

# Extending Unification Formalisms

G. Erbach    M. van der Kraan    S. Manandhar    H. Ruessink    W. Skut  
C. Thiersch\*

## Abstract

This paper describes some of the results of the project *The Reusability of Grammatical Resources*. The aim of the project is to extend current grammar formalisms with notational devices and constraint solvers in order to aid the development of reusable grammars. The project took the Advanced Linguistic Engineering Platform (ALEP) as its starting point. ALEP allows two levels of extension: additional syntactic expressions (“syntactic sugar”) and specialised external constraint solvers. Our syntactic additions to ALEP comprise support for LFG coherence and completeness and an extended notation for phrase-structure rules. The project has developed solvers for set constraints, set operations, linear precedence constraints and guarded constraints. The extensions were formally specified and implemented as Prolog modules and hence have a wider applicability. These can be either used standalone or integrated within existing grammar formalisms. We have successfully integrated the all constraint solvers with the ProFIT typed feature system and these have been partially integrated into ALEP.

## 1 Introduction

The development of large-coverage grammars for NLP systems is a time-consuming and expensive process, and therefore it is an important requirement that grammars be reusable for different applications. In the past, this has often not been the case because grammars were developed in formalisms that were dependent on particular processing models.

---

\* Authors' affiliations: Gregor Erbach (erbach@coli.uni-sb.de) and Wojciech Skut (skut@coli.uni-sb.de): Universität des Saarlandes, Saarbrücken; Mark van der Kraan (Mark.vanderKraan@let.ruu.nl) and Herbert Ruessink (Herbert.Ruessink@let.ruu.nl): Stichting Taaltechnologie, Utrecht; Suresh Manandhar (Suresh.Manandhar@ed.ac.uk): Language Technology Group, University of Edinburgh; Craig Thiersch (thiersch@kub.nl): Instituut voor Taal- en Kennistechnologie, Tilburg. The project was supported by the Commission of the European Communities through the programme *Linguistic Research and Engineering* (LRE-061-61).

In the past decade, there has been a move toward declarative specifications of grammatical knowledge, and a convergence between the formalisms used in computational and theoretical linguistics. This development has led to the widespread use of typed feature formalisms [Carpenter, 1992] (also known as *unification formalisms* due to the name of the operation for combining feature structures), which are today used in almost all large grammar development projects.

One such typed feature formalism is the Advanced Linguistic Engineering Platform (ALEP) [BIM-SEMA, 1993] [Meylemans, 1994], on which the work in our project is based.<sup>1</sup> Although ALEP is a typed feature formalism offering an attractive integrated grammar development environment it offers only a very impoverished typed feature logic in comparison to other systems such as ALE [Carpenter and Penn, 1994], CUF [Dorna *et al.*, 1994] and TFS [Zajac, 1992].

It is uncontroversial to say that there is still a gap between the high-level descriptions used by linguists and the expressive means of current grammar formalisms such as ALE, ALEP, CUF and TFS. This is especially true for the kinds of descriptions that are offered by relatively low-level grammar formalism such as ALEP. Although encodings of high-level constructs may be possible within existing formalisms, such an encoding suffers from several disadvantages:

1. *Grammars will lose clarity when encoded in a low-level formalism.* For example, grammars lose clarity when high-level LP constraints are hand-coded into low-level constructs that are available in the formalism.
2. *The existing feature formalism may not have the necessary constructs to achieve an efficient encoding.* This is especially true if one (disjunctive or underspecified) data structure must be expanded to several data structures in

---

<sup>1</sup>ALEP was designed by the European Community project ET6/1 and implemented under the project ET9.

the existing formalism and generalisations are lost.

3. *The encoding may not be re-usable if non-standard constructs of a formalism are employed.* This is the case if a formalism supports ad-hoc constructs that are not formally well-understood or makes calls to the underlying programming language.

For developing reusable grammars that do not suffer from these problems, the linguist needs better notational devices and some high-level logical descriptions interpreted by specialised reasoning modules or constraint solvers.

Since it is impossible to foresee the requirements of all current and future grammatical theories, the project has conducted a survey of datatypes used in current grammatical theories, and selected those which are widely used, have the potential of surviving in the future, and can be implemented relatively efficiently. General-purpose datatypes have been preferred over very specialised ones.

