TR-I-0045

Operations on Typed Feature Structures: Motivations and Definitions

タイプ付き素性構造に対する操作: 動機および諸定義

Rémi Zajac
レミ・ザジャック

October, 1988

Abstract

In the spirit of functional programming (à la LISP), we propose a systematic framework for introducing a variety of extensions to unification-based grammar formalisms. Feature-structures are interpreted as objects which have a set structure, and not as denotation of sets. We introduce associated functions derived from set theory. For example, union is defined as a weak form of unification. We have developed an experimental interpreter to test these ideas. The basic data structure of the language is a Typed Feature Structure, as the list is the basic data structure of LISP.

This report presents the motivations of this approach, and describes the formal framework used for the language, which integrate functions and equations on Typed Feature Structures. The interpreter of the language and the grammar formalism will described in a forthcoming report.

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

Operations on Typed Feature Structures : Motivations and Definitions

Rémi Zajac*

ATR Interpreting Telephony Research Laboratories
Twin 21, MID Tower
2-1-61 Shiromi, Higashi-ku, Osaka, 540 Japan

[zajac%atr-ln.atr.junet@uunet.uu.net]

Abstract: A Machine Interpretation system will presumably use various specialized linguistic programming languages. At some point, there will probably be a transition from an attributed tree to a feature structure, and vice versa. A common point between attribute structures and feature structures is that attribute structures can be considered degenerated feature structures.

On the other hand, most grammar formalisms using unification of feature structures introduce a variety of augmentations that hamper the declarativeness of the formalism and monotonicity of computation of these equational formalisms.

In the spirit of functional programming (à la LISP), we propose a framework for such augmentations based on the interpretation of feature structures as objects (not as denotations). We introduce the associated functions derived from set theory. For example, union is defined as a weak form of unification.

We have developed an experimental functional interpreter to test these ideas. The basic data structure of the language is a typed feature structure. Operations avaible include conditional instructions and sequence of instructions, boolean expressions, assignment, unification, and various functional operations on typed feature structures. Definitions include type definitions, template definition and (recursive) function definitions. This interpreter has been integrated in an Earley parser. The grammar formalism offers the grammar writer enhanced expressive power and allows a more modular approach to grammar development.

This report gives the motivations of this approach, and describes the formal framework used for the language. The interpreter of the language and the grammar formalism will be described in a forthcoming report.

Key Words: Equational formalisms, unification, functionnal formalisms, feature structures, attribute structures, grammar formalisms.

* Visiting researcher from GETA, UJF-CNRS, 38041 Grenoble, cedex 53X, France.

TABLE

1. Introduction

1.1 Motivations
1.2 Equational and functional operations
1.4 Feature structures as hierarchical sets of attributes

2. Operations

2.1 Preliminaries
2.2 Equivalence and equality
2.3 Subsumption and inclusion orderings
2.4 GLBs : conjunction and union
2.5 LUBs : generalisation and intersection
2.6 Difference

3. Conclusion

Appendix: recursive definitions of operations on h-sets

References

1. INTRODUCTION

1.1 Motivations

A. Integration of different computational models for computational linguistics

As argued in [Boitet 88], a Machine Interpretation system is likely to use different Specialized Languages for Linguistic Programming (SLLP), and may be more than text Machine Translation systems. For analysis, we have to differentiate between at least four layers of input representation. Each layer uses a different representation which has its own computationnal model.

layer	*computational model*	*Input*	*ouput*
UNDERSTANDING	logic formalism	syntactic structures	semantic formula
SYNTAX	CF grammar	lemmatized words	syntactic structures
MORPHOLOGY	Finite State Automata	word form	lemmatized word
PHONOLOGY	Hidden Markov Model	acoustic frames	word forms

Figure 1 : layers of representation of a spoken utterance

If we make the (reasonable) assumption that each task is programmed using the fitted formalism, we have at least three different formalisms (not counting the speech recognition part). Morphology performs the segmentation of word forms and accesses the dictionnary: the basic formalism would be an (augmented) regular grammar. The syntactic part performs the recognition of syntagms and computes associated syntactic functions: the associated formalism would be an attributed context-free grammar, possibly augmented with tree transformations. The understanding part computes a «semantic interpretation» of the input, that is, a mapping from a syntactic structure to a formula of a semi-formalized domain. This domain would include (not exhaustively) linguistic semantics, pragmatics and domain knowledge. The associated formalism would be a logic programming language based on feature structures.

There are already formalisms used to compute each layer from the others, but there are none to compute the mapping of an attributed tree to a feature structure. Such a task is necessary at the end of a (successful) analysis, for transfer and maybe for generation steps. We see two possibilities for such a computation. One is to use a rewriting system that creates a feature structure from an attributed tree (and conversely for the generation part). Another is to imbed the computation of feature structures in the context-free grammar formalism, and at some point to treat the attribute structure as a (degenerated) feature structure. In any case, we have to relate attributes and feature structures.

Attribute systems are widely used in augmented context free grammars, particularly for writing compilers, but also in MT systems (e.g., METAL [Slocum 84, Bennet and Slocum 85]). They are also used in SLLPs designed to write the grammars of a MT system: ROBRA, GRADE, Q-systems, etc. [Boitet et al. 80, Nakamura et al. 84, Colmerauer 71]. Attributes are generaly associated to nodes of a tree structure. In the simplest case, an attribute structure is a set of variable-value pairs. A variable has one value and is possibly typed (string, integer,...). A slight extension allows a variable to have a set of (symbolic) values. The type of each variable is generally declared, in order to check consistency during compilation, and to allow efficient internal representations (ROBRA, GRADE). Another extension is to use a tree of attributes (in Q-systems, for example), where an attribute CATEGORY may have a ADJUNCT value which, in turn, has an ADJECTIVE value. If complex attribute structures are generaly depicted as trees, they might also be depicted as graphs with symbols on arcs, as shown in Figure 2.

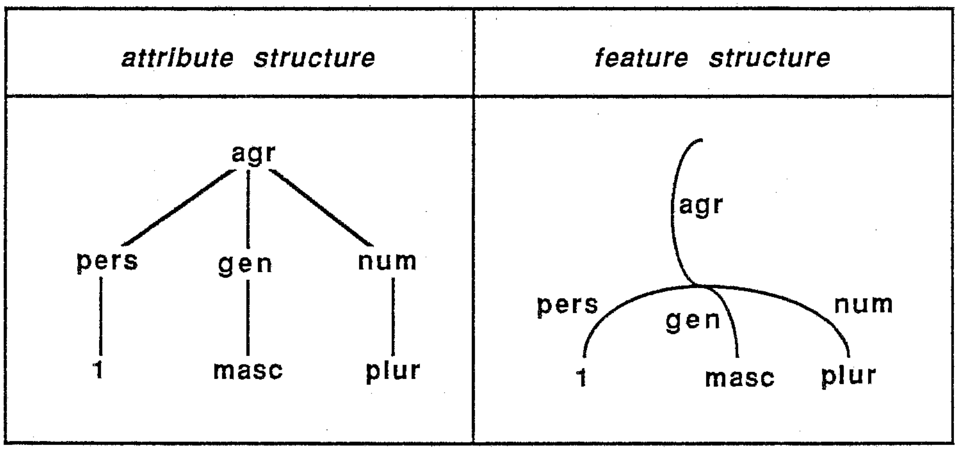


Figure 2 : an attribute structure and an equivalent feature structure.

Putting this analogy forward, we want to see how operations on attribute structures, e.g. inclusion (-DANS- predicate of Q-systems) or union of two attribute structures could be related to operations on feature structures (e.g. subsumption and unification).

B . Toward a practical formalism for building large systems

If we consider feature structures to be the basic data structure for computational linguistic, there are various operations that could be used with this data structure in different frameworks :

- unification in a logic programming language, LOGIN [Aït-Kaci and Nasr 86];

- function application in a functional programming language, KL-ONE [Brachman and Schmolze 85].

So far, unification is the only operation used in formalisms for computational linguistics. We argue that the use of a functional paradigm imbedding descriptions of typed feature structures will offer more expressive power to the user, making descriptions more concise. Futhermore, a grammar formalism adapted to use this functional framework would enhance grammar writing, grammar modifying, and grammar debugging. And last but not least, a more concise grammar (with fewer rules) will run faster than an equivalent grammar written in a pure unification formalism.

Our functional framework offers:

conditional constructions: they can be used to factor in one rule slightly different cases that, using unification only, must be expressed in several rules having the same production part. This makes descriptions more concise as the same part is only written once. Modifications are also easier, because they are done only at one place, not repeated with all similar rules. And as there is only one rule which have a production $A \rightarrow B\ C$, it is easier to point out an error in debugging.

user defined functions: the grammar writer can define his own functions as a composition of other functions (recursive definitions are allowed). For example, for expressing some agreement condition that is frequently used, instead of repeating the same code all over the grammar, one need only define this condition as a function and call the function where the agreement must hold. The gain in concision and ease of debugging is clear.

string and number types: these types are treated as atomic types. The usual functions associated with these types are defined.

equational operations: unification and generalisation are implemented; complement and disjunction are planned.

