, it should not be surfing.

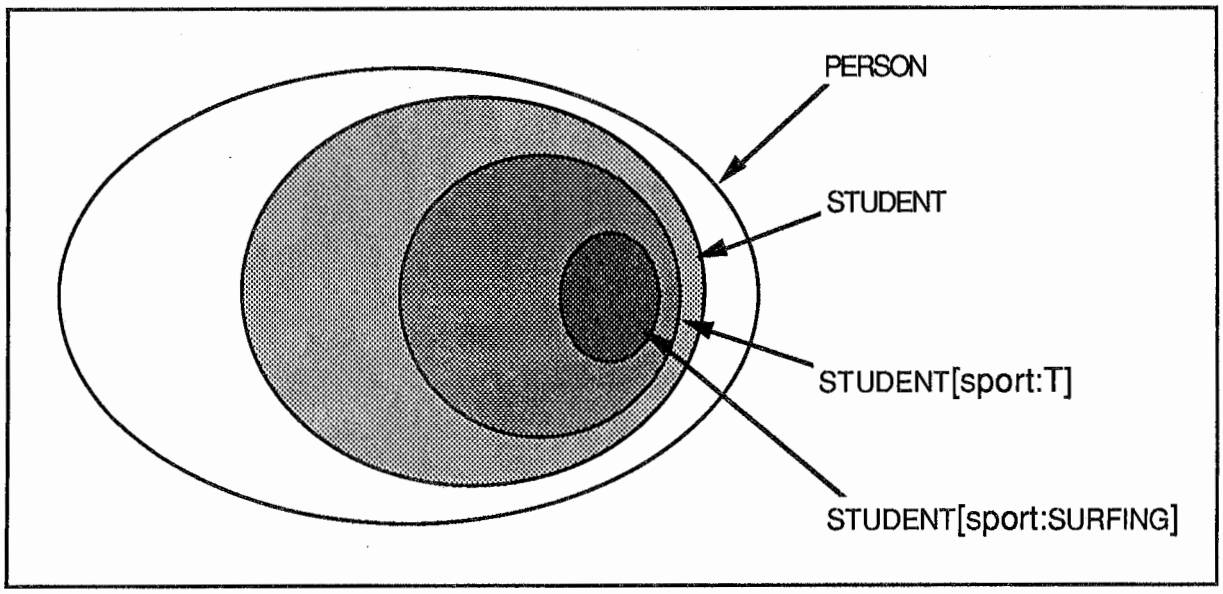

Figure 10: type as set semantics for negation.

The semantics of complemented types is described in [Aït-Kaci 84]. The negation is described in [Smolka 88] and is defined there as: $\neg$(label: TYPE) = $\neg$label: T $\vee$ label: $\neg$TYPE.

Our definition of negation for typed terms is then:

$$\begin{aligned} \neg A[l_1: S_1, \ldots l_n:S_n] &= \neg A \vee A[\neg(l_1: S_1), \ldots \neg(l_n:S_n)] \\ &= \neg A \vee A[\neg l_1:T \vee l_1: \neg S_1, \ldots \neg l_n:T \vee l_n:\neg S_n] \\ &= \neg A \vee A[l_1: \neg S_1, \ldots l_n:\neg S_n] \end{aligned}$$

The EVAL function is augmented with the following definition:

(4) $\quad EVAL(\neg t_1) = \neg\, EVAL(t_i)$

All those extensions are very basic ones, and give the formalism a good expressive power. Further developments can be envisaged once full logical expressions have been implemented (see for example [Aït-Kaci 84 Chap 7], [Kasper 88]).

5. Applications to Natural Language Understanding

We shall present here some input/ouput typed feature structures for the transfer and generation grammars described in the appendix. Note that this example is built to examplify some features of the RETIF formalism, and thus the grammars have been kept as simple as possible. We refer to [Emele 89, Zajac 89] for more details on transfer and generation grammars.

