A Survey of Systems for Implementing HPSG Grammars

Leonard Bolc
Krzysztof Czuba
Anna Kupść
Małgorzata Marciniak
Agnieszka Mykowiecka
Adam Przepiórkowski

October 31, 1996

Streszczenie • Abstract

Przegląd systemów do implementacji gramatyk HPSG

Niniejszy raport stanowi przegląd najbardziej popularnych systemów umożliwiających implementację gramatyk formalnych języka naturalnego. Selekcji systemów dokonano ze względu na ich przydatność do implementacji gramatyk HPSG. Dla każdego z uwzględnionych systemów (TFS, CUF, ALE, ALEP, PAGE, ProFIT, CL-ONE i ConTroll) opisano przyjęte na jego potrzeby założenia dotyczące hierarchii typów, sposobu zapisu struktur atrybutów, możliwości implementacji reguł gramatyk typu HPSG (reguły LP/IP, reguły leksykalne). Omówione zostały też cechy użytkowe poszczególnych systemów: dostępność, jakość dokumentacji, wymagania sprzętowe i programowe, efektywność oraz środowisko programistyczne. W części końcowej raportu zawarto szczegółowe porównanie opisywanych systemów.

A Survey of Systems for Implementing HPSG Gramamrs

In this report, we present an overview of some chosen grammar development systems which could be applied to implement HPSG grammars. The selection contains descriptions of the following systems: TFS, CUF, ALE, ALEP, PAGE, ProFIT, CL-ONE and ConTroll. We describe these systems with respect to the following criteria: the kind of type system used and its expressiveness, in particular the possibility to encode a type inheritance hierarchy; the constraint system employed; the expressiveness of the description language (disjunction, negation, LP rules, etc.) and built-in data types, e.g., sets; availability of lexical rules; declarativity of grammar description; the possibility to influence the computation rule, i.e, control mechanisms; system time performance on example grammars; system architecture: PS rules, built in parsers/generators, morphology components, etc.; foreign language interface, grammar tracing and debugging tools, graphical user interface, modular grammar development; documentation and support. The last section of the report presents a detailed comparison of the systems.

Contents

1 Introduction

In this paper we present an overview of some chosen grammar development formalisms which could be applied to implement and verify an HPSG-style grammar for Polish developed by our team.[1] There are other similar surveys available (e.g., Gardent (1993), Backofen *et al.* (1993)) but they are not well-suited for our purposes and they do not include newer systems.

The systems presented below were chosen from a bigger collection proposed by Przepiórkowski (1995) on the basis of our initial criteria gathered by Kupść (1995). Our aim is to investigate the suitability of currently available grammar development systems to express HPSG grammars. We hope to select systems which could be applicable at various grammar development stages, i.e., it/they should allow fast prototyping of new ideas in order to verify them and to support theoretical efforts as well as be sufficiently effective to provide a tool for our final implementation. Since these requirements seem to be contradictory, at least as far as available systems are concerned, finding two systems — one closely implementing mechanisms of HPSG and another one, more efficient, but perhaps requiring some changes in grammar architecture during implementation — seems to be a more plausible task.

The basic criterion we applied when selecting computational formalisms for this report was the requirement that they should provide an implementation of a version of a typed feature structure logic, preferably as used in HPSG. Another crucial criterion was availability of the system and of the software it requires. After taking these criteria into consideration, we concentrated on the following systems: ALE (Carpenter and Penn, 1995), ALEP (Groenendijk and Simpkins, 1994), CL-ONE (Manandhar, 1994b), ConTroll (Götz, 1995), CUF (Dörre and Dorna, 1993), ProFIT (Erbach, 1994b), PAGE/$\mathcal{TDL}$ (Krieger and Schäfer, 1994a, 1994b) and TFS (Emele, 1993).[2] In the report we decided to examine the systems with respect to (among others): the kind of type system used and its expressiveness, in particular the possibility to encode a type inheritance hierarchy; the constraint system employed; the expressiveness of the description language (disjunction, negation, LP rules, etc.) and built-in data types, e.g., sets; availability of lexical rules; declarativity of grammar description; the possibility to influence the computation rule, i.e, control mechanisms; system time performance on example grammars; system architecture: PS rules, built in parsers/generators, morphology components, etc.; foreign language interface, grammar tracing and debugging tools, graphical user interface, modular grammar development; documentation and support.

This overview presumes general familiarity with formalisms claimed to underly HPSG, cf. e.g. King (1989), Carpenter (1992), King (1994). Definitions of some basic notions, including notions such as open vs. closed world interpretation of a type hierarchy, can be also found in section 10.[3]

The following sections describe the individual systems according to the criteria mentioned above. Each section ends with a recommendation of the system. The systems are described chronologically (the order depending roughly on dates of first publications), i.e., in the following order: TFS, CUF, ALE, ALEP, PAGE, ProFIT, CL-ONE, ConTroll. The last section of this document presents a comparison of the systems.

[1]The authors are grateful to Gregor Erbach, Thilo Götz, Suresh Manandhar, Gerald Penn and Wojtek Skut for detailed comments on preliminary versions of the report and/or for the encouragement. Especially, comments provided by Gerald turned out to be so detailed and insightful that he might conceivably be added to the list of authors. Needless to say, only the authors should be blamed for any remaining errors and misunderstandings.

[2]Some omissions of this report are Troll (Gerdemann, 1993), developed at the University of Tübingen, and UBS (Stolzenburg *et al.*, 1996), developed at the University of Koblenz.

[3]Actually, judging from the comments we received, the notion of open/closed world interpretation turned out to be much more controversial than we had assumed. We should probably emphasize right at the beginning of this report that we borrowed the definitions of these notions from Gerdemann (1995), the most explicit work on this topic (within NLP) that we came across.

2 TFS

2.1 Introduction

The Typed Feature Structure representation formalism, cf. Emele and Zajac (1990b) and Zajac (1992), was developed by Martin Emele and Rémi Zajac within the German Polygloss project at IMS, University of Stuttgart. The following description presents briefly the main ideas of the system, Emele (1993):

> The Typed Feature Structure (TFS) representation formalism is an attempt to provide a synthesis of several of the key concepts of unification-based formalisms (feature structures), knowledge representation languages (inheritance) and logic programming (logical variables and declarativity). The inheritance-based constrained architecture embodied in the TFS system integrates two computational paradigms: the *object-oriented* approach offers complex, recursive, possibly nested, record objects represented as typed feature structures with attribute-value restrictions and (in)equality constraints, and multiple inheritance: the *relational programming* approach offers declarativity, logical variables, non-determinism with backtracking, and existential query evaluation.

2.2 Syntax and Data Structures

2.2.1 Type System

The TFS system allows direct encoding of an HPSG sort hierarchy. The set of types has to constitute a *partial order* but it is not required to constitute a *bounded complete partial order*. The predefined type `T` is the maximal element in the hierarchy (*top*), all other types are its subtypes. The attributes introduced by a type are declared in the following way (the notation resembles the AVM notation):

(1)
```
sign[PHON: nelist,
     SYNSEM: synsem].
```

Subtypes of a given type can be declared in two equivalent ways, cf. (2). In this example, `word` and `phrase` are the subtypes of `sign`, they inherit attributes (if any) from the supertype.

(2) a.
```
sign = word | phrase.
```
b.
```
word < sign.
phrase < sign.
```

Multiple inheritance is available in TFS, i.e., a subsort can inherit from several supersorts, see (3).

(3)
```
sort < supersort1, supersort2.
```

The multiple type definition, i.e., the possibility to define a type in several separate (and logically treated as disjoint) declarations, is a convenient feature of TFS which is useful to encode type constraints, see 2.2.5.

2.2.2 Lists

In TFS, a list hierarchy can be defined as in (4):

(4)
```
list = elist | nelist.
nelist [FIRST: T,
        REST: list].
```

The attributes FIRST and REST are provided by the system and need not be declared.
In order to use a Lisp-like notation for lists, the four following lines of type definitions have to be copied into the TFS program:

```
:NIL elist.       ;; type symbol for the empty lists.
:CONS nelist.     ;; type symbol for the non-empty list.
:CAR FIRST.       ;; feature symbol for the head of
                  ;; a non-empty list.
:CDR REST.        ;; feature symbol for the tail of
                  ;; a non-empty list.
```

This allows the use of the angle brackets ('< >) to enclose list elements and the period ('.') operator to construct lists. Both definitions in (5) have the same effect.

(5)
```
PHON [<"jan" "lubi "jana">].

PHON [<"jan". <"lubi". <"jana">>>].
```

TFS does not provide the concept of sets.

2.2.3 Variables

Structure sharing in TFS is encoded by variables. A variable consists of '#' and a name, e.g., #1, #X. Example (6) shows a way of encoding the Head Feature Principle as a constraint on the type phrase.

(6)
```
phrase
 [SYNSEM:[LOCAL:[CAT:[HEAD: #X]]],
  DTRS:[HEAD_DTR:[SYNSEM:[LOCAL:[CAT:[HEAD: #X]]]]]].
```

Operators '=' or '=/=' allow for type restrictions on variables. In example (7), the additional constraint on #X defines the finite verb phrase.

(7)
```
phrase
 [SYNSEM:[LOCAL:[CAT:[HEAD: #X=verb[VFORM:fin]]]],
  DTRS:[HEAD_DTR:[SYNSEM:[LOCAL:[CAT:[HEAD: #X]]]]]].
```

2.2.4 Predicates

In TFS, predicates can be defied via a definite clause mechanism similar to that of Prolog (using the ':-' operator).
The append operation for lists can be formulated in the following way:

(8)
```
append (<>,#Ls,#Ls).
append (<#X.#Xs>,#Ys,<#X.#Zs>) :-
       append (#Xs,#Ys,#Zs).
```

In TFS it is possible to add relational constraints to the type definition. The same operator :- specifies the condition the description on the left hand side has to satisfy. In (9), the phonology of the phrase is defined as the concatenation of the PHON values of the subject daughter and the head daughter.

(9)
```
phrase [PHON: #p1_2,
        ...
        DTRS: [HEAD-DTR: [PHON: #p2,
                          ...    ],
               SUBJ-DTR: [PHON: #p1,
                          ...    ]]] :-
    append (#p1,#p2,#p1_2).
```

2.2.5 Macros

TFS offers a macro facility to define shorthands. The definition of a macro consists of its name followed by '`:=`' and the macro description. Macros are very useful for encoding lexical entries and for simplifying descriptions of complex structures and grammar principles. For example, the following non-parametric macro encodes the Head Feature Principle:

(10)
```
hfp :=
 [SYNSEM:[LOCAL:[CAT:[HEAD: #X]]],
  DTRS:[HEAD_DTR:[SYNSEM:[LOCAL:[CAT:[HEAD: #X]]]]]].
```

Macros can be combined with type definitions in order to impose inherited constraints on types. E.g., the requirement that all phrases should fulfill the Head Feature Principle (`hfp`) and Valence Principle (`valp`) can be expressed in the following way:

(11)
```
phrase & hfp & valp.
```

(11) defines a constraint on `phrase` which will be satisfied by all its subtypes. Thus, if `wf-phrase` is declared as a subtype of `phrase`, it fulfills both `hfp` and `valp`.
As stated above, a type can have multiple definitions. In (12), `wf-phrase` has several definitions corresponding to different Immediate Dominance Schemata (`IDS`) and Constituent Order Principles (`COP`). All these constraints are disjunctive and `wf-phrase` has to fulfill *one* of them, not *all* of them. On the other hand, since `wh-phrase` is a subtype of `phrase`, the constraints imposed on the latter are inherited by the former.

(12)
```
wf-phrase & IDS1 & COP1.
wf-phrase & IDS2 & COP2.
       .....
wf-phrase & IDSn & COPn.
```

Macros (especially parametric ones) are also very useful for encoding lexical entries[4]. They can be used to factor out large amounts of information shared by such entries. For example, each nominal lexical entry has to specify the values of such categories as `CASE`, `GENDER`, etc. Once macros such as (13) are present in the grammar, complex lexical descriptions can be simplified to such as the one in (14):

(13)
```
M1 := Gender (m1).
Gender(#X) :=
    [SYNSEM:[LOCAL:[CONT:[INDEX:[GENDER:#X]]]]].
```

(14)
```
word[PHON: <"Jan">] & Noun & Nom & M1 & Third.
```

2.3 Computational Aspects

Components TFS has no specialized modules such as a parser or a generator.