1.2 Equational and functional operations

Grammar formalisms based on unification of feature structures (hereafter f-structures which are, most generally directed acyclic graphs) are equational formalisms. In the following, we shall take as a representative example, the D-PATR formalism [Shieber 86]). A so-called «unification grammar» is an augmented context-free grammar. A f-structure is associated with each symbol of a rule. The output of a parser is the f-structure associated with the initial symbol of the grammar (one for each successful parse) for a given input string. To define the f-structure that corresponds with a given input string, the linguist writes equalities between sub-parts of the f-structures associated with each symbol of a rule; the order in which the equations are written is not important. During computation, for a given parse, grammar rules add these constraints on the f-structures which are computed. As the equalities are added, they are immediatly interpreted using unification. At the end of the process (recognition of the initial symbol of the grammar), the smallest f-structure that verifies all the accumulated constraints is output.

At first, this kind of grammar formalism was used by theoretical linguists to formalize such-and-such linguistic/semantic phenomena. But these formalisms were not used to implement real size parsers. As far as we know, there is no such parser for English. On the other hand, actual real size parsers based on context-free grammars (such as the one used in the EPISTLE system developed at IBM [Heidorn et al. 82, Jensen and Heidorn 83]) use generally simple attributes, with possibly attached transformations (as in the METAL system [Slocum 84]).

The subject of this paper is to investigate to what extend the basic qualities of both kinds of formalisms can be retained in a single grammar formalism. One possible way of doing this is to devise a single framework that could account for f-structures *and* for hierarchical attributes of the kind used in practical systems such as Q-systems [Colmerauer 71]. We hope this might possibly lead to a formalism that could be used for developing both practical systems and more speculative formal models.

Let us take an example (from [Shieber 86]) and consider the grammar rule in Figure 3. The application of this rule on a string of two f-structures (Figure 4) builds a new f-structure (Figure 5) by «merging» the two previous ones and adding some new elements to it; the two previous f-structures are not recoverable. Outside the scope of the rule, the f-structure cannot be accessed using NP and VP «entry points», thus some parts of this f-structure are «hidden».

```
S --> NP   VP

    <S cat> = s
    <NP cat> = np
    <VP cat> = vp
    <S head> = <VP head>
    <S head subject> = <NP head>
```

Figure 3 : a grammar rule using unification.

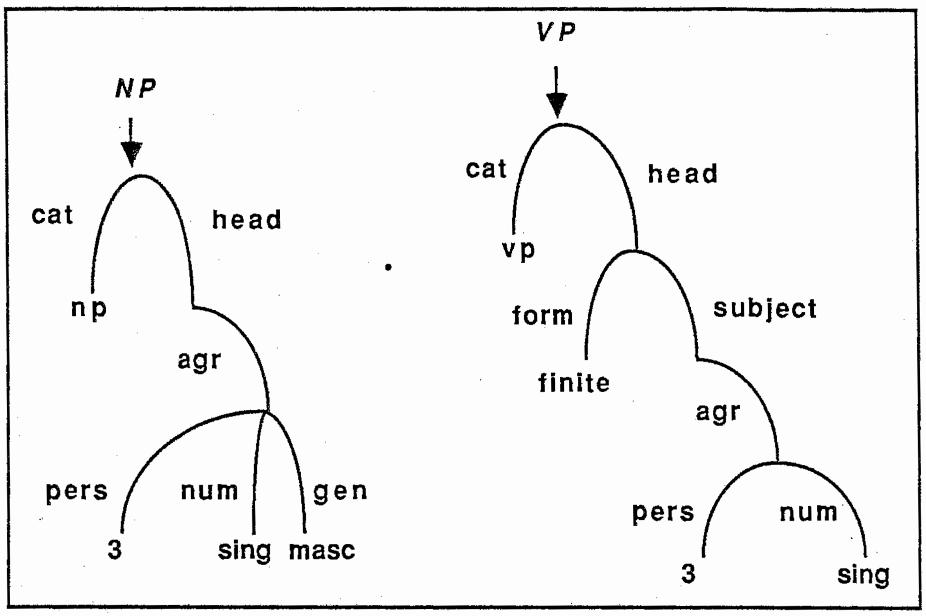


Figure 4 : a string of two feature structures.

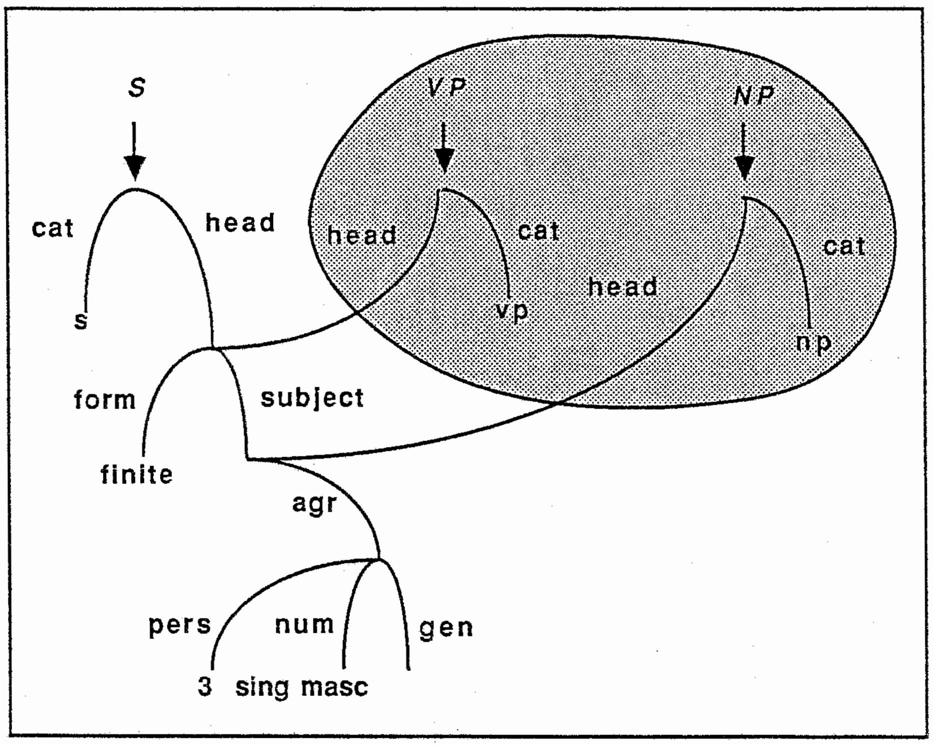


Figure 5 : the result of an application of a rule using unification on f-structures.

The same rule can be written with an attribute grammar using conditional assignment and boolean functions. Let us assume that the attribute system of the grammar is hierarchical and can be represented as an f-structure without sharing. The rule could then be written as in Figure 6. Applied on the same string of f-structures (Figure 4), it builds a new attribute structure without any explicit link with the two structures associated with NP and VP. The *if* part checks if the rule can be applied using boolean operators (AND) and boolean functions (=, $agr). The third line of the *then* part is an assignment of the *union* of the values of <S head subject> and < NP head> to <S head subject>. Note that we have a *sequence* of operations. Thus, the order in which the operations are written is important. The application of the rule builds a completly new attribute structure (Figure 7), with no (explicit) links to the others. As the structures of NP and VP could be withdrawn, the resulting attribute structure is smaller than the f-structure of Figure 5 (no «hidden» parts).

```
S –> NP  VP
if (<NP cat>=np and <VP cat>=vp and $agr(<NP>, <VP>))
then
   <S cat> := s ;
    <S head> := <VP head> ;
     <S head subject> := <S head subject> + <NP head>
endif;
```

Figure 6 : a grammar rule using conditional assignment.