The project developed solvers for the following kinds of constraints:

1. Set descriptions and set operations
2. Linear precedence constraints
3. Guarded constraints

These constraint solvers are the first of its kind to appear within a typed feature formalism. Currently we have successfully integrated all the constraint solvers with the ProFIT [Erbach, 1995] typed feature system. Furthermore, since each of the constraint solvers is written as a separate Prolog library they are portable into other feature formalisms.

Secondly, the project developed the following syntactic extensions which were integrated into ALEP:

1. LFG completeness and coherence
2. Extended phrase-structure rules

We believe that in the future large-scale grammar development should be based on using general-purpose high-level grammar specifications. Furthermore, there should be concrete standards that large-scale grammars should adhere to if they are meant to be portable and modular. Achieving portability would make it independent of any underlying programming language implementation or the hardware implementation. To achieve this goal it is important that all the high-level constructs have

a mathematical basis. On the other hand, modularity is essential if large grammars can be constructed from smaller ones and vice versa without too much re-engineering. In addition, the feature formalism should also be extensible to allow easy integration of newer developments.

In order to address the portability issue, the project has developed formal specifications of each of the extensions. The constraints solvers are backed by a declarative semantics and a sound and complete operational semantics. The modularity aspect is addressed to a certain extent, since each of the extensions that the project has developed is implemented as a separate module. However the empirical study of modular development of large-scale grammars goes beyond the scope of the project.<sup>2</sup>

From a commercial perspective we believe that the extensions developed in this project together with efficient compilation techniques could serve as a rapid low-cost development platform for language engineering products that make use of modern constraint-based grammars in their products.

In the following sections we will provide an overview of each of these extensions.

## 2 Set Descriptions and Set Operations

Set descriptions are widely used in linguistic theories to model phenomena the handling of which implies storing and passing information about possibly large collections of certain linguistic objects such as subcategorisation frames, discourse referents, quantifiers, extraposed or fronted constituents etc. Set descriptions are used in Head-Driven Phrase-Structure Grammar (HPSG) [Pollard and Sag, 1994], Lexical-Functional Grammar [Kaplan and Bresnan, 1982] and others.

Despite this widespread use, current grammar formalisms lack support for set descriptions. In implemented grammars, sets are often approximated by encoding as lists. Such a low-level encoding strongly constrains the class of operations that can be performed. In certain cases such a restricted version of set operations might prove sufficient, albeit not without a loss in both efficiency and formal transparency of the formalisation. For example, subcategorisation frames may be modelled by lists, but in free word order languages the arguments on such a subcat-list can be bound off in almost any order, which is often accounted for by splitting grammar rules into several instances depending on whether they apply to the *1st* argument,

---

<sup>2</sup>There are other EC funded projects such as LS-GRAM that are addressing this issue.

the 2nd argument, 3rd argument etc. Alternatively, another solution would be to write a special purpose parser that treats the subcategorisation list as an unordered set. Of course, such a solution poses an obvious obstacle for the portability and reusability of grammars.

Apart from that, there also exist phenomena whose treatment does require the use of set operation. For instance, the substitution of *set union* operation by the *append* operation, i.e. the usual solution in formalisms which do not support set operations, does not guarantee correct results if the two sets have elements in common. Thus, if e.g. the union of two non-disjoint sets of discourse referents is to be determined, the result sensibly should not contain multiple occurrences of referents since this would lead to unmotivated multiple solutions.

Using *append* instead of *set union* can also lead to termination problems if the arguments are only partially instantiated. Furthermore, *append* can cause unnecessary backtracking if a partially instantiated list gets further instantiated after an *append* operation.

The project has developed a complete constraint solver for set descriptions which permits the specification of *set memberships*, *set descriptions* and *fixed cardinality set descriptions*. In addition the full range of set operations such as *subset*, *intersection* and *union* constraints are supported.

### Syntax of Set Constraints

Our syntax for set constraints is given below. The intuitive meaning of the constructs is self-explanatory; for the logical and operational semantics, refer to formal description given in [Manandhar, 1994] upon which the implementation is based.