System requirements TFS is implemented in Common Lisp (e.g., MCL, Allegro, Lucid, CMU CL, CLISP, AKCL). The system is freely available for non-commercial purposes by anonymous ftp from **`ftp.ims.uni-stuttgart.de`** in the directory **`pub/TFS`**. Files for different Lisp dialects and architectures (Sun, Macintosh, IBM PC), documentation and examples of HPSG grammars can be also obtained from this site.

Efficiency TFS is not very efficient. The aim of the authors was to create the formalism for designing, implementing and testing formal linguistic models, so the system should not be treated as a programming language for creating real applications.

[4] In TFS, there is no special notation for lexical entries and lexical rules.

TFS does not provide the means to make program execution more efficient. But the order of constraints stated in grammar is important. TFS interpreter tries to resolve a query, subsort and sort disjunctions in the order in which they appear in the source code. So the more frequently used constraint should be declared before the others.

Debugging TFS supports a trace facility with several useful options, e.g., it is possible to print definitions applied in a proof, trace the unification process and trace changes of the symbols from a specified list.
The whole type system as well as only parts of it can be printed. The internal definitions of the knowledge base can be inspected, i.e., feature structures can be printed in the fully expanded form with the macros used in the descriptions.

Environment TFS does not provide any special working environment and it is not integrated with any editor.

File Organization The grammar can be split into several files collected in one directory, which allows some degree of modular organization in TFS grammars.

Graphical User Interface TFS does not include any graphical user interface in its basic version. For some versions involving Common Lisp dialects (LISP machines and Mac IIs) the implementation of TFS supports graphic output and a menu-based window interface.

2.4 Development Stage

Support The system is not maintained anymore and TFS users should not expect any help from the authors in case of detecting errors or other difficult situations. Problems can occur with installation of the TFS system on Lisps running under recent operating systems (e.g., there is no TFS version for CMU CL running on Sun SPARC machine under Solaris system).

Documentation

- Emele (1993) — introductory information about TFS;
- Kuhn (1993) — the list of commands, a description how to implement the basic ideas of HPSG, examples of simple grammars;
- Keller (1993) — the description of chapter 9 of Pollard and Sag (1994) implementation in TFS;
- Zajac (1991) — documentation of an out-of-date TFS version.

Recommendation TFS can be recommended for testing HPSG grammars, but not for creating real applications. The weak feature of the system is its low efficiency which makes processing of large grammars very time consuming.

3 CUF

3.1 Introduction

The Comprehensive Unification Formalism has been developed at IMS, University of Stuttgart, within the DYANA research project as an implementational tool for grammar development. CUF has been designed to provide mechanisms broadly used in current unification based grammar formalisms such as HPSG or LFG. The main features of CUF are a very general type system and relational dependencies on feature structures. It is a constraint based system, with no specialized components, e.g., for morphology or transfer. It can be seen as a general constraint solver and it does not include a parser or a generator.

3.2 Syntax and Data Structures

CUF feature terms consist of three kinds of constraints: type information, partial descriptions of feature structures and sort goals. There is no special notation for lexical rules, linear precedence rules, phrase structure rules, etc.

3.2.1 Type System

The type system implemented in CUF is very general and allows a precise description of the feature structure universe. Type interdependencies are stated in full propositional logic, which makes defining all kinds of type hierarchies possible (Backofen *et al.* (1993), Dörre and Dorna (1993)). The type information is defined in the form of type axioms which are arbitrary boolean expressions over types (disjunction `s;t`, conjunction `s&t`, negation `~t`). In particular, it is possible to define disjoint types, which is impossible in systems like ALE. Compare the following type definitions:[5]

(15) a. ALE: `sign sub [word, phrase].`

b. CUF: `sign = word | phrase.`

Only the CUF axiom declares that the types `word` and `phrase` are disjoint and that the type `sign` has no (direct) subtypes other than `word` and `phrase`. In order to obtain the same effect in ALE it is necessary to use definite clauses.[6] For other examples of type hierarchies which are easily defined in CUF and require complicated definitions in ALE-like type systems see Hegner (1994).
Although the CUF type system is very expressive, it is not *general* as defined in Manandhar (1993). Manandhar (1993) calls a type system *general* "if there are no restrictions imposed on the right hand side of a type definition, i.e., if at least arbitrary conjunctions of type symbols, feature selection and variables are permitted." In CUF, no variables are allowed in type axioms. As a result, it is not possible to state certain HPSG principles (e.g., the Head Feature Principle) directly in the type definitions as it is the case, e.g., in TFS. The principles have to be defined as CUF *sorts*, see 3.2.3 for examples.
The type universe is prestructured and the following type axioms are built in:

(16) a. for all types `t`: `t < top`, `bottom < t`;

b. `top = afs | cfs`;

c. `list = elist | nelist`;

d. `number | string | afs_symbol | prolog < afs`;

e. `elist = {[]}`;

f. `nelist ::  'F':top, 'R':list.`

where `afs` is the set of all atomic feature structures, `cfs` is the set of all complex feature structures.
All undeclared constant symbols are of type `afs_symbol`. Although this feature can be useful, e.g., for fast prototyping of large lexicons, it can also lead to obscure errors. Sets are not provided in CUF and have to be modelled as lists.[7]
Due to the open-world approach, two types in CUF are inconsistent only when they are declared as such, otherwise they are consistent. This has direct impact on the way the feature appropriateness is modelled in CUF, see section 3.2.2.
Typing is not obligatory. Type information is fully checked during compilation and runtime type checking is reduced to a necessary minimum.

[5] `t|s` stands for `(t;s) &~(t&s)`, see (König, 1995, p. 16).
[6] In ALE type disjointness can be also very often deduced on the basis of feature appropriateness.
[7] Prolog notation for lists `[Head|Tail]` is also available.

3.2.2 Feature Declarations

Features in CUF are interpreted as unary partial functions. The declaration

(17) $\mathtt{t}_0$ :: $\mathtt{f}_1$:$\mathtt{t}_1$,..., $\mathtt{f}_n$:$\mathtt{t}_n$.

declares features $\mathtt{f}_i$ as functions with domain $\mathtt{t}_0$ and range $\mathtt{t}_i$. In addition to this fairly standard interpretation, polyfeatures are allowed, i.e., a feature can appear in multiple feature declarations. Polyfeatures are also interpreted as unary functions but their domains are restricted to the corresponding types. Polyfeatures can be useful for redeclaration of a feature in a subtype of a type in order to restrict its value. However, polyfeatures lead to processing overhead and should not be overused.
In addition to the above-mentioned boolean operations on types, for types with features (i.e., elements of `cfs`) the operation of feature selection is defined (`f:t`).
It should be noted that the existence of polyfeatures and the open world philosophy of CUF can lead to effects which are not necessarily in accord with the HPSG philosophy. For instance, it is difficult to express total well-typedness of feature structures. With the following declaration

(18) $\mathtt{t}_0$:: f:$\mathtt{t}_1$

the feature term with a new, undeclared feature g

(19) f:$\mathtt{t}_1$ & g:$\mathtt{t}_1$

is a valid description of a subset of $\mathtt{t}_0$. In such a case, the CUF system gives a warning. Should the feature g be introduced for another type which is not declared disjoint with $\mathtt{t}_0$, no warning is given and processing overhead can occur.
In CUF, the above mentioned Head Feature Principle can be defined by means of feature terms:

(20)
```
synsem:loc:cat:head: H &
dtrs:head_dtr:synsem:loc:cat:head: H.
```

It is, however, difficult to combine this definition with the corresponding type as it is easily done in ALE:

(21)
```
phrase cons (synsem:loc:cat:head:H,
               dtrs:head_dtr:synsem:loc:cat:head:H).
```

CUF's sorts offer a potential solution.

3.2.3 Sorts

Sorts in CUF correspond to generalized relations on feature structures (Dörre and Eisele (1991)) and can be understood as a CUF counterpart of Prolog predicates. Sorts are (possibly recursive) feature terms and they can occur where types and feature selection expressions occur with the exception of negation scope (see also Dörre and Eisele (1991)). Sorts have pseudofunctional syntax: an n-ary sort corresponds to an n+1-ary relation. Type declarations can be added for sort parameters (e.g., `append(list,list) -> list` in (22)).
Sorts can be used to implement HPSG relational dependencies (e.g., `append` in the Valency Principle, `valp`) and HPSG principles directly:

(22)
```
valp :=
 subjvalp;
 compvalp.

valp :=
  ~head_struc.
```

```
subjvalp :=
 synsem:loc:subj: Subj &
 dtrs:head_dtr:synsem:loc:subj:append(synsems(Dtr),
                                       Subj) &
 dtrs:subj_dtr: Dtr.

compvalp :=
 synsem:loc:comps: Comps,
 dtrs:head_dtr:synsem:loc:comps:append(synsems(Dtrs),
                                        Comps) &
 dtrs:comp_dtrs: Dtrs.

append(list,list) -> list.
append(elist, L) := L.
append([H|T], L) := [H|append(T,L)].
```

3.2.4 Type Constraints and Lexicon

Sorts provide a way to implement structure-sharing which cannot be encoded in the type system (22). They do not, however, provide the same functionality as type constraints in ALE. The ALE declaration in (21) will be enforced on all subtypes of `phrase`. In CUF, the same effect can be obtained only by redefining/calling again the corresponding sorts. For an example solution in a similar system STUF-III see Geißler and Kiss (1994). The lack of constraint inheritance can possibly lead to a relatively difficult and error-prone implementation of large hierarchies.
CUF does not provide any special mechanisms for lexical entries and lexicon definition. In particular, there is no special syntax for lexical rules (see Geißler (1994) for a possible implementation).

3.3 Computational Aspects

System Modules The CUF system is composed of an incremental compiler and an interpreter. The compiler provides mechanisms for incremental compilation which can be very useful when developing a full-size grammar. However, these mechanisms are not very reliable (see implementation error list in Dörre *et al.* (1996a)) and a full recompilation may be required. Apart from file handling the main tasks of the compiler are type information translation, in particular processing of polyfeatures, and compilation of sort definitions. The compiler can perform expansion of deterministic[8] goals if this is required by the user (default behaviour).
The CUF interpreter uses a bounded SLD-resolution strategy. The user can influence the order in which goals are called and change the default Prolog-like expansion order.

Control Statements Sort declarations in CUF can be augmented with control statements `wait` and `lazy_goal`.
These constructs can be used to fine tune the computation rule. A declaration of the form

(23) `lazy_goal(`<*sort name*>`/`<*arity*>`)`

placed in a control file (see section 3.3) prevents <*sort name*>/< *arity*> goals from being considered for deterministic expansion.
Goal expansion can be delayed by the following statement:

(24) `wait(`<*sort name*>*(*<*delay arg 1*>*,...,*<*delay arg N*>*)* `->` <*delay arg 0*>`).`

where <*delay arg n*> are delay paths for all arguments of the sort (including the implicit argument). All <*sort name*> goals will be delayed until all arguments are more specific than the provided delay declarations. The delay paths are ignored by the interpreter if there are no more undelayed goals or a delayed goal is deterministic.

[8] A goal is called deterministic if there is only one applicable clause for this goal.

Resolution Strategy Deterministic goals which are not blocked by a `lazy_goal` declaration are expanded first. Then the first undelayed goal is expanded. If all goals are delayed, a warning is issued and the first goal is taken for expansion. If the search depth bound is reached, a warning is printed out and the interpreter backtracks to look for another solution.[9]
The ability to influence the computation rule is a crucial feature needed to implement grammars of reasonable size and time performance.

Sort indexing A declaration of the form

(25) `index_table(`<*sort name*>/<*arity*>`)`

can appear in a CUF description file (see section 3.3) in which <*sort name*>/<*arity*> is defined. It is allowed for sorts whose first argument is atomic. As a result an index table for the specified sort is created with the first argument as the key. This feature can be useful when implementing large lexicons with fast look-up.

