# CUF – A Formalism for Linguistic Knowledge Representation

Jochen Dörre and Michael Dorna

Institut für maschinelle Sprachverarbeitung
Universität Stuttgart
D–W–7000 Stuttgart 1
email: {jochen,michel}@ims.uni-stuttgart.de

**Abstract**

We describe the formalism CUF (Comprehensive Unification Formalism), which has been designed as a tool for writing and making computational use of any kind of linguistic description ranging from phonology to pragmatics. The motivations for its major design decisions are discussed.

CUF is an offspring of the line of theory-neutral universal grammar formalisms like PATR-II [SUP+83] and STUF-II [BKU88, Dör91, DR91]. Like these it is based on defining feature structures and relations over these as encodings of linguistic principles and data. However, it is radically more expressive, since it allows the definition of arbitrary recursive relational dependencies without tying recursion to phrase structure rules. Complex restrictions needed in semantic interpretation like anaphora resolution and presupposition checking can thus be stated in the same description language as the syntactic restrictions, providing the basis for highly integrated linguistic processing.

CUF can be roughly characterized as a feature structure description language similar to Kasper/Rounds logic [KR90] combined with the possibility of stating definite clauses over feature terms. Moreover, feature structures are typed, with the types possibly being ordered in a hierarchy. The CUF type discipline allows for an axiomatic statement of global restrictions on the structures in which the program is to be interpreted providing enough redundancy in the descriptions to detect mistakes without burdening the grammar writer with tedious repetitions.

Although it is useful to think of CUF as a kind of pure (typed) Prolog in which first-order terms have been replaced by feature terms, there are two important respects in which this analogy is misleading. First of all, CUF makes a clear distinction between the purely declarative logical specification and the control statements which are used to guide the proof procedure without compromising the logical semantics of the specification. Second, we use a syntax especially well suited for a direct description of feature structures.

The connection to logic programming can be explored even further by taking into account that CUF is an instance of constraint-logic programming (CLP) of the very general Höhfeld/Smolka scheme [HS88]. This provides us not only with a sound and complete proof procedure, but also equips us with the right paradigm to attack the efficiency problems associated with highly modular specifications, as for instance proposed by GB theory.

## 1 Introduction

The formalism CUF (Comprehensive Unification Formalism) is one of the results of the ESPRIT project DYANA.[1] One major aim of the project was to establish the formal foundations of grammar formalisms which would cover as many as possible of the extensions to simple unification formalisms that had been proposed in the linguistic literature, especially in the fields of Head-Driven Phrase

[1]ESPRIT Basic Research Action 3175: Dynamic Interpretation of Natural Language

Structure Grammar (HPSG, [PS87] and Lexical-Functional Grammar (LFG, [KB82]). Hence, the name CUF. These extensions included formal devices like disjunction, negation, typing feature structures, operators for set or list manipulation, constraints to express functional uncertainty, subsumption relations, or even arbitrary functional and relational dependencies in features structures as well as nonmonotonic devices. [DEW$^+$90] gives a catalog of these potential extensions, describing their linguistic relevance as well as the technical difficulties of their integration.

## 1.1 Relational Dependencies

The formalism that was designed as a result of this first study, however, departed radically from the view of adding more and more "extensions" to a PATR-II-like base formalism [DE91]. Instead the reducibility of most of the above mentioned extensions to a very general form of relational dependency was exploited, resulting in a very simple though very expressive specification language. We can think of these relational dependencies as a generalization of the well-known template mechanism of PATR-II in two ways. Firstly, we can add parameters to templates as in

```
TRANSITIVE(VFORM) := ...
```

Next, we stop regarding templates merely as macro notation, but instead allow them their own ontological status. Thus we regard a "template call" like any other device to describe feature structures as a unary predicate denoting the set of described structures. Note that it now makes sense to have "templates" with recursive definitions as in:

```
vcomp_star(X) := X ; (vcomp: vcomp_star(X)).
```

With this definition we can simulate a functional uncertainty constraint $f_1\, vcomp^* = f_2$ (the infinite disjunction $f_1 = f_2 \vee f_1 vcomp = f_2 \vee f_1 vcompvcomp = f_2 \vee \ldots$) by conjoining to the description of $f_1$ the term `vcomp_star(`$f_2$`)`. These generalized templates are called "sorts" in CUF.[2]

Another example is the 2-place parametric `append`, which given feature-structure encodings[3] of two lists "yields" the concatenation of these. If we use more than one definition, these are taken disjunctively as in PROLOG.

```
append(elist, L) := L.
append('F':F & 'R':R, L) := 'F':F & 'R':append(R,L).
```

Some remarks on syntax are in order here. Feature terms are constructed using boolean operators `&` (conjunction), `;` (disjunction), and `~` (negation), as well as feature selection (`<feature>: <feature term>`) over atomic forms of feature terms, which are atoms and types (written in lowercase) or variables (uppercase). Parametric sorts are feature terms as well, as are their arguments. Hence, we can use the term `'R':append(R,L)` to describe a feature structure with an `'R'`-feature, whose value has to be the concatenation of whatever structures are bound to `R` and `L`.

Now, an untyped CUF specification is just a collection of such definitions. There are no special descriptive devices like lexical entries or grammar rules, since CUF resembles the DCG formalism in treating grammar rules simply as a macro notation for definite clauses of a special form (see [DE91]).

[2] There are a lot of other formalisms (TFS, STUF-III, UD, TDL, ALE) that have taken the same step and have introduced means to define what we like to term general relational dependencies. The unary form is sometimes called "type". We refrained from using this notion, since types in logic or in programming languages are normally used to classify syntactic entities, whereas the CUF sorts are not intended to be used in that way. Actually, sorts are pretty useless for syntactic type checking, since it is in general undecidable whether a feature term denotes some (or only) structures of a given defined sort. Nevertheless, CUF supports real types, which structure the universe more coarsely than do sorts, but which are used for type checking and type inference (see below).

[3] We use the features `'F'` for first element and `'R'` for tail of the list, as well as the atom `elist` for the empty list.

### 1.2 Expressive Power

It should be clear by now that our formalism is as expressive as (pure) PROLOG, allowing the definition of arbitrary first-order predicates. However, this expressivity alone does not turn CUF into a programming language. Unlike PROLOG, CUF does not assume a fixed processing strategy. Instead, we want to regard CUF definitions as purely logical specifications that can be used to accomplish diverse computation tasks. For instance, in the case of grammatical descriptions, the definitions should be usable for parsing *and* for generation.

The actual *processing strategy* for each task has to be defined on a separate level. No extra-logical control constructs, like for instance the cut in PROLOG, are mixed into the logical part of the language. This strict separation of logic and control facilitates grammar development and maintenance.

### 1.3 Type Discipline

The philosophy behind CUF's type system is the following.

- Types are used to coarsely prestructure the universe of feature structures. This includes the (indirect) specification of a type hierarchy as well as the declaration of which features are compatible with which types and what the types of their respective values are, i.e. the declaration of which data structures are considered as legal.
- Typing is not obligatory, i.e., the grammar writer is free to add or omit type information which would support error-checking during compilation and may lead to short-cuts in the proof. The system tries to infer missing type information.
- Types are an alternative means of describing sets of feature structures, and complement the descriptive device of sorts by providing full negation on a propositional level. Types are thus just ordinary feature terms like sorts, with a logical semantics. No specific syntax for typing is needed.