$S, T \longrightarrow$	
$exists(T)$	set membership
$\forall x \in y : T$	forall constraint
$\{T_1, \dots, T_n\}$	set description
$\{T_1, \dots, T_n\} =$	fixed card. set desc.
$x \cup y$	union
$x \cap y$	intersection
$\supseteq x$	subset
$disjoint(x, y)$	disjointness

Disjoint union is not available in the implementation, but it can be defined as follows by employing set disjointness and set union operations:

$$x \uplus y =_{def} disjoint(x, y) \sqcap (x \cup y)$$

This operation can be used to model the way the arguments of a head are successively bound off in the

process of forming a sentence. In the following example, the set of arguments subcategorised for by the head daughter of a phrase is the disjoint union of the set of arguments and the set of elements subcategorised for by the mother node.

$$(1) \quad \begin{array}{l} mother\text{-}node : subcat : y \sqcap \\ daughters : \left[ \begin{array}{l} head\text{-}dtr : subcat : x \uplus y \sqcap \\ arguments : x \end{array} \right] \end{array}$$

Linguistic theories employ set descriptions for various purposes such as *representation of semantic content*, *slashfeature* for encoding long-distance dependencies, *quantifier store*, *background context* information etc. The use of the set constraint solver permits direct writing and execution of these grammars without resorting to list-based encodings. Thus representations such as the HPSG lexical entry of the attributive adjective *red* given in (2) (see [Pollard and Sag, 1994] pp. 329) can be represented as is virtually without any changes. The availability of such a direct encoding obviates the need for low-level non-portable encodings.

From a computational perspective, generally speaking constraint solving with set descriptions is a potentially non-deterministic exponential-time operation. However, it should be noted that the constraint solver only creates *choice-points* only if there is no alternative but to search for solutions. Furthermore, if only a subset of the available set constraints are employed then a deterministic behaviour from the constraint solver is guaranteed. Consistency of (formulae in) the sub-logic consisting of just set-memberships, intersection and subset constraints can be computed deterministically. Hence a judicious use of set constraints does not cause a great deal of processing overhead.

$$(2) \quad \left[ \begin{array}{l} \text{CATEGORY} \left[ \begin{array}{l} \text{MOD } N' \left[ \begin{array}{l} \text{INDEX } \boxed{1} \\ \text{RESTR } \boxed{3} \end{array} \right] \\ \text{HEAD } \textit{adj} \\ \text{SUBCAT } \langle \rangle \end{array} \right] \\ \text{CONTENT} \left[ \begin{array}{l} \text{INDEX } \boxed{1} \\ \text{RESTR } \left\{ \left[ \begin{array}{l} \text{QUANTS } \langle \rangle \\ \text{NUCLEUS } \left[ \begin{array}{l} \text{RELN } \textit{red} \\ \text{INST } \boxed{1} \end{array} \right] \end{array} \right\} \cup \boxed{3} \end{array} \right. \end{array} \right]$$

### 3 Guarded Constraints

The project has developed a specialised constraint solver for guarded constraints. Guarded constraints are used in logic programming to delay a constraint if not enough information is available for its deterministic execution. Such situations arise fre-

quently in natural language processing when the same grammar is used bidirectionally for parsing and generation.

Therefore, it is a natural move to include guarded constraints into grammar formalisms. The implementation of guarded constraints supports the following general purpose syntax:

$$\exists x_1, \dots, x_n \text{ case} ( \begin{array}{l} [ \text{condition}_1 \Rightarrow \text{action}_1, \\ \dots \\ \text{condition}_n \Rightarrow \text{action}_n \\ ] \\ \text{else } \text{action}_{n+1} \end{array} )$$

Each of the  $\text{action}_i$  can be any term or another guarded constraint. Each of the  $\text{condition}_i$  also known as *guard* is restricted to one of the following forms (the variables  $\exists x_1, \dots, x_n$  stand for existentially quantified variables):

$$\text{condition} \rightarrow \begin{array}{l} \langle \text{feature\_term} \rangle \\ | \text{exists}(\langle \text{feature\_term} \rangle) \\ | \text{precedes}(x, y) \end{array}$$

The constraint *if G then S else T* can then be thought of as syntactic sugar for the statement  $\text{case}([G \Rightarrow S]) \text{ else } T$ . The constraint *if G then S else T* can also be thought as syntactic sugar for the constraint  $\text{if } G \text{ then } S \sqcap \text{if } \neg G \text{ then } T$ . However the former representation is more efficient.

Guarded constraints can be thought of as *conditional constraints* whose execution depends on the presence of other constraints. The action  $S$  is executed if the current set of constraints *entail* the guard  $G$ . The action  $T$  is executed if the current set of constraints *disentail* the guard  $G$ . If the current set of constraints neither entail nor disentail  $G$  then the execution is *blocked* until more information is available.