Foreign Language Interface The relatively rich and well-defined foreign language interface is a strong feature of CUF. It is possible to define foreign sorts which can be general Prolog predicates, i.e., it is possible to call Prolog from CUF. This provides an easy way to incorporate through the corresponding Prolog foreign interface specialized sorts implemented in other languages, e.g. C, which can help achieve better performance. `wait` declarations can be used for foreign sorts in the same way as for native ones.
The CUF implementation defines also the CUF abstract data type which can be directly used to manipulate CUF objects, e.g., in foreign sorts or to construct CUF goals. Also, the implementation provides pretty printing predicates for CUF terms.

File Organization The CUF system distinguishes between description and control files. In description files it is possible to place type axioms, feature and sort declarations and sort definitions. Description files contain the declarative part of a CUF program. It is also possible to define `foreign` sorts, use the indexing declarations and change flag settings. In control files, `wait` declarations are collected. It is also allowed to define Prolog predicates. Distinct commands are provided for loading and unloading description and control files. In order to make processing large grammars easier and facilitate modular grammar development it is possible to define a configuration file specifying all description and control files required for a program.

Debugging Facilities Commands for compiled code inspection are provided. It is possible to trace a proof interactively and choose goals for expansion. Tools for a post mortem inspection of the proof tree are also available. The system provides pretty printing predicates which can be configured for debugging purposes. Also, an Emacs interface is available.

3.4 Development Stage

CUF has been used to implement relatively large grammars, the largest example is the German grammar in the Verbmobil project. The system is available free of charge from `http://www.ims.uni-stuttgart.de/cuf/` and requires Quintus or SICStus Prolog. Currently, support is available from the authors.

Documentation

- Dörre and Eisele (1991), Dörre and Dorna (1993) — presentation of the fundamental CUF concepts;
- Dörre *et al.* (1996a) — a fairly detailed implementation description and commands required to compile and debug grammars;

[9] It is also possible to force expansion of all deterministic goals as the last compilation step.

- König (1995) — an introductory CUF tutorial.

System robustness The current version is fairly stable. The known errors (Dörre *et al.* (1996a)) involve the incremental compilation mechanism and can lead to long development times for large grammars in (rare) cases in which the compiler loops and the whole system has to be restarted.

Recommendation From the linguistic point of view, developing HPSG grammars with CUF can require some additional effort to implement mechanisms often applied in the theory. Implementing large type hierarchies with plenty of constraints with structure-sharing which are very common in current approaches can be relatively difficult. Due to the restricted type system (as defined by Manandhar (1993)), some arbitrary decisions as to the grammar architecture have to be made by the grammar developer.
On the other hand, the system provides features necessary for development of full-size grammars, such as computation rule control declarations, and a reasonable debugging environment. Hence, it is less suitable for direct testing of HPSG grammars in their pure form. The system is considered to be relatively efficient for a constraint based one and could be applied to implement a performance oriented grammar.

4 ALE

4.1 Introduction

The Attribute Logic Engine (ALE, Carpenter and Penn, 1995), created at the Carnegie Mellon University, is an integrated phrase structure parsing and definite clause logic programming system in which terms are typed feature structures. It is based on Kasper-Rounds logic (Kasper and Rounds, 1986), a typed version of which was defined in Carpenter (1992).

4.2 Syntax and Data Structures

4.2.1 Types

ALE is a language with strong typing, which means that every feature structure must have a type. Types are arranged in an inheritance hierarchy declared by the user. The transitive closure of the subtyping relation should form a partial order and be bounded complete (every pair of types which have a common subtype must have a unique most general common subtype). This last condition is not very restrictive since type specifications which violate it can be changed by adding intermediate subtypes, although this operation can decrease the readability of the hierarchy. Example (26) contains type specifications of an English gender hierarchy.

(26)
```
gender sub [animate, neut].
  animate sub [masc,fem].
    masc sub [].
    fem sub [].
```

ALE is based on an open world interpretation of the type hierarchy, i.e., there may be objects in the denotation of a non-minimal type which are not in the denotation of any of the minimal (maximally specific) subtypes of this type. (See section 10 for discussion.)
Types in ALE can be intensional or extensional. For extensional types, two feature structures with all features values being token-identical are token-identical.[10] In ALE, only maximally specific types can be extensional.
There is a special pre-defined type `bot`, which is the unique most general type (every type is a subtype of `bot`).

[10] As we are reminded by Gerald Penn (p.c.), this is actually just an approximation of the definition of extensionality: it breaks down when cyclic structures are considered, cf. (Carpenter, 1992. 126–129).

4.2.2 Feature Structure Descriptions

In ALE, descriptions of feature structures can consist of type names, (Prolog-like) variables, selection (<*feature*> : <*description*>), conjunction (<*desc*> , <*desc*>) or disjunction (<*desc*> ; <*desc*>) of descriptions, inequations (`=/=` <*desc*>), or they can take the form of path equality (<*path*> `==` <*path*>).
ALE allows disjunction and conjunction but does not support general negation of descriptions or polymorphic types. The conjunction of two descriptions is interpreted as unification of feature structures. The path inequality is satisfied by the persistent absence of token-identity while equational descriptions are satisfied by token-identity.
Feature structures in ALE are ***totally well typed.*** This condition of feature appropriateness forces each type to specify which features it can be defined for and what values such features can take. Appropriateness specifications are inherited by a type from its supertypes. Example (27) includes the specification of the list of signs (`sign_list`) which can be empty (`e_list`) or not (`ne_sign_list`). The non empty list has two features representing the head (`hd`) and the tail (`tl`).

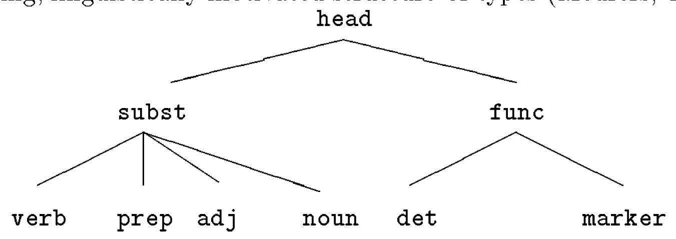
(27)
```
sign_list    sub   [ e_list, ne_sign_list].
ne_sign_list sub   [ ]
             intro [ hd: sign,
                     tl: sign_list].
```

The appropriateness conditions in ALE cannot be cyclic, i.e., no feature specification can contain an attribute which is of the same or more specific type. This means that the specification given in (28) is not allowed since it leads to non-terminating inference. (The only finite feature structures which could meet the specification and be totally well-typed would have to be cyclic and there is no most general cyclic structure.) This condition, of course, does not rule out recursive structures such as lists, cf. (27).

(28)
```
person sub   [male,female]
       intro [father:male,
              mother:female].
male   sub   [].
female sub   [].
```

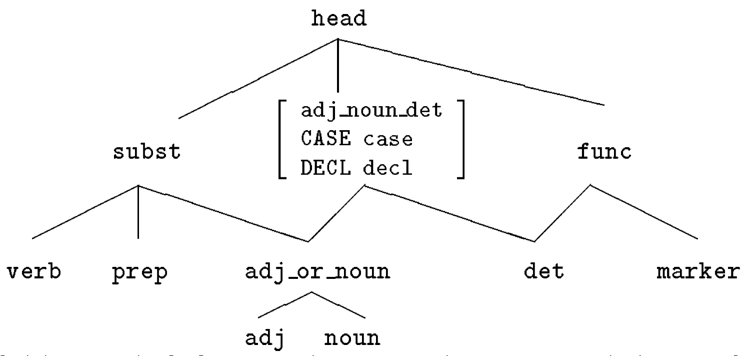
The second condition on appropriateness, the Feature Introduction Condition, states that every feature has to be introduced at a unique most general type (the set of types appropriate for a given feature must have a most general element). This condition makes it impossible to declare the following, linguistically motivated structure of types (Meurers, 1994):

(29)



The problem arises while adding the case and declension attributes. They are appropriate for adjectives, nouns and determiners and there is no single place to introduce them in the above hierarchy. The obvious solution is to introduce a new type `adj_noun_det` serving as the supertype for these three types. The result would, however, violate the condition of bounded completeness. One solution would be to introduce a new type serving as a unique greatest lower bound for `subst` and `adj_noun_det` types. The final hierarchy, far from the original linguistic intuitions, is shown in (30).[11]

[11]As pointed out by Thilo Götz (p.c.), the difference between (29) and (30) becomes crucial under open-world interpretation of these type systems: in this case there might be objects in denotations of the newly introduced types which are not in denotations of any maximally specific types. Hence, the reformulation of the type system discussed here changes linguistic ontology in a non-trivial way.

(30)


A type definition can include constraints expressing some restrictions on feature structures of the type. Constraints are introduced by the `cons` operator and may consist of arbitrary descriptions. Example (31) describes a type `x` as having two non token-identical values for features `f` and `g`.

(31)
```
x cons (f:X, g:=/=X).
```

Type constraints can also be expressed with the help of a relational constraint introduced by the optional operator `goal`. Assuming that the `agreement_principle` is encoded via definite clauses, example (32) expresses the fact that every feature structure of type `word` has to fulfill this condition.

(32)
```
word cons W goal agreement_principle(W).
```

4.2.3 Macros

Abbreviations for ALE descriptions can be defined by means of macros. They cannot contain calls to definite clauses and cannot be recursive but can be parameterized. Macro definitions have the following form:

(33) <*macro_name*>(<*parameters*>) `macro` <*description*>.

The macro call begins with the '@' operator. The macros allowing for the traditional list notation are built-in (and are called without @, e.g., `[one, two]`).

4.2.4 Lexical Entries and Rules

Lexical entries in ALE are encoded in terms of rewriting rules and have the following form:

(34) <*word*> `--->` <*description*> .

Example (35) shows a sketchy description of the word 'ball'.

(35)
```
ball ---> (syn: noun,
           sem: sphere
           ;
           syn: noun,
           sem: social_event).
```

Lexical rules in ALE are just a special form of unary phrase structure rules with an additional mechanism to alter morphology. They operate on feature structures described by lexical entries and they are processed during the compilation of the grammar, when the closure under lexical rule application is computed and new lexical feature structures are added. Lexical rules can be applied to their own output or to the output of other lexical rules. There exists a special variable defining the allowable depth of lexical rule application. The syntax of lexical rules is illustrated in (36), which shows a part of the lexical rule describing conversion of English singular nouns into plural.

(36)
```
plural_n lex_rule
    (n, num:sing)   ** >   (n, num:plu)
    morphs
       goose becomes geese,
       (X,F) becomes (X, F, es) when fricative(F),
       (X,ey) becomes (X, [e,y,s]),
       (X,y) becomes (X, [i,e,s]),
       X becomes (X,s).
```

The first element of the rule (`n, num:sing`) is a description selecting lexical entries to which the rule can be applied. After the '** >' operator, the description of the output entry is given. In this case, the value of the `num` feature is changed. After the `morphs` operator, rules for converting string patterns are specified. Within the morphological sequences, atoms can be used as a shorthand for the lists of their characters. Lexical rules can have procedural attachments (introduced by `when`). For each category satisfying the input description, a lexical rule generates new feature structures which satisfy the output description (with multiple solutions allowed). As no default values are assumed, all feature values from the input feature structure have to be explicitly stated in the output description (even if they remain unchanged). After the category information is processed, the morphological part of the rule is used to produce one word for each word matching the conditions listed. Only the first match is used.

4.2.5 Grammar Rules

Grammar rules in ALE represent phrase structure schemata with procedural annotations allowed. Rules are not expanded to features structures at compile time but are compiled to some intermediate form which prevents useless rule instantiation. The rules are evaluated from left to right. Each rule consists of a mother category, at least one daughter category (introduced by the `cat>` operator) along with additional procedural attachments expressed as ALE definite clauses (after the `goal>` operator). The operator `cats>` can be used to represent a list of daughters of unspecified length. Example (37) shows a part of the possible encoding of HPSG's Schema 2.