The type system of CUF is described in detail in Section 2.3.

### 1.4 Acknowledgement

It should be mentioned that although CUF was developed and implemented on its own, its development was largely influenced not only by the existing STUF-II system, in the development of which the first author was involved, but also by the ideas that at the same time lead to the system STUF-III [Sei92]. These ideas have been worked out jointly with Roland Seiffert at IBM Germany [DS91].

## 2 CUF– The Language

This section describes the language constructs of the declarative specification language of CUF together with their logical semantics. For those not interested in the formal details we have tried to describe the semantics of CUF informally in the text while giving the precise formalization only in figures.

Fig. 1 presents an overview of all kinds of language constructs that can be used to compose a CUF program, including its control part, which is described in Section 3.

CUF program

logical part: CUF specification

typing information

- type (hierarchy) axioms
- feature declarations
- sort declarations

- clauses (defining sorts)

control part: CUF control statements (task dependent)

- delay patterns
- index declarations
- subgoal order declarations[a]

[a] not yet implemented

Figure 1: Possible Parts of a CUF Program

## 2.1 Feature Algebras and Feature Terms

In order to give an adequate formalization of CUF expressions, we need the notion of a *feature algebra*. This is only a slight generalization of the well-known concept of a feature structure [Shi86, KR86, KR90] by dropping the restriction that it has a single root, i.e. we are talking about general directed graphs whose edges have a "deterministic" labelling (no two edges leaving one node are labelled the same) and some of whose terminal nodes are labelled. Edge labels and terminal node labels come from the two alphabets $\mathcal{F}$ of *feature symbols* and $\mathcal{C}$ of *constant symbols*, respectively.

The central building block of our language is the notion of a feature term, which informally denotes a *set* of feature structures. Now, an interesting point to note about feature algebras is that there is a natural correspondence between a node and the structure of its extension, i.e. the subalgebra of all nodes which are reachable via feature paths from that node. Hence, instead of talking about sets of (single-rooted) feature structures as the denotations of feature terms, we can use a single feature algebra as a domain of interpretation and let feature terms denote subsets of this domain. This way of modelling is in fact essential if we want to express that parts of the structures denoted by different terms are actually shared.

Features and constants are primitive notions in CUF which cannot be defined to have a specific meaning. They play roles similar to term constructors and atoms in PROLOG. What corresponds to predicates are the *sorts* (alphabet $\mathcal{S}$) and the *types* (alphabet $\mathcal{T}$).

A type is a primitive form of feature term and is always interpreted as a *subset* of the interpretation feature algebra. Hence, by the above considerations, we can think of it as denoting some set of feature structures. As mentioned in the introduction, types are used to coarsely structure the universe of feature structures. We will see below that only certain sets of feature structures can be defined as

| | | |
|---|---|---|
| Alphabets: | $\mathcal{F}$ | feature symbols (letters `f`, `g`, `h`), including the symbols `'F'` and `'R'` |
| | $\mathcal{C}$ | constant symbols (letters `a`, `b`, `c`), including the empty list symbol `[]` |
| | $\mathcal{S}$ | sort symbols of any arity (letters `s`, arity function $ar$) |
| | $\mathcal{T}$ | type symbols (letters `t`), including `top, bottom, afs, cfs, list, nelist, elist` |
| | $VAR$ | variables (letters `X`, `Y`, `Z`) |

In order to keep the semantic notions simple, we assume that sorts and types are first mapped to predicates before getting an interpretation, i.e. with each $n$-ary sort symbol `s` we associate a predicate $p_s$ of arity $n+1$ and with each type symbol `t` a unary predicate $p_t$. The combined set of these predicate symbols will be denoted by $\mathcal{P}$.

Figure 2: Symbol Sets

**Definition 1 (Feature Algebra)** *A feature algebra $A$ is a pair $(\mathcal{D}^{\mathcal{A}}, \cdot^{\mathcal{A}})$ consisting of the domain (or carrier) $\mathcal{D}^{\mathcal{A}}$, which is a nonempty set, and the interpretation function $\cdot^{\mathcal{A}}$ defined on $\mathcal{F}$ and $\mathcal{C}$ such that:*

- *$f^{\mathcal{A}}$ is a unary partial function on $\mathcal{D}^{\mathcal{A}}$ for each feature symbol $f$;*
- *$a^{\mathcal{A}} \in \mathcal{D}^{\mathcal{A}}$ for each constant $a \in \mathcal{C}$;*
- *if $a \neq b$ then $a^{\mathcal{A}} \neq b^{\mathcal{A}}$ $(a, b \in \mathcal{C})$;*
- *no feature is defined on a constant.*

**Definition 2 (Extended Feature Algebra)** *An extended feature algebra is a feature algebra $(\mathcal{D}^{\mathcal{A}}, \cdot^{\mathcal{A}})$ whose interpretation function is also defined on $\mathcal{P}$ such that:*

- *$p^{\mathcal{A}}$ is a nonempty subset of $(\mathcal{D}^{\mathcal{A}})^k$ for each $p \in \mathcal{P}\backslash\{p_{bottom}\}$ of arity $k$;*
- *$p^{\mathcal{A}}_{top} = \mathcal{D}^{\mathcal{A}}$; $p^{\mathcal{A}}_{bottom} = \emptyset$.*

Figure 3: Definitions: Feature Algebra, Extended Feature Algebra

types.

Sorts may have feature terms as arguments, in which case they are called parametric. An $n$-place sort denotes an $n+1$-place relation over the nodes of the interpretation feature algebra, which again we will consider informally as an $n+1$-place relation over feature structures. The additional parameter comes from the fact that a sort expression $\mathtt{s}(t_1, \ldots, t_n)$ will denote a set like any other feature term, i.e. sorts are notated in a functional style. We will make this more precise in the examples below.

*Variables* denote single objects (nodes) of the interpretation feature algebra, i.e. are first-order. They can only occur in clauses and are always interpreted local to a clause.

The symbols of types, constants, sorts, features and variables are required to be pairwise disjoint.

Now that all basic concepts have been introduced we can proceed to give the syntax and formal semantics of feature terms. The syntax specification in Fig. 4 is to be read as a context-free production, in which $s$ and $t$ (sometimes with indices) denote feature terms.[4]

[4]Parentheses ("(", ")") can be used too. In order to avoid parentheses we assume the following operator precedences (highest to lowest): `:` `~` `&` `;` `|` `=` `<` .