An example of transfer from Japanese to English

The input structure is shown in Figure 11: it could be the structure produced by application of an abstract communicative act grammar on the result of a surface parser, as described in [Kogure, Yoshimoto et al. 88]. The output structure (Figure 12) is produced by the application of the transfer grammar on this structure. Note that the process is monotonic and works by addition of new information: the result is a structure which contains all information of the input structure. This property also makes the transfer grammar reversible.

```
TRANSFER-ACA
[japanese:J-PLAN-SCHEMA
          [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                    [relation:REQUEST,
                     agent:#20,
                     recipient:#19,
                     object:J-PROP
                            [relation:J-OKURU,
                             agent:#19,
                             recipient:#20,
                             object:J-TOUROKUYOUSHI],
                    manner:INDIRECTLY-BY-ASKING-POSSIBILITY]]]
```

Figure 11: Japanese input for transfer.

```
[english:E-PLAN-SCHEMA
        [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                  [relation:#1=REQUEST,
                   agent:#12=E-SPEAKER,
                   recipient:#11=E-HEARER,
                   object:#6=E-PROP
                            [relation:E-SEND,
                             agent:#10=E-HEARER,
                             recipient:#9=E-SPEAKER,
                             object:#7=E-REGISTRATION-FORM],
                   manner:#4=INDIRECTLY-BY-ASKING-POSSIBILITY]],
 japanese:J-PLAN-SCHEMA
          [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                    [relation:#1,
                     agent:#3=J-SPEAKER,
                     recipient:#2=J-HEARER,
                     object:#5=J-PROP
                              [relation:J-OKURU,
                               agent:#2,
                               recipient:#3,
                               object:#8=J-TOUROKUYOUSHI],
                     manner:#4]],
 translate-obj:[english:#6,
                japanese:#5,
                translate-obj:[english:#7, japanese:#8],
                translate-rec:[english:#9, japanese:#3],
                translate-agt:[english:#10, japanese:#2]],
 translate-rec:[english:#11, japanese:#2],
 translate-agt:[english:#12, japanese:#3]]
```

Figure 12: result of Japanese-English transfer.

An example of English generation

The input structure (Figure 13) is a part of a structure that could be produced by a transfer grammar: the level of description is the level of sematic relations. After application of the generation grammar, the output structure (Figure 14) describes the constituent structure and the associated string of lexems (feature «phon»).

```
E-PROP
[relation:E-SEND,
 agent:E-HEARER,
 recipient:E-SPEAKER,
 object:E-REGISTRATION-FORM]
```

Figure 13: a fragment input for English generation.

```
PHRASAL-SIGN
[phon:#39=<#1="send"
           ; #38=<#2="I"
           ; #36=<#9=a | the
           ; #37=<#10="registration form">>>>,
 relation:#31=LEXICAL-SIGN
              [phon:#40=<#1>,
               syn:CATEGORY
                   [head:#29=[lexem:#1],
                    subcat:<#27=PHRASAL-SIGN
                                [phon:#6=<#2>,
                                 relation:#2,
                                 syn:CATEGORY
                                     [head:#3=[lexem:#2],
                                      subcat:#4=<>],
                                 dtrs:TREE
                                      [head-dtr:LEXICAL-SIGN
                                                [phon:#7=<#2>,
                                                 syn:CATEGORY
                                                     [head:#3,
                                                      subcat:#4]],
                                       comp-dtrs:<>]]
                           ; #32=<#28=PHRASAL-SIGN
                               [phon:#17=<#9
                                          ; #11=<#10>>,
                                relation:#10,
                                syn:CATEGORY
                                    [head:#12=[lexem:#10],
                                     subcat:<>],
                                dtrs:TREE
                                     [head-dtr:PHRASAL-SIGN
                                               [phon:#11,
                                                syn:CATEGORY
                                                    [head:#12,
                                                     subcat:#13=<LEXICAL-SIGN
                                                                 [phon:#18=<#9>,
                                                                  syn:CATEGORY
                                                                 [head:
                                                                  [lexem:#9]]]>],
                                                dtrs:TREE
                                                     [head-dtr:LEXICAL-SIGN
                                                               [phon:#15=<#10>,
                                                                syn:CATEGORY
                                                                    [head:#12,
                                                                     subcat:#13]],
                                                      comp-dtrs:<>]],
                                      comp-dtrs:#13],
                               spec:#9]
                           ; #30>>]],
agent:PHRASAL-SIGN
      [phon:#23=<#20="you">,
       relation:#20,
       syn:CATEGORY
           [head:#21=[lexem:#20],
            subcat:#22=<>],
       dtrs:TREE
            [head-dtr:LEXICAL-SIGN
                      [phon:#24=<#20>,
                       syn:CATEGORY
```

```
                              [head:#21,
                               subcat:#22]],
             comp-dtrs:<>]],
recipient:#27,
object:#28,
syn:CATEGORY
    [head:#29,
     subcat:#30],
dtrs:TREE
     [head-dtr:#31,
      comp-dtrs:<#27
                 ; #33=<#28>>]]
```