(37)
```
schema2 rule
  (phrase,
   synsem:loc:cat:subj: Subj,
   head_dtr: HeadDtr,
   comp_dtrs: Comps)
===>
 cat> (HeadDtr,
       word,
       synsem:loc:cat:(subj: Subj,
                       comps: Comps)),
 cats> Comps,
 goal> case_principle(HeadDtr).
```

4.3 Computational Aspects

System availability and requirements ALE is available from:
`http://macduff.andrew.cmu.edu/ale/index.html`.
It is implemented in Prolog and either a SICStus or Quintus Prolog compiler is required.

Efficiency The authors estimated the speed of ALE running definite clause programs at roughly 15% of the speed of the SICStus 2.1 interpreter. In practice this gives an acceptable parsing time. Also, incremental compilation feature is very useful.

Procedural model Definite clause programs and grammars are compiled into a Prolog abstract machine instructions which are interpreted by an emulator compiled from type specifications (Carpenter and Penn, 1994, p. 1). For parsing, ALE compiles the grammar specification into a bottom-up, dynamic chart parser. Definite clauses are also compiled into Prolog. The convenient feature is incremental compiling - it is possible to compile separately the type hierarchy, type constraints, the attribute-value logic and definite clauses.

Debugging It is possible to test the results of the program compilation. One can check type existence, type subsumption and unification, feature existence and feature introduction in a given type, type constraints and the inherited appropriateness function. It is also possible to see the complete type specification, lexical entries and definite clauses in feature structure format. Descriptions can be evaluated in order to find their most general satisfiers. ALE contains a mini-interpreter allowing one to traverse and edit an ALE parse tree.

Graphical user interface ALE has no graphical interface of its own, but there are at least two front ends (HDRUG and Pleuk) adapted for ALE.[12] An Emacs-based interface is also available. Information about these programs and their ftp addresses can be found on the ALE www page.

4.4 Development Stage

Support The system is stable, support from the authors is available and the group of users is large. A new version is in preparation, whose features will include: functional syntax for relations, restricted negation parameter, parametric types, a tracer and a generator (Gerald Penn, p.c.). (This version will be, however, upward compatible.)

Documentation

- Carpenter (1992) — the theoretical background of ALE;
- Carpenter and Penn (1995) — the user's guide of the system's version 2.0.1.

Recommendation Relatively numerous applications of ALE have proved that it can express and process rather complex grammars. It can be recommended for non-toy applications in spite of the fact that it does not offer any support for large scale projects (e.g., version management).

5 ALEP

5.1 Introduction

The Advanced Linguistic Engineering Platform (ALEP, Simpkins, 1994b), created at the Advanced Information Processing Group at BIM, Belgium, is a platform for developers of linguistic software and provides a distributed, multitasking (Motif-based) environment containing tools for developing grammars and lexicons. The system comes with a unification-based formalism and parsing, generation and transfer components. A facility to implement two-level morphology is also provided.

The ALEP formalism has been developed on the basis of the ET-6.1 project (Alshawi *et al.*, 1991)[13]. The core of the ET-6.1 formalism follows a rather conservative design and ALEP is very much a traditional rule based grammar development environment. The rules are based on a context free phrase structure backbone with associated types. As a result, implementing HPSG grammars in ALEP is far from natural. As stated in Genabith *et al.* (1994), "one can write (in ALEP) not 'HPSG' grammars but 'HPSG-inspired' grammars." The main problem is the

[12] Actually, Pleuk relies on a GM interface, which is not supplied with SICStus Prolog anymore (Gerald Penn, p.c.).

[13] The most significant changes concern the type system and the definition of categories.

lack of relational extension and a very limited type system. This results in poor modularity and maintainability of grammars.

5.2 Syntax and Data Structures

5.2.1 Type Declarations

ALEP is strongly typed: the arity of every feature term, the names of its attributes and the type of their values have to be specified in type declarations. The syntax of such declarations is as follows:

(38) `type(`<*type_name*>`:{`<*feature_declarations*>`},` <*doc_atom*>`).`

where *doc_atom* is an arbitrary documentation string and a *feature_declaration* is of the form:

(39) <*feature_name*> `=>` <*feature_type*>.

A *feature_type* can be one of the predefined types: atom, list, tuple, functor, Boolean or a user defined type, cf. examples below:

- atom: `num => atom({sing,plural})`,
- list: `gaps => list` (description of a list containing elements of any (but one) type), `subcat => list(type({args}))` (list of elements of type `args`),
- tuple (representing a pair of elements): a declaration of `gaps => tuple` allows in rules and lexical entries values such as `gaps => (gaps_in, gaps_out)`,
- functor: a definition of `lf => functor` permits use of any Prolog term, e.g. `lf => exists(X,man(X))`,
- Boolean type (<*feat*> `=> boolean([`<*atom_sets*>`]))` allows for the operators of disjunction (`;`), conjunction (`&`) and negation (`~`), e.g. `agr => boolean ([{sg,pl}, {first, second, third}])` mandates a value of `agr => ((sg&second);third))`,
- user defined type:

```
feat => type (syn: { cat => atom({n,v,det,pn,s,np,vp}),
                     subcat => list(type({sign:{}}),
                     lf => functor}, ' comments').
```

Within a type declaration, it is possible to define default values for attributes. Structure sharing can be specified by means of variables.
There are no sets (and set operations) so sets have to be modelled as lists; and because of the lack of functional relations, such as `append`, the operations on lists are themselves quite complicated. Using difference lists is only a partial solution to the problem since they model only the disjoint set union and make signs bigger and less readable.

5.2.2 Type Hierarchy

Type structures in ALEP are strictly hierarchical, and each type can have only one supertype — no multiple inheritance is supported. Type hierarchy is used only during compilation: after that the structures are present in the database in their fully expanded form. The syntax of hierarchical relationship declarations as follows:

(40) <*super_type*> `>` <*sub_types*>.

where <*sub_types*> can be a type hierarchy itself, cf. (41) below:

(41) `animal > { domestic  > {dog,cat}, wild}.`

The attributes introduced by a supertype are inherited by all of its subtypes. The subtypes of a certain type cannot further specify attribute values already specified on this type (with the exception of a Boolean type).
ALEP allows for specifying many type hierarchies at the same time, e.g. for different languages. A special kind of rules for transferring linguistic structures from one type system to another is provided.

5.2.3 Macros, Abbreviations

Macros are an integral part of the ALEP formalism. Macro definitions have the form of:

(42) `macro ( <name> [ <arguments> ], <body> ).`

Macros can include calls to other macros. The following example shows the definition of a macro which expands to type `agr` using arguments as attribute values:

(43)
```
macro(agr[G,N,P,D], agr:{group=>G,num=>N,
                         pers=>P,def=>D}).
```

Another way of simplifying the code is by using category feature declarations. They allow a single feature to be an abbreviation of a complete type by the following definition:

(44) `<ld_category_feature> ( <variable>, <LD>).`

The definition given in (45) allows for writing `np < [det n]` instead of the rule given in (46).

(45)
```
ld_category_feature(Cat,ld:{syn:{cat=>Cat}, sem{}}).
```

(46)
```
ld:{syn:{cat=>np}} <
  [ ld: { syn : { cat=> det}}
    ld: { syn: { cat => n }} ].
```

5.2.4 Lexical Entries

A lexical entry has a form of <*TAG*> ~ <*LD*> where *TAG* is an identifier and *LD* is a linguistic description. An example of a lexical entry is as follows:

(47)
```
some ~
  ld:{ spec => apply:  {language=>en},
       pho  => phones: {string[some|R],rest=> R},
       syn  => syntax: {cat => det},
       log  => logform:{lf  =>exists(X,N),var=>X,
                        scope=>N}}.
```

The example describes the phonetic form of the word `some`, its category (`syn`) and semantics (`log`). The first feature, `spec`, indicates which language the entry belongs to.

5.2.5 Phrase Structure Rules

A phrase structure rule is a linguistic structure of the form
<*LD*> `<` `[` <*daughters* >`]`.
where *LD* is a linguistic description and *daughters* is the list of dominated LDs. The daughters of a phrase structure can be specified independently of their order using braces { }. The disjunction of daughters and their optionality can also be expressed. An illustrative example of typical S→ NP VP rule is shown in (48) (Schütz, 1994a) (assuming that `ph` is a subtype of `ld`).

(48)
```
ph:{syn  => syntax:{cat=>s, gaps=>gaps:{in=>[],
                                        out=>[]}},
          log  => logform:{lf=>VP},
          spec => specifier:{lang:en,gram=>toy,
                                     id=>'S_NP_VP'} }
      <  [ ph:{ syn => syntax: {cat=>np},
                log => logform:{lf=>NP} },
           ph:{ syn => syntax: {cat=>vp},
                sem => semantics: {subj=>NP},
                log => logform {lf=>VP} }}
          ].
```

The algorithms used for synthesis and analysis implement a head-based parser and require head daughters of rules to be specified. They are described by declarations of the form (49) in which *specifier_names* describe which rules this declaration is valid for:

(49) `select_analysis_head (` <*specifier_names*>`,` <*root_ld*>`,` <*head_ld*>`).`

To express that the second daughter (i.e. VP rather than NP) is the head of the rule cited in (48) one should write (assuming that `phase_parse` is a specifier subsuming the cited specifier description):

```
(50)    select_analysis_head([phase_parse],
                             ph: {syn=> syntax:{cat=>s}},
                             ph: {syn=> syntax:{cat=>vp}}).
```

5.3 Computational Aspects

System availability and requirements ALEP has been designed and implemented by the Advanced Information Processing Group at BIM, Belgium and SEMA for the LRE project of the European Commission. The basic version (not for system developers) runs on SUN Sparc and Intel 486 machines under SunOS 4 and 5 (Solaris). 70Mb of swap memory should be reserved, recommended memory size: 32Mb. ALEP is not a public domain system, information on the system can be obtained from http://www.anite-systems.lu/alep/. The technical support is availble, e-mail: alep-support@anite-systems.lu.

Procedural model, efficiency ALEP is implemented in Prolog. It includes two different analysis algorithms (a non-deterministic bottom-up head-out parser and a structure sharing algorithm) and one synthesis algorithm. All are based on the notion of 'head' which is not exactly the linguistic head but serves as the starting point of analysis or synthesis. The choice of the rule head element can change failure/success result of parsing. The parsing efficiency depends on the coding techniques used. Generally, strongly lexicalized grammars tend to be slower than less lexicalized ones (the bottom-up algorithm is sensitive to lexical ambiguity).
ALEP divides linguistic processing into a number of distinct sequential phrases: segmentation/two-level morphology, lifting, parsing, refinement, synthesis, lowering. The rules used in these processes have the same format and are distinguished by the use of specifiers (which belong to the special type `spec`). There are many ways to control the analysis. One of them is defining *key declarations.* Key declarations are given as a part of a type system and they are used to index rules and lexical entries. The key declaration identifies an attribute value which is to be used as a key. They are not required but make the linguistic processing more efficient.

Debugging A post-mortem debugging provides an interface to a stored log of events from linguistic processing.

Environment, graphical user interface The ALEP environment includes many tools for text handling, lingware development and debugging, and linguistic processing. There are also several additional tools for browsing and editing object descriptions, writing, compiling and executing

tasks, online browsing of the manuals. Linguistic and environment resources can be shared in public or private databases. The system comes with demonstration lingware (German analysis, transfer and generation in English).
ALEP has a graphical, customisable interface based on MOTIF and Emacs. There are possibilities for pretty printing and handling large structures produced during linguistic processing.

5.4 Development Stage

Support The first operational version of the system came in mid-1993. In 1996, version 3.0 became available. The system is sufficiently stable and rich, packaged with user documentation and installation aids.

Documentation The documentation of ALEP is relatively large (hundreds of pages) and consists of many documents: descriptions of the theoretical background of the system, user guides for the ALEP environment and supporting tools, notes on the included lingware (the list of more than 60 related titles is included in Groenendijk and Simpkins (1994)). Both Dvi and PostScript files are supplied in ALEP distribution. Info files are also available on line using the *xminfo* graphical viewing tool.