The constraint solving machinery needed for implementing guards on feature constraints has been worked out in [Smolka and Treinen, 1994] and [Ait-Kaci and Podelski, 1994]. Our implementation extends this to permit guards on set-memberships and guards on precedence constraints. Our implementation is restricted to what is known as *flat guards* since guards cannot be embedded in our implementation. However, this restricted language appears to be sufficient for linguistic applications.

#### 4 Linear precedence constraints

A notion of linear order has various uses in linguistic descriptions. Its most obvious use is the modelling of word order phenomena. Other uses are

in natural language semantics in the description of temporal precedence relations and underspecified quantifier scope relations.

The linear precedence constraint solver has successfully been employed to analyse complex word-ordering phenomena in German. The surface form of a sentence (*i.e.* the sequence of words in a sentence) is represented by a set whose elements are pointers to the individual words together with a collection of ordering or *precedence* constraints between the elements. Thus the set given by:

$$(3) \{ \text{john, her, loves} \}$$

without any ordering constraints is intended to model the case where no ordering restrictions are placed between the 3 words. Thus all the sentences in (4) are permitted by this specification.

- (4) a. John her loves.
- b. Her John loves.
- c. John loves her.
- d. Loves John her.
- e. Loves her John.
- f. Her loves John.

However we would want to rule out the ungrammatical ones. Thus for instance, if we now add the additional constraints:

$$(5) \text{john} < \text{loves} \sqcap \text{loves} < \text{her}$$

then we can only derive the sentence given in (4c).

#### Syntax of LP Constraints

In the following we describe the syntax of the linear precedence constraints supported by our implementation. The implementation is based on the work described in [Manandhar, 1995].

$$S, T \rightarrow \begin{array}{l} \text{precedes}(y) \\ | \text{precedes\_equals}(y) \\ | \text{fst\_daughter}(y) \\ | \text{dom\_precedes}(y) \\ | \text{dom\_precedes\_equals}(y) \\ | \text{if precedes}(x, y) \text{ then } S \text{ else } T \end{array}$$

To enable the specification of complex word order phenomenon, following [Reape, 1993], a *dom* (set-valued) feature is introduced which collects all the leaf nodes of a phrase (*i.e.* all the pointers to words belonging to a phrase).<sup>3</sup>

<sup>3</sup>[Reape, 1993] uses *sequences* whereas we use sets.

- (6)  $cat : n \sqcap$   
 $phon : man \sqcap$   
 $subcat : \{cat : det \sqcap dom : x\} \sqcap$   
 $pos : y \sqcap$   
 $dom : (exists(y) \sqcap \supseteq x)$

This lexical entry states that the domain of a common noun includes itself ( $y$ ) and the domain of the subcategorised determiner ( $x$ ).

Similarly,  $dom\_precedes$  can be employed to specify relative ordering between domains. Thus for example, to state that (every element of) domain  $x$  precedes (every element of) domain  $y$ , where both  $x$  and  $y$  are set-valued, we can employ a description such as:

- (7)  $arg1 : (dom : (x \sqcap dom\_precedes(y))) \sqcap$   
 $arg2 : (dom : (y))$

Typically for treating word-ordering regularities in the German *Mittelfeld* constraints such as *pronouns precede non-pronouns* need to be expressed. By employing a combination of set operations, guarded constraints and precedence constraints this can be stated as follows<sup>4</sup>:

- (8)  $\forall x, y \in Dom :$   
 $if\ x = pron : + \sqcap y = pron : -$   
 $then\ x \sqcap precedes(y)$

## 5 An Extended Syntax for Phrase-Structure Rules

The project implemented an extended syntax for phrase-structure rules in ALEP. The syntactic extensions are optionality and arbitrary repetition of sequences of daughter nodes. The extended syntax is compiled out during the compilation of the grammar and the lexicon. This is combined with the automatic instantiation of certain (user-defined) features of the mother and daughter nodes in the rule. The automatic instantiation of some features is required to allow the grammar writer to access information about the optional daughters in rules that make use of these extensions.

### Notation

The extensions for the optionality and arbitrary repetition of daughters in a rule are inspired by the use of regular expressions on the right-hand side of LFG phrase structure rules. The extensions cover sequences (lists) of daughter nodes and they can be nested. The following example illustrates this. In this rule the ? prefix operator indicates optionality and \* indicates arbitrary repetition.