Figure 14: the result of English generation.

Conclusion

The main characteristics of the formalism are (1) type inheritance which provides a clean way of defining classes and sub-classes of objects, and (2) the rewriting mechanism based on unification of typed feature structures which provides a very powerful and semantically clear mean of specifying and computing *relations* between classes of objects. In this respect, the formalism can be compared favourably with PROLOG (see the append example in appendix 2).

So far, we have used this formalism to develop sample grammars for transfer [Zajac 89] and generation [Emele 89] in order to demonstrate the feasability of this approach for natural language generation and transfer. It can also be used for parsing, in a DCG-like style, as shown in [Aït-Kaci 84, pp161-165], and therefore seems to be useful in more general natural language applications.

The interpreter has been implemented in Common Lisp, and runs on Vax and Symbolics. Some implementation work is still needed to achieve greater efficiency, and to provide a better user interface. However, the system can already be used for the development of NLP systems, and the speed is no longer a limiting factor (even on a Micro-Vax).

The main developments envisaged in a second step are the implementation of multiple inheritance and the implementation of negation. The introduction of functional application ought to be studied.

References

Hassan AIT-KACI, 1984, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, Ph.D. Thesis, University of Pennsylvania.

Hassan AIT-KACI, 1986, An Algebraic Semantics Approach to the Effective Resolution of Type Equations, *Theoretical Computer Science* 45, pp 293-351.

Hassan AIT-KACI and Roger NASR, 1986, LOGIN: a Logic Programming Language with Built-in Inheritance, *J. of Logic Programming*, 3, pp 185-215.

Martin EMELE, 1989, A Generation Model Using a TFS Rewriting System with Inheritance, submitted for the *2nd European Workshop on Narural Language Generation*, 6-8 April, University of Edinburgh.

Andreas EISELE and Jochen DÖRRE, 1988, Unification of Disjunctive Feature Descriptions, *Proc. of the 26h Annual Meeting of the ACL*, 7-10 June, Buffalo, pp 286-294.

Pierre ISABELLE and Eliot MACKLOVITCH, 1986, Transfer and MT Modularity, *Proceedings of COLING-86*, Bonn.

Ron KAPLAN and J. BRESNAN, 1982, Lexical Functional Grammar, a Formal System for Grammatical Representation. In J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, The MIT Press, 1982, pp 173-381.

Robert T. KASPER and William C. ROUNDS, 1986, A Logical Semantics for Feature Structures, *Proceedings of the 24th Annual Meeting of the ACL*, 10-13 June, Columbia University, New-York, pp 257-266.

Robert T. KASPER, 1987, A Unification Method for Disjunctive Feature Descriptions, *Proceedings of the 25th Annual Meeting of the ACL*, 6-9 June, Stanford University, pp 235-242.

Robert T. KASPER, 1988, Conditional Descriptions in Functional Unification Grammar, *Proceedings of the 26h Annual Meeting of the ACL*, 7-10 June, Buffalo, pp 233-240.

Martin KAY, 1984, Functional Unification Grammar: a Formalism for Machine Translation, *Proceedings of COLING-84*.