Recommendation The size of the system and its rather complicated structure result in great effort needed at the beginning so ALEP should not be recommended as the experimental tool for novices. Also, many formal devices of HPSG are not available. On the other hand, the efficiency of the system and its rich environment make an attempt at implementing "HPSG-style" grammars worth considering. Bredenkamp and Hentze (1995) and Genabith *et al.* (1994) show some interesting ideas of using ALEP mechanisms to model elements of HPSG. Because of the very rich environment ALEP may be attractive for large scale projects.

6 PAGE

6.1 Introduction

PAGE (Platform for Advanced Grammar Engineering) is a grammar development environment and run-time system which evolved from the DISCO project (Uszkoreit *et al.*, 1994) at the German Research Center for Artificial Intelligence (DFKI). The system has been designed to facilitate development of grammatical resources based on typed feature logics. It consists of several specialized modules which provide mechanisms that can be used to directly implement notions of HPSG, LFG and other grammar formalisms. PAGE provides a general type system ($\mathcal{TDL}$), a feature constraint solver ($\mathcal{UD}i\mathcal{N}e$), a chart parser, a feature structure editor (FEGRAMED), a chart display to visualize type hierarchies and an ASCII-based command shell. In the remainder, we will concentrate mostly on the $\mathcal{TDL}$ module which is best documented and constitutes the fundamental part of the environment. The description of $\mathcal{TDL}$ is based on Krieger and Schäfer (1994a) and Krieger and Schäfer (1994b).[14]

[14]Unfortunately, we did not succeeded in installing and testing the system, so the information provided here and in section 10 is based solely on the $\mathcal{TDL}$ documentation. This also means that our understanding of $\mathcal{TDL}$ is lesser than that of the other systems considered in this report.

6.2 Syntax and Data Structures

6.2.1 Type Hierarchy

The type information is specified in the $\mathcal{TDL}$ (Type Definition Language) module. $\mathcal{TDL}$ provides HPSG-style type hierarchies and it depends on $\mathcal{UD}i\mathcal{N}e$, which in turn is an untyped feature constraint solver.

$\mathcal{TDL}$ distinguishes the following types: sort types, which are unstructured, i.e., they can have no features, avm types, which allow features, built-in types, which are provided by Common Lisp, and atoms.[15] Sorts are given the closed world semantics, i.e., a conjunction of two sort types is inconsistent unless one of the types subsumes the other or they have an explicitly specified greatest lower bound. In contrast, avm types have a (default) open world semantics, i.e., the greatest lower bound of two avm types a and b which share no common subtype is $a \wedge b$. It is also possible to declare exhaustive and disjoint partition of types or specify incompatible types (Krieger and Schäfer, 1994a, p. 11).

The type hierarchy comes basically with no pre-defined structure, although some default type definitions and declarations are provided and it is up to the grammar writer to load them. They include, among others, declarations of basic built-in types (numbers, strings, symbols, atoms), the list type, the difference list type, and basic rule types (Krieger and Schäfer, 1994b, p. 11).

Subtypes can be defined via the operator `:<` which defines the hierarchy of sorts and types without feature declarations. The type hierarchy constitutes a partial order (Krieger and Schäfer, 1994a, p. 26).

6.2.2 Feature Structures

The following declaration defines the type `v_agr` using the general syntax of type definition (`:=`):

(51) a. `v_agr := [GENDER PERSON].`

b. $\begin{bmatrix} \textit{v\_agr} \\ \text{GENDER } [\,] \\ \text{PERSON } [\,] \end{bmatrix}$

Most general feature structure of type `v_agr` is shown in (51b). As the definition provides no type constraints for the features `GENDER` and `PERSON`, the top avm type `[]` is assumed. Attribute values can be atoms, feature structures, disjunctions, distributed (named) disjunctions, lists, functional constraints, template calls or negated values. Also, coreference tags (indicated by the hash sign `#`) are allowed, which makes the type system general in the sense defined in Manandhar (1993). Thus, the following $\mathcal{TDL}$ expression

(52)
```
headed_phrase :=
  [SYNSEM.LOC.CAT.HEAD #Head,
   DTRS.HEAD_DTR.SYNSEM.LOC.CAT.HEAD #Head].
```

describes a feature structure with the corresponding structure sharing:

(53) $\begin{bmatrix} \textit{headed\_phrase} \\ \text{SYNSEM}|\text{LOC}|\text{CAT}|\text{HEAD } \boxed{1} \\ \text{DTRS}|\text{HEAD-DTR}|\text{SYNSEM}|\text{LOC}|\text{CAT}|\text{HEAD } \boxed{1} \end{bmatrix}$

[15] Atoms in $\mathcal{TDL}$ can be strings, numbers or (Lisp) symbols. Symbols need to be (single) quoted.

This means also that all subtypes of `headed_phrase` will inherit the structure sharing. Coreferences are allowed in the scope of negation, so it is possible to encode that two features must not share values.

6.2.3 Logical Operations

$\mathcal{TDL}$ atoms, types and feature structures can be combined using the logical operators: `&` (conjunction, unification, inheritance), `|` disjunction, `^` (exclusive or) and `~` (negation).[16]

The operator `&` is treated differently depending on the context it appears in. It is interpreted as (multiple) inheritance if a type specification occurs in a conjunction at the top level, i.e., with no path specified. In the example below, the type `joan` from (54) is conjoined with the information about phonology, `[PHON 'joan]`. The resulting type inherits all constraints of the types `headed_phrase` and `phon-orth`:

```
(54)    phon-orth := [PHON, ORTH].
        joan := phon-orth & headed_phrase & [PHON 'joan].
```

Expanding the type `joan` results in the following feature structure:

(55)
$$\begin{bmatrix} \textit{joan} \\ \text{PHON } \textit{joan} \\ \text{ORTH } [\ ] \\ \text{SYNSEM}|\text{LOC}|\text{CAT}|\text{HEAD } \boxed{1} \\ \text{DTRS}|\text{HEAD-DTR}|\text{SYNSEM}|\text{LOC}|\text{CAT}|\text{HEAD } \boxed{1} \end{bmatrix}$$

$\mathcal{TDL}$ offers both simple and distributed disjunctions. The distributed disjunction

```
noun := [NUMERAL %1(yes, no),
         CASE    %1(str, lex)].
```

corresponds to the following simple one

```
noun := [NUMERAL yes, CASE str] | [NUMERAL no, CASE lex].
```

Distributed disjunctions can involve multiple attributes and the value lists can be of an arbitrary length. In both kinds of disjunction it is not allowed to use coreferences referring to any values outside the alternative. However, in the case of distributed disjunctions it is allowed to declare coreferences referring to a value in the corresponding position of the alternative, e.g.:

```
(56)    noun := [NUMERAL %1(yes & #1, no & #2),
                 AGR     %1([CASE str, NUMERAL #1],
                            [CASE lex, NUMERAL #2])].
```

6.2.4 Functional Constraints

The mechanism of functional constraints can be used to compute a value of an attribute using a Lisp function. The function can be given the values of other attributes in a feature structure.

[16] $\mathcal{TDL}$ provides also non-monotonic definitions for avm types which can be used to implement defaults or lexical rules, see (Krieger and Schäfer, 1994b, p. 23).

Coreference tags are used to transmit the values to the function call. The following example (taken from Krieger and Schäfer (1994b)) specifies that the value of the attribute `WHOLE` has to be computed using the LISP function `String-Append`:

(57)
```
add-prefix :=
    [ WORD  #word,
      PREFIX #prefix,
      WHOLE #whole]
    where
      (#whole = String-Append (#prefix, #word)).
```

`where`-clauses can be used in an analogous way to specify constraints on coreference values within a structure:

(58)
```
equal := [A #a, B #b] where (#a = #b).
```

Coreference tags can be attached to any $\mathcal{TDL}$ term by the `&` connector, (Krieger and Schäfer, 1994b, p. 22), i.e., if `#a` is specified as the feature structure in (51a), this is encoded in the following declaration:

(59)
```
equal := [A #a & v_agr, B #b] where (#a = #b).
```

Functional constraints can also appear in rules as procedural attachments. The evaluation of functional constraints is postponed until all parameters are instantiated.

6.2.5 Lists

$\mathcal{TDL}$ offers a rich list notation which leads to a relatively simple encoding of grammar principles and possibly more efficient grammars. The empty list is represented by the sort `*null*` and nonempty lists are of the avm type `*cons*` with features `FIRST` and `REST`. The notation allows for:

(60) plain lists inclosed in `< >`: e.g.,

- `simple := [the-list <first, second, third>].`
- $$\begin{bmatrix} \textit{simple} & \\ \text{THE-LIST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{first} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{second} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{first} \\ \text{REST} & \text{*null*} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(61) dotted lists inclosed in `< >`, with the last element separated by `.`; the last element fills the last `REST` attribute: e.g.,

- `dotted := [the-list <first, second .  third>].`
- $$\begin{bmatrix} \textit{dotted} & \\ \text{THE-LIST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{first} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{second} \\ \text{REST} & \text{third} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(62) difference lists inclosed in `<!  !>`, with the special `LAST` attribute at the top level which shares its value with the last `REST` attribute: e.g.,

- `difference := [the-list <!  first, second !>].`
- $$\begin{bmatrix} \textit{difference} & \\ \text{THE-LIST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{first} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{second} \\ \text{REST} & \boxed{1} \end{bmatrix} \end{bmatrix} \\ \text{LAST} & \boxed{1} \end{bmatrix}$$

(63) plain lists with open end; the last element is ... and the value of the last `REST` attribute is the top type `[ ]`:

- `open := [the-list <first, second, ...>].`
- $$\begin{bmatrix} \textit{open} & \\ \text{THE-LIST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{first} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{second} \\ \text{REST} & [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(64) plain lists of a particular type and a specified length, e.g., lists of type `sign` of length 0, 1 or 2 can be specified as:

- `short-sign := [the-list <...>:sign:(0,2)].`
- $$\begin{bmatrix} \textit{short-sign} & \\ \text{THE-LIST} & \left\{ \begin{array}{l} \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{sign} \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{sign} \\ \text{REST} & \text{*null*} \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \text{sign} \\ \text{REST} & \text{*null*} \end{bmatrix} \\ \textit{*null*} \end{array} \right\} \end{bmatrix}$$

6.2.6 Rules

$\mathcal{TDL}$ provides the operator `-->` which can be used to define rules in the grammar. `-->` does not assume any specific interpretation of the rule definitions. In fact, this interpretation can be specified by user who can supply a function which translates rule definitions into feature structures. This is particularly useful when grammars are to be processed by different parsers using different rule representations.

The following definitions are provided and will be loaded with other built-in types:[17]

(65)
```
*rule* :=
   [ ] --> < ... >,
   status: RULE.
unary-rule :=
   *rule* --> < [ ] >.
binary-rule :=
```

[17]As noted before, the grammar writer may prefer not to load the definitions of the built-in types.

```
    *rule* --> < [ ], [ ]>.
ternary-rule :=
    *rule* --> <[ ], [ ], [ ]>.
```

The keyword `status:` is required if the rules are to be processed by the DISCO parser which comes with PAGE. In this case, the rule

(66)
```
np :=
  binary-rule --> <Det, Noun>.
```

is translated into (Krieger and Schäfer, 1994b, p. 24):

(67)
$$np \Rightarrow \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{binary-rule}} \\ \textsc{args} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{*cons*}} \\ \textsc{first} & \text{det} \\ \textsc{rest} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{*cons*}} \\ \textsc{first} & \text{noun} \\ \textsc{rest} & \text{*null*} \end{array}\right] \end{array}\right] \end{array}\right]$$

6.2.7 Templates

$\mathcal{TDL}$ provides a (parameterized) templates abbreviation facility. It is particularly useful in writing large grammars. Template definitions in $\mathcal{TDL}$ must not be recursive. Parameters used in templates are preceded by the `$` sign and can be assigned a default value via the `=` symbol, cf. (68), Krieger and Schäfer (1994b).

