

Sorted Feature Terms and Relational Dependencies

Jochen Dörre

University of Stuttgart
Institut für maschinelle Sprachverarbeitung
Keplerstr. 17
D-7000 Stuttgart 1, Germany
e-mail: jochen@adler.philosophie.uni-stuttgart.de

Roland Seiffert

IBM Deutschland GmbH
Institute for Knowledge Based Systems
P.O. Box 80 08 80
D-7000 Stuttgart 80, Germany
e-mail: seiffert@ds0lilog.bitnet

Abstract

In this paper we describe the key ideas of the unification-based grammar formalism STUF that is being developed and used within the LILOG project at IBM Germany. STUF integrates *feature terms* with *sorts* and *recursive definitions of relations*. We provide a twofold motivation for our approach: First, we show how sorts can be exploited to reduce the structural complexity—especially the number of structural disjunctions—of feature terms. Second, we demonstrate how our relational dependencies help to write more concise and elegant grammars in principle-based approaches like HPSG. The main contribution of this paper is to provide a complete formal semantics for our formalism that is also applicable to other related formalisms currently being investigated.

Contents

1	Introduction	3
2	The Formalism	3
2.1	Sorts	4
2.2	Avoiding Structural Disjunctions	5
2.3	Integrating Relations	8
2.4	Relational Dependencies vs. Sorts	10
3	Defining Grammars	12
3.1	Principle-based Approach	13
4	Formalization	16
4.1	Feature Algebras with Relations	16
4.2	Denotation of Feature Terms and Models	17
5	Implementational Issues	18
6	Conclusion	20
A	An Example Grammar	22

1 Introduction

In this paper we describe the key ideas of the unification-based grammar formalism STUF¹ that is being developed and used within the LILOG project at IBM Germany. STUF integrates *feature terms* with *sorts* and *recursive definitions of relations*. This is in the spirit of a new consensus of recent formalisms used in computational linguistics that can be characterized as *principle-based* approaches to grammar. Principles state relations over (typed) feature structures, incorporating phonological strings and constituent structures as integral parts. Examples for such formalisms are HPSG ([PS87]), TFS ([ZE90]), CUF ([DE91]), and also the knowledge representation language LIFE ([AKP90]). We show how the concepts of these formalisms can be further enriched and integrated into one formalism, STUF, providing a rigorous though simple semantics. It is argued that exactly this chosen combination allows for a very elegant formulation of HPSG-style grammars. Also, we give a translation of relational dependencies to definite clauses of first-order logic that fits exactly into the generalized Constraint Logic Programming scheme of [HS88]. This opens up the treasure of results and techniques known in CLP and promises that an efficient implementation of STUF is possible.

2 The Formalism

Central to our formalism is the notion of a feature term coined by [KR86] and extended and generalized by [Smo88, Smo89]. It allows us to specify sets of feature structures in a linearized feature-matrix oriented notation. This differentiation between feature structures and their descriptions is crucial since it allows us to extend the descriptonal devices to include e.g. disjunction or negation without having to stipulate new kinds of underlying structures. We even can abstract away from the concrete notion of a feature structure. We only assume a domain of discourse to contain unspecified elements, where feature applications are reflected by functional dependencies between those elements. In the following the letters s, t, t_1, \dots will always denote feature terms. The syntactic forms of feature terms are given by the context-free production in Figure 1.

The simplest forms are variables and sorts. Feature terms may contain variables to state sharing of structures, i.e. they serve the same purpose as path equations in Kasper-Rounds logic. Sorts denote subsets of the semantic domain. They are described in more detail below. A term $f : t$ denotes the set of those elements for which the feature f leads to an element in the denotation of t . Conjunction, disjunction, and negation are set intersection, union, and complement, respectively.

Notation: In the following examples we assume notational conventions for variables

¹Once, this acronym had a meaning. But evolution didn't stop ...

s, t	\longrightarrow	X	a variable
		A	a sort
		$f:t$	feature selection
		$s&t$	conjunction (intersection)
		$s;t$	disjunction (union)
		$\neg t$	negation (complement)

Figure 1: The Syntax of Feature Terms

and lists like in Prolog. Variable names always start with capital letters whereas sorts, relations, and features start with a lowercase letter. We assume a uniform encoding of lists, e.g. with attributes *first* and *rest* for nonempty lists and the atom *nil* for the empty list. As in Prolog we will write

$[X Y]$	for a list with first element being X and the rest of the list being Y
$[\]$	for the empty list <i>nil</i>
$[X_1, X_2, \dots, X_n]$	as an abbreviation of $[X_1 [X_2 \dots[X_n [\]]\dots]]$, i.e. this stands for the list $X_1X_2\dots X_n$.

2.1 Sorts

The intended meaning of sorts is to denote subsets of the domain of discourse. The integration of sorts into feature terms follows [Smo88]. The only difference is, that we made it possible to define the sort structure within our formalism.

Sorts come in three varieties: *atoms*, *primitive sorts*, and *defined sorts*. For atoms we use the letters a, b, c, \dots and for sorts other than atoms we write A, B, C, \dots . An *atom* is assumed to denote a singleton subset of the domain that is disjoint to the set denoted by any other atom or primitive sort. A *primitive sort* denotes an arbitrary, not further specified subset of the domain. However, we can declare primitive sorts to be disjoint to other primitive sorts. *Defined sorts* can be built from arbitrary sorts using boolean connectives in a definition of the form $A = \text{sexpr}$. Figure 2 shows the syntax of these sort expressions. The connectives are interpreted in the usual way, i.e. for example $A \sqcap B$ denotes the intersection of the sets denoted by A and B .