Kiyoshi KOGURE, Kei YOSHIMOTO, Hitoshi IIDA, and Teruaki AIZAWA, 1989, The Intention Translation Method, A New Machine Translation Method for Spoken Dialogues, submitted for *IJCAI-89*, Detroit.

Ikuo KUDO and Hirosato NOMURA, 1986, Lexical-Functional Transfer: A Transfer Framework in a Machine Translation System based on LFG, *Proceedings of COLING-86*, Bonn, 112-114.

Masako KUME, Gayle K. SATO, and Kei YOSHIMOTO, 1989, A Descriptive Framework for Translating Speaker's Meaning - Towards a Dialogue Translation System between Japanese and English, *4th Conference of ACL-Europe*, Manchester.

Kuniaki MUKAI, 1988, Partially Specified Terms in Logic Programming for Linguistic Analysis, *Proc. of the International Conference on Fifth Generation Computer Systems*, Nov.28-Dec.2, Tokyo, pp 479-488.

Carl POLLARD and Ivan A. SAG, 1987, *Information-based Syntax and Semantics*, CSLI, Lecture Notes Number 13.

Stuart M. SHIEBER, 1986, *An Introduction to Unification-based Approaches to Grammar*, CSLI, Lecture Notes Number 4.

Gert SMOLKA, 1988, *A Feature Logic with Subsorts*, LILOG-REPORT 33, IBM Deutschland GmbH, Stuttgart.

Kei YOSHIMOTO, Kiyoshi KOGURE, 1988, *Japanese Sentence Analysis by means of Phrase Structure Grammar*, ATR Technical Report TR-I-0049.

David A. WROBLEWSKI, 1987, Nondestructive Graph Unification, *Proc. of the 6th National Conference on AI, AAAI-87*, July 13-17, Seattle, pp 582-587.

Rémi ZAJAC, 1989, A Transfer Model Using a TFS Rewriting System with Inheritance, *27th Annual Meeting of the ACL*, 26-29 June, Vancouver.

Appendix: a sample of transfer and generation grammars

1. A grammar for surface/abstract Japanese communicative act representation

```
:KB J-CA

J-APPLY-SIMPLE-DISCOURSE-HEURISTICS =  _ replaces the variables for speaker
                                       _ and hearer with instances
                                     [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                                               [agent:J-SPEAKER,
                                                recipient:J-HEARER]] .

J-PLAN-SCHEMA =  _ toplevel plan recognition rule
                 _ (it should be of course a disjunction of others)
               J-APPLY-SIMPLE-DISCOURSE-HEURISTICS
               [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                         [relation:REQUEST,
                          agent:#agent,
                          recipient:#recipient,
                          object:#object=[agent:#recipient],
                          manner:INDIRECTLY-BY-ASKING-POSSIBILITY],
                surface:J-SURFACE-COMMUNICATIVE-ACT
                        [relation:J-INTERROGATE-IF,
                         agent:#agent,
                         recipient:#recipient,
                         object:J-SURFACE-COMMUNICATIVE-ACT
                                [relation:J-CAN,
                                 agent:#agent,
                                 object:J-SURFACE-COMMUNICATIVE-ACT
                                        [relation:J-RECEIVE-FAVOR,
                                         agent:#agent,
                                         object:#object,
                                         source:#recipient]]]] .
```