| Form | | | Significance | Denotation $[\![\cdot]\!]^{\mathcal{A}}_{\alpha}$ |
|---|---|---|---|---|
| $s,t$ | $\rightarrow$ | $\mathtt{s}(t_1,\ldots,t_n)$ | sort symbol $\mathtt{s}/n$ | $\{d_0 \mid \langle d_0,\ldots,d_n\rangle \in p^{\mathcal{A}}_s \wedge \bigwedge_{i=1}^{n} d_i \in [\![t_i]\!]^{\mathcal{A}}_{\alpha}\}$ |
| | $\mid$ | `t` | type symbol | $p^{\mathcal{A}}_t$ |
| | $\mid$ | `c` | constant symbol | $\{c^{\mathcal{A}}\}$ |
| | $\mid$ | `X` | variable symbol | $\{\alpha(\mathtt{X})\}$ |
| | $\mid$ | $\mathtt{f}\!:\!t$ | selection | $\{d \in \mathcal{D}^{\mathcal{A}} \mid f^{\mathcal{A}}(d) \in [\![t]\!]^{\mathcal{A}}_{\alpha}\}$ |
| | $\mid$ | $s$ & $t$ | conjunction (intersection) | $[\![s]\!]^{\mathcal{A}}_{\alpha} \cap [\![t]\!]^{\mathcal{A}}_{\alpha}$ |
| | $\mid$ | $s$ ; $t$ | disjunction (union) | $[\![s]\!]^{\mathcal{A}}_{\alpha} \cup [\![t]\!]^{\mathcal{A}}_{\alpha}$ |
| | $\mid$ | ~$s$ | negation (complement) | $\mathcal{D}^{\mathcal{A}} \backslash [\![s]\!]^{\mathcal{A}}_{\alpha}$ |

Figure 4: Syntax and Semantics of Feature Terms

We have to add two remarks on the syntax of feature terms. First, the use of negation is restricted: it is not allowed to use a sort in the scope of a negation. This is a consequence of restricting our clauses (defined later) to be definite. Second, some further feature terms are lists and strings in the syntax of Edinburgh PROLOG, of which the latter will be treated as constants.[5]

The denotations given in Fig. 4 are interpreted relative to the interpretation feature algebra $\mathcal{A}$ (an extended algebra) and a variable assignment $\alpha$, which maps variables to elements of $\mathcal{D}^{\mathcal{A}}$. It should also be noted that although constants and variables pick out single objects in the interpretation feature algebra $\mathcal{A}$, we let their occurrences in feature terms always denote (singleton) sets in order to have a uniform syntax.

**Example 1** Saturated phrases of HPSG ([PS93]) can be described by a CUF feature term `synsem:loc:cat:subcat:[]`. □

**Example 2** To require two variables `X` and `Y` to denote the same object we can write `X & Y`. To express inequality we write `X & ~Y` or `~X & Y`. Hence, a path inequation $\langle f\rangle \neq \langle g\rangle$, denoting structures in which the root's $f$-path and the root's $g$-path lead to different nodes, can be expressed by the feature term `f:X & g:~X`. □

**Example 3** A variable `X` will be restricted to a type `type` by the conjunction `X & type`. This can be useful in a feature term where we want to restrict parameters of a sort or restrict the range of a feature to certain types:

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

In this example the type `nelist` for non-empty lists restricts the denotation of both parameters of the sort `append/2` and also the values of the features `f` and `h`. □

[5]The list constructor `[ | ]` maps to a feature term with the features 'F' and 'R'. Thus, the second clause of `append` on page 4 could equivalently be written `append([F|R],L) := [F|append(R,L)]`. As in PROLOG `[a,b,c,...]` abbreviates `[a|[b|[c|...|[]...]]]`. Strings are notated by enclosing characters in quotation marks (").

## 2.2 Clauses

A **sort definition** consists of one or more clauses of the form

$$\mathtt{s}(t_1,\ \ldots, t_n)\ \mathtt{:=}\ t_0\mathtt{.}$$

where s is sort of rank $n$ and the $t_i$ are feature terms. Such a clause states that the denotation of the given sort expression (left-hand side) covers at least the denotation of $t_0$ (for each variable assignment). What this means precisely for the associated predicate $p_s$ of s is expressed in the following conditional (where $\alpha$ is a variable assignment).

$$\langle d_0, \ldots, d_n\rangle \in p_s^{\mathcal{A}} \quad \mathbf{if} \quad \exists\alpha\ (d_0 \in [\![t_0]\!]_\alpha^{\mathcal{A}} \wedge \ldots \wedge d_n \in [\![t_n]\!]_\alpha^{\mathcal{A}})$$

**Example 4** The *Head Feature Principle*[6] of HPSG can be implemented as:

```
head_feature_principle :=
        synsem:loc:cat:head: Head &
        dtrs:head_dtr:synsem:loc:cat:head: Head.
```

The example illustrates that path equalities can be expressed in CUF in terms of variable sharings of path values. □

**Example 5** A list of elements of type synsem can be defined as:

```
list_of_synsem := [].
list_of_synsem := [synsem|list_of_synsem].
```

**Remark:** A more general definition like

```
list_of(X) := [].
list_of(X) := [X|list_of(X)].
```

is also a possibility for defining lists of certain elements of one type. This, however, is pretty useless in most cases, since because of the logical variable X all elements of this list have to be identical! To get the intended effect, X would have to be a variable over types, i.e. a second-order variable. The current CUF system does not support higher order variables. □

## 2.3 Type Information

CUF allows for the specification of a type hierarchy through a set of propositional type axioms. Types can then be used to restrict the interpretation of features as well as parametric sorts. This is done using feature declarations, which declare domain and range types of a feature, and sort declarations, which declare argument and result types for sorts.

| | | | |
|---|---|---|---|
| $s,t$ | $\rightarrow$ | `t` | type symbol |
| | $\mid$ | `c` | constant symbol |
| | $\mid$ | $s$ `&` $t$ | conjunction (intersection) |
| | $\mid$ | $s$ `;` $t$ | disjunction (union) |
| | $\mid$ | `~`$s$ | negation (complement) |

Figure 5: Syntax of Type Terms in CUF

### 2.3.1 Type Axioms

Type axioms are essentially boolean expressions over type and constant symbols.[7] Fig. 5 presents the syntax of **type terms**. Here $s$ and $t$ are type terms. Notice that these are simply feature terms without variables, sorts or features. The formal semantics is as above.

There is also some additional syntactic saccharine to allow a more compact notation for type axioms. First of all the equality operator `=` (corresponding to equivalence) and the subtype operator `<` (corresponding to implication) are supported. Both can be eliminated in the usual way ($s$ `<` $t$ means `~`$s$ `;` $t$). Another notational device are disjointness constraints

$$t_1 \mid \ \ldots \mid t_n.$$

These may appear by themselves as axioms or as the right-hand side of an equality axiom $s = t_1 \mid \ \ldots \mid t_n$ or as the left-hand side of a subtyping axiom $t_1 \mid \ \ldots \mid t_n < t$. They are used to express the pairwise disjointness of the named types. The isolated form is considered as a short form of the conjunction over all formulas $\tilde{}(t_i \,\&\, t_j)$ such that $1 \leq i < j \leq n$, i.e., all pairs of different $t_i$ and $t_j$ have to have empty intersections. The other two forms, where disjointness constraints are embedded, express the conjunction of the disjointness — seen as an isolated constraint — with the original formula in which the operator '`|`' is replaced by '`;`' (see Fig. 6).[8]

$$\begin{array}{rcl} t_1 \mid \ \ldots \mid t_n & \equiv & \bigwedge_{1\leq i<j\leq n} \tilde{}(t_i \,\&\, t_j) \\ s = t_1 \mid \ \ldots \mid t_n & \equiv & (s = t_1 \,;\ \ldots \,;\ t_n) \wedge \bigwedge_{1\leq i<j\leq n} \tilde{}(t_i \,\&\, t_j) \\ t_1 \mid \ \ldots \mid t_n < t & \equiv & (t_1 \,;\ \ldots \,;\ t_n < t) \wedge \bigwedge_{1\leq i<j\leq n} \tilde{}(t_i \,\&\, t_j) \end{array}$$

Figure 6: Interpretation of Disjointness Constraints

By using these transformations all kinds of type axioms can be expressed as type terms. So the formalization of the semantics can be restricted to these cases.