Besides these three types of sorts there are the two additional sorts \top and \perp with a fixed meaning. They denote the entire domain and the empty set, respectively.

Obviously, the sets denoted by sort expression defined in this way form a distributive lattice with set inclusion being the order relation and intersection and union being the meet and join operations. In our system these operations are implemented

$$\begin{array}{l}
\text{sexpr} \rightarrow A \\
a \\
\text{sexpr} \sqcap \text{sexpr} \\
\text{sexpr} \sqcup \text{sexpr} \\
\neg \text{sexpr}
\end{array}$$

Figure 2: The Syntax of Sort Expressions

using a propositional theorem prover that operates on bit vectors for the internal representation of sorts². The description of that algorithm lies beyond the scope of this paper.

Sorts serve two purposes, a syntactic and a semantic one. In the syntax they are used by a typing scheme similar to that of [Car90]. For example, for well-typed expressions for which a certain sort is specified (or inferred) only certain features are allowed, whose values in turn have to obey sort restrictions. However, in this paper we will not give details of this syntactic use of sorts.

Semantically, sorts are a means to coarsely structure the domain of interest. The sorts definable in our system can constitute a hierarchy, and one way of refining the information about a certain object which is known to be of some sort A would be to go to a subsort of A . Compared to the usual atomic values in feature structures, the unification of two different sorts does not necessarily lead to inconsistency, but instead depends on the sort hierarchy and we get the greatest common subsort of the two, if it exists. Also, sorts are compatible with feature specifications, i.e. objects denoted by sorts may have features.

2.2 Avoiding Structural Disjunctions

Disjunctive feature terms are a well-known source for very hard computational problems. But there is an interesting class of disjunctions for which these problems can be avoided using our approach of sorted feature terms.

Consider the following example specifying all possible structures of a feature `agr` that is intended to model agreement information in a grammar:

```

agr: num: (sg; pl) &
    gen: (fem; masc; neut) &
    per: (1st; 2nd; 3rd)

```

The values of `num`, `gen` and `per` are disjunctions of atoms.

²similar to that of [She89]

A typical instance of such an agreement structure is the following part of the definition of the word “go”:

```
word(go) ==>
  ... agr: ( num: pl
             ; num: sg & ( per: 1st ; per: 2nd )
             )
```

The structural disjunctions in this example can be removed and can be very efficiently handled by the sort unification part of STUF. Given the description of all possible agreement structures above, we introduce a primitive sort—i.e. a sort that is some unspecified subset of the universe—for each of the atoms occurring as values of the embedded features of `agr` and specify that the primitives under each one of the features are pairwise disjoint. Then we define one sort for each of those features as the disjunction of the embedded primitives. Finally, we introduce a sort `agr` as the conjunction of all the sorts that have been introduced for the embedded features:

```
primitive sg, pl.
disjoint  sg, pl.
num =     sg; pl.
primitive fem, masc, neut.
disjoint  fem, masc, neut.
gen =     fem; masc; neut.
primitive 1st, 2nd, 3rd.
disjoint  1st, 2nd, 3rd.
per =     1st; 2nd; 3rd.

agr =     num & gen & per.
```

All possible structures with the feature `agr` are now given by

```
agr: agr
```

After having declared the sorts we must change the definition of the verb “go” accordingly:

```
word(go) ==>
  ... agr: ( pl
             ; sg & ( 1st ; 2nd )
             )
```

Semantically, this method introduces “agreement objects”, i.e. linguistic objects that are classified according to their agreement properties.

The above structure turns out to be equivalent to the following one under the additional assumption that all values of the feature `agr` must be of the sort `agr`. This definition says exactly what the grammar writer would like to express, namely, that the agreement restrictions of “go” are simply that it cannot be a third singular form:

```
word(go) ==>
... agr: not (3rd & sg)
```

Note that there is a little semantic difference between the two representations: Using features `num`, etc., allows the grammar writer to completely omit specifications for one of those features. This is semantically different from specifying the feature and a disjunction of all possible values. But the intended meaning in these cases is always the disjunction! Leaving the feature out is a hack to overcome the problems of specifying too many disjunctions and thereby slowing down the unification process. Our representation using sorts reflects precisely the intended meaning without introducing cumbersome structural disjunctions.

There are quite a lot of situations in a typical unification grammar that allow for a similar optimization. We expect that consequent usage of this technique can cut down the number of structural disjunctions considerably.

But probably, grammar writers do not want to change their usual way of dealing with features like `agr`. Therefore, we want the compiler to do this optimization to free them of the burden.

Note that a sufficient condition to make this optimization applicable is that we have a feature with a finite range of possible values and that there is no direct reference to these values using variables or path equivalences. In principle, this property can be decided by the compiler.

Obviously, in the above example this is usually met since all agreement features are handled together. A common situation is:

```
... agr: X
... ... agr: X
```

Values of embedded features are not referred to individually, i.e. there are no specifications like the following one:

```
... agr: num: X
... ... agr: num: X
```

But this decision turns out to be problematic to make when implementing an incremental compiler since it needs the whole grammar and lexicon to check the preconditions for the optimization. Therefore, we introduce a special directive `finite` in the STUF language that identifies features that allow for the optimization:

```
finite
  agr: num: (sg; pl)
       gen: (fem; masc; neut)
       per: (1st; 2nd; 3rd)
```

With this `finite` declaration the compiler will apply the optimization to the specified values of the `agr`-feature. This will result in the sort definitions and the new structure of the agreement specification in the lexical entry of “go” shown above.