2. Some examples for Japanese

```
J-SCA-1 = _ Japanese structure produced by the parser for
          _ "watashi-ni tourokuyousi-wo o-okuri itadake masu ka?"
          _ the evaluation produces the abstract communicative act:
          _ (kbl-eval 'j-sca-1)
          J-PLAN-SCHEMA
          [surface:J-SURFACE-COMMUNICATIVE-ACT
                   [relation:J-INTERROGATE-IF,
                    agent:#15,
                    object:J-SURFACE-COMMUNICATIVE-ACT
                           [relation:J-CAN,
                            agent:#14,
                            object:J-SURFACE-COMMUNICATIVE-ACT
                                   [relation:J-RECEIVE-FAVOR,
                                    agent:#14,
                                    object:J-PROP
                                           [relation:J-OKURU,
                                            agent:#16,
                                            recipient:#15,
                                            object:J-TOUROKUYOUSHI]],
                            source:#16]]] .
```

```
J-ACA-1 = _ abstract communicative act for
          _ "watashi-ni tourokuyousi-wo o-okuri itadake masu ka?"
          _ the evaluation produces the surface communicative act:
          _ (kbl-eval 'j-aca-1)
          J-PLAN-SCHEMA
          [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                    [relation:REQUEST,
                     agent:#18,
                     recipient:#17,
                     object:J-PROP
                            [relation:J-OKURU,
                             agent:#17,
                             recipient:#18,
                             object:J-TOUROKUYOUSHI],
                     manner:INDIRECTLY-BY-ASKING-POSSIBILITY]] .

J-ACA-TRANS-1 = _ abstract communicative act for
                _ "watashi-ni tourokuyousi-wo o-okuri itadake masu ka?"
                _ The evaluation produces the English abstract communicative act:
                _ (kbl-eval 'j-aca-trans-1 '(j-ca ej-trans))
                _ To produce the English phonological string:
                _ (kbl-eval 'j-aca-trans-1 '(j-ca ej-trans e-ca e-gen))
                TRANSFER-ACA
                [japanese:J-PLAN-SCHEMA
                          [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                                    [relation:REQUEST,
                                     agent:#20,
                                     recipient:#19,
                                     object:J-PROP
                                            [relation:J-OKURU,
                                             agent:#19,
                                             recipient:#20,
                                             object:J-TOUROKUYOUSHI],
                                     manner:INDIRECTLY-BY-ASKING-POSSIBILITY]]] .

J-SCA-TRANS-1 = _ the Japanese input for plan recognition and transfer:
                _ "watashi-ni tourokuyousi-wo o-okuri itadake masu ka?"
                _ the slot 'japanese.surface' is the result of the parser
                _ the evaluation produces the abstract communicative act
                _ representation and the English part:
                _ (kbl-eval 'j-sca-trans-1 '(j-ca ej-trans e-ca e-gen))
                TRANSFER-ACA
                [japanese:
                 J-PLAN-SCHEMA
                 [surface:J-SURFACE-COMMUNICATIVE-ACT
                          [relation:J-INTERROGATE-IF,
                           agent:#22,
                           object:J-SURFACE-COMMUNICATIVE-ACT
                                  [relation:J-CAN,
                                   agent:#21,
                                   object:J-SURFACE-COMMUNICATIVE-ACT
                                          [relation:J-RECEIVE-FAVOR,
                                           agent:#21,
                                           object:J-PROP
                                                  [relation:J-OKURU,
                                                   agent:#23,
                                                   recipient:#22,
                                                   object:J-TOUROKUYOUSHI]],
                                   source:#23]]]] .
```

3. A transfer grammar

```
:KB EJ-TRANS

TRANSFER-ACA =  _ transfer rule for abstract communicative acts.
               [english:E-PLAN-SCHEMA
                        [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                                 [relation:#rel,
                                  agent:#e-agt,
                                  recipient:#e-rec,
                                  object:#e-obj,
                                  manner:#manner]],
                japanese:J-PLAN-SCHEMA
                         [abstract:J-ABSTRACT-COMMUNICATIVE-ACT
                                  [relation:#rel,
                                   agent:#j-agt,
                                   recipient:#j-rec,
                                   object:#j-obj,
                                   manner:#manner]],
               translate-agt:TRANS[english:#e-agt, japanese:#j-agt],
               translate-rec:TRANS[english:#e-rec, japanese:#j-rec],
               translate-obj:TRANS[english:#e-obj, japanese:#j-obj]] .

TRANS = _ transfer rules for the propositional part.
       SEND | REG-FORN | INTERLOCUTORS.

SEND  =  _ a relation: translate recursively the arguments.
        [english:E-PROP
                 [relation:E-SEND,
                  agent:#e-agt,
                  recipient:#e-rec,
                  object:#e-obj],
         japanese:J-PROP
                  [relation:J-OKURU,
                   agent:#j-agt,
                   recipient:#j-rec,
                   object:#j-obj],
         translate-agt:TRANS[english:#e-agt, japanese:#j-agt],
         translate-rec:TRANS[english:#e-rec, japanese:#j-rec],
         translate-obj:TRANS[english:#e-obj, japanese:#j-obj]] .

REG-FORM =  _ a noun.
            [english:E-REGISTRATION-FORM, japanese:J-TOUROKUYOUSHI] .

INTERLOCUTORS = SPEAKER | HEARER.

SPEAKER = [english:E-SPEAKER, japanese:J-SPEAKER].

HEARER = [english:E-HEARER, japanese:J-HEARER].
```