- (9)  $NP \rightarrow ?[ Det ] *[ Adj ] [ N ] ?[ PP_{of} ]$

The rule given in (9) permits noun phrases such as *water, the book, green bananas, the fat old man, a picture of Charles V, etc.*

### Compilation

The optionality and repetition operators are not interpreted at run-time but are removed during grammar compilation. A rule that matches the format of (10) or (12) is replaced the rules given under (11) and (13). This elimination process also applies to the resulting rules and it proceeds until all occurrences of the optionality and repetition operator have been removed. The symbols  $s$  and  $u$  in the examples stand for possibly empty sequences of nodes and  $t$  stands for a non-empty sequence. The first example shows the standard approach to eliminate the optionality operator.

- (10)  $X \rightarrow s ?[ t ] u$

- (11) a.  $X \rightarrow s u$   
 b.  $X \rightarrow s t u$

Rules that contain the repetition operator result in a slightly more complex set of rules.

- (12)  $X \rightarrow s *[ t ] u$

- (13) a.  $X \rightarrow s u$   
 b.  $X \rightarrow s Q u$   
 c.  $Q \rightarrow t Q$   
 d.  $Q \rightarrow t$

A rule matching (12) is replaced by four different rules. The first of these represents the fact that the sequence  $t$  need not occur at all. The other three rules allow the sequence  $t$  to be repeated any number of times. The symbol  $Q$  is a newly created symbol that does not occur elsewhere in the grammar.

There are also other rule sets for replacing a rule \*, which achieve the same descriptive result. The implementation for the ALEP environment employs the approach illustrated by (13) because it avoids the use of empty production rules and of left-recursive rules. It is known that such rules can be problematic for certain analysis and synthesis strategies. ALEP is designed as an open environment and this requires compatibility with a range of different implementations for analysis and synthesis.

### Instantiation of Features

The use of optionality and arbitrary repetition creates a problem in that certain daughters may or may

<sup>4</sup>A more detailed and technical example can be found in [Manandhar, 1995].

not be present when the rule is used during synthesis or analysis<sup>5</sup>. Specifically, threading of information such as string or slash features, and accessing information about the content of the optional or repeated nodes is not possible. Consider the following example (in a DCG-like notation):

```
(14) % verbal complement rule:
      %   VP --> V * [ Complement ]
      node(vp, [], In-Out) -->
          node(v, Subcat, In-Mid),
          * [ node(_, _, Mid-Out) ].
```

This example illustrates two problems. Firstly, the `Subcat` information has to be related to the complements that are instantiated during synthesis or analysis. As represented here there is no interaction between the arbitrarily repeated complement and the subcategorisation list. The second problem is that the threading of the string information is correct only if there is one complement. The following examples illustrate the (undesirable) results when there are zero or two complements present:

```
(15) node(vp, [], In-Out) -->
      node(v, Subcat, In-Mid).
```

```
(16) node(vp, [], In-Out) -->
      node(v, Subcat, In-Mid),
      node(_, _, Mid-Out),
      node(_, _, Mid-Out).
```

The compilation of rules in ALEP is therefore extended with the automatic instantiation of two pairs of features. The first is used to represent threading features and the second is used to represent a list of daughters for each mother of a rule. The representation of the list of daughters makes use of a pair of features in order to handle the added nodes created when compiling rules that contain arbitrary repetition (the node  $Q$  in (13)). In these cases the two features are employed as a difference list to collect all occurrences of the sequence  $t$ . This information percolates upwards to the mother node  $X$  of the rule. The second daughters feature will always be the empty list for nodes that appear in the grammar writer's rules.

The grammar writer has to declare which features are to be used by the compiler to represent threading information and the lists of daughters, for instance by including Prolog facts such as the following in the grammar files.

```
(17) daughters_feature(  

      node(_, _, Ds, _, _), Ds).  

      second_daughters_feature(  

          node(_, _, _, Extra, _), Extra).  

      threading_feature_in(  

          node(_, _, _, _, In-_), In).  

      threading_feature_out(  

          node(_, _, _, _, -Out), Out).
```

<sup>5</sup>That is, when one or more of the rules that replaced the original rule are used.

```
      node(_, _, Ds, _, _), Ds).  

      second_daughters_feature(  

          node(_, _, _, Extra, _), Extra).  

      threading_feature_in(  

          node(_, _, _, _, In-_), In).  

      threading_feature_out(  

          node(_, _, _, _, -Out), Out).
```