As a side effect, the usage of a symbol in a type axiom classifies (defines) it as a type or constant.

[6]This principle encodes the percolation of head attributes along the head projection line.

[7]Actually, what we call constants in the syntax of our language is a special form of type, which more accurately could be termed constant type. Nevertheless, we will continue to call them constants and only if it is not clear from the context we shall describe whether the singleton set or the element itself is meant. The system distinguishes constant and type symbols by requiring that constants are written in enumeration braces like `{a,b,c}` when appearing in a type axiom. Such an enumeration is nothing else than a disjunction. This special syntactic form is needed, because symbols in type axioms are considered by default as (non-constant) types.

[8]Notice that the disjointness operator is *not* a binary logical connective. Instead it can be regarded as logically denoting `;` (OR) with a side effect of asserting additional axioms, which actually enforce the disjointness of the types in such a group. Even if we restricted disjointness constraints to pairs of types the meaning of $s = t_1 \mid t_2$ is different from the meaning of $s = t_1$ `xor` $t_2$, with `xor` being the logical exclusive-or connective. The latter merely require $s$ to denote the same as $(t_1 \,;\ t_2)$ `&` `~`$(t_1$ `&` $t_2)$.

**Example 6** In HPSG ([PS93]) there are two disjoint types of signs. So, we simply write

`sign` $=$ `phrase` $|$ `word`.

to partition the universe into phrasal and lexical signs. The following declaration has a similar effect:

```
phrasal < sign.
word < sign.
word | phrase.
```

The last axiom denotes the disjointness of lexical and phrasal signs too. But the subtype axioms are not exclusive. There can be elements which are signs but neither phrases nor words, i.e., the term `sign & ~phrase & ~word`, which would be inconsistent when using the equation above, is satisfiable here. □

**Example 7** To define an agreement type for a German grammar we can describe a cross classification of person, number and case attributes in the following way:

```
agreement = person & number & case.
person = first | second | third.
number = plural | singular.
case = nom | dat | acc | gen.
```

**Remark 1:** Observe that we are not using features here. These types can be used to classify whole feature structures, for instance, as `nom & third & singular`, thereby excluding the other possibilities. We can even include dependencies between certain combinations of types as in `gender < third & singular`, which could be used to express that only third singular forms are subclassified for gender.

**Remark 2:** It should be mentioned that the first axiom is *not* necessary to guarantee the compatibility of the types `person`, `number`, and `case`, like in some other systems. In CUF any interpretation of the types which is not invalidated by the axioms is possible. Hence, in the absence of other information, intersections of types are not considered to be empty. □

**Example 8** The two values of a boolean type can be defined by `boolean = {-,+}`. Further we can define the values of specific features to be certain enumeration types such as `person = {1, 2, 3}` or `case = {nom, acc, dat, gen}`. Of course, the features using these ranges have to be defined too. □

Sometimes it takes considerable effort to declare all constants in a description. Therefore we have included an automatic constant classification for all non-parametric non-defined symbols used in clauses. Furthermore, we allow the use of strings which are classified as constants (see section 2.3.2).

Finally a remark on conjunction of feature terms containing constants: according to our logic the conjunction of a constant and a selection as well as the conjunction of two non-identical constants is inconsistent. The compiler detects these immediate inconsistencies and reports a type error (*constant-complex clash* and *constant-constant clash*, respectively).

### 2.3.2 Basic Structure of the Universe

The following **built-in types** predefine the structure of the universe: `top` (the universe), `bottom` (the empty denotation), `afs` (atomic feature structures), `cfs` (complex feature structures), `list` (lists), `elist` (the empty list), `nelist` (non-empty lists), `string` (strings), and the denotation of the empty list, the constant `[]`.[9] The interpretation of `top` and `bottom` is fixed as the universe and the empty set. The following axioms define the relations between the other built-in types in the notation introduced above:

```
top = afs | cfs.
list = elist | nelist.
elist = { [] }.
string < afs.
```

Additionally each constant type automatically becomes a subtype of `afs` and each type used as a domain of some feature (see below) becomes a subtype of `cfs`.

The features `'F'` and `'R'` define a list structure. They come with the following built-in domain-range restrictions:

`'F'`: `nelist` $\rightarrow$ `top` and
`'R'`: `nelist` $\rightarrow$ `list`.

They are used as internal selectors of the first element and the tail of a list, respectively. Since features are defined on `nelist`, this type automatically becomes a subtype of `cfs`.

In general it is not allowed to use these built-in types in type axioms, because this can change the structure of the universe, i.e. the logic. The only exception is the usage of a built-in type as a supertype, because this cannot change the denotation of the supertype. For the same reason it is not allowed to use a built-in type as a domain for a user-defined feature.

**Example 9** If we did not restrict the possible domains for features, we could define a feature `dummy`: `top` $\rightarrow$ `top`. By this definition we would describe a logic containing no constants. (They would all cause type errors.) □

### 2.3.3 Feature Declarations

Declarations of features are bundled with respect to a common domain type. They have the following format:

$$t_0 \;\; :: \;\; \mathtt{f}_1 : t_1 \, , \; \mathtt{f}_2 : t_2 \, , \; \ldots \, , \; \mathtt{f}_n : t_n \, .$$

where $t_i$ is any type term and $\mathtt{f}_i$ is any feature symbol. By this declaration each feature $\mathtt{f}_i$ gets a domain $t_0$ and a range $t_i$.

**Example 10** A definition of the HPSG type "sign" and its features has the form

```
sign = word | phrase.

sign   :: phon:   nelist,
          synsem: synsem,
          qstore: list.
phrase :: dtrs:   constituent_structure.
```

[9]Notice that our syntax allows the use of `elist` and `[]` interchangeably in the clauses part, whereas in the type information part `[]` is written in braces.

**Remark:** In the original HPSG theory ([PS93]) the range of feature qstore is a set. Currently CUF does not support sets, so we have to simulate a set in the form of a list. □

By using the same feature symbol in different declarations we may get a form of polymorphism which we call **feature polymorphism**. Such a feature is called a **polyfeature**.

During grammar development it is often convenient not to have to define all features. We therefore allow undefined features in clauses. If the CUF system finds an undefined feature in a clause, it will generate a warning and assume a standard definition. This standard definition does not restrict the domain and the range in any way.

### 2.3.4 Sort Declarations

A sort s of rank $n$ can be restricted by a declaration to have arguments of a certain type. Sort declarations have the form

$$\texttt{s(}t_1\texttt{, ..., }t_n\texttt{) -> }t_0\texttt{.}$$

where $t_i$ are type terms. $t_0$ is the type of the (implicit) result argument of the corresponding relation.