4. A grammar for surface/abstract English communicative act representation

```
:KB E-CA

E-APPLY-SIMPLE-DISCOURSE-HEURISTICS =  _ replaces the variables for speaker
                                       _ and hearer with instances
                                     [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                                              [agent:E-SPEAKER,
                                               recipient:E-HEARER]] .

E-PLAN-SCHEMA = E-APPLY-SIMPLE-DISCOURSE-HEURISTICS
               [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                        [relation:REQUEST,
                         agent:#42,
                         recipient:#41,
                         object:#43=E-PROP[agent:#41],
                         manner:INDIRECTLY-BY-ASKING-POSSIBILITY],
                surface:E-SURFACE-COMMUNICATIVE-ACT
                       [relation:E-INTERROGATE-IF,
                        agent:#42,
                        recipient:#41,
                        object:E-ASK-MODALITY
                               [relation:E-CAN,
                                agent:#41,
                                object:#43]]] .
```

5. Some English examples

```
E-PROP-1 = _ The evaluation produces the surface representation and
           _ the phonological string "send I (a the) registration form".
           _ (kbl-eval 'e-prop-1 '(e-ca e-gen))
           E-PROP
           [relation:E-SEND,
            agent:E-HEARER,
            recipient:E-SPEAKER,
            object:E-REGISTRATION-FORM].

E-SCA-1 = _ the English surface representation for
          _ "could you send me the registration form"
          _ the evaluation will produce the english phonological string
          _ and the abstract communicative act representation.
          _ (kbl-eval 'e-sca-1 '(e-ca e-gen))
          E-PLAN-SCHEMA
          [surface:E-SURFACE-COMMUNICATIVE-ACT
                   [relation:E-INTERROGATE-IF,
                    agent:#45=E-SPEAKER,
                    recipient:#44=E-HEARER,
                    object:E-ASK-MODALITY
                           [relation:E-CAN,
                            agent:#44,
                            object:E-PROP
                                   [relation:E-SEND,
                                    recipient:#45,
                                    object:E-REGISTRATION-FORM]]]] .
```

```
E-ACA-1 = _ the English abstract representation for
          _ "could you send me the registration form"
          _ the evaluation will produce the surface representation and the
          _ phonological string: (kbl-eval 'e-aca-1 '(e-ca e-gen))
          E-PLAN-SCHEMA
          [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                    [relation:REQUEST,
                     agent:#40=E-SPEAKER,
                     recipient:#39=E-HEARER,
                     object:E-PROP
                            [relation:E-SEND,
                             agent:#39,
                             recipient:#40,
                             object:E-REGISTRATION-FORM],
                     manner:INDIRECTLY-BY-ASKING-POSSIBILITY]].


E-ACA-TRANS-1 =   _ The English abstract representation for
                  _ "could you send me the registration form"
                  _ The evaluation produces the Japanese abstract and
                  _ surface representations.
                  _ (kbl-eval 'e-aca-trans-1 '(e-ca ej-trans j-ca))
                  _ or the English phonological string:
                  _ (kbl-eval 'e-aca-trans-1 '(e-ca e-gen))
     TRANSFER-ACA
     [english: E-PLAN-SCHEMA
          [abstract:E-ABSTRACT-COMMUNICATIVE-ACT
                    [relation:REQUEST,
                     agent:#25=E-SPEAKER,
                     recipient:#24=E-HEARER,
                     object:E-PROP
                            [relation:E-SEND,
                             agent:#24,
                             recipient:#25,
                             object:E-REGISTRATION-FORM],
                     manner:INDIRECTLY-BY-ASKING-POSSIBILITY]]] .


E-SCA-TRANS-1 =   _ the English surface representation for
                  _ "could you send me the registration form"
                  _ the evaluation produces the Japanese surface representation:
                  _ (kbl-eval 'e-input-1 '(e-ca ej-trans j-ca)) or the English
                  _ surface string: (kbl-eval 'e-input-1 '(e-ca e-gen))
     TRANSFER-ACA
     [english: E-PLAN-SCHEMA
          [surface:E-SURFACE-COMMUNICATIVE-ACT
                   [relation:E-INTERROGATE-IF,
                    agent:#26=E-SPEAKER,
                    recipient:#27=E-HEARER,
                    object:E-ASK-MODALITY
                           [relation:E-CAN,
                            agent:#27,
                            object:E-PROP
                                   [relation:E-SEND,
                                    agent:#27,
                                    recipient:#26,
                                    object:E-REGISTRATION-FORM],
                            source:#27]]]] .
```