(68)
```
a-template($inherit=*avm*, $attrib=PHON, $value):=
          $inherit & [ $attrib #1 & $value,
                       COPY #1].
```

(69a) defines the `top-level-call` type which corresponds to the structure in (69b):

(69) a. `top-level-call := @a-template( ).`

b.
$$\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{top-level-call}} \\ \textsc{phon} & \boxed{1} \\ \textsc{copy} & \boxed{1} \end{array}\right]$$

The `inside-call` type can be defined in the following way:

(70)
```
inside-call :=
    [ TOP-ATTRIB @a-template ($inherit = phon,
                              $value = "hello",
                              $attrib = MY-PHON) ].
```

Note that the default parameters' values (`*avm*` and `PHON` for `$inherit` and `$attrib` respectively) are overridden. (70) will result in the following feature structure:

(71)
$$\left[\begin{array}{ll} \multicolumn{2}{l}{\textit{inside-call}} \\ \textsc{top-attrib} & \left[\begin{array}{ll} \multicolumn{2}{l}{\textit{phon}} \\ \textsc{my-phon} & \boxed{1}\,\text{“hello”} \\ \textsc{copy} & \boxed{1} \end{array}\right] \end{array}\right]$$

6.2.8 Instances

Although $\mathcal{TDL}$ does not have any special syntax for lexical entries, it provides the concept of (type) instances which are useful to encode the lexicon. Instances do not belong to the type hierarchy and in this respect they differ from types. They can inherit from types and their syntax is similar to that of types. Generally, no types inherit from types corresponding to lexical entries and the number of lexical entries is large. Thus, the main advantage of using instances is keeping the type hierarchy small. Multiple instances of a type are allowed and they can have additional idiosyncratic information. E.g., in (72), `john` and `bob` are instances of the type `noun & sing-masc`, i.e., inherit from the types `noun` and `sing-masc` and have additional, specific `PHON` value added.[18]

(72)
```
john := noun & sing-masc & [PHON "john"].
bob := noun & sing-masc & [PHON "bob"].
```

The parser is responsible for taking into account all possible instances of a given type.

6.3 Computational Aspects

6.3.1 Processing Type Information

Symbolic Simplification Before type hierarchy information is used, a strictly symbolic simplification is applied to any $\mathcal{TDL}$ expression until a normal form is obtained (it is possible to choose between the conjunctive or the disjunctive normal form). This simplification procedure helps to avoid unnecessary unifications and calls to the hierarchy using simple syntactic reduction rules well-known form the propositional logic, e.g., de Morgan's laws, distributivity, absorption, idempotence, etc.

$\mathcal{TDL}$ introduces a total lexicographic order on type expressions. All subformulas (conjuncts or disjuncts) in a normal form are ordered according to this order. Sorted normal forms can be compared without having to consider different permutations of subexpressions, i.e., the commutativity law is not included among the reduction rules, which avoids the time complexity and possible termination problems it introduces.

Reduction rules which use type inheritance information are applied at compilation time and runtime.

Memoization A memoization algorithm is applied in $\mathcal{TDL}$ in order to reuse the preprocessed type expressions. Precomputed simplified types are stored in four memoization tables (implemented via LISP hash tables): for disjunctive and conjunctive normal form and with or without type hierarchy information. These intermediate results can be reused for further computations. A number of functions allow to manipulate information stored in the memo tables.

Controlled Type Expansion $\mathcal{TDL}$ introduces a mechanism of type expansion which is comparable to total well-typedness in ALE.[19] Similarly to unification, type expansion is a structure-building and consistency-checking operation.

[18]The reader might be wondering, how the system knows that `john` and `bob` are instances and not types since the definitions in (72) look precisely like type definitions we presented earlier. In fact, a $\mathcal{TDL}$ grammar consists of a sequence of possibly nested environments and specific definitions and declaration are allowed in particular environments. E.g., instances are allowed in the instance environment which starts with the command `begin :instance.` and continues to the closing `end :instance.`. All definitions like in (72) which occur in the `:instance` environment are treated as instance definitions. Other environments are e.g. `:template`, `:type`, `:domain`.

[19]Feature structures in $\mathcal{TDL}$ do not have to be totally well-typed. Well-typedness as well as feature appropriateness can be optionally checked at compilation time and runtime.

Typically, $\mathcal{TDL}$ grammars operate on unexpanded, i.e., normally small feature structures reflecting only relevant features, which is beneficial from the efficiency point of view. In the case of recursive type definitions, operating on unexpanded type definitions is the only possibility. However, all type constraints have to be unfolded in the final solution since they all contribute to grammaticality checking.

In general, type expansion may be inefficient and can cause termination problems (as in the case of recursive avm types) (Zajac, 1992). $\mathcal{TDL}$ provides sophisticated tools for the control of type expansion. Every feature structure can be associated with the success/failure potential for unification and the order of type expansion can depend on this information: in conjunctions, constraints with the highest failure probability are evaluated first, in disjunctions, constraints with the highest success probabilities are considered first.

It is possible to restrict type expansion only to certain (sub)types, specify paths for expansion or indicate the depth of expansion for a type. Also, it is possible to define a partial order in which attributes are considered for expansion and interactively choose disjunction alternatives for expansion. These options can be used independently or can be combined or changed dynamically during processing. In the case of recursive types, the implemented memoization technique leads to lazy expansion, i.e., types which have already been expanded under a subpath and occur at a node where no other features or types are specified are not expanded again. In general, it is the responsibility of the grammar implementor to ensure that type expansion will terminate by supplying the corresponding control statements.

6.3.2 Implementation

GUI Two graphical tools are provided which are very useful during the grammar development cycle. FEGRAMED is an interactive feature structure editor and the $\mathcal{TDL}$ grapher is an application displaying type hierarchies. FEGRAMED allows the user to view feature structures corresponding to types and instances. Hiding irrelevant information is a particularly useful feature when viewing large feature structures. FEGRAMED commands are among PAGE shell commands and can appear in $\mathcal{TDL}$ grammars.

The $\mathcal{TDL}$ grapher can be used to display the type hierarchy defined in a grammar. It is possible to call FEGRAMED, $\mathcal{TDL}$2LaTeX and execute $\mathcal{TDL}$ commands (e.g., expand types) from it, e.g., in order to debug a grammar.

Also, feature structures can be printed out in LaTeX format using $\mathcal{TDL}$2LaTeX. $\mathcal{TDL}$2LaTeX is a configurable tool generating LaTeX output which enables the user to add $\mathcal{TDL}$ feature structures to LaTeX documents. It is possible to print large structures by using various fonts and/or hiding irrelevant information.

A $\mathcal{TDL}$ interface for Emacs is available as well. However, it works only with the ALLEGRO COMMON LISP implementation of PAGE.

Specialized Components As mentioned in the introduction, PAGE is a multicomponent system with specialized components to handle the type information ($\mathcal{TDL}$) and feature constraints and unification ($\mathcal{UD}i\mathcal{N}e$). PAGE comes with the DISCO parser (bidirectional chart parser) which can be used to test grammars. The parser can be replaced by a user-defined one. The system does not provide any specialized components to handle morphology, transfer, etc.

System Requirements and Availability PAGE is written in LISP and implementations in Allegro Lisp, Lucid Lisp and CLISP under UNIX are advertised to be available. However, the PAGE shell and the $\mathcal{TDL}$ Emacs mode are available only with ALLEGRO COMMON LISP. The graphical interface is written in portable C and uses X Window System and Motif to implement the graphical tools.

The system is available on request from the Computational Linguistics group at the DFKI Saarbrücken (`http://www.dfki.uni-sb.de`).

6.4 Development Stage

Documentation, Support PAGE comes with a description of its components but no consistent documentation of the system as a whole is provided. The PAGE shell provides a help command which lists all defined commands. Limited support from the authors can be expected. However, a fairly comprehensive documentation on $\mathcal{TDL}$ is available.

- `http://cl-www.dfki.uni-sb.de/cl/systems/` — introductory information on PAGE;
- Krieger and Schäfer (1994b) — description of the $\mathcal{TDL}$ system from the user's point of view with example runs;
- Krieger and Schäfer (1994a) — underlying concepts of $\mathcal{TDL}$.

Recommendation $\mathcal{TDL}$/PAGE has been used to implement a few large grammars. The biggest grammar implemented in $\mathcal{TDL}$ is probably the DISCO grammar which contains a type hierarchy with several hundreds of (non-lexical) types. $\mathcal{TDL}$ has been used to implement grammars in the Verbmobil project and to develop English grammars at the CSLI.

In general, PAGE is an interesting system for HPSG-style grammars. It provides full boolean algebra on feature structures, a rich notation and various mechanisms which can be used to implement grammars relatively close to the theory. Powerful graphical tools facilitate the grammar development process significantly.

7 ProFIT

7.1 Introduction

Prolog with Features, Inheritance and Templates (ProFIT) is an extension of Prolog. ProFIT programs consist of data type declarations and Prolog definite clauses. ProFIT provides mechanisms to declare an inheritance hierarchy and define feature structures. Templates are widely used to simplify descriptions, encode constraint-like conditions and abstract over complex data structures. The system has been developed at Universität des Saarlandes, Saarbrücken.

7.2 Syntax and Data Structures

7.2.1 Types

Types in ProFIT constitute a hierarchy with the most general type `top`. ProFIT's type hierarchy forms a partial order but need not be a bounded complete partial order, which contrasts, e.g., with the approach of Carpenter (1992). All types are declared as subtypes of `top`. The immediate subtypes of `top` (but no other type) need not be introduced in the same declaration, cf.(73) (and even not in the same file) but they have to be declared only once, before they are used in clauses.

The subtypes which are declared in the same list are disjoint, i.e., `type1` and `type2` (as well as `subtype1` and `subtype2`) are disjoint.

(73)
```
top > [type1, type2].
  type1 > [subtype1, subtype2].
top > [type3].
  type3 > [subtype3].
```

Immediate subtypes of a type can be specified as elements of the cross-product ('`*`') of type lists (dimensions), (74). In the example, `wine` has six subtypes: `(france, white), (france, rose), (france, red), (italy, rose), (italy, white), (italy, red)`. Types included in each list are also subtypes of `wine`. ProFIT defines multiple inheritance for types which originate from different dimensions (multi-dimensional inheritance, Erbach (1994a)).[20] In (74), `edelzwicker` inherits both from `france` and `white`, i.e., it is a subtype of type `(france, white)`.

(74)
```
wine > [france, italy] * [white, rose, red].
  france > [edelzwicker, pinot_noir].
  white > [edelzwicker, riesling].
```

ProFIT "...adopts the open-world semantics, and two types are considered consistent unless they are explicitly declared as disjoint",[21] (Erbach, 1994a, p. 4). See however the discussion in section 10.

ProFIT distinguishes intensional and extensional types. "Two terms which are of extensional type are identical if they have a most specific type (which has no subtype) and if all features are instantiated to ground terms", (Erbach, 1994b, p. 7). Extensional types have to be introduced as immediate subtypes of `top` and prefixed with the '`-`' sign, (75). Their subtypes are extensional, all other types are intensional. "Two intensional terms are identical only if they have been unified", (Erbach, 1994b, p. 7).

(75)
```
top > [wine, -list].
```

Types which have no subtypes do not have to be explicitly defined as such, i.e., a declaration such as: `subtype > []` is not necessary.

Only appropriateness conditions can be declared within the inheritance hierarchy. No type constraints are allowed, which implies that HPSG principles cannot be declared as constraints imposed on types. However, a substitute of constraints can be encoded by means of templates, see section 7.2.4.

7.2.2 Features

Features are introduced in type declarations by the keyword `intro`, as in (76).

(76)
```
sign > [phrase, word]
  intro [phon, synsem].
```

[20] Only orthogonal multiple inheritance seems to be allowed, i.e., if a type inherits from two supertypes, both of which have feature f specified with different appropriateness conditions, ProFIT behaves in an unexpected way.

[21] This statement concerns two types which origin from different dimensions; types from the same dimension are always disjoint.