2.3 Integrating Relations

We extend the syntax of feature terms to include the form:

$$A(t_1, \dots, t_n) \quad \text{a relational dependency}$$

where t_i are all feature terms. Hence, relational dependencies are used in our syntax as function applications. The meaning of such a term depends on the $n+1$ -place relation A , which can be introduced through relation definitions, as described below. A relational dependency term $A(t_1, \dots, t_n)$ now denotes the set of values which the additional argument, let’s call it the 0-argument, can take, when the other arguments are in the denotations of their respective t_i . Suppose for example a 3-place relation *append* on lists whose 0-argument is the concatenation of the other two arguments. The definition of this relation is given below. Now, the term with the variables X , Y , and Z

```
f: X &
g: Y &
h: Z &
i: append(X, append(Y,Z))
```

denotes a structure whose value of the feature `i` is the concatenation of the values of the features `f`, `g`, and `h`. The meaning of relational dependencies does not imply an order of evaluation as one might assume for this function application syntax. Our semantics is completely declarative, also in this respect. Hence, a relational dependency may be used to generate its arguments from its ‘result’ value, or to propagate side-effects from one argument to another.

For example, conjoining the above term with

g: [b] &
i: [a,b,c]

would only yield a non-empty denotation if the values of **f** and **h** are taken to be [a] and [c], respectively.³

A $n+1$ -place relation A is defined through a set of defining clauses of the form:

$$A(t_1, \dots, t_n) ==> t_0. \quad \text{where all } t_i \text{ are feature terms}$$

Multiple defining clauses for one relation are taken disjunctively. The meaning of such a defining clause is that the terms t_0 to t_n give us a sufficient condition on the description of objects u_0 to u_n , respectively, in order to be in the relation A . Hence, relation A contains the cross-product of the denotations of t_0 to t_n . For example, the definition of the recursive relation *append* can be given as follows.

`append([],L) ==> L.`
`append([F|R],L) ==> [F|append(R,L)].`

Another interesting relation is the relation *true* as defined by:⁴

`true(_) ==> _.`

Although the relation `true` holds for arbitrary pairs of objects of the universe, we can use an expression `true(t)` to restrict possible variable assignments to those for which t does not denote the empty set. Notice that `true(t)` is equivalent to \top if and only if t does not denote the empty set. Otherwise it is equivalent to \perp .

For instance, we could write $t \ \& \ \text{true}(X \ \& \ Y)$ in order to add to the feature term t the additional constraint that the variables **X** and **Y** have to be identified, i.e. the whole term denotes the same as t , if **X** and **Y** denote the same singleton, independently from what it actually is. Otherwise it denotes the empty set.

Actually, the introduction of definable relations means that we are proposing some sort of logical programming language where a very powerful term syntax is used, including disjunction and relational dependencies, making the relational body of a clause superfluous. Notice that the right-hand side of a defining clause is just the term for the implicit 0-argument.

An important observation is that unary relations and sorts are semantically the same types of objects, namely sets over the semantic domain. Also syntactically there is no difference between “application” of a unary relation and a sort. Since the only difference between a unary relation and an ordinary sort is that a unary relation is defined via general feature terms, we will call such relations simply generally-defined sorts. The term “relational dependencies” shall henceforth only refer to terms of at least one parameter, i.e. terms that refer to relations of at least two arguments.

³or anything that is subsumed by that

⁴We use the symbol ‘ \top ’ for the sort \top .

2.4 Relational Dependencies vs. Sorts

In this section we show that it is in fact possible to eliminate parametric relational dependencies and encode all relations with generally-defined sorts, however, where the resulting feature structures have additional features holding additional information. To see this, consider the following definition of a parameter-free sort `app`.

```

app ==> arg1: [] &
        arg2: L &
        res: L.
app ==> arg1: [H|T] &
        arg2: L &
        res: [H|TL] &
        aux: (app & arg1:T & arg2:L & res:TL).

```

`app` defines the sort of all feature structures whose `res`-features are the concatenation of the values of their `arg1`- and `arg2`-features, respectively. The general strategy to make use of the relation which such a sort establishes is to introduce a new feature somewhere which has as its value this sort together with its argument features and argument values for them. With this we enforce that the relation holds between whatever we specified as argument values. In most cases these values will be given using variables which appear elsewhere in the structure such that we really establish a relation between those parts of the structure. Note that, what would be the result in the parametrized version is simply treated as an argument in this approach, as e.g. the argument called `res` above.

An example:

```

f: X &
g: Y &
h: append(X,Y)

```

can also be expressed by

```

f: X &
g: Y &
h: Z &
dummy: (app & arg1: X & arg2: Y & res: Z)

```

Thus, the `dummy`-feature is used to perform a “subcomputation”. For the same reason we need the `aux`-feature in the definition of `app`.