6. A grammar for English generation

```
:KB E-GEN

LIST = END | CONS .

APPEND = APPEND0 | APPEND1 .

APPEND0 = [whole:#1=LIST, front:<>, back:#1] .

APPEND1 = [front:<#2 ; #4>,
           back:#5=LIST,
           whole:<#2 ; #3>,
           patch:APPEND[front:#4, back:#5, whole:#3]].


MINOR =  _ Lexical head: relates lexem with the phonological string
        LEXICAL-SIGN
        [phon:<#6>,
         syn:CATEGORY [head:[lexem:#6]]] .

MAJOR = MAJOR1 | MAJOR2 .

MAJOR1 =  _ saturated phrasal sign =
          _ head phrasal sign + single complement phrasal sign
         PHRASAL-SIGN
         [phon:#9,
          syn:CATEGORY
              [head:#7,
               subcat:<>],
          dtrs:TREE
               [head-dtr:PHRASAL-SIGN
                         [phon:#11,
                          syn:CATEGORY
                              [head:#7,
                               subcat:#8=<[phon:#10]>]],
                comp-dtrs:#8],
          patch0:APPEND[whole:#9, front:#10, back:#11]] .

MAJOR2 =  _ unsaturated phrasal sign =
          _ head lexical sign + complement phrasal sign*
         MAP
         [phon:#13,
          syn:CATEGORY[head:#12],
          dtrs:TREE
               [head-dtr:LEXICAL-SIGN
                         [phon:#14,
                          syn:CATEGORY[head:#12]]],
          patch0:APPEND[whole:#13, front:#14, back:#15],
          patch1:[whole:#15]] .

MAP = MAP0 | MAP1 .

MAP0 = PHRASAL-SIGN
       [syn:CATEGORY[subcat:#16],
        dtrs:TREE
             [head-dtr:LEXICAL-SIGN [syn:CATEGORY [subcat:#16]],
              comp-dtrs:<>],
        patch1:[whole:<>]] .
```

```
MAP1 = PHRASAL-SIGN
       [patch:MAP
              [syn:CATEGORY[subcat:#17],
               dtrs:TREE
                    [head-dtr:LEXICAL-SIGN[syn:CATEGORY[subcat:#18]],
                     comp-dtrs:#20],
               patch1:[whole:#22]],
        syn:CATEGORY[subcat:#17],
        dtrs:TREE
             [head-dtr:LEXICAL-SIGN[syn:CATEGORY[subcat:<#19=[phon:#21] ; #18>]],
              comp-dtrs:<#19 ; #20>],
        patch1: APPEND[front:#21, back:#22]] .

PRONOUN =  _ Pronouns define fully saturated simple NPs
          MAJOR
          [relation:#23,
           dtrs:TREE
                [head-dtr:MINOR
                          [syn:CATEGORY
                               [head:[lexem:#23],
                                subcat:<>]]]] .

NP =  _ abstracted NP head will be filled with relation slot,
      _ determiner with SPEC slot
     MAJOR
     [relation:#24,
      dtrs:TREE
           [head-dtr:MAJOR
                     [dtrs:TREE
                           [head-dtr:MINOR
                                     [syn:CATEGORY[head:[lexem:#24]]],
                            comp-dtrs:<>]],
            comp-dtrs:<MINOR
                       [syn:CATEGORY[head:[lexem:#25]]]>],
      spec:#25] .

E-PROP = E-VP-DITRANS | E-VP-TRANS | E-VP-INTRANS.

E-VP-DITRANS = _ partial tree for ditransitive vp
               _ with relation, recipient, and object.
              MAJOR
              [relation:#26,
               recipient:#27,
               object:#28,
               dtrs:TREE
                    [head-dtr:#26,
                     comp-dtrs: _ XP (Obj2)
                                <#27 #28>]] .
```

```
CLAUSE = MAJOR
         [relation:#29,
          agent:#32,
          recipient:#30,
          object:#31,
          dtrs:TREE
               [head-dtr: _ VP (Head)
                          VP-DITRANS
                          [relation:#29,
                           recipient:#30,
                           object:#31],
                comp-dtrs:<#32>]] .

E-DECLARE-MODALITY = MAJOR
                     [relation:#33,
                      agent:#35,
                      object:#34,
                      dtrs:TREE
                           [head-dtr: _ VP (Head)
                                      MAJOR
                                      [dtrs:TREE
                                            [head-dtr:#33,
                                             comp-dtrs: _ VP (Comp)
                                                        <#34>]],
                            comp-dtrs: _ XP (Subj)
                                       <#35>]] .

E-ASK-MODALITY = MAJOR
                 [relation:#36,
                  agent:#37,
                  object:#38,
                  dtrs:TREE
                       [head-dtr:#36,
                        comp-dtrs: _ XP (Subj)
                                   <#37 #38>]] .
```