All features introduced for a type, say `sign`, are inherited by all its subtypes. Thus both `phrase` and `word` have features `phon` and `synsem` declared. It is possible to impose type restrictions on the introduced features, (77). Feature values of the type `top` have to be omitted[22] in the declaration (`phon` value is presumed to be `top`). Subtypes can introduce features of their own in addition to those inherited from a supertype. In (77), `phon`, `synsem` and `dtrs` are the appropriate features for the `phrase` type.

(77)
```
sign > [phrase, word]
  intro [phon,
         synsem: synsem].
  phrase intro [dtrs: head_struc].
```

Features must not be reintroduced on the subtypes of types for which they were introduced. As a consequence, features values cannot be further specified on subtypes. The description in (78) is illegal because the `subj_dtr` feature is reintroduced by the `head_comps_struc` type, which is a subtype of `head_struc` (analogously for `head_subj_struc`).

(78)
```
phrase intro [dtrs: head_struc].
  head_struc> [head_comps_struc, head_subj_struc]
              intro [head_dtr: sign,
                     comp_dtr: sign_list,
                     subj_dtr: sign_list].

  head_comps_struc intro [subj_dtr: e_list].
  head_subj_struc  intro [comp_dtr: e_list].
```

The correct declaration for the `head_struc` type is given in (79) (assuming the appropriate declaration of the list hierarchy). Thus, unlike in HPSG, the *head_struc* type introduces only one feature. However, its subtypes have all features introduced.[23]

(79)
```
head_struc> [head_comps_struc, head_subj_struc]
                intro [head_dtr: sign].

  head_comps_struc intro [comp_dtr: sign_list,
                          subj_dtr: e_list].
  head_subj_struc  intro [subj_dtr: sign_list,
                          comp_dtr: e_list].
```

7.2.3 ProFIT Terms

ProFIT terms can be mixed with the standard Prolog terms, i.e., ProFIT terms can be Prolog terms and Prolog clauses can use ProFIT terms as arguments. Note, however, that Prolog lists and ProFIT's representation of lists are incompatible. They have different internal representations: subtypes of `list > [e_list, ne_list]` are represented as typed feature structures and they do not match the representation of Prolog lists. In order to avoid a complicated list hierarchy and translation of Prolog list operations into ProFIT, sortal restrictions on features whose values are lists can be omitted and Prolog lists can be used instead. However, in that case the appropriateness conditions have to be handled by the grammar writer.

[22](Erbach, 1994b, p. 4) writes that they can be omitted but from our implementation experience it follows that actually they must not be present.

[23]In contradistinction to (Erbach, 1994b, p. 4) FIC need not be satisfied.

Typed feature terms in ProFIT can be specified as types (`<` <*type*>), feature–value pairs (<*feature*>`!`<*value*>) or conjunction of terms (<*term*>`&`<*term*>). Disjunction of terms is added as syntactic sugar (<*term*>`or`<*term*>), since all disjuncts are processed as separate clauses. In addition, ProFIT terms include variables (variables are represented in the same way as in Prolog), template calls, section 7.2.4, and finite domain values, section 7.2.5. Additional syntactic expressions for feature search are provided.

ProFIT terms are translated into Prolog terms and then processed. However, it is possible to prevent translation from ProFIT to Prolog terms. Single (back-) quoting of a ProFIT term, i.e., prefixing it with a `‘` sign, enables treating terms which look like ProFIT terms as Prolog terms, i.e., prevent translation. Double (back-) quoting (`‘‘`) of a ProFIT term prevents only the main functor from translation whereas the arguments are translated into Prolog.

7.2.4 Templates

ProFIT provides a macro facility called templates. A template consists of a name followed by the operator `:=` and a template description, see (80). The template name can be parameterized and the template body can be an arbitrary ProFIT description. Templates cannot be defined recursively.

(80) <*name*> `:=` <*description*>.

Templates are used as an abbreviation device. They are particularly useful in specifying the lexicon, see (81). In this example, the usual DCG notation is used to encode the lexical entry, `john`. Since templates may use variables as arguments, all items which have similar properties can be easily declared.

(81)
```
noun(Case) := <word &
                 synsem!local!cat!head!(<noun &
                                          case! Case).
@noun(nom) --> [john].
```

A template is called by the name prefixed with the ‘`@`’ sign. Since templates are expanded at the compile time, some degree of partial execution of clauses can be achieved.

One can impose restrictions on types using templates. Although it is not possible to impose constraints on types directly in the type hierarchy, templates allow for a combination of types with grammar principles. Also, calls to the constraints can be placed in a grammar rule (defined in the example as a DCG rule), (82). Instead of encoding the Head Feature Principle (`hfp`) as a Prolog goal, one can do this directly in the rule body (by calling the `@h_phrase` template).

(82)
```
hfp := synsem!local!cat!head!X &
       dtrs!head_dtr!synsem!local!cat!head! X.

h_phrase := <phrase & @hfp.

(@h_phrase & synsem!local!cat!subcat! []) -->
  (<word & Subj),
  (@h_phrase & synsem!local!cat!subcat! [Subj]).
```

Note, however, that these are not real HPSG constraints, i.e., they are not inherited by subtypes, and they are valid only when the template is called (other occurrences of `phrase` will not satisfy `hfp`).

7.2.5 Finite Domains

A finite domain is a special ProFIT construct which restricts feature values to a finite set of atoms, cf. (83).

(83) <*domain_name*> `fin_dom` <*finite_set_of_atoms*>.

Within finite domains, all boolean combinations of atoms, i.e., negation (`~`), conjunction (`&`), disjunction (`or`), can be encoded. Elements of finite domains are followed by the '`@`' sign and the name of the finite domain. In certain cases this affix can be omitted. Below, we present an example of the finite domain `case`. The `not_nom_subj` template disallows case values to be nominative.

(84)
```
case fin_dom [nom, gen, dat, acc, inst, loc].

not_nom_subj := case! (~nom) @case.
```

7.3 Computational Aspects

7.3.1 Components

The ProFIT system comes without any processing components of its own. No parser, morphology or generation components are provided by the system. Lexical rules have to be implemented as well. However, ProFIT can make use of other existing components which can be integrated with the system using Prolog as the interface language. In the currently available version (ProFIT 1.54), the ALE parser is used for processing HPSG grammars. It is also possible to use standard DCG Prolog rules for parsing.

7.3.2 Implementation

System requirements The ProFIT system can be run with SICStus or Quintus Prolog. In case of SICStus, version 2.1#9 or higher is required. ProFIT is freely available by anonymous ftp from `ftp.coli.uni-sb.de` in the directory `pub/profit`.

Efficiency Since ProFIT is an extension of Prolog and no interface is needed, the system itself is very fast and efficient. "ProFIT is 5 to 10 times faster than a system which implements a unification algorithm on top of Prolog", (Erbach, 1994b, p.10).

Procedural model ProFIT uses Prolog built-in unification procedures. The system does not allow for control of the unification process. The standard Prolog strategy for executing goals is used.

It is possible to force partial execution of clauses by using templates instead of Prolog goals.

Debugging Standard Prolog procedures for debugging can be used. In addition, it is possible to check inheritance relations and feature appropriateness by switching on/off the `sort_checking` and `feature_search` directives. The debugging mechanism is not very handy.

Graphical user interface No GUI is provided. ProFIT terms are printed with the Prolog predicate `print/1`. Only those features that have been instantiated are printed out. Terms can be printed in the ProFIT format or the internal Prolog representation can be used. ProFIT terms can be printed out in the LaTeX format as attribute-value matrices.

Environment Only Prolog environment.

File organization Type declarations, finite domains and template declarations can be split into several files. In that case, the following `multifile` declaration is necessary in the file which is loaded first:

(85)
```
:- multifile '>'/2.
:- multifile fin_dom/2.
:- multifile ':='/2.
```

7.4 Development Stage

Support The system is stable but comes with no warranty. Bug reports can be sent to Gregor Erbach, `erbach@coli.uni-sb.de`.

Documentation

- Erbach (1994a) — description of multi-dimensional inheritance underlying ProFIT inheritance system;
- Erbach (1995) — manual distributed with the system;
- Erbach (1994b) — the system description.

Recommendation The system is very efficient. ProFIT terms are translated into Prolog representation and Prolog's unification procedure is used directly. However, since ProFIT is a relatively low-level system, and because of its weak type system, it is not recommended for testing HPSG grammars.

8 CL-ONE

8.1 Introduction

CL-ONE extends ProFIT's typed feature structure description language. The system comprises constraint solvers for set descriptions, linear precedence and guarded constraints. Set and linear precedence constraints are defined over constraint terms (cterm), which are internal CL-ONE data structures. Unification of sets differs from standard Prolog terms unification. The system has been designed in the project *The Reusability of Grammatical Resources*, at the University of Edinburgh and at Universität des Saarlandes, Saarbrücken.

8.2 Syntax and Data Structures

This section is based on the 1994 version of the system manual (Manandhar, 1994b). Since the system is currently under development, it is possible that the syntax may have evolved.

8.2.1 Types

CL-ONE provides no typed feature structures representation of its own. The system uses ProFIT's data type representation language.

8.2.2 Sets

Set constraints are an extension to the usual ProFIT syntax for ProFIT terms. This means that they can be stated directly in the term declaration. The set constraints can be imposed only on the (sub-) types of the (CL-ONE) built-in `set` type. The type is implicitly declared as a subtype of `top`. Logical foundations for the set constraints are presented in Manandhar (1994a).

CL-ONE allows the following set descriptions:

- the empty set: { },
- a set description: {<*T1*>,<*T2*>, ... <*Tn*>} — in this case, set members may unify and set cardinality can be less than *n*,
- fixed cardinality set descriptions: `fixed_card({`<*T1*>,...,<*Tn*>`})`,
- set membership relation: `exist(`<*X*>`)` — the current description contains element <*X*>, i.e., `Y & exist(X)` says that `X` belongs to `Y`,
- subset relation: `subset(`<*T*>`)`[24] — the current description is a subset of <*T*>, i.e., `S & subset(T)` defines `S` as a subset of `T`,
- set union: `union(`<*T1*>`,`<*T2*>`)` stands for the union of all members of sets <*T1*> and <*T2*>,
- set intersection: `intersection(`<*T1*>`,`<*T2*>`)`,
- disjoint sets: `disjoint(`<*T1*>`,`<*T2*>`)` verifies whether <*T1*> and <*T2*> are disjoint sets and assigns the corresponding truth value,
- disjoint set union: `dis_union(`<*T1*>`,`<*T2*>`)`.

With this apparatus in hand, set-valued subcategorization requirements of a ditransitive verb can be stated as follows:

(86)
```
ditr_verb :=
  <word &
  synsem!local!cat!(subj! fixed_card({@np}) &
                    comps! fixed_card({@np, @np}) &
                    head! <verb).
```

[24] The manual (Manandhar, 1994b, sec. 1) says "...current description includes all members of T" but form our implementation experience it follows this is the other way round: the current description *is included* in T, i.e., is its subset (this notation mismatch was reported to the author).

(87) shows a set-valued analogue of the subcategorization principle formulated using `dis_union`. ProFIT does not allow for relational constraints, thus the predicate `synsems_to_phrases` has to be defined via Prolog definite clauses:

(87)
```
subcat_principle(<phrase &
  synsem!local!cat!subcat X &
  head_dtr!synsem!local!cat!subcat! dis_union(X,Y) &
  comp_dtr! CompY) :-
synsems_to_phrases(Y, CompY).
```

8.2.3 Linear Precedence Constraints

LP constraints are employed to represent the surface order of constituents irrespectively of the system's procedural model. The surface form of a phrase is represented as an unordered sequence of words, i.e., a set, and LP rules provide acceptable orderings.

In CL-ONE, the syntax for LP constraints extends the ProFIT term syntax, as in the case of sets. The linear precedence rules are formulated as relations between sets representing individual words or phrases. The details of logical foundations for this implementation are presented in Manandhar (1995).