**Example 11** The sort append/2 can be declared and defined in the following manner:

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

**Remark:** The sort declaration here really restricts the possible usages of append/2. To see this assume the feature term append([], atom) without any declaration of append/2. By virtue of the first clause the denotation (the instantiation of the result argument) would be "atom". With the declaration above this term will cause a type error since its denotation would be empty. □

### 2.3.5 Semantics of Type Information

A type system of a CUF specification, i.e., a set of type axioms, feature declarations, and sort declarations, has a logical interpretation and contributes to the semantics of the whole program in just the same way as the clauses do, namely by restricting the possible models. However, contrary to the clauses defining sorts, it is decidable for a type system whether a given feature term is satisfiable with respect to it or not. This fact is used during compilation, when, after having checked the consistency of the type system itself, each feature term of a clause is tested for satisfiability with respect to it. Unsatisfiable feature terms indicate a type error.

Actually, the logic used for the type system is only slightly more powerful than propositional logic. In order to support this claim, we present here a formalization of the type logic which is a variant of propositional modal logic.[10]

First, observe that feature algebras can be seen as a special kind of Kripke structures in which worlds are the nodes and accessibility is given by the feature transitions between nodes.[11] However, there is not only this semantic correspondence, which we will have a closer look at immediately, but

[10] For modal treatments of feature logics see also [Bla92, BS92, Rea91].

[11] More precisely, since there is more than one feature and features are partial functional, we are talking about *partial functional multiframes.*

also syntactically a feature selection acts like a modal operator. We regard a formula $f\!:\!\phi$, where $\phi$ is some feature term, as true at a node of the feature algebra if it has an $f$-successor and $\phi$ is true at that node. Features are sentence operators with a semantics completely analogous to the modal possibility $\diamond$.

Let us consider the modal logic over the propositional variables $\mathcal{T}$ (the types) and the possibility operators $\mathcal{F}$ (the features). To emphasize the modal interpretation of features we shall write $\langle f\rangle\phi$ in the place of $f\!:\!\phi$. Fig. 7 presents the interpretation structure as well as the satisfaction relation $\mathcal{M} \models_n \phi$ (read: $\phi$ is true in $\mathcal{M}$ at node $n$).

$$
\begin{array}{lcl}
\text{Structure } \mathcal{M} & = & (N, \{R_f \mid f \in \mathcal{F}\}, \{V_t \mid t \in \mathcal{T}\}) \\
 & & \quad \text{where} \quad N \subseteq \mathbb{N} \\
 & & \qquad\qquad\quad R_f \subset N \times N \text{ is a partial function for each } f \\
 & & \qquad\qquad\quad V_t \subseteq N \text{ for each } t \\
\mathcal{M} \models_n t & \text{iff} & n \in \mathcal{V}_t \\
\mathcal{M} \models_n \langle f\rangle\phi & \text{iff} & \exists n : nR_f n' \wedge \mathcal{M} \models_{n'} \phi \\
\mathcal{M} \models_n \phi\&\psi & \text{iff} & \mathcal{M} \models_n \phi \ \wedge\ \mathcal{M} \models_n \psi \\
\mathcal{M} \models_n \tilde{}\phi & \text{iff} & \mathcal{M} \not\models_n \phi
\end{array}
$$

Figure 7: Deterministic Propositional Modal Logic

Now, the purely propositional type axioms state general relationships between types that hold at every node of the structure, i.e., they have to be *valid* (true at every node) in a model. Recall, that besides the user-defined type axioms there are type axioms for built-in types as given in Section 2.3.2.

To express the meaning of feature declarations in this logic, we have to consider all domain-range pairs that are defined for a given feature. Say, $f : D_1 \mapsto R_1, \ldots f : D_n \mapsto R_n$ are the domain-range restrictions for $f$, then the logic underlying these restrictions is given by the axioms

$$
\begin{array}{l}
\langle f\rangle\texttt{top} \leftrightarrow \bigvee_{i=1}^{n} D_i \\
\bigvee_{i=1}^{n} D_i \rightarrow \texttt{cfs} \\
D_1 \rightarrow \langle f\rangle R_1 \\
\ldots \\
D_n \rightarrow \langle f\rangle R_n
\end{array}
$$

The first axiom states that $D_f = \bigvee_{i=1}^{n} D_i$ is exactly the domain of $f$, which implies that $f$ is total on this subset of the universe.[12] The second axiom takes care of the fact that elements of $D_f$ are classified as complex feature structures, i.e., they cannot be constants. The other axioms reflect the fact that specific domains come together with specific ranges.

In order to check consistency of the propositional axioms we employ a propositional theorem prover. It is interesting to consider what has to be done additionally to guarantee consistency of the system including the axioms containing the feature modalities. First, observe that for a feature with only one domain-range restriction we actually have to do nothing. We can just interpret the feature such that its domain is $D_1$ and its range is $R_1$. This trivially validates the two axioms $\langle f\rangle\texttt{top} \leftrightarrow D_1$ and $D_1 \rightarrow \langle f\rangle R_1$. However, when using polymorphic features inconsistencies due to feature declarations

[12]Note that our choice of interpreting the feature declarations such that $f$ has to be total on $D_f$ is no actual restriction for the type language. Suppose we wanted to state a feature declaration $f : D \mapsto R$ without the requirement that $f$ is defined everywhere in $D$, then we simply write `D' < D` and `D' :: f:R` letting the new symbol `D'` denote the subset of `D` where `f` is total.

may result, as is illustrated in the next example. In the section on the implementation below we will describe a method with which the information about domain-range restrictions can be used to calculate new type axioms, such that consistency of pure type terms with respect to the whole axiom system (including those induced by feature declarations) can be checked by only looking at the propositional ones.

**Example 12** Let $f : D_1 \mapsto R_1$ and $f : D_2 \mapsto R_2$ be two domain-range restrictions for $f$ and let $R_1 \mid R_2$ be an axiom. Now, we can conclude $D_1$ & $D_2 \Rightarrow \langle f\rangle(R_1$ & $R_2) \Rightarrow \langle f\rangle\texttt{bottom} \Rightarrow \texttt{bottom}$. This conclusion could not be drawn without the axioms with the feature modalities. □

A sort declaration $\mathtt{s}(t_1, \ldots, t_n) - > t_0$ cannot be expressed in this logic. The corresponding axiom in first-order logic is

$$\forall x_0 \ldots x_n : \ \mathtt{s}(x_0, \ldots, x_n) \rightarrow t_0(x_0) \wedge \ldots \wedge t_n(x_n),$$

where the $t_i(x_i)$ are the corresponding first-order translations for the modal formulae $t_i$.

As mentioned earlier, welltypedness of a feature term now simply means satisfiability together with all axioms of the type system.

#### 2.3.6 Types vs. Sorts

Ontologically, types and argumentless sorts are the same. Both denote subsets of the universe of feature structures. Also, both can be defined to have certain features and to have certain relationships to other types/sorts. There are, however, two important differences in the definability of these two kinds of feature terms which are also an explanation for their different procedural treatment. Sorts may employ variables in their definitions, types don't. This causes types to be insensitive to the graph structures of their elements. Actually, whenever some feature graph is an element of a type, then so is its tree unfolding. In other words, types can only differentiate between feature structures that differ in some path values or in the definedness of some paths, whereas sorts can require certain path equations or inequations to hold, i.e., they can describe structural coreferences. On the other hand types can be defined using full propositional logic, including full negation, whereas sorts are defined through definite clauses.

Procedurally, types are handled in the unification part of the resolution algorithm, i.e., after each step, type consistency is checked. Sorts, however, drive the resolution in general with their definite clauses.