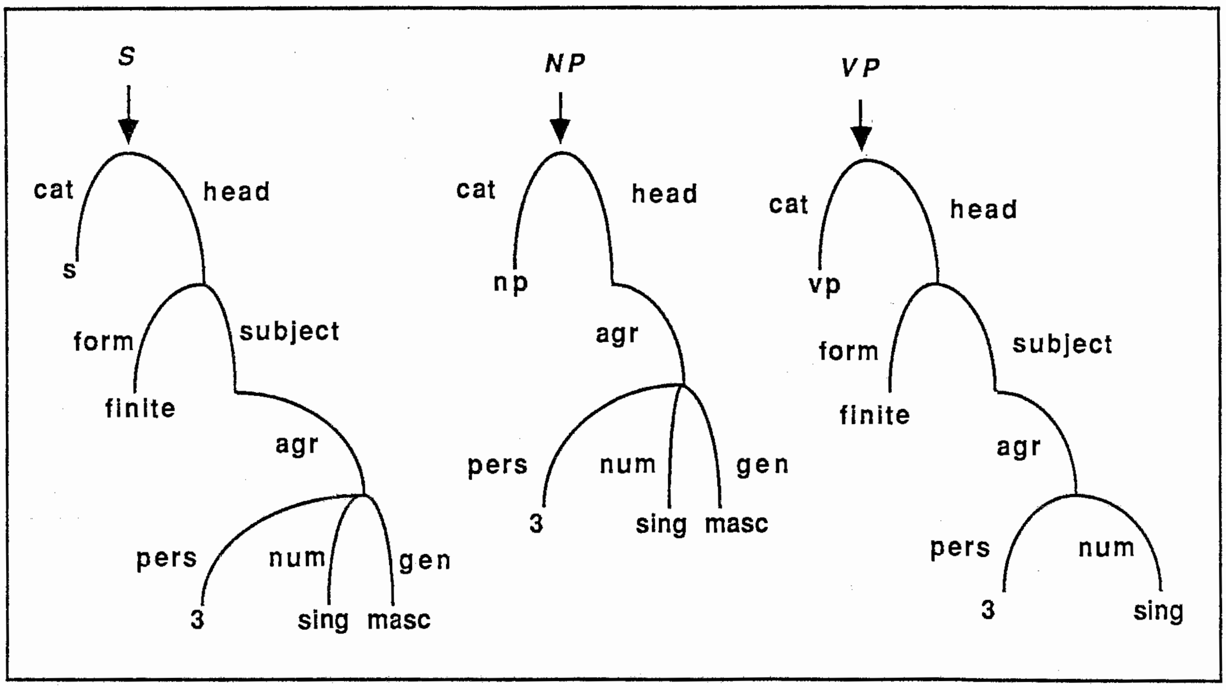


Figure 7 : the result of an application of a rule using conditional assignment in attribute structures.

We have two different computational frameworks that use the same kind of data structure. In the unification framework, the computation is expressed by a set of equations attached to each rule; in the second framework, the computation is expressed by a sequence of conditional assignments. The right hand side of an assignment is a functional expression that uses a variety of operations on f-structures. We can draw an analogy with PROLOG computational behavior for unification based-formalisms, and with LISP for the second one. The spirit of the integration of those two frameworks –equational and functional– is quite similar to the integration of LISP and PROLOG [Robinson and Sibert 82].

1.3 Feature structures as hierachical sets of attributes

We want to view attribute structures as a special case of feature structures. As attribute structures are always interpreted as objects and not as denotations of sets, we also have to interpret feature structures as objects standing for themselves. On the contrary, the logical approach of feature structures leads to interpreting them as denotation of sets, and not as sets themselves. In the following, we shall speak of feature structures with the object interpretation in mind, that is, feature structures as hierarchical sets of attributes.

The main differences with the denotative interpretation are:

1. The ordering used in the denotative interpretation (subsumption) is the reverse of the ordering on hierarchical sets of attributes (inclusion);

2. As feature structures are viewed as objects standing for themselves (similar to record data structures in programming languages), one can directly manipulate these objects, without worrying about the semantics of these manipulation on the denotated sets.

We use the interpretation of an f-structure as a *hierarchical set of attributes* (hereafter h-sets). A path is interpreted as the name of a set, the elements of this set being the values of the path. This interpretation is valid at all levels down to the bottom of the f-structure where atomic features are interpreted as features whose sets of values are bound to be empty. This interpretation is made possible by the fact that an f-structure is non ordered (but directed) and without repetition (for the successors of any given feature). Let's take an exemple (as adequate meaningful linguistic examples are very difficult to build for all interesting cases, we shall mainly use formal examples).

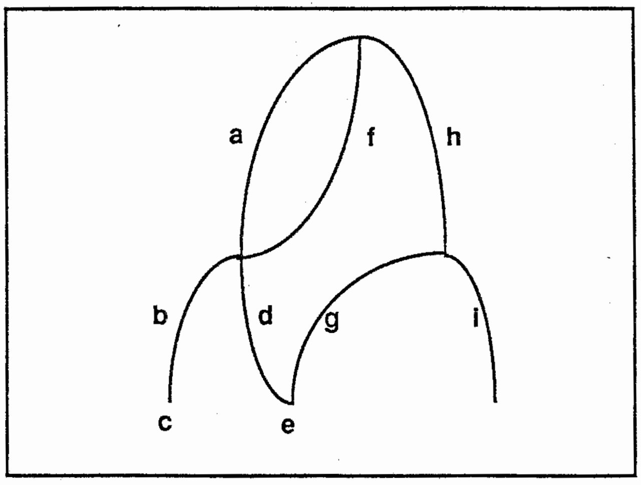


Figure 8: a feature structure.

Interpreting the graph of Figure 8 as a hierarchical set of attributes level by level gives

```
1  {a, f, h} ;
2  a={b, d} ;   f={b, d} ;   h={g, i} ;
3  b={c} ;   d={e} ;   g={e} ;   i = {} = Ø ;
4  c={} ;   e={} .
```

The whole structure is written `{a{b{c}, d{e}}, f{b{c}, d{e}}, h{g{e}, i}}`. Recall that an attribute structure does not involve sharing, thus it is not represented here.

In order to be able to introduce other types than set and atomic types, we define explicitly the type of features: in each node of the graph, we put the name of the type and each value on an arc. An atomic value will have the null (bearing the null type) as a successor. We add in each node an index which is used to represent sharing, and we shall speak of co-indexing of paths and of sub-structures. In the following, such a structure will be called a Typed Feature Structure or *tfs*.

In the traditionnal view of attribute structures as trees, (see for exemple [Colmerauer 71, Boitet, Bachut et al. 88]), types, indexes, *and values* are put in nodes. Figure 10 gives the dual view of the tfs as a tree-like attribute structure.

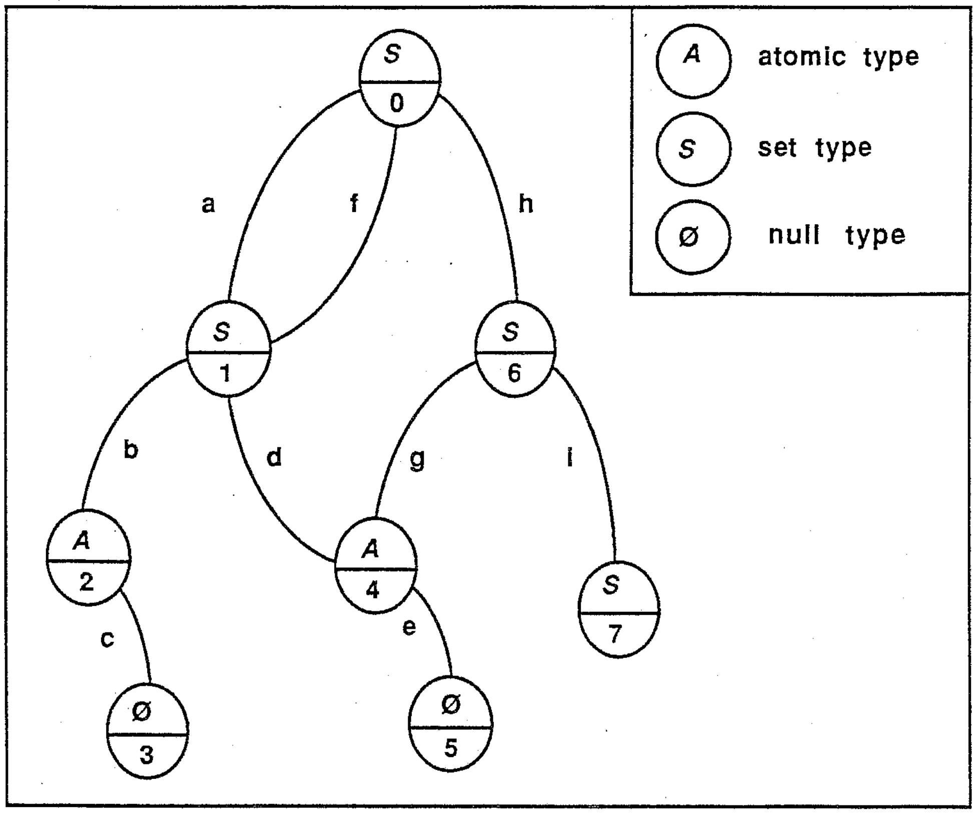


Figure 9 : an f-structure with representation of types.