LP constraints are defined over the (CL-ONE) pre-defined `lp_set` type which is a subtype of `top`. CL-ONE allows for the formulation of constraint terms precedence and immediate precedence of terms. `precedes(<`*CTerm*`>)` defines a constraint term which precedes <*CTerm*>, i.e., these two constraint terms, `precedes(<`*CTerm*`>)` and <*CTerm*>, are related by the `precedes` relation. `imm_precedes(<`*CTerm*`>)` defines a constraint term which immediately precedes <*CTerm*>. Precedence constraints are also formulated over sets of constraint terms. `dom_precedes(<`*Set*`>)` defines a set of constraint terms in which all elements precede the elements of <*Set*>, (domain precedence). The predicate `fst_daughter(<`*Set*`>)` defines a constraint term such that it is the leftmost element of domain <*Set*>. Constraint terms cannot occur as feature values directly in the structure since their unification differs from the standard Prolog unification. LP constraints in ProFIT structures must be declared over singleton sets of cterms, not over cterms directly, cf. (88).

(88)
```
lp! (lex_head_lp! {LH_CTerm} &
     det_lp! {precedes(LH_CTerm)}).
```

The LP constraint in (88) defines a relation between the determiner and the lexical head of a phrase. The value of `det_lp` is a set such that all its elements precede all elements of the set `{LH_CTerm}`, i.e., `LH_CTerm`. This means that the determiner is forced to precede the lexical head. Note that since the `precedes` relation is used, instead of `imm_precedes`, the correct placement of modifiers can be obtained.

8.2.4 Guarded Constraints

Guarded constraints are applied to delay a constraint until enough information is available for its deterministic execution.[25] There are two types of guarded constraints in CL-ONE: case statements (with an extension for existentially quantified variables) and the empty set check.

[25]However, they may cause problems in case of two-directional natural language processing.

Case Statements Case statements formulate (a list of) conditional expressions which relate a condition with a certain action to be executed, cf. (89):

(89) <*feature_name*>`!case([`<*condition*>`#>`<*action*>`]) else` <*action1*>.

Several conditional expressions <*condition*> `#>` <*action*> can be used in a single case statement. In that case they must be separated by a comma on the `case` list. Any ProFIT description can be used as <*condition*> apart from set descriptions and case statements. There is no variable binding between conditions and the rest of the structure. Thus, variables used in the condition part will not be recognized in the rest of the structure. In the action part further case statements can be embedded. <*action*> is a Prolog predicate and will be called when the corresponding condition is entailed by the context. If the condition is disentailed by the context, <*action1*> will be executed. The `else` part need not be overtly specified, in this case <*action1*> is understood as `true`. If the condition is neither entailed nor disentailed by the context, the constraint will be delayed until more information is instantiated. In (90), if the value of `f1` equals `plus` then the procedure succeeds, whereas in case of the `minus` value it will fail. If the `f1` value is not instantiated at the moment, the constraint execution is delayed.

(90)
```
f1! case([(<plus) #> true,
          (<minus) #> fail]).
```

If we assume the value of `f1` to be `boolean` (with the only subtypes `plus` and `minus`), (90) can be written as follows:

(91)
```
f1! case([(<minus) #> fail]).
```

This statement guarantees that the procedure will succeed if and only if `f1` is specified as `<plus` (since the `else` part is omitted, the remaining action is `true`).

Existentially quantified variables extend the syntax of case statements in that they allow Prolog variables used in the <*condition*> part to be recognized in other parts of the structure, i.e., variable binding between conditions and rest of the structure is now feasible. We will not quote here the precise syntax of this constraint for two reasons: the system is being modified presently and the syntax of this constraint (as not very convenient) probably will be changed. Besides, the integration of this constraint with ProFIT has not been tested. However, the reader interested in the details is referred to (Manandhar, 1994b, sec. 3.2).

Empty set check The empty set check is an additional guarded constraint which allows to execute/delay the constraint in case of set emptiness.

(92) `if_empty_set(`<*Then*>`,`<*Else*>`)`

where both <*Then*> and <*Else*> represent Prolog predicates. (93) specifies the `comps` value as a subset (of `X`) and the constraint is executed if `comps` is the empty set.

(93)
```
comps! subset(X) & if_empty_set(true, fail).
```

8.3 Computational Aspects

8.3.1 Components

The CL-ONE system is distributed with two parsers: a left-corner parser and a chart parser, as well as a (head-driven) generator. A facility for handling lexical rules is provided. There is no morphology component. Constraint solvers for set constraints, LP rules and guarded constraints have been implemented as separate modules. Thus it is possible to integrate them with other systems. The constraint solvers have been also successfully implemented as an extension of ALEP.

8.3.2 Implementation

System requirements CL-ONE runs under SICStus Prolog and the ProFIT system. For the earlier version of the system SICStus Prolog 2.1#9 is required, although the system is currently under development and the next release will require SICStus Prolog 3.0. The system is available via anonymous ftp from: `ftp.coli.uni-sb.de`, directory `pub/coli/proj/rgr/cl-one`.

Efficiency The high efficiency of ProFIT does not entirely carry over to CL-ONE. This is due to the set unification procedure which cannot be implemented very suitably in the earlier version of SICStus Prolog. The used algorithm is extremely memory consuming, which can even cause the interruption of program execution. The next release of the system based on SICStus 3.0 should be free from these processing problems.

Procedural model The system consists of constraint solvers for sets, linear precedence and guarded (conditional) constraints. There are two types of guarded constraints which delay the execution in case of insufficient information for the deterministic behaviour of the program. A separate unification algorithm for unification of sets exists.

Debugging The debugging environment is not very friendly, standard Prolog commands can be used. The debugger tests the satisfiability of constraints as well as the appropriateness of ProFIT terms.

GUI CL-ONE terms are printed in the ProFIT format.

Environment CL-ONE has the same environment possibilities as ProFIT.

8.4 Development Stage

Support The system is currently reimplemented into a new platform thus only previous versions are accessible. On-line support form the author(s) is available, e-mail: `suresh@minster.cs.york.ac.uk`.

Documentation

- Manandhar (1994b) — sketchy user's guide;
- Manandhar (1994a) — description of set constraints;
- Manandhar (1995) — description of LP constraints;
- Erbach *et al.* (1995) — general description of the system and the project the system has been developed in.

The system is under construction; very sketchy manual and system description.

Recommendation Because of its novel features such as set and LP constraints, the systems is worth consideration as a system for implementation of constraint-based grammars. However, it is hard to test the system at the current stage of development.

9 ConTroll

The ensuing description is based mainly on Götz (1995), Götz (1996) and Meurers (1996).

9.1 Introduction

ConTroll is being developed at the Seminar für Sprachwissenschaft, University of Tübingen, as a system specifically designed for implementing HPSG grammars. ConTroll is a constraint-based formalism with relational extension, it contains no phrase structure backbone. "One can view ConTroll as CUF augmented with arbitrary implicational constraints on objects" (Götz, 1995).[26] Just as CUF, ConTroll is a constraint solver and does not include additional components (parser, generator, etc.).

This section describes version 0 the system. Since ConTroll is, at this stage, unstable and being rapidly developed, the end product is likely to greatly differ from the version dealt with here.

9.2 Syntax and Data Structures

The ConTroll grammar consists of three main parts: a type system (i.e., type hierarchy and appropriateness conditions), implicational constraints and definite relations. There is also some syntactic saccharine for lexical entries.

9.2.1 Type System

The types in ConTroll follow King (1994), i.e., they constitute a semi-lattice with joins and the greatest type `bot`. An important feature of ConTroll is the closed-world philosophy of type hierarchy, i.e., every linguistic object is of exactly one most specific type.[27] The closed-world approach implies that all the minimal (maximally specific) types are disjoint. There is no division into intensional and extensional types in ConTroll (all types are intensional in the sense of Carpenter and Penn (1995)).

The type system (signature) is written as an indented tree, e.g.:

(94)
```
type_hierarchy

bot
  sign cat:cat agr:number phon:list
    phrase dtr1:sign dtr2:sign
    word
  cat
    np
    vp
```

[26] *Implicational constraints* are ConTroll's counterpart of ALE's or TFS's *constraints on types*, cf. section 9.2.3 below.

[27] Note that it differs in this respect from ALE and CUF, but adheres to the philosophy of HPSG.

```
      sv
      s
    number
      singular
      plural
  .
```

Types which multiply inherit should be syntactically marked as such by adding '`&`' in front of any subsequent occurrence of these types, e.g., the case hierarchy for Polish (cf. Przepiórkowski (1996)) can be represented by:

(95)
```
type_hierarchy

case
    str
      nom
      acc
      sgen
    gen
      &sgen
      lgen
    lex
      &lgen
      dat
      ins
      loc
.
```

Part of the type system is predefined. Notably, there is type `string` such that all undeclared types are assumed to be of this type.[28] Moreover, type `list` is defined, with two subtypes: `e_list` and `ne_list`. There are two features appropriate for `ne_list`, namely `HD` and `TL`. As syntactic sugar, the Prolog list notation can be used for lists (cf. examples below).

9.2.2 Feature Declarations

Feature specifications on types are inherited by their subtypes, which can, in turn, further specify the values of these features. Unlike in ALE, the feature introduction condition (FIC) does not have to hold (i.e., polyfeatures are allowed, cf. King and Götz (1993)) and, unlike in CUF, the linguistic objects are assumed to be totally well-typed (only and all features declared on a type have to be present).

In the current version of ConTroll, features on a type have to be specified in the same line in which the type has been introduced, cf. (94) above. Unlike in TFS or ALE, only appropriateness conditions can be specified on types in the type system: other constraints (including grammar principles) have to be specified via implicational constraints (see below). However, just as in those two systems, every object of a given type has to satisfy all the constraints defined on this type (both in the type system and as implicational constraints).

[28] Note analogy to CUF's `afs_symbol`.

9.2.3 Constraints

Constraints are expressed over feature terms, which are built out of types, attributes, variables and relations in an ordinary way. The allowed connectives are conjunction '`,`', disjunction '`;`' and negation '`~`', though neither variables, nor relational constraints are allowed in the scope of negation. ConTroll provides Prolog-like notation for lists (cf. section (9.2.1)), it does not, however, provide any mechanism for handling sets.

Implicational Constraints These are the every-day HPSG constraints, defined on types, cf. (96). The meaning of the '`==>`' operator is that any object satisfying the description which is its left-hand side argument must also satisfy its right-hand side argument.

(96)
```
phrase ==> ( ( cat: s,
               dtr1: (np_sign(Num),
                      phon: Phon1),
               dtr2: (cat: vp,
                      agr: Num,
                      phon: Phon2))
           ; ( cat: vp,
               agr: Num,
               dtr1: (cat: sv,
                      agr: Num,
                      phon: Phon1),
               dtr2: (cat: s,
                      phon: Phon2))

           ),
           phon: append(Phon1, Phon2).

word ==> ( (np_sign(singular),
            phon: [(john; mary)])
         ; (np_sign(plural),
            phon: [(cats; dogs)])
         ; (cat: vp,
            phon: [(runs; jumps)],
            agr: singular)
         ; (cat: vp,
            phon: [(run; jump)],
            agr: plural)
         ; (cat: sv,
            phon: [(knows; thinks)],
            agr: singular)).
```

In this approach, lexical entries are nothing more than constraints on the type *word*. The example above illustrates also that the consequents of implicational constraints can involve disjunctions, expressed with '`;`', lists and relational constraints, e.g., `np_sign` and `append` (see below). Structure sharing is indicated with variables, e.g., `Num`, `Phon1`. An interesting feature of ConTroll is that it allows complex antecedents in implicational constraints, e.g.:

(97)
```
dtrs:head_struc ==> head_feature_principle.
```

Relational Constraints ConTroll's relational constraints are based on CUF's sorts, i.e. they have a functional format:

(98)
```
append(list,list) **> list.
append([],List) := List.
append([H|T],L) := [H|append(T,L)].

np_sign(number) **> sign.
np_sign(Num) := sign, cat:np, agr: Num.
```

Note that with each relation comes its specification (obligatorily, unlike in CUF), represented with the operator '`**>`'.

9.2.4 Lexicon

Syntactic Sugar Even though lexical entries are formally just implicational constraints on the appropriate lexical type (e.g., *word*), ConTroll provides a special operator '`--->`' for lexical entries, e.g.: `john ---> cat:np.`. In order to use this format, the grammar writer has to define two predicates, i.e