Consider the following example of a grammar rule and the corresponding four compiled rules:

```
(18) node(vp, [], [V|Subcat], [], _)
      -->
          node(v, Subcat, _, [], _),
          * [ Complement ].
```

```
(19) a. node(vp, [],
      [node(v, Subcat, A, [], In-Out) | []],
      [], In-Out)
      -->
          node(v, Subcat, A, [], In-Out).
```

```
      b. node(vp, [],
      [node(v, Ds, A, [], In-Out) | Ds],
      [], In-Out)
      -->
          node(v, Ds, A, [], In-Mid),
          node(q12, _, Ds, [], Mid-Out).
```

```
      c. node(q12, _,
      [node(A, B, C, D, In-Mid) | DsIn],
      DsOut, In-Out)
      -->
          node(A, B, C, D, In-Mid),
          node(q12, _, DsIn, DsOut, Mid-Out).
```

```
      d. node(q12, _,
      [node(A, B, C, D, In-Mid) | DsOut],
      DsOut, In-Out)
      -->
          node(A, B, C, D, In-Out).
```

The instantiation of the threading feature takes place automatically after the expansion of the grammar writer's rules. It applies by default, that is to say, the relevant feature pairs (e.g., the *out*-feature of one daughter and the *in*-feature of the next daughter) are not unified if the value of both is specified by the grammar writer.

The instantiation of the list of daughters is straightforward in the compiled rule (19a). The only occurring daughter is placed on the list. Since in that case the list of daughters contains only the verb, the subcat list is forced to be the empty list. The other compiled rules make use of difference lists for collecting daughters into one list.

## 6 LFG Completeness and Coherence

Lexical Functional Grammar (LFG) offers an elegant treatment of subcategorisation formulated by

the completeness and coherence conditions. These conditions require that an f-structure must contain all (completeness) and only those (coherence) grammatical functions that its local predicate governs.

The concept of grammaticality in LFG rests crucially on these two conditions, as can be seen in the *grammaticality condition* of LFG which states that “a string is grammatical if and only if it is assigned a complete and coherent f-structure” [Kaplan and Bresnan, 1982, page 212].

#### *Completeness Condition*

An f-structure is *locally complete* if and only if it contains all the governable grammatical functions that its predicate governs. An f-structure is *complete* if and only if it and all its subsidiary f-structures are locally complete.

#### *Coherence Condition*

An f-structure is *locally coherent* if and only if all the governable grammatical functions that it contains are governed by a local predicate. An f-structure is *coherent* if and only if it and all its subsidiary f-structures are locally coherent.

To put it simply, the *completeness condition* blocks sentences such as *John likes* since not all the arguments of *likes* are saturated (assuming that *likes* is a transitive verb). On the other hand, the *coherence condition* blocks sentences such as *John likes Mary Sandy* since there is one too many noun phrase which is not subcategorised by any verb.

One aspect of the LFG treatment of completeness and coherence that is attractive for other frameworks as well is that subcategorised elements can be bound in any order, because they are not represented on a list with a fixed order but referred to by their grammatical function name.

Coherence can be encoded in the ALEP formalism, but completeness is non-monotonic and goes beyond the expressive power of ALEP. The project has adopted an approach to coherence and completeness that employs NIL and ANY types. The implementation of the extension provides an automatic expansion of the grammar writer’s type system to include the required NIL and ANY types.

The NIL types serve to enforce coherence. The lexical entry of, for instance, a verb specifies that the features representing the functions for which the verb does not subcategorise must have a value of the NIL type. This type is incompatible with

any other type and the instantiation of the values of these features is therefore blocked.

Completeness is checked with the use of ANY types. The features that represent the governable grammatical functions of a lexical entry are declared to be of the ANY type. This type can unify with anything else, and when this occurs the result of the unification is a subtype of the ANY type. Any structure which contains any uninstantiated ANY values at the end of processing is considered ungrammatical. ANY values thus constitute a non-monotonic external constraint extension of ALEP and in this respect this extension exceeds the notion of syntactic sugar.

## 7 Implementation and Availability

The project has produced two implementations of the extensions. One implementation extends the ALEP formalism, while the other (called CL-ONE) exists as an extension of (Sicstus) Prolog. The two implementations have slightly different purposes:

- The ALEP implementation is aimed at grammar writers who want to make use of the comfortable grammar development provided by ALEP to develop natural language grammars.
- The CL-ONE implementation is aimed at grammar writers who don’t use ALEP. Because the CL-ONE implementation is designed as an *extension* to Prolog, it allows a seamless integration into Prolog-based NLP applications.