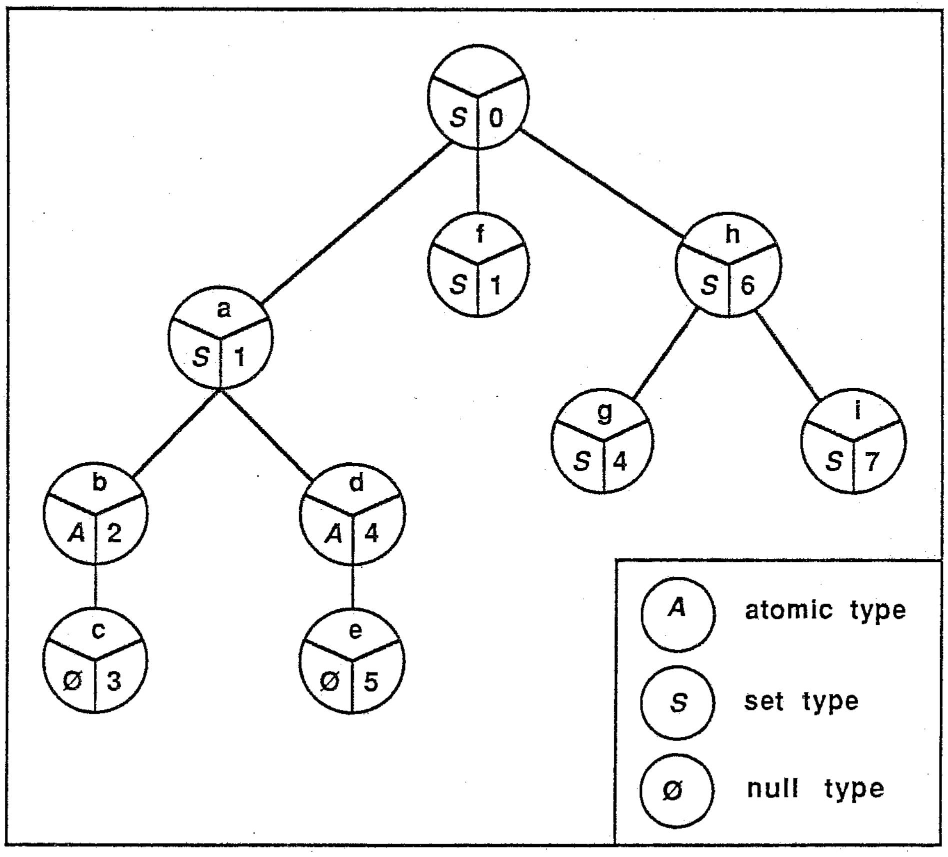


Figure 10 : an f-structure with values on nodes.

If we write an h-set as `{a{b{c}, d{e}}, f{b{c}, d{e}}, h{g{e}, i}}`, the union operation used in the rule in Figure 6 has a very simple recursive definition. Let us first define the ordinary union of two sets of symbols. In the following definitions, e*x* will represent a feature, l*x* a list of features, and f*x* a feature-value structure (either atomic or complex). Parentheses have their usual meaning. When a set is written between angle brackets, the elements of the set are treated as a list of elements: <{l*x*}> ⟶ l*x*.

Definition 1: union of sets

```
1  {} + {l2} ⟶ {l2}
2  {l1} + {} ⟶ {l1}
3  {e1, l1} + {e1} ⟶ {e1, l1}
4  {e1, l1} + {e2} ⟶ {e1, <{l1}+{e2}> }
5  {l1} + {e2, l2} ⟶ ( {l1} + {e2} ) + {l2}
```

The usual properties of ordinary sets for union (idempotency, commutativity, associativity and null element) follow directly from this definition. For dealing with hierarchies of sets, we modify this definition slightly to get:

Definition 2: union of h-sets

```
1     {} + {l2} ⟶ {l2}
2     {l1} + {} ⟶ {l1}
3     {e1{l1}, l2} + {e1{l3}} ⟶ { e1{<{l1}+{l3}>} } + {l2}
4     {f1, l1} + {f2} ⟶ {f1, <{l1}+{f2}> }
5     {l1} + {f2, l2} ⟶ ( {l1} + {f2} ) + {l2}
```

From the exemples of rules given in section 1.2 (figures 1 and 4), one can guess that the algebraic operation involved in unification, that is, conjunction [Rounds and Kasper 86] or join [Aït-Kaci 86] bears some similarities with union of two h-sets. This is the fact which is stressed in this paper and we shall see that the main difference lies, as one can again guess, in the treatement of sharing. To account for this, we have to give a more precise definition of a typed feature structure. As the main goal of this paper is not to provide a comprehensive theoretical account but rather to introduce some new practical operations, we ask the reader to refer to [Aït-Kaci 86] for a sound mathematical account. We follow the main points of his presentation.

2. OPERATIONS

2.1 Preliminaries

The example in Figure 9 is written `{a{b{c},d{e}},f{b{c},d{e}},h{g{e},i}}` as an h-set. Representing sharing is done with indexes, and co-indexed sub-structures are represented only once; we shall also omit indexes that occur only once in the structure: `{a.1{b{c},d.4{e}},f.1,h{g.4,i}}`. The complete developed representation has three components: the set of paths, associated indexes, and associated types:

Paths	Indexes	Types
ε	0	S
a	1	S
ab	2	A
abc	3	Ø
ad	4	A
ade	5	Ø
f	1	S
fb	2	S
fbc	3	A
fd	4	S
fde	5	A
h	6	S
hg	4	A
hge	5	Ø
hi	7	S

Figure 11 : a developed representation of a TFS.

Let us now introduce some formal notations before defining some operations on TFS.

The graph skeleton: TFS-graph

Let **L** be a set of label symbols, and **I** a set of indexes included in **N**, the set of natural integers. A set of paths **P** is a sub-set of $\mathbf{L}^*$ and has the following connexity property:

(p1) $\forall$ x, y $\in$ $\mathbf{L}^*$, xy $\in$ **P** $\Rightarrow$ x $\in$ **P**.

A TFS is a finite data structure: a node of a graph has a finite number of successors and the number of sub-graphs is finite. A node has a finite number of successors:

(p2) $\forall$ x $\in$ **P** {xa $\in$ **P** / a $\in$ **L**} is finite.

A TFS may contain cycles, and therefore the set of paths of the graph is infinite but regular. It means that the number of sub-graphs is finite. A sub-TFS is $\mathbf{P}\backslash x=\{y \,/\, xy \in \mathbf{P}\}$. A regular graph $\mathbf{P}$ has the property

(p3) $\forall x \in \mathbf{P}, \{\mathbf{P}\backslash x\}$ is finite.

and we shall call a *TFS-graph* a connex regular graph.

The co-indexing relation

If two nodes have the same index, it means that their successors are identical (and not only equal). Practically, it means that there are two ways (paths) for accessing the sub-structure dominated by the two nodes. This could be used to achieve side-effects, for example to enforce constraints linking sub-parts of TFSs using unification. A co-reference relation defines an equivalence $\equiv$ relation between paths: $x.i \equiv y.j \Leftrightarrow i=j$. This relation has the following right-invariance property:

(p4) $\forall x, y, z \in \mathbf{P} : x \equiv y \Leftrightarrow xz \equiv yz.$

As each TFS defines a particular equivalence relation between paths, the equivalence relation in a TFS x will be written $\equiv_x$ (and its set of paths, P_x). The number of nodes in a TFS is finite, as are the number of indexes: the equivalence relation is said to be of finite index. The equivalence relation does not impose a particular indexing, and to avoid re-indexing on graphs, we shall only compare the relations defined by an indexing and not the indexing itself. We shall call such a relation, a *co-indexing relation.*

Types

In the graph, each path is typed. Let $\mathbf{T}$ be a partially ordered signature of type symbols with a top element $\top$, and bottom element $\bot$. In our formalism, without type refinement, $\mathbf{T}$ is a flat signature where the ordering is $\top<$ SET$<\bot$, $\top<$ ATOMIC$<\bot$, and $\top<$ NULL$<\bot$. It could be pictured as

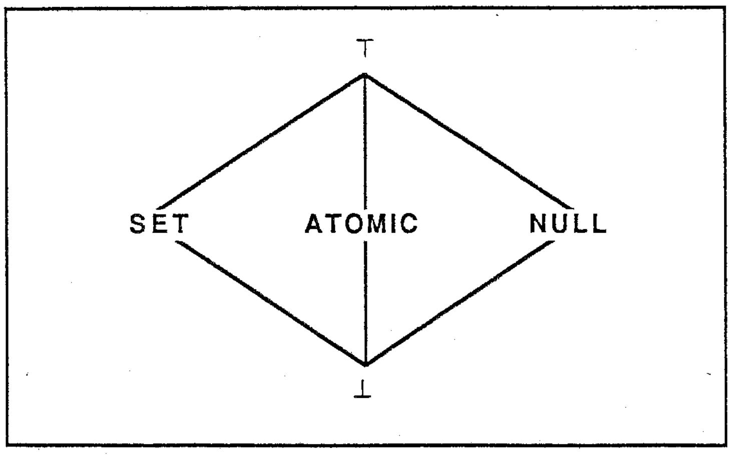


Figure 12: lattice of types.

Thus, the type calculus which is involved is very simple. The motivation is that we have been primarily interested in combining the attribute approach and the feature structure approach, and not in the logic of feature terms: we needed only a mean of defining domains and sub-domains for feature structures: each value of a feature must belong to the domain of the feature. For example, in the prototype implementation of the interpreter, string and number types are introduced as refinements of atomic type. As the definition of domains is purely syntactical, it is not described here. It does not seem too difficult to add type inheritance to this approach.

We define the type associated with each path of a graph $\mathbf{P}_X$ as a (total) *type function* τ_X from $\mathbf{L}^*$ to $\mathbf{T}$ where $\forall\, w \in \mathbf{L}^* - \mathbf{P}, \tau_X(w) = \top$.