## 3 CUF – The Implementation

The current CUF system (Version 2.25) consists of a compiler, an interpreter and graphical interface with a debugger, i.e. a development tool for CUF descriptions. The implementation was done in Quintus PROLOG and C under UNIX and X. The named parts of the system are presented in this section schematically. Furthermore, we explain how the declarative CUF specifications are treated procedurally.

### 3.1 Compiler

The compilation steps are presented in a schematic way because we think things are getting clearer with a little abstraction from details. E.g., in practice it is not always necessary to do all the compilation steps mentioned below for the compilation of a CUF description.

The compilation of a CUF description can be divided into the following parts:

- **File Handling**: file access; scanning and parsing; symbol table management; dependency checking between already compiled files and new ones using occurrence tables;
- **Translation of Type Information**: translation of type axioms, feature and sort declarations; treatment of polyfeatures; building of the axiom system;
- **Clause Translations**: negation transformations; disjunction extraction; clause compilation; code optimization.

A further part is the compilation of interpreter control statements which will be discussed in Section 3.2.

### 3.1.1 File Handling

The incremental compiler treats several physical files of CUF descriptions in a quasi-parallel manner as if it were only one logical file. If a file was changed and should be compiled, "dependent" parts of previously compiled files will be recompiled. This is necessary because parts like feature or sort declarations are represented in the internal structures generated during the compilation. So if some of these declarations were changed, the parts in which they were used have to be recompiled.

This recompilation step does not start from scratch. It suffices to start with a scanned and parsed internal representation of the CUF description code. So the file access is restricted to the compilation of a new file.

The symbol table makes sure that the symbols used in several parts of a CUF description, e.g. in several files, have always the same usage, i.e. that the sets of constant, type and sort symbols are pairwise disjoint. E.g. a type symbol used in a type axiom must not be defined as a sort.

### 3.1.2 Translation of Type Information

This part of the compilation treats type axioms as well as feature and sort declarations. It prepares these constructs for the compilation of clauses, i.e. some of the gained information will be added to the internal clause representation.

Starting with type axioms the compiler builds incrementally an axiom system which will be used for type unification, i.e. for testing satisfiability of conjunctions of type terms. Feature declarations as well as sort declarations are translated into declaration tables. During the translation of feature declarations the axiom system is extended by axioms stating that each domain is a subtype of `cfs`. Next, the compiler classifies each feature into the class of either monofeatures or polyfeatures. A monofeature has only one declaration, and a polyfeature has at least two of them. Under certain conditions polyfeatures cause the addition of new axioms, which we explain in the following.

We have defined features as total functions over the union of the declared domains. So we have to make sure that if there exists a non-empty intersection between some domains, the feature should map into the intersection of the corresponding ranges. This is only possible if the intersection of the ranges is also non-empty. On the other hand, if the intersection of the ranges is the empty set, the domains must be disjoint too.

This is the basic idea of the algorithm which treats each feature and may increase the axiom system dynamically. The procedure takes all $i$-tuples of domain-range pairs of each feature, respectively, where $i \leq n$ and $n$ is the number of declarations for that feature. If an $i$-tuple of ranges has no intersection, an axiom will be added to the axiom system stating that the corresponding domains must have no intersection. Since other range intersections could be affected by this new axiom, the algorithm has to compute a kind of closure over these additions, proceeding until no axiom can be added any more.

After a successful treatment of all polyfeature domain-range pairs the axiom system guarantees that if there exists an intersection of the declared domains, then the intersection of corresponding ranges is non-empty in at least some interpretation. Now the axiom system is complete, and each type and constant is tested whether it has an empty denotation. In this case a type error will be reported.

### 3.1.3 Sort Translation

The last steps were primarily preparation steps for the clause compilation. Now, the global type constraints will be added to restrict the denotation of feature terms. The main effect is that the type information in declarations will be propagated by unification into the internal representation of feature structures.

The first part of clause compilation transforms negated feature terms by pushing negation down into the term structure as far as possible. The equivalences used for transformation are shown in Fig. 8. We should highlight the elegant treatment of negated selections. Because a feature `f` is a

| | | |
|---|---|---|
| `~cfs` | $\equiv$ | `afs` |
| `~afs` | $\equiv$ | `cfs` |
| `~(f:`$t$`)` | $\equiv$ | `~`$D_f$ `; f:~`$t$ |
| `~(`$s$ `&` $t$`)` | $\equiv$ | `~`$s$ `; ~`$t$ |
| `~(`$s$ `;` $t$`)` | $\equiv$ | `~`$s$ `& ~`$t$ |
| `~~`$t$ | $\equiv$ | $t$ |

Figure 8: Equivalence Transformations of Negated Feature Terms

total function on its domain $D_f$, the handling of negation is simplified a lot. `~`$D_f$ is a propositional formula where $D_f$ is the disjunction of all declared domains of `f`. This part of the disjunction denotes the part of the universe where `f` is not defined. The other part, on which `f` must be defined, will be treated recursively by pushing down the negation in front of the value `t` of `f`. The equivalence transformations produce a normal form in which only constants, types, and variables are negated. So the interpreter can be restricted to handle these negation cases only.

The second part of the clause translation adds the "result parameter" to the sort to produce a predicate (see Section 2.2). The translation function for feature terms is defined in Fig. 9. The input of the translation is a "result variable" and a feature term which should be translated. The output is a subgoal list. The type information is used for checking domains and ranges of features and parameters of sorts.

The equalities "=" are implicitly handled by the internal unification of the typed feature structures which are built from the equalities for each variable at compile time. The disjunctions $\mathtt{d}_i$, the polyfeature constraints $\mathtt{fdecl}_i$ and the inequalities "$\neq$" remain as goals in the subgoal list, i.e., they are delayed at this moment.

A definite clause will be compiled of a CUF clause $\mathtt{a}(t_1,\ldots,t_n)$ `:=` $t_0$ in the following way:

$$\mathtt{a}(\mathtt{X}_0,\ \mathtt{X}_1,\ldots,\mathtt{X}_n) \leftarrow \mathbf{trans}(\mathtt{X}_0,t_0\ \&\ D_0),\ \mathbf{trans}(\mathtt{X}_1,t_1\ \&\ D_1),\ \ldots,\ \mathbf{trans}(\mathtt{X}_n,t_n\ \&\ D_n).$$

with new variable symbols $\mathtt{X}_0, \ldots, \mathtt{X}_n$ and $\mathtt{a}(D_1,\ldots,D_n)\text{->}D_0$. As in Fig. 9 we use the sort declarations in this translation.