Here we want to argue, that the use of parameters is not only of aesthetical interest, but also has practical advantages. First of all, in an implementation of the

parameter-free approach the space required by “subcomputations”, which in that approach belong to the result, cannot be reclaimed by a garbage collector. This space can be very large if we think of recursive functions such as `append`. We can only reclaim this space if we know for sure that no further reference is made to information available through a “subcomputation”-feature, and this is exactly what is the difference between the two approaches. In the parameter approach we cannot refer to intermediate results of subcomputations. We abstract away from these by using parameters and giving back only the actual result of a functional application. On the other hand “subcomputation”-features allow for referring to intermediate results. We have discussed this issue at some length, because we want to argue that in some sense the HPSG feature *dtrs* has the same effect as subcomputation features and we can avoid them in a parametrized setting, without giving up anything of the elegance and declarativity of the principle-based encoding of grammar exemplified by HPSG.

But there are also cases in which the use of attributes should be preferred over using parameters. The first two advantages have to do with the benefits of feature structures over first-order terms, namely the variable arity and the possibility to identify arguments by keywords. Thus, if we want to have relations with lots of arguments, maybe we even don’t want to fix beforehand how many, for which when used only some arguments shall be given leaving the rest unspecified, i.e. \top , we better use attributes. Another motivation for having relations expressed through attributes is the following. Consider the case when we have two or more related structures, e.g. the in- and out-list of a difference list pair, which we want to pass up or down unaltered through a series of sorts. In this case it is advantageous to be able to glue together the two components in a feature structure. It is clear, that all of this can be done in the formalism.

It is also interesting to consider the question of what has to be added to a version of the formalism with just generally-defined sorts to overcome the above-mentioned deficiencies. Actually, one relational dependency of one parameter is enough:

```
extract_res( res: X ) ==> X.
```

This definition gives us an operator which, as the name suggests, extracts the result of its parameter, which means here, the feature `res`. We can use this operator to eliminate subcomputation features. For example, we could formulate the second clause of `app` as:

```
app ==> arg1: [H|T] &
        arg2: L      &
        res: [H|extract_res(app & arg1:T & arg2:L)].
```

A similar solution to this problem of nonparametrized sorts is pursued in the work of Zajac and Emele [ZE90, EHMZ90]. Their formalism of typed feature structures

allows the addition of “conditions”, introduced with the symbol $:-$, in the definition of sorts (which they call types). Thus the second clause of `app` becomes (adapted to our syntax):

```

app ==> arg1: [H|T] &
        arg2: L      &
        res: [H|TL]
:- (app & arg1:T & arg2:L & res:TL).

```

On the first look this seems to be a type mismatch, since on one hand we define `app` as a typed feature structure, i.e. a sort, but on the other hand we want to use it as a condition. However, it is clear that we can view the sort specified after a $:-$ as the condition of this sort being nonempty, being consistent. Thus, this operator simply ignores the structure built by this sort, as long as it exists. We get exactly the same effect as with our 1-place sort `true`.

3 Defining Grammars

A simple and instructive example is the definition of PATR-II-style grammars in our formalism. Similar to the view in HPSG we do not assume a separation between a level of strings and a level of feature structures, but assume strings to be encoded as feature structures and define the meaning of grammars simply with a relation over feature structures. A PATR-II grammar rule can be given as a sequence of feature terms:

$$t_0 \longrightarrow t_1, \dots, t_n$$

In our formalism we want to view such a rule as a macro notation for

```

phrase( $t_0$ ) ==>
  append(phrase( $t_1$ ), append(... append(phrase( $t_{n-1}$ ), phrase( $t_n$ ))...)).

```

which defines `phrase` as a 2-place relation from feature structures to terminal strings. The definition simply says that the terminal string of the LHS, `phrase(t_0)`, is the concatenation of the strings of the RHS, i.e. the `phrase(t_i)`, $i = 1 \dots n$.

For the lexicon we can directly specify

```

phrase( $t_w$ ) ==> [ $w$ ].

```

for each entry with word w and associated feature term t_w . The denotation of the grammar is now simply the denotation of the relation `phrase`, i.e. we have defined the grammar completely inside our logic.

Alternatively we could have defined the inverse relation of `phrase`, lets call it `type`, because it gives us the grammatical type of a phrase. Interestingly its formulation is not as straightforward as the one of `phrase`. Assume lexical entry and rule form as above, then we get the definitions

$$\text{type}([w]) \implies t_w.$$

for each lexical entry and

$$\begin{aligned} \text{type}(\text{append}(W_1, \text{append}(\dots, \text{append}(W_{n-1}, W_n) \dots))) \implies \\ t_0 \ \& \ \text{true}(\text{type}(W_1) \ \& \ t_1) \ \& \ \dots \ \& \ \text{true}(\text{type}(W_n) \ \& \ t_n). \end{aligned}$$

for each rule.

We use “calls” to the auxiliary relation `true` of the form `true(type(W_i) & t_i)` to constrain W_i . This expression has the same effect as we would get if we could constrain W_i to be the result of applying the inverse relation of `type` to t_i .

The definition of `type` can easily be extended if we want to encode strings with difference lists to avoid explicit appending.

$$\text{type}([w|R], R) \implies t_w.$$

$$\begin{aligned} \text{type}(L_0, L_n) \implies t_0 \ \& \\ \text{true}(\text{type}(L_0, L_1) \ \& \ t_1) \ \& \ \dots \ \& \ \text{true}(\text{type}(L_{n-1}, L_n) \ \& \ t_n). \end{aligned}$$

With this definition `type(ϕ , [])` denotes the same structures as `type(ϕ)` for the definition above. See Appendix A for an example of a PATR-II grammar defined in this way.