Typed Feature Structure

A TFS x is then a triple $<\mathbf{P}_X, \equiv_X, \tau_X>$ where $\mathbf{P}_X$ is TFS-graph labelled on $\mathbf{L}^*$, $\equiv_X$ is a co-indexing relation on $\mathbf{P}_X$, and τ_X is a type function from $\mathbf{L}^*$ to $\mathbf{T}$.

Well-Formed Typed Feature Structure

A well-formed TFS is a TFS where the properties of each type are respected: a sub-TFS having a set type can be anything, a sub-TFS having an atomic type is reduced to a node at most and a sub-TFS having the null type is null. We have only to state the last two properties:

(p5) 1) $\forall$ x $\in$ **P** / τ(x)=A, $\forall$ y $\in$ **P** / x $\equiv$ y, $\forall$ a $\in$ **L**, there exists at most one ya $\in$ **P** and τ(ya)=$\emptyset$.

2) $\forall$ x $\in$ **P** / τ(x)=$\emptyset$, $\forall$ a $\in$ **L**, xa $\notin$ **P**.

Consistent Typed Feature Structure

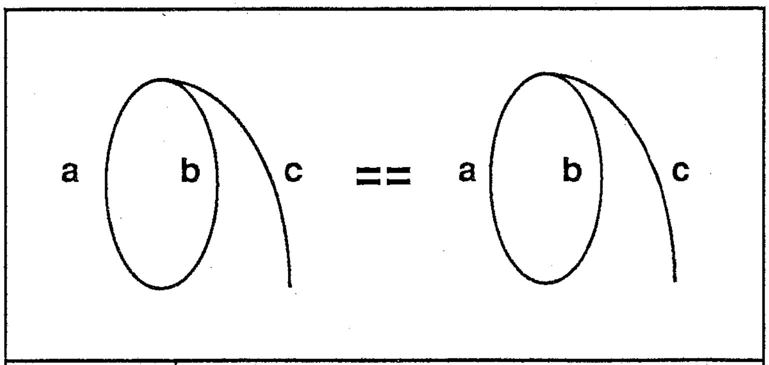
The type $\top$ denotes the whole universe, and is associated with the *undefined* value. The type $\perp$ denotes overdefined (i.e. inconsistent) information and its associated value is by convention the empty set. A TFS that containts the $\perp$ type or does not respect (p5) should then be intrepreted itself as inconsistent. A consistent TFS is a well-formed TFS where no $\perp$ type appears and if $\perp$ does occur in a TFS or if (p5) does not hold, then the TFS belongs to the class of $\perp$. This is achieved by defining a relation $\Downarrow$, called bottom-smashing, where t1 $\Downarrow$ t2 if and only if $\perp$ occurs in t1 or t1 does not have (p5), and $\perp$ occurs in t2 or t2 does not have (p5), and all equivalence classes are singletons except [$\perp$]. We shall now use TFS in place of «the quotient set of well-formed TFSs modulo the bottom-smashing relation».

2.2 TFS equivalence and TFS equality

As attribute structures and feature structures can be seen as a dual representation of each other, in the traditional attribute framework, there is no notion of co-indexing, or «sharing» of values. This is the main difference of feature structures. In that sense, feature structures can encode more information: not only the information represented as a hierarchy of features and values but also the information that two features share the same set of values. If two features, say f1 and f2, are co-indexed their values are not only equal, but more: identical. Operations on attribute structures do not take identities into account and operations on feature structures do. As we work on only one kind of structure, typed feature structures, we can nevertheless define two kinds of operations: those taking identities into account (written with a doubling of the operator sign), and those ignoring identities.

Two TFSs are equivalent if they have the same structure, co-indexing, and types:

(d1) $S{==}T \Leftrightarrow P_S = P_T \;\wedge\; \equiv_S = \equiv_T \;\wedge\; \tau_S = \tau_T$

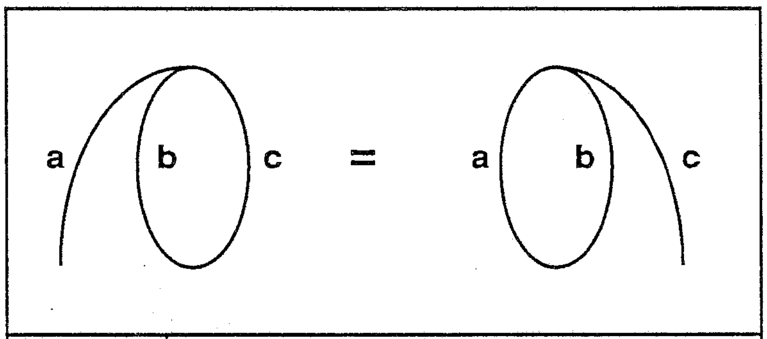


X==Y	• same features and values • same indentities

Figure 13: equivalence between tfs.

Two TFSs are equal if they have the same structure and the same types, but not necessarily the same co-indexing:

(d2) $S{=}T \Leftrightarrow P_S = P_T \;\wedge\; \tau_S = \tau_T$



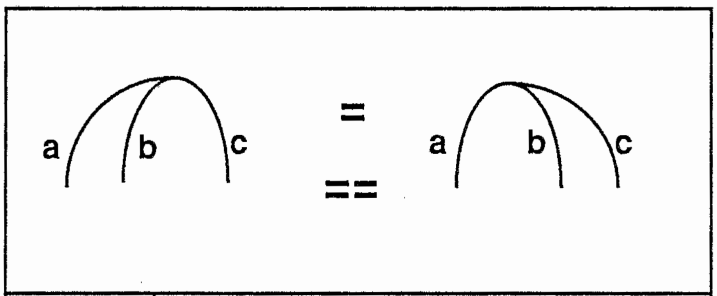
X=Y	• same features and values • not the same indentities

Figure 14: equality between tfs.

It is easy to see that

(p7) $== \;\supseteq\; =$.

As the difference lies in co-indexing, when there is no sharing, those two definitions are reduced to the definition of equality on hierarchical attribute structures:



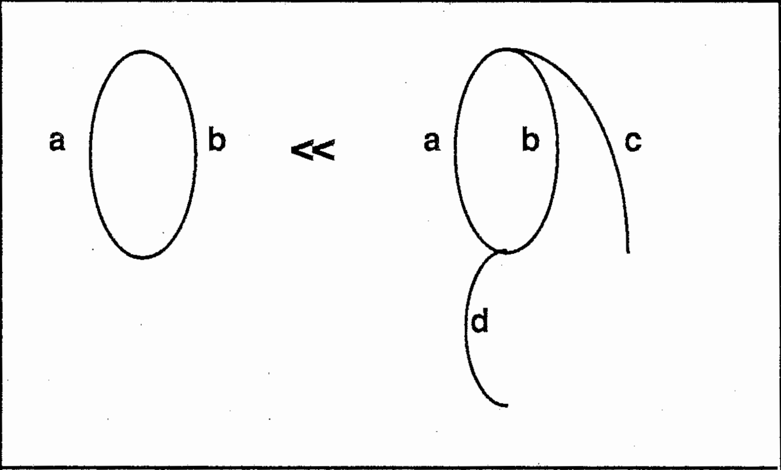
X=Y and X==Y	• same features and values • no identity

Figure 15: equality and equivalence hold.

2.3 Subsumption and inclusion orderings

A TFS T is *strongly included* in a TFS T if T contains *less* information than S:

(d3) $S>>T \Leftrightarrow P_S \supseteq P_T \wedge \equiv_S \supseteq \equiv_T \wedge \tau_T \supseteq \tau_S$



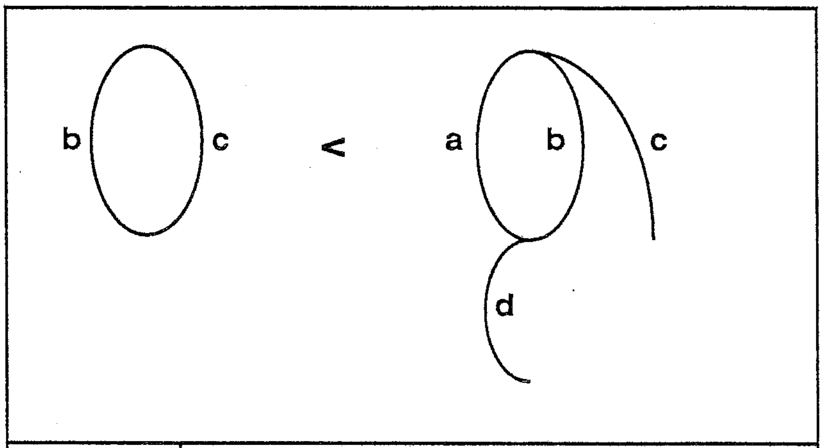
X << Y	• features and values of X are in Y • indentities of X are in Y

Figure 16: strong inclusion.