The translated clauses have the form $h \leftarrow C, B$, where $h$ is a relational atom which contains the system internal representation of feature and type constraints in its parameters, $B$ is a list of such relational atoms, and $C$ is a list of unresolved inequalities and polyfeature constraints which are treated separately from relational atoms.

| | | |
|---|---|---|
| $\mathbf{trans}(\mathtt{X}_0, \mathtt{a}(t_1,\ldots,t_n))$ | $:=$ | $\mathtt{a}(\mathtt{X}_0, \mathtt{X}_1,\ldots,\mathtt{X}_n)$, $\mathbf{trans}(\mathtt{X}_0, D_0)$, $\mathbf{trans}(\mathtt{X}_1, t_1\ \&\ D_1), \ldots, \mathbf{trans}(\mathtt{X}_n, t_n\ \&\ D_n)$ with new variable symbols $\mathtt{X}_i$ where $i > 0$, and $\mathtt{a}(D_1,\ldots,D_n)\mathtt{->}D_0$. |
| $\mathbf{trans}(\mathtt{X}, \mathtt{c})$ | $:=$ | $\mathtt{X} = c$. |
| $\mathbf{trans}(\mathtt{X}, \mathtt{t})$ | $:=$ | $\mathtt{X} = t$. |
| $\mathbf{trans}(\mathtt{X}, \mathtt{Y})$ | $:=$ | $\mathtt{X} = \mathtt{Y}$. |
| $\mathbf{trans}(\mathtt{X}, \mathtt{f}{:}t)$ | $:=$ | $(i)$ if $\mathtt{f}$ is a monofeature: $\mathtt{X} = \mathtt{f}{:}\mathtt{Y}$, $\mathtt{X} = D_f$, $\mathtt{Y} = R_f$, $\mathbf{trans}(\mathtt{Y}, t)$ with new variable symbol $\mathtt{Y}$ and $\mathtt{f}{:}D_f \rightarrow R_f$.<br>$(ii)$ if $\mathtt{f}$ is a polyfeature: $\mathtt{X} = \mathtt{f}{:}\mathtt{Y}$, $\mathtt{X} = D_f$, $\mathbf{trans}(\mathtt{Y}, t)$, $\mathtt{fdecl}_i(\mathtt{X},\mathtt{Y})$ with new variable symbol $\mathtt{Y}$ and $\mathtt{fdecl}_i(\mathtt{X},\mathtt{Y}) := (\mathtt{X} \neq D_i \vee \mathtt{Y} = R_i)$ for all $\mathtt{f}{:}D_i \rightarrow R_i$, and $D_f = \bigvee_i D_i$. |
| $\mathbf{trans}(\mathtt{X}, \tilde{\ }t)$ | $:=$ | $\mathtt{X} \neq \mathtt{Y}, \mathbf{trans}(\mathtt{Y}, t)$ with new variable symbol $\mathtt{Y}$. |
| $\mathbf{trans}(\mathtt{X}, s\ \&\ t)$ | $:=$ | $\mathbf{trans}(\mathtt{X}, s), \mathbf{trans}(\mathtt{X}, t)$. |
| $\mathbf{trans}(\mathtt{X}, s\ ;\ t)$ | $:=$ | $\mathtt{d}_i(\mathtt{X},\vec{Y})$ with new sort symbol $\mathrm{d}_i$ and the clauses:<br>$\mathtt{d}_i(\mathtt{X},\vec{Y}) \leftarrow \mathbf{trans}(\mathtt{X}, s)$.<br>$\mathtt{d}_i(\mathtt{X},\vec{Y}) \leftarrow \mathbf{trans}(\mathtt{X}, t)$.<br>$\vec{Y}$ holds all variables of $(s\ ;\ t)$ that are nonlocal to $s$ or $t$. |

Figure 9: Translation of Feature Terms in CUF

The intermediate code used by the CUF interpreter is an optimization of the translation code given above. Disjuncts with empty denotation are removed, and if the user wants to expand all deterministic goals, this will be done at compile time too. Furthermore, some inequalities can be removed, e.g., if the arguments are no more unifiable or the arguments have been made identical in an earlier step. Of course, the latter case would cause an error.

## 3.2 Interpreter

### 3.2.1 Control Statements

Currently CUF provides two kinds of control statements: index declarations and delay patterns.

The index declarations trigger the compiler to generate tables for sorts which should be indexed over the first parameter of a sort, e.g. for lexicon access. The format of an index declaration is

`index_table(`*sort*/*arity*`).`

Delay patterns allow the specification of user-defined strategies for the interpretation of CUF specifications. The circumstances under which the propagation of a sort should be delayed are described by delay specifications. The format of delay patterns is

`delay(`*sort*/*arity*, *Delayed Parameters*`).`

*Delayed Parameters* is a list of delay specifications of the form $i{:}Path$ where the path statement is optional. The positive integer $i$ stands for the $i^{th}$ parameter, so $i \leq arity$. 0 is used for the "result parameter". A delay specification for the $i^{th}$ parameter may look like $i{:}\mathtt{f}_1{:}\ \ldots\ {:}\mathtt{f}_n$. This means that if the value of the path $\mathtt{f}_1{:}\ \ldots\ {:}\mathtt{f}_n$ is uninstantiated evaluation is delayed. If the list of delay specifications has more than one element, e.g., for different parameters or different paths of one parameter, the statements mean a conjunction of the delay conditions: "delay if A is variable `and` B is variable". Therefore, if one of the "delayed parameter paths" gets a value, the sort will be expanded. Disjunction of delay conditions is expressed using more than one dalay pattern for the same sort.

**Example 13** `append/2` as defined in Example 11 may cause problems with termination, if parameters are not sufficiently instantiated. The sort may generate lists containing only variable elements. But if we define a delay pattern `delay(append/2,[0,1])` which requires some instantiation of the first or the result parameter, this problem and similar ones can be handled very elegantly. Notice that with this delay pattern we would still be able to use `append/2` in "reverse", generating splits of a given list, as in `[a,b,c,d] & append(Part1,Part2)`. □

### 3.2.2 Proof Strategy

In order to prove a goal — given as a sort possibly with parameters — the interpreter basically performs a kind of SLD-resolution with the clauses. The actual processing order can be fine-tuned using the control statements described above. Of course, if no control of this kind is defined, the interpreter works as well.

The semantic foundation for this proof procedure is given by the CLP scheme of Höhfeld and Smolka [HS88]. According to this we can employ the generalized SLD-resolution as described in Fig. 10, since we can view our clauses as consisting of relational atoms and of constraints, the satisfiability problem of the latter being decidable. There, goals contain a number of relational atoms $r_i(\vec{X_i})$ over distinct variables as well as a complex constraint (letters: $\phi$ and $\psi$). Bodies of clauses have the same form as goals and heads consist of one relational atom over distinct variables. Notice that with these assumptions resolution involves only substituting $\vec{X}$ for $\vec{X_0}$ in $\phi$ and solving the conjunction of this constraint and $\phi_0$. Actually, this differs only from ordinary (PROLOG) resolution in the fact

| | |
|---|---|
| $r_0(\vec{X}) \wedge \mathit{RelAtoms} \wedge \phi_0$ | goal list |
| $r_0(\vec{X_0}) \leftarrow r_1(\vec{X_1}) \wedge \ldots \wedge r_n(\vec{X_n}) \wedge \phi[\vec{X_0}, \vec{X_1}, .., \vec{X_n}]$ | clause to be resolved with |
| $r_1(\vec{X_1}) \wedge \ldots \wedge r_n(\vec{X_n}) \wedge \mathit{RelAtoms} \wedge \psi$ | result goal list ($\psi$ is the solved form of $\phi_0 \wedge \phi[\vec{X}, \vec{X_1}, \ldots, \vec{X_n}]$, if that exists) |

Figure 10: Generalized SLD resolution step

that unification is generalized to constraint solving, i.e., partial solutions are not only stored as variable bindings, but we also have to administer constraints restricting further variable bindings in an internal state. However, SLD-resolution describes only the general scheme of the proof. The actual strategy for goal selection, i.e. the computation rule, has a dramatic influence on the size of the search space that has to be considered. It is this computation rule that is refined by the addition of delay statements.