3.1 Principle-based Approach

Having understood the approach described above to define the grammatical relationship between strings and feature structures a further generalization becomes evident. If we give up our macro notation for rules, but instead specify the relationship directly, we are no longer restricted to concatenation as the only operation available on strings, since we are now dealing explicitly with them. For example, we can define the word order domain operations of [Rea89]. Also, we are no longer restricted to a one-dimensional set of rules, which enumerate all possible local trees, but rather may have a whole hierarchy of rules, expressed through a hierarchy of `phrase`-like sorts, including e.g., principles that hold on local trees no matter which rule is going to be used.

This approach is exactly the one taken in HPSG.⁵ There the sort `sign` corresponds to our sort `phrase` and a grammar is given by ultimately defining this sort. Note that the actual grammar rules, the defining clauses of `rule`, have to contribute only very few information, because most of the information constraining a phrasal sign is inherited in an object-oriented way from principles.

However, the grammar architecture we want to propose deviates from the approach of [PS87] in one important respect. We do not assume that the whole constituent structure of signs responsible for the derivability of a phrasal sign is accessible from that sign via a feature, the feature *dtrs*. We want to eliminate this feature and instead model principles and rules as relational dependencies.

The main motivation why we *want* to do this is because the structures that we regard as the result of an analysis and also intermediate results are immensely smaller. The main reason why we *can* do this is the following. The attribute *dtrs* actually is needed by the principles and the grammar rules to be able to enforce relations between daughters and mother category, i.e. the structure containing the *dtrs* attribute, of a local tree. Principles, which in essence are generalizations which hold over the whole set of rules, can only be established if we are able to talk about the same mother, the same head daughter and the same complement daughter categories. This is achieved in a nonparametrized approach, where only one structure can be constrained, by attaching the daughters of local trees to the structure of the mother via the feature *dtrs*. However, principles and rules, as formulated for HPSG, are local in the sense that only one local tree is regarded at once. They never talk about head or complement daughters of some daughter category or of any other part of the structure, e.g. of the subcategorization list. Also, lexical signs never make direct use of the notion of constituency, an assertion which is called the Locality Principle in [PS87, pp.143–144]:⁶

... Our answer is that all of these locality restrictions follow from the assumption that *the SUBCAT elements of lexical signs specify values for SYNTAX and SEMANTICS but crucially not the attribute DAUGHTERS*. This *Locality Principle*, we suggest, is a universal constraint on lexical signs.

These considerations lead us to the claim, that principles and rules are better formalized as relations of three independent arguments, mother, head daughter and the list of complement daughter categories, which in our approach means, they are

⁵There is a superficial difference in that in [PS87] principles are stated as general axioms, whereas in our approach any clause has to contribute to the definition of one specific sort. However, as our example shows, this poses no problem, since all the principles in our example can be written such that they can simply be applied to the sort `headed_phrase` and we believe that even if we extend the fragment the properties which are expressed using principles can always be fixed to be properties of specific sorts.

⁶The stress is that of [PS87]

```

sign ==> headed_phrase.
sign ==> word.

headed_phrase ==> constituent_order_principle(HD,CD) &
                 universal_principles(HD,CD) &
                 rule(HD,CD).

universal_principles(HD & sign, CD & signs) ==>
    subcat_principle(HD,CD) &
    head_feature_principle(HD) &
    semantics_principle(HD,CD).

subcat_principle(syn:loc:subcat:append(S1,S2), S2) ==>
    syn: loc: subcat: S1.

head_feature_principle(syn:loc:head:X) ==>
    syn: loc: head: X .

semantics_principle(HD & sem:cont:SH, CD) ==>
    sem: ( cont: successively_combine_semantics(SH,CD) &
          indices: collect_indices([HD|CD]) ).

signs ==> [].
signs ==> [sign|signs].

/* language-dependent principles (for phrasal signs) */
constituent_order_principle(HD,CD) ==>
    phon: collect_phon(order_constituents([HD|CD])).

```

Figure 3: Parametrized version of the HPSG-style approach, principles

3-place relations (appearing as 2-place relational dependencies).

Figures 3 and 4 present the straightforward formulation of the grammar architecture of HPSG where principles and rules have the two parameters *head daughter* and *complement daughter list*. This approach has clear computational benefits against one with an explicit *dtrs* structure, since the Locality Principle is built into this architecture and hence, no constituent structures with their dependent signs have to be constructed explicitly, e.g. during parsing. The constituent structure only exists as an implicit relation between the constructed signs.

```

/* rules of the grammar */
/* rule 1 */
rule(syn:loc:lex:minus, [_]) ==>
    syn: loc: subcat: [].

/* rule 2 */
rule(syn:loc:(head:inv:minus &
    lex:plus) , _ ) ==>
    syn: loc: subcat: [_].

/* rule 3 */
rule(syn:loc:(head:inv:plus &
    lex:plus) , _ ) ==>
    syn: loc: subcat: [].

```

Figure 4: Parametrized version of the HPSG-style approach, rules

4 Formalization

We hope that we have given enough examples for the simplicity and perspicuity of our formalism to motivate the reader to also work her way through the following formalization.

4.1 Feature Algebras with Relations

Feature algebras are a generalization of the usual feature structures as formalized for example as a special kind of finite automata in [KR86]. Following [Smo88, Smo89] we devise a Tarskian open-world semantics for our feature terms. We extend the original notion of a feature algebra as given in [Smo89] and [DR90] by including arbitrary relations. Henceforth the notion of a feature algebra shall refer to this extended version given below.