The CL-ONE implementation is based on the typed feature language ProFIT [Erbach, 1995]. However, the constraint solvers for sets, linear precedence and guarded constraints are also available as stand-alone modules, and can be used in conjunction with other feature constraint languages, or for other problems such as reasoning about underspecified sets, or temporal constraint solving in applications such as scheduling.

It is possible to use the ALEP environment for developing grammars with the extensions, and CL-ONE to integrate these grammars into NLP applications. Several grammars, especially with a focus on previously problematic German word order phenomena, have been implemented and tested with both implementations. This has shown the expressiveness of the system, and demonstrated that it is reasonably efficient.

The extensions are accompanied by user manuals. The ALEP extensions will be available with

the ALEP system. The stand-alone implementation CL-ONE is freely available. Further information can be obtained through the World Wide Web: <http://coli.uni-sb.de/info/projects/rgr.html>

## 8 Conclusion

Almost all grammar formalisms in use today share a typed feature structure language as a common core. But so far there is no widely recognised standard for grammar formalisms because the common core just does not provide enough expressive power, and different kinds of extensions are needed by grammar writers to model different phenomena (e.g. multiple inheritance, or lambda expressions for semantics). We believe that the extensions developed in our project (sets, LP constraints, tree constraints) will be an indispensable part of any future standard for grammar formalisms. We expect grammars developed in such a standard to have a good chance of being reusable for different applications.

The availability of high-level constructs within a feature formalism would be a great help for grammar writers since they can devote their attention to the linguistic specification rather than devoting their efforts into learning to live with the inadequacies of a low-level formalism. The task of efficient execution of such high-level grammars is then delegated to compilation techniques and smart constraint solvers. Needless to say, such an approach would greatly speed up product development cycles and reduce developmental costs in those applications which require a grammar module.

## References

- [Aït-Kaci and Podelski, 1994] Hassan Aït-Kaci and Andreas Podelski. Functions as Passive Constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, 16(4):1–40, July 1994.
- [BIM-SEMA, 1993] BIM-SEMA. ALEP System Documentation: The ALEP Linguistic Subsystem, Version 1.0. Technical report, Commission of the European Communities, March 1993.
- [Carpenter and Penn, 1994] Bob Carpenter and Gerald Penn. *The Attribute Logic Engine: User's Guide (Version 2.0.1)*. Carnegie-Mellon University, Pittsburgh, December 1994.
- [Carpenter, 1992] Bob Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, 1992.
- [Dorna *et al.*, 1994] Michael Dorna, Jochen Dorrer, and Joerg Junger. *CUF User's Guide: Version 2.30*. Institut fuer maschinelle Sprachverarbeitung (IMS), Universitaet Stuttgart, Germany, July 1994.
- [Erbach, 1995] Gregor Erbach. PROFIT: Prolog with Features, Inheritance and Templates. In *Seventh Conference of the European Chapter of the Association for Computational Linguistics (EACL'95)*, Dublin, Ireland, March 1995.
- [Kaplan and Bresnan, 1982] Ronald M. Kaplan and Joan Bresnan. Lexical-Functional Grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173 – 281. MIT Press, Cambridge, Massachusetts, 1982.
- [Manandhar, 1994] Suresh Manandhar. An Attributive Logic of Set Descriptions and Set Operations. In *32nd Annual Meeting of the ACL*, pages 255–262, Las Cruces, New Mexico, 1994.
- [Manandhar, 1995] Suresh Manandhar. Deterministic Consistency Checking of LP Constraints. In *Seventh Conference of the European Chapter of the Association for Computational Linguistics (EACL'95)*, Dublin, Ireland, March 1995.
- [Meylemans, 1994] Paul Meylemans. ALEP - Arriving at the next platform. *ELSNNews*, 3(2):4–5, 1994.
- [Pollard and Sag, 1994] Carl Pollard and Ivan Andrew Sag. *Head-driven Phrase Structure Grammar*. Chicago: University of Chicago Press and Stanford: CSLI Publications, 1994.
- [Reape, 1993] Mike Reape. Getting Things in Order. In Wietske Sijtsma and Arthur van Horck, editors, *Discontinuous Constituency*. Berlin: Mouton de Gruyter, 1993.
- [Smolka and Treinen, 1994] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.
- [Zajac, 1992] Rémi Zajac. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182, 1992.