One should notice that strong inclusion is nothing more than the *reverse* of subsumption. In the same manner, we shall call weak subsumption the reverse of inclusion. In our approach, we consider that TFS stand as *objects rather than denotations*. This is coherent with the interpretation of attribute structures. One can also notice that the duality object/denotation is similar to the distinction made by [Kaplan and Bresnan 82] between feature structures and feature descriptions.

A TFS T is *included* in a TFS S if, except for the co-indexing, T contains less information than S:

(d4) $S > T \Leftrightarrow P_S \supseteq P_T \wedge \tau_S \supseteq \tau_T$

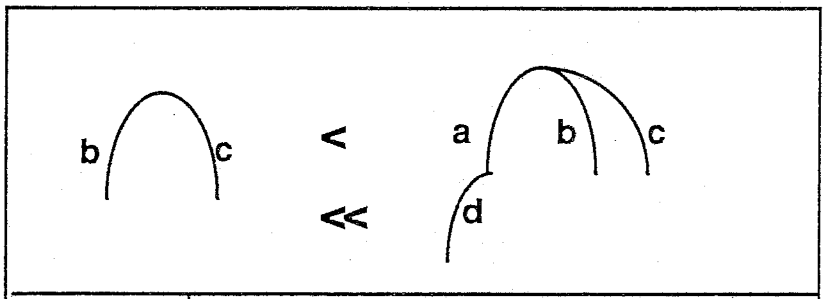


X < Y	• features and values of X are in Y • forget indentities

Figure 17: inclusion.

In this exemple, strong inclusion does not hold. And as for equivalence and equality, we have:

(p8) $>> \supseteq >$.



X < Y and X<<Y	• features and values of X are in Y • no indentities, only equalities

Figure 18: strong inclusion and inclusion hold.

The reverse of these orderings, that is subsumption and weak subsumption, which are extensions of the ordering on **T** to TFSs, can be used to define least upper bounds (lub) and greatest lower bounds (glb) on TFSs, which then have a lattice structure derived from the structure of **T**.

We now extend the parallelism between equivalence and equality to other operations.

2.4 GLBs : conjunction and union

We first introduce *union* of two f-structures. The union of two f-structures is the union of features at the first level *and,* recursively, the union of the values of two features having the same name. Let's take as an example the following f-structures:

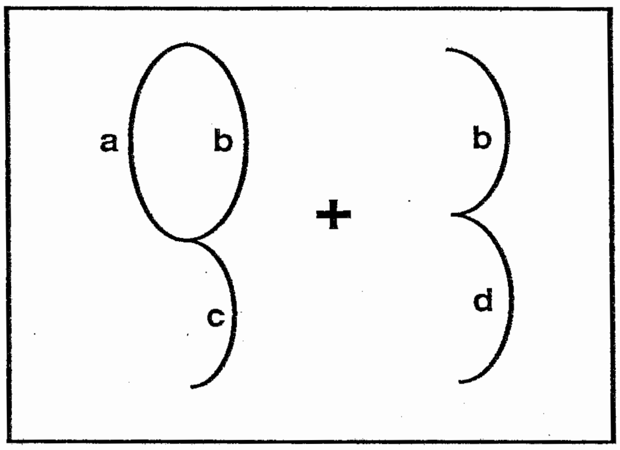


Figure 19: union of feature and values.

As for the *union of h-sets*, the union X + Y = Z level by level gives (empty values are not written)

```
1.    {a, b}
2.    a={c} ;  b={c, d}
```

and the whole resulting structure is:

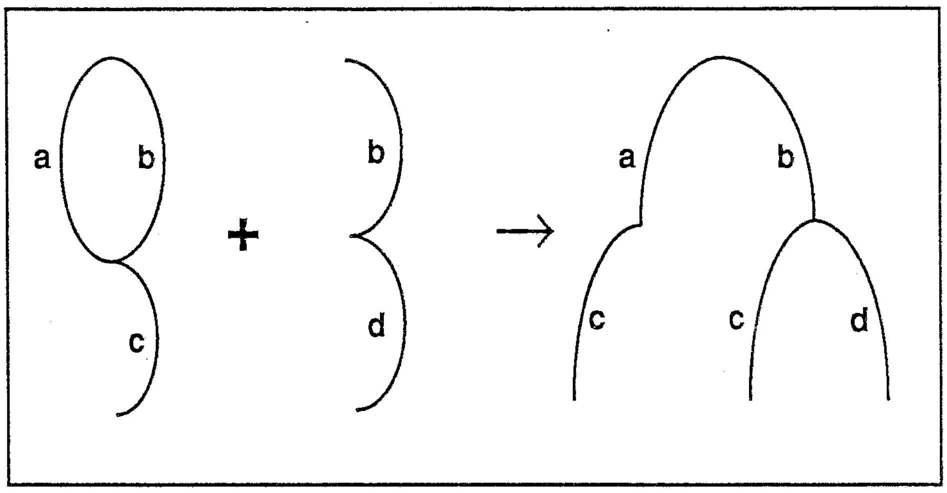


Figure 20: union of feature and values X+Y=Z: do not take identities into account.

and *not*

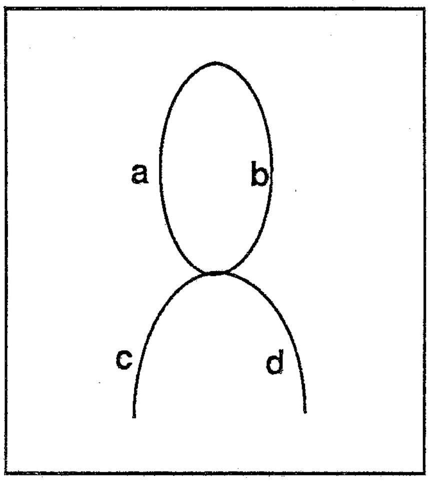


Figure 21: conjunction of feature structures X++Y=Z'

If we consistently use the h-set interpretation, <X a>={c} and there is no path <Y a>. Therefore, <Z a> must have the same value {c}. It would be quite an unexpected *side effect* for <Z a> to get a different value when there is no H-set of the same name in Y! In fact, the structure Z' is the *conjunction* of X and Y, where <X a> is bound to have the same value as <X b>, and all operations that affect one must also affect the other.

As we shall see, the same two interpretations will also appear for intersection. These two kinds of operations are perfectly legitimate. The first is quite natural, interpreting f-structures as h-sets of attributes with a local view restricted to two arcs (variable → value). The second is quite natural reasoning on f-structures as graphs, with a global view of the structure. A consequence is that, under the equational interpretation, a path is a means to access a part of the graph, that is, a path is a *pointer*. Under the functional interpretation, a path is a *variable* that returns a value and can be assigned a value. Of course, if there are no identities at all, the two operations are equivalent. This suggests introducing a run-time parameter for a grammar to take or not to take sharing into account.

If sharing is not taken into account, unification is reduced to a «pseudo-unification», similar to the pseudo-unification used in the default mode of the Universal Parser developed at Carnegie-Mellon's Machine Translation Center [Tomita 88].

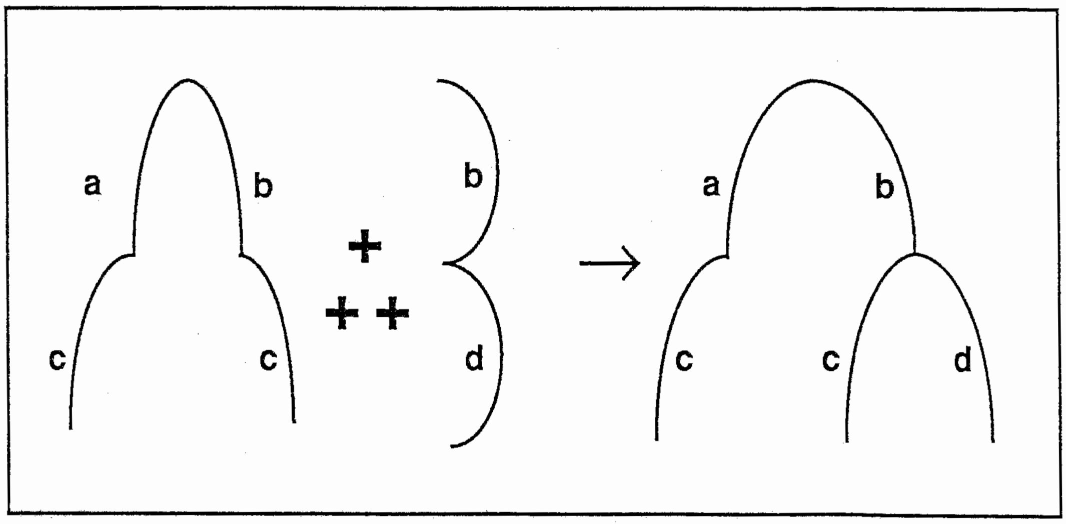


Figure 22: conjunction or union of feature structures without co-indexing.