As a general strategy the computation rule employed in the CUF system always selects *deterministic goals*, if such goals are present. A deterministic goal is a goal for which no choice point needs to be introduced, i.e. for which only one clause has satisfiable constraints. Only if no such goals are left over the left-most nondeterministic goal which is not delayed according to a delay statement will be expanded. The search induced by the nondeterminism is performed depth-first, using chronological backtracking. In the exceptional case where all goals are delayed, the first one of the goal list will be expanded, optionally indicating this condition to the user. The user may restrict the recursion using a depth bound. This guarantees termination, but the proof procedure is no longer complete.

### 3.2.3 Constraint Handling

As mentioned in Section 3.1.3 there are three kinds of constraints to solve in a prove of a CUF goal: equality, inequality and polyfeature constraints. The extended unification for typed feature structures is used for solving immediately equality constraints during compile time as well as runtime. But the translation generates inequalities too, which cannot always be resolved at compile time. And if there exists polyfeatures we have to handle the constraints on feature domains and ranges as well. So we have to say something more about inequality and polyfeature constraints.

The treatment of inequalitites is reduced to three cases:

- An inequality constraint is satisfied, if the unification of both arguments fails.
- An inequality constraint is never satisfiable, if the arguments are identical. In this case a prove fails immediately.
- If none of the previous cases hold, delay the evaluation.

A polyfeature constraint $\texttt{fdecl}_i(\texttt{X}, \texttt{Y}) := (\texttt{X} \neq D_i \vee \texttt{Y} = R_i)$ is treated the following way:

- If Y is subsumed by $R_i$, or X and $D_i$ are not unifiable, the constraint is satisfied.
- If Y and $R_i$ are not unifiable, then unify X and $\neg D_i$.
- If X is subsumed by $D_i$, then unify Y and $R_i$.
- If none of the previous cases hold, delay the evaluation.

The constraints are handled always before an expansion of a nondeterministic goal (see Section 3.2.2), or, if some are remaining in the end, these have to be checked too. If some constraints cannot be resolved, they are reported together with the result.

# 4 Future Work and Conclusion

Besides the descriptions of HPSG grammars CUF is currently being used for implementations of

- a theory of underspecified discourse representation structures, which gives an elegant solution to some semantic scoping problems [Rey93, Kön92] and
- a grammar of German based on GB theory.

There is also work going on in refining the evaluation strategy. Currently we are testing an extension of the resolution method by a memoization technique which is a generalization of Earley Deduction [Dör93]. Furthermore, we plan to add macros for DCG-like grammar rules and for operators of Lambek categorial grammar.

In conclusion let us state that CUF breaks with the traditional view of grammar formalisms, in which labour is divided between linguists designing grammars inside the formalism and specialists for parsing and generation developing the processing machinery for the formalism independently from individual grammars. This division of labour is now (partly) supported inside the formalism itself with the descriptive specification of grammar and (fine-tuning of) control being separate parts of a CUF program. With this move we achieve a much larger flexibility in the specification language without compromising declarativity. The ability to define new operators as well as, to a certain degree, their procedural behaviour justifies to regard CUF more as a formalism *toolbox* of formalisms in the traditional view.

Moreover, since descriptions of very different linguistic fields from phonology to semantics can be expressed in terms of relations between feature structures and since CUF's evaluation strategy is especially well suited to strongly interacting constraints of very different parts of a description, we expect it to be an ideal basis for highly integrated linguistic processing.

# References


[BKU88] Gosse Bouma, Esther König, and Hans Uszkoreit. A flexible graph-unification formalism and its application to natural-language processing. *IBM Journal of Research and Development*, 32(2):170–184, March 1988.

[Bla92] Patrick Blackburn. Modal logic and attribute value structures. ITLI Prepublication Series LP-92-02, Institute for Language, Logic and Information, University of Amsterdam, March 1992. To appear in *Diamonds and Defaults*, edited by M. de Rijke, Studies in Logic, Language and Information, Kluwer.

[BS92] Patrick Blackburn and Edith Spaan. A modal perspective on the computational complexity of attribute value grammar. Logic Group Preprint Series No. 77, Department of Philosophy, University of Utrecht, Heidelberglaan 8, NL–3584 CS Utrecht, April 1992.

[DE91] Jochen Dörre and Andreas Eisele. A Comprehensive Unification-Based Grammar Formalism. DYANA Deliverable R3.1.B, ESPRIT Basic Research Action BR3175, Jan. 1991.

[DEW+90] Jochen Dörre, Andreas Eisele, Jürgen Wedekind, Jo Calder, and Mike Reape. A Survey of Linguistically Motivated Extensions to Unification-Based Formalisms. Dyana deliverable R3.1.A, ESPRIT Basic Research Action BR3175, 1990.

[Dör91] Jochen Dörre. The Language of STUF. In O. Herzog and C.-R. Rollinger, editors, *Text Understanding in LILOG*, Lecture Notes in Artificial Intelligence 546, pages 39–50. Springer-Verlag, 1991.

[Dör93] Jochen Dörre. Generalizing Earley deduction for constraint-based grammars. DYANA deliverable, 1993. This volume.

[DR91] Jochen Dörre and Ingo Raasch. The STUF Workbench. In O. Herzog and C.-R. Rollinger, editors, *Text Understanding in LILOG*, Lecture Notes in Artificial Intelligence 546, pages 55–62. Springer-Verlag, 1991.

[DS91] Jochen Dörre and Roland Seiffert. Sorted Feature Terms and Relational Dependencies. IWBS Report 153, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, February 1991.

[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.

[KB82] Ronald M. Kaplan and Joan Bresnan. Lexical-Functional Grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press, Cambridge, Mass., 1982.

[Kön92] Esther König. Generation from underspecified discourse representation structures. Submitted for publication, Nov. 1992.

[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.

[KR90] Robert T. Kasper and William C. Rounds. The logic of unification in grammar. *Linguistics and Philosophy*, 13(1):35–58, 1990.

[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.

[PS93] Carl Pollard and Ivan A. Sag. Head-Driven Phrase Structure Grammar. Draft of June 24, 1991. Course materials of C. Pollard, 3rd European Summer School on Logic, Language and Information, Saarbrücken, Germany. Will be published by Chicago University Press and CLSI Publication, 1993.

[Rea91] Mike Reape. An introduction to the semantics of unification-based grammar formalism. Deliverable R3.2.A, DYANA — ESPRIT Basic Research Action BR3175, 1991. to appear.

[Rey93] Uwe Reyle. Dealing with ambiguities by underspecification: Construction, representation, and deduction. To appear in the Journal of Semantics, 1993.

[Sei92] Roland Seiffert. How could a good system for practical NLP look like? In *Coping with linguistic ambiguity in typed feature formalisms*, pages 83–92, Workshop at the 10th European Conference on Aritificial Intelligence, Vienna, Austria, 1992.

[Shi86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes 4. Center for the Study of Language and Information, Stanford University, 1986.

[SUP+83] Stuart M. Shieber, Hans Uszkoreit, Fernando C.N. Pereira, J.J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In J. Bresnan, editor, *Research on Interactive Acquisition and Use of Knowledge*. SRI International, Artificial Intelligence Center, Menlo Park, Cal., 1983.