7. The English lexicon

```
DET = a | the .

E-SPEAKER = PRONOUN[relation:"I"] .

E-HEARER = PRONOUN[relation:"you"] .

E-REGISTRATION-FORM = NP[relation:"registration form", spec:DET] .

E-SEND = MINOR[syn:CATEGORY[head:[lexem:"send"]]] .

E-CAN = MINOR[syn:CATEGORY[head:[lexem:"could"]]] .
```

Appendix 2 : PROLOG and RETIF

We give here the definitions for append written in PROLOG and in RETIF. In PROLOG, argument terms are identified by position; in RETIF they are identified by name. Apart from syntactical differences one should note the similarity of the definitions. The similarity is not restricted to syntax, and the computational behavior too is very similar in this case. In particular, for a given input, the set of solution is the same (modulo the difference of data structures). However, it is possible in RETIF to enforce type checking very naturally, but this is impossible in PROLOG without adding cumbersome patches.

APPEND in PROLOG

```
append([], W, W).
append([X|Y], Z, [X|U]) :- append(Y, Z, U).


?- append(X, Y, [a,b]).

X = [a,b]
Y = []

X = [a]
Y = [b]

X = []
Y = [a, b]
```

APPEND in RETIF

```
APPEND = [front:<>, whole:#W=LIST, back:#W]
       | [front:<#X ; #Y>,
          back:#Z=LIST,
          whole:<#X ; #U>,
          patch:APPEND[front:#Y, back:#Z, whole:#U]].

QUERY = APPEND[whole: <a b>].


(kbl-eval 'query)

[front:<>,
 back:#1=<a b>,
 whole:#1]

[front:#1=<a>,
 back:#2=<b>,
 whole:#3=<#1 ; #2>,
 patch:[front:<>, back:#2, whole:#2]]

[front:<#1=a ; #2=<#3=b>>,
 back:#4=<>,
 whole:<#1 ; #6=<#3>>,
 patch:[patch:[front:<>, back:#4, whole:#4],
        front:#2,
        back:#4,
        whole:#6]]
```