We shall first define the greatest lower bound for the subsumption ordering, which we shall call *conjunction*, and write S++T=L. One should notice that, for strong inclusion ordering, conjunction will be a *least upper bound*: we prefer to use subsumption to make comparisons easier with other works. For S++T, we have to put in the set of paths all paths of S and T, and to take as the co-indexing relation, the transitive closure of the union of $\equiv_S$ and $\equiv_T$. This is not enough, and we have also to preserve the right-invariance property and add to any class of the partition of $\mathbf{P}_S \cup \mathbf{P}_T$, all paths xz such that yz belongs to the class and $x \equiv_L y$. All these paths are also added in $\mathbf{P}_L$. The limit of this construction is called the right-invariant transitive closure and written $^{[*]}$. The type of each path is the meet on **T** of the types of the paths of the class to which the path belongs.

(d5) $L = S{+}{+}T \Leftrightarrow$

1) $\mathbf{P}_L = (\mathbf{P}_S \cup \mathbf{P}_T)^{[*]}$;

2) $\equiv_L = (\equiv_S \cup \equiv_T)^{[*]}$;

3) $\forall x \in \mathbf{P}_L, \tau_L(x) = \wedge\{\tau_S(y), \tau_T(y) / x \equiv_L y\}$,
and the result is taken modulo bottom-smashing.

The glb for the weak subsumption ordering, which we shall call *union*, and write S+T=L, is a bit simpler, because we don't have to take the closure of the co-indexing relation:

(d6) $L = S+T \Leftrightarrow$

1) $\mathbf{P}_L = \mathbf{P}_S \cup \mathbf{P}_T$;

2) $\forall x, y \in \mathbf{P}_S \cap \mathbf{P}_T,\ x \equiv_S y \wedge x \equiv_T y \Leftrightarrow x \equiv_L y$,

$\forall x, y \in \mathbf{P}_S - \mathbf{P}_T,\ x \equiv_S y \Leftrightarrow x \equiv_L y$,

$\forall x, y \in \mathbf{P}_T - \mathbf{P}_S,\ x \equiv_T y \Leftrightarrow x \equiv_L y$;

3) $\forall x \in \mathbf{P}_L,\ \tau_L(x) = \bigwedge\{\tau_S(y), \tau_T(y) / x \equiv_L y\}$,

and again, the result is taken modulo bottom-smashing.

2.5 LUBs : generalisation and intersection

The lub for the subsumption ordering is called *generalisation*, and written S*T=U. We simply have to take the intersection for the sets of paths and for the co-indexing relations, and the join of the types for each path in the intersection.

(d7) $U = S*T \Leftrightarrow$

1) $\mathbf{P}_U = \mathbf{P}_S \cap \mathbf{P}_T$;

2) $\equiv_U = \equiv_S \cap \equiv_T$;

3) $\forall x \in \mathbf{P}_U,\ \tau_U(x) = \tau_S(x) \vee \tau_T(x)$;

As S and T are well-formed consistant TFSs, 1) and 2) preserve the well-formedness, and 3) preserves the consistency.

In the graphic view, the application of the definition is as follows: if an arc is not in both H-sets, remove it. Apply this operation from top to bottom. At the bottom, if an atomic element is not in both, remove it.

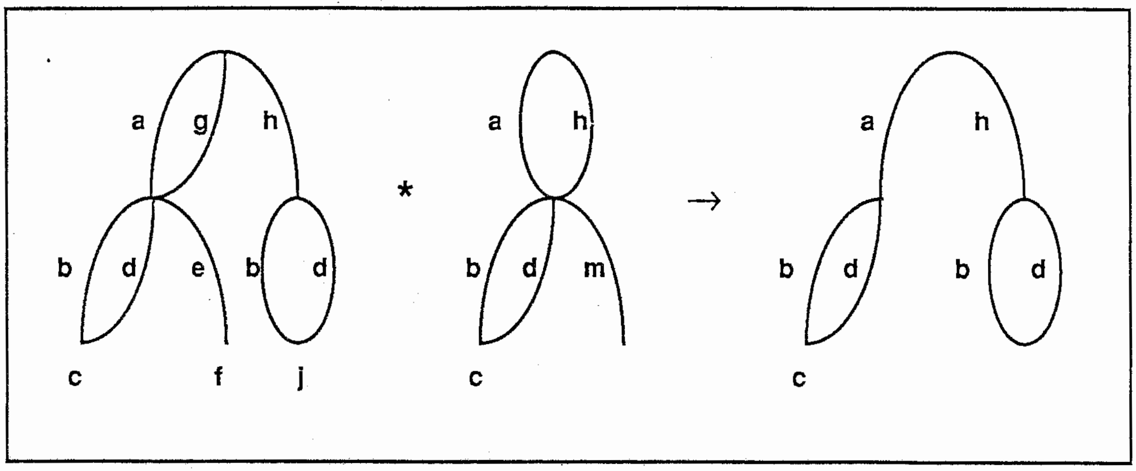


Figure 23: generalisation of TFS.

The lub for the inclusion ordering, which we shall call *intersection*, and write S**T=U, is included in the lub for subsumption. As for generalisation, we take the intersection of paths, but for the co-indexing relation we take the union of co-indexing relations, and not the intersection. To preserve the right-invariance property, we have to take in fact the right-invariant transitive closure, as for conjunction.

(d8) $U = S^{**}T \Leftrightarrow$

1) $\mathbf{P}_U = (\mathbf{P}_S \cap \mathbf{P}_T)^{[*]}$;

2) $\equiv_U \; = \; (\equiv_S \cup \equiv_T)^{[*]}$;

3) $\forall x \in \mathbf{P}_U, \tau_U(x) = \bigvee\{\tau_S(y), \tau_T(y) \,/\, x \equiv_U y\}$,

and the result is taken modulo bottom-smashing.

Reasoning on the graphic view, the intersection takes the largest common sub-graph, respecting identities when they exist. In fact, this means that taking the union of co-indexing relation, some identities may be *added.*

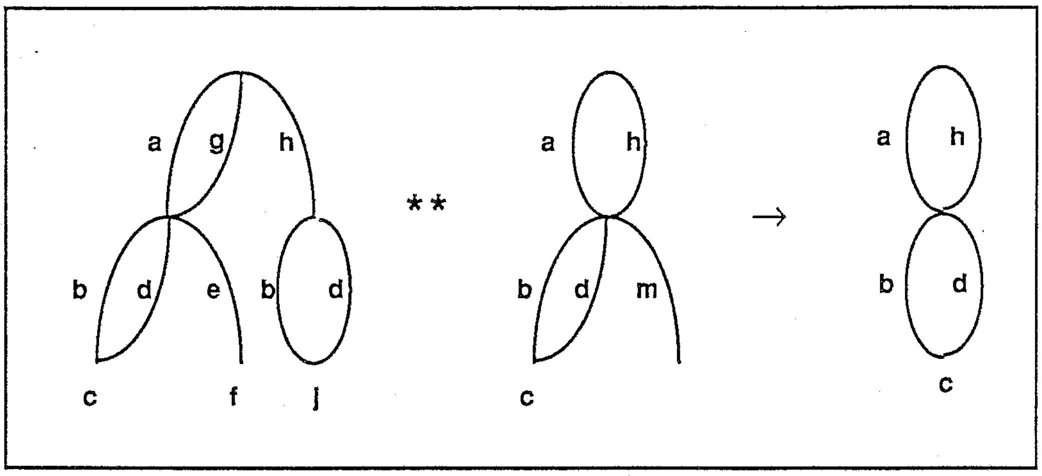


Figure 24: intersection of TFS.

Finally, we have a scale of operations on TFS (where ⊇ denotes the subsumption ordering):

(p9) $** \supseteq *$ and $++ \supseteq +$

We can think of conjunction and intersection as operations that preserve identities (i.e. co-indexing) whenever they exist, the first one taking the union of graphs and the second one the intersection of graphs. Similarily, we can think of generalisation and union as operations that preserve only equality, the first one taking recursively the intersection of values and the second one, the recursive union of values; the co-indexing relation is preserved for sub-parts which are not directly involved in the recursive intersection or union.

2.6 Difference

The difference of two TFS is first the difference of the sets of paths. But in general, the resulting set does not describe a connex graph: we have to remove all non connex sub-graphs that do not have the same root as the first operand.