We assume a signature containing the following sets:

- an alphabet \mathbf{S} of sort (relation) symbols containing the symbols \top and \perp .
- a set of symbols for singleton sorts $\mathbf{Sg} \subset \mathbf{S}$
- the arity function $Ar : \mathbf{S} \mapsto N_0$, such that $Ar(A) = 0$ for all $A \in \{\top, \perp\} \cup \mathbf{Sg}$.
- an alphabet \mathbf{F} of feature symbols.

Additionally we assume an infinite set \mathbf{V} of variables, disjoint to \mathbf{S} .

A feature algebra \mathcal{A} consists of a nonempty set $D^{\mathcal{A}}$, the domain of objects, and an interpretation function $\cdot^{\mathcal{A}}$ defined on \mathbf{S} and \mathbf{F} such that:

- $\top^{\mathcal{A}} = D^{\mathcal{A}}$
- $\perp^{\mathcal{A}} = \emptyset$
- for every sort A : $A^{\mathcal{A}} \subseteq (D^{\mathcal{A}})^{Ar(A)+1}$, i.e. $A^{\mathcal{A}}$ is a set of $Ar(A)+1$ -tuples over $D^{\mathcal{A}}$
- singleton sorts are mapped onto distinct singleton sets
- for every feature f : $f^{\mathcal{A}}$ is a unary partial function on $D^{\mathcal{A}}$.
- no feature is defined on the element of a singleton sort

4.2 Denotation of Feature Terms and Models

A variable assignment is a mapping from variables to the elements of some feature algebra. The denotation of a feature term with respect to a feature algebra \mathcal{A} under an assignment α is a subset of $D^{\mathcal{A}}$ as given in Figure 5.

$\llbracket x \rrbracket_{\alpha}^{\mathcal{A}}$	$:= \{ \alpha(x) \}$
$\llbracket f:t \rrbracket_{\alpha}^{\mathcal{A}}$	$:= \{ d \in D^{\mathcal{A}} \mid f^{\mathcal{A}}(d) \in \llbracket t \rrbracket_{\alpha}^{\mathcal{A}} \}$
$\llbracket s \& t \rrbracket_{\alpha}^{\mathcal{A}}$	$:= \llbracket s \rrbracket_{\alpha}^{\mathcal{A}} \cap \llbracket t \rrbracket_{\alpha}^{\mathcal{A}}$
$\llbracket s; t \rrbracket_{\alpha}^{\mathcal{A}}$	$:= \llbracket s \rrbracket_{\alpha}^{\mathcal{A}} \cup \llbracket t \rrbracket_{\alpha}^{\mathcal{A}}$
$\llbracket \neg t \rrbracket_{\alpha}^{\mathcal{A}}$	$:= D^{\mathcal{A}} - \llbracket t \rrbracket_{\alpha}^{\mathcal{A}}$
$\llbracket A(t_1, \dots, t_n) \rrbracket_{\alpha}^{\mathcal{A}}$	$:= \{ d_0 \in D^{\mathcal{A}} \mid \exists d_1, \dots, d_n. \langle d_0, \dots, d_n \rangle \in A^{\mathcal{A}} \wedge \bigwedge_{i=1, \dots, n} d_i \in \llbracket t_i \rrbracket_{\alpha}^{\mathcal{A}} \}$

Figure 5: Denotation of Feature Terms in \mathcal{A} under assignment α

The last line is meant to also include sorts of arity 0. In this case the denotation of A reduces to $A^{\mathcal{A}}$ itself.

We call a feature term *consistent* iff it has a nonempty denotation for some interpretation and some variable assignment. Otherwise it is *inconsistent*.

Relation definitions are the formulae of our logic. Ideally the set of defining clauses for a relation A should completely define this relation, i.e. should give necessary and sufficient conditions elements to stand in this relation. However, we will motivate a restriction on the use of negation in our operational semantics below which allows for an especially simple translation of relation definitions to definite clauses in first-order logic with equality. Due to this restriction we can give a simplified semantics to relation definitions, since they will never be used as necessary conditions.

Definition 1 (Model) *A feature algebra $(D^{\mathcal{A}}, \cdot^{\mathcal{A}})$ models a set of defining clauses \mathcal{C} iff for any $A \in \mathbf{S}$ with $Ar(A) = n$, if $A(t_1, \dots, t_n) \implies t_0 \in \mathcal{C}$ and there is an assignment α such that $\exists d_0 \dots d_n : d_i \in \llbracket t_i \rrbracket_{\alpha}^{\mathcal{A}} (i = 0 \dots n)$, then $\langle d_0, \dots, d_n \rangle \in A^{\mathcal{A}}$.*

According to this definition we can always construct a trivial model by letting the denotation of every relation be the full cross-product over the domain with the respective arity: $A^A = (D^A)^{Ar(A)+1}$. The denotation of sorts for which no definition exists is free. Notice also that each variable is interpreted as being universally quantified in the defining clause it occurs in. In [DE91] this semantics is discussed in greater detail. The authors point out the problematic interaction of general negation and predicate completion that views relational definitions also as necessary conditions.

5 Implementational Issues

In this section we will present some of the basic ideas underlying our current STUF implementation. First, recall that our grammars employ a quasi-functional notation to define relations. We can make these relations explicit by systematically introducing an additional argument for all relations. We now present a translation function *trans* (Figure 7) that converts a given definition $r(\vec{s}) ==> t$ into a new, equivalent definition of the form $r(X, \vec{X}) \leftarrow t'$, where t' is a formula built of conjunctions and disjunctions of *basic constraints* (Figure 6). We call the first four forms *primitive*

$f(X) = Y$, where f is a feature, X and Y are variables
 $X \in A$, where A is a sort, X is a variable
 $X = Y$, where X and Y are variables
 $X \neq Y$, where X and Y are variables
 $r(X_0, \dots, X_n)$, where r is a (defined) n -ary relation; X_0, \dots, X_n is
 usually written as \vec{X}

Figure 6: The Basic Constraints

constraints. They are just the feature constraints of ordinary feature logic with subsorts [Smo88].

Note that we don't have a translation scheme for general negation $\neg t$. This is not an accident as will become clear soon.

As an example consider the definitions for `subcat_principle` and `append` from our HPSG fragment:

```

subcat_principle(syn: loc: subcat: append(S1,S2),S2) ==>
    syn: loc: subcat: S1.

append(nil,Y) ==> Y.
append(cons&(first:X1)&(rest:X),Y) ==>
    cons&(first:X1)&(rest:append(X,Y)).

```

$\text{trans}(r(\vec{s}) \implies t)$	$\Rightarrow r(X, \vec{X}) \leftarrow \text{trans}(X, t) \wedge \text{trans}(X_i, s_i)$, for all X_i in \vec{X} and corresponding s_i in \vec{s} , and X and all X_i are new variables
$\text{trans}(X, f : t)$	$\Rightarrow f(X) = Y \wedge \text{trans}(Y, t)$, where Y is a new variable
$\text{trans}(X, A)$	$\Rightarrow X \in A$
$\text{trans}(X, \neg A)$	$\Rightarrow X \in \bar{A}$
$\text{trans}(X, Y)$	$\Rightarrow X = Y$
$\text{trans}(X, \neg Y)$	$\Rightarrow X \neq Y$
$\text{trans}(X, t_1 \& t_2)$	$\Rightarrow \text{trans}(X, t_1) \wedge \text{trans}(X, t_2)$
$\text{trans}(X, t_1 ; t_2)$	$\Rightarrow \text{trans}(X, t_1) \vee \text{trans}(X, t_2)$
$\text{trans}(X, r(\vec{t}))$	$\Rightarrow r(X, \vec{X}) \wedge \text{trans}(X_i, t_i)$, for all X_i in \vec{X} and corresponding t_i in \vec{t} , and all X_i are new variables

Figure 7: The Translation Function *trans*

The function *trans* applied to all definitions in the example yields:

$$\begin{aligned} & \text{subcat_principle}(M, HD, CDs) \leftarrow \\ & \quad \text{syn}(M) = M1 \wedge \text{loc}(M1) = M2 \wedge \text{subcat}(M2) = S1 \wedge \\ & \quad \text{syn}(HD) = HD1 \wedge \text{loc}(HD1) = HD2 \wedge \text{subcat}(HD2) = S \wedge \\ & \quad \text{append}(S, S1, CDs). \end{aligned}$$

$$\begin{aligned} & \text{append}(Y, X, Y) \leftarrow \\ & \quad Y \in \text{nil}. \\ & \text{append}(Z, Y, X) \leftarrow \\ & \quad Z \in \text{cons} \wedge \text{first}(Z) = X1 \wedge \text{rest}(Z) = Zs \wedge \\ & \quad X \in \text{cons} \wedge \text{first}(X) = X1 \wedge \text{rest}(X) = Xs \wedge \\ & \quad \text{append}(Zs, Xs, Y). \end{aligned}$$

Equations between simple variables are already eliminated through variable substitution.

If we exclude general negation then each of the definitions resulting from *trans* can be easily transformed into an equivalent set of *normal form clauses*. The normal form is defined as

$$r_0(\vec{X}_0) \leftarrow \phi \wedge r_1(\vec{X}_1) \wedge \dots \wedge r_n(\vec{X}_n)$$

ϕ is a formula containing arbitrary conjunctions and disjunctions of primitive constraints. If our system contains only definitions in normal form—definite clauses—

then this fits nicely into the refined Constraint Logic Programming scheme described in [HS88]. In fact, our definite relations correspond exactly to the relational extensions of simple feature logic as described in [Smo88] for which efficient constraint solvers exist. This immediately gives us an operational semantics for solving the relational constraints, which is a generalization of SLD-resolution. In this approach we can resolve in a sequence of goals $r_0(\vec{X}) \wedge RelAtoms \wedge \phi_0$ with the above clause by replacing $r_0(\vec{X})$ by $r_1(\vec{X}_1) \wedge \dots \wedge r_n(\vec{X}_n)$ and replacing ϕ_0 by a solved form of $\phi_0 \wedge \phi \wedge \vec{X} = \vec{X}_0$, if that exists. This constraint solving step corresponds to the term unification of the head in conventional logic programming. Our implementation of STUF is based on this SLD-resolution scheme and thanks to [HS88] we know that this approach is sound and complete.