(d9) $D = S–T \Leftrightarrow$

1) $P_D = P_S - \{xy \in P_S \,/\, x \in P_T\}$,

2) $\forall\, x, y \in P_D,\ x \equiv_D y \Leftrightarrow x \equiv_S y$,

3) $\forall\, x \in P_D, \tau_D(x) = \tau_S(x)$;

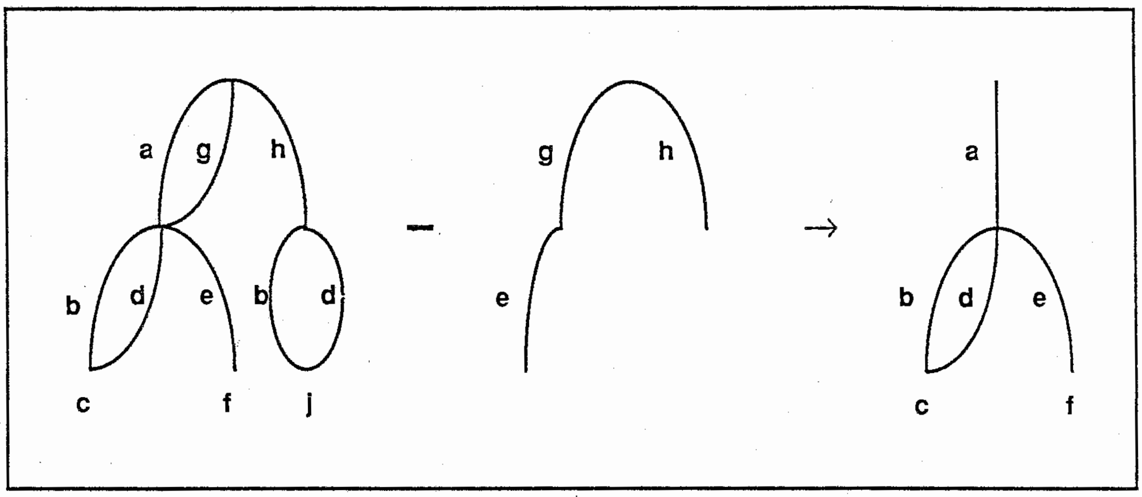


Figure 25: difference of TFS.

The application of the definition is straightforward: from top to bottom, remove all arcs which are in both. Take as a result the connex graph which has the same top as operand_1. Note that there could be another interpretation of H-set difference where only the leaves of operand_2 are removed in operand_1.

3. CONCLUSION

From a logical point of view, feature structures are interpreted as denotations of sets (feature descriptions). But the dual view (interpretation as objects) is equally valid and is the most natural for many linguists. Using this interpretation and the anlaogy that can be drawn with attribute structures, we have proposed a single framework in which an attribute structure is defined as a kind of a degenerated feature structure (it cannot represent sharing of information). This framework makes possible to extend classical and powerful operations on attribute structures to feature structures. Introduction of predefined types such as string and number types (with their associated operations) are derived from the traditional use of these types in attribute structures, and are defined as refinements of the atomic type.

When there is no sharing at all, feature structures behave exactly like hierarchical attribute structures. We can use this property to parametrize the interpreter: using an attribute interpretation («no-share» mode), all sharing is removed, and all operations are reduced to attribute operations that do not take indentities into account. With a graph interpretation («share» mode), all operations are avaible (see «pseudo-unification» in [Tomita 88]).

This report has given the motivations of this approach, and described the formal framework used for the language. We have developed an experimental functional interpreter to test these ideas. The basic data structure of the language is a typed feature structure. Operations available include conditional instructions and sequence of instructions, boolean expressions, assignment, unification, and various functional operations on typed features structures. Definitions include type definitions, template definition and (recursive) function definitions. This interpreter has been integrated in an Earley parser developed by Yves Nicolas [Nicolas 88]. The grammar formalism offers the grammar writer enhanced expressive power and allows use of a more modular approach to grammar developement. The interpreter of the language and grammar formalism will be described in a forthcoming report.

Acknowledgements

As a reference on attribute structures, I have used the work done during the French MT National Project on the specifications of a new software factory for MT [Boitet, Bachut et alii 88]. I am very grateful to Yves Nicolas, Christian Boitet and Kiyoshi Kogure for numerous and encouraging discussions.

ANNEX: Recursive definitions of operations on h-sets

We give the recursive definitions of operations on h-sets. They do not take indexing and typing into account. They could be used as a first level of specifications for implementing the actual operations involving indexing and type calculus. In fact, in the actual prototype implementation, the various algorithms were derived directly from these specifications, which made implementation, debugging and modification very simple.

In the following definitions, e*x* will represent a feature, l*x* a list of features, and f*x* a feature-value structure (either atomic or complex). Parentheses have their usual meaning. When a set is written between angle brackets, the elements of the set are treated as a list of elements: <{lx}> → lx. The boolean value TRUE is noted T and the boolean value FALSE is noted F.

Definition 1: union of H-sets

```
1     {} + {l2} → {l2}
2     {l1} + {} → {l1}
3     {e1{l1}, l2} + {e1{l3}} → { e1{[{l1}+{l3}]} } + {l2}
4     {f1, l1} + {f2} → {f1, [{l1}+{f2}] }
5     {l1} + {f2, l2} → ( {l1} + {f2} ) + {l2}
```

Definition 2: intersection of H-sets

```
1     {} * {l2} → {}
2     {l1} * {} → {}
3     { e1{l1}, l2 } * { e1{l3} } → { e1{[{l1}*{l3}]} }
4     {f1, l1} * {f2} → {l1} * {f2}
5     {l1} * {f2, l2} → ( {l1}*{f2} ) + ( {l1}*{l2} )
```

Definition 3: difference of H-sets

```
1     {} - {l2} → {}
2     {l1} - {} → {l1}
3     {e1{l1}, l2} - { e1{l3} } → {l2}
4     {f1, l1} - {f2} → {f1} + ( {l1} - {f2} )
5     {l1} - {f2, l2} → ( {l1} - {f2} ) - {l2}
```

Definition 4: membership in H-sets

```
1     f ∈ {} → F
2     e{l} ∈ {e{l1}, l2} → l = l1 or e{l} ∈ {l2}
3     f ∈ {f1, l1} → f ∈ {l1}
```

Definition 5: equality of H-sets

```
1     {l1} = {l2} → ∀ x ∈ {l1}, x ∈ {l2} and ∀ x ∈ {l2}, x ∈ {l1}
```

Definition 6: inclusion of H-sets

```
1     {} < {l2} → T
2     {l1} < {} → F
3     {e1{l1}} < {e1{l2}, l3} → {l1} < {l2} or {e1{l1}} < {l3}
4     {f1} < {f2, l2} → {f1} < {l2}
5     {f1, l1} < {l2} → {f1} < {l2} and {l1} < {l2}
```

REFERENCES

Hassan Aït-Kaci, 1984, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, PhD. Dissertation, University of Pennsylvania.

Hassan Aït-Kaci, 1986, *An Algebraic Semantics Approach to the effective Resolution of Type Equations*, Theoretical Computer Science 45, pp 293-351.

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

W. Bennett and J. Slocum, 1985, *The LRC Machine Translation System*, Computational Linguistics 11/2-3, April-September.

Ch. Boitet, P. Guillaume and M. Quezel-Ambrunaz, 1980, *Manipulation d'arborescences et parallélisme: le système ROBRA* , COLING-80.

Ch. Boitet, D. Bachut, N. Verastegui and R. Gerber, 1988, *ARIANE portable, Dossier des Spécifications Externes, Le langage TETHYS*, GETA-ADI.

R.J. Brachman and J.G. Schmolze, 1985, *An Overview of the KL-ONE Knowledge Representation System*, Cognitive Science 9/2, pp 171-216.

Alain Colmerauer, 1971, *Les SYSTEMES-Q, un formalisme pour analyser et synthétiser des phrases sur ordinateur*, Groupe TAUM, Université de Montréal.

Marc Dymetman, 1987, *RATP: un nouveau formalisme de la classe des grammaires d'unification*, Unpublished paper, April 1987.

G.E. Heidorn, K. Jensen, L.A. Miller, R.J. Byrd and M.S. Chodorow,1982 , *The EPISTLE text-critiquing system*, IBM Syst. Journal, 21/3.

K. Jensen and G.E. Heidorn, 1983, *The fitted parse : 100% parsing capability in a syntactic grammar of English*, Proc. of the Conf. on Applied Natural Language Processing, pp 93-98, Santa-Monica, California, February.

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, *Proc. of the 24th Annual Meeting of the ACL*, 10-13 June, Columbia University, New-York, pp 257-266.

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

J. Nakamura, J. Tsujii and M. Nagao, 1984, *Grammar Writing System (GRADE) of Mu-Machine Translation Project and its Characteristics*, COLING-84.

Yves Nicolas, 1988, *Pragmatic Extensions to Unification Based Formalisms*, ATR Interpreting Telephony Research Laboratories.

Carl Pollard and Ivan A. Sag, 1987, *Information-based Syntax and Semantics*, CSLI, Lectures Notes Number 13.

J. A. Robinson and E. E. Sibert, 1982, LOGLISP: an alternative to PROLOG, in *Machine Intelligence*, volume 10, J.E. Hayes, D. Michie and Y-H. Pao eds., Ellis Horwood Limited, pp 399-419.

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

Jonathan Slocum, 1984, *METAL: the LRC machine translation system*, ISSCO Tutorial on Machine Translation, Lugano, Switzerland, April 2-6.

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

Masaru Tomita (ed.), 1988, *The Generalized LR Parser/Compiler Version 8.1: User's Guide*, CMU-CMT-88-MEMO, 26 January 1988.

-0-0-0-0-0-0-0-0-0-0-0-0-0-