Another observation is that a very common optimization technique from conventional logic programming can be integrated into our framework: Some of the (non-relational) constraints of ϕ are associated with the head of a definite clause (ϕ_{Head}) and others with the body (ϕ_{Body}), i.e. $\phi = \phi_{Head} \wedge \phi_{Body}$.

$$r_0(\vec{X}_0) \leftarrow \phi_{Head} \wedge \phi_{Body} \wedge r_1(\vec{X}_1) \wedge \dots \wedge r_n(\vec{X}_n)$$

ϕ_{Head} should be very simple but impose very strong constraints. Then it can be used to efficiently cut down the search space significantly. When selecting a clause, we first unify with ϕ_{Head} and if this fails, we reject the clause immediately. Typical constraints in ϕ_{Head} are sort restrictions on the variables occurring in the head of the clause. For instance, in the second clause of `append` we could have

$$\begin{aligned} \phi_{Head} &= Z \in cons \wedge X \in cons \\ \phi_{Body} &= first(Z) = X1 \wedge rest(Z) = Zs \wedge first(X) = X1 \wedge rest(X) = Xs \end{aligned}$$

6 Conclusion

We have presented a formalism for the flexible specification of unification-based grammars which integrates feature terms with sorts and recursive definitions of relational dependencies. Grammars as well as any auxiliary notions like *order-constituents* in HPSG are simply defined as relations over feature structures. We don't have to assume additional levels of description for terminal strings, constituent structure, functions, We view all these as feature structures and relations on them. The declarative definition of relational and functional dependencies is the central building block of our formalism. Taking serious relational dependencies leads us to a grammar architecture for HPSG-style grammars based on principles and rules that are conceptualized as relations between the categories of a local tree. With this approach the phrasal signs that have to be built can be immensely smaller. We have provided a rigorous direct semantics of the formalism as well as

a translation of relational definitions with feature terms to definite clauses of first-order logic. Since these can be viewed as the relational extension of a well-known constraint language in the sense of [HS88], we can rely on their results concerning an operational semantics.

Appendix A: An Example Grammar

In the following we give an example of a PATR-II grammar converted to STUF, as described in Section 3, as well as the definition of its signature, i.e. defining the symbols which are atoms or primitive sorts, the sort hierarchy, and the types of features which shall be used.

section types.

```
%% S -> NP VP
%%      <S head>=<VP head>
%%      <VP head subject>=<NP head>
```

```
type(PhonIn,PhonOut) ==>
  cat: s &
  head: VPHead &
  true( type(PhonIn,PhonInt) &
        cat: np &
        head: NPHead
      ) &
  true( type(PhonInt,PhonOut) &
        cat: vp &
        head: ( VPHead &
                subject: NPHead )
      ).
```

```
%% VP -> V
%%      <VP head>=<V head>
```

```
type(PhonIn,PhonOut) ==>
  cat: vp &
  head: VHead &
  true( type(PhonIn,PhonOut) &
        cat: v &
        head: VHead
      ).
```

```
%% Word tiber
%%      <cat>=NP
%%      <head agr gender>=masc
%%      <head agr person>=third
%%      <head agr number>=sg
```

```

type(cons(tibor,X),X) ==>
  cat: np &
  head: agr: ( gender: masc &
              person: third &
              number: sg   ).

%% Word weint
%%   <cat>=V
%%   <head form>=finite
%%   <head subject agr person>=third
%%   <head subject agr number>=sg
type(cons(weint,X),X) ==>
  cat: v &
  head: ( form: finite &
          subject: agr: ( person: third &
                          number: sg   ) ).

cons(X,Y) ==>
  car: X &
  cdr: Y.

true(_) ==> _..

section signature.

atom s, np, vp, v,
     masc, third, sg,
     finite,
     tibor, weint.

cat, head, agr, gender, person, number, form, subject:: '$T$' -> '$T$'.

atom      nil.
primitive cons.
list      == cons ; nil.

car:: cons -> '$T$'.
cdr:: cons -> list.

```

References

- [AKP90] Hassan Ait-Kaci and Andreas Podelski. Is there a meaning to LIFE? Draft paper, Nov. 1990.
- [Car90] Bob Carpenter. Typed feature structures: Inheritance, (in)equality and extensionality. In *Proceedings of the Workshop on Inheritance in Natural Language Processing*, Tilburg University, The Netherlands, 1990.
- [DE91] Jochen Dörre and Andreas Eisele. A comprehensive unification-based grammar formalism. Deliverable R3.1.B, DYANA — ESPRIT Basic Research Action BR3175, 1991. to appear.
- [DR90] Jochen Dörre and William C. Rounds. On Subsumption and Semi-Unification in Feature Algebras. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 300–310, Philadelphia, PA., 1990.
- [EHMZ90] Martin Emele, Ulrich Heid, Stefan Momma, and Rémi Zajac. Organizing linguistic knowledge for multi-lingual generation. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki, Finland, 1990.
- [HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, October 1988. To appear in the *Journal of Logic Programming*.
- [KR86] Robert T. Kasper and William C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the ACL, Columbia University*, pages 257–265, New York, N.Y., 1986.
- [PS87] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics*. CSLI Lecture Notes 13. Center for the Study of Language and Information, Stanford University, 1987.
- [Rea89] Mike Reape. A logical treatment of semi-free word order and bounded discontinuous constituency. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, pages 103–110, Manchester, England, 1989.
- [She89] M. J. Shensa. A computational structure for the propositional calculus. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Detroit, Michigan, USA, 1989.

- [Smo88] Gert Smolka. A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, May 1988. To appear in the Journal of Automated Reasoning.
- [Smo89] Gert Smolka. Feature constraint logics for unification grammars. IWBS Report 93, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, November 1989. To appear in the Proceedings of the Workshop on Unification Formalisms—Syntax, Semantics and Implementation, Titisee, The MIT Press, 1990.
- [ZE90] Rémi Zajac and Martin Emele. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki, Finland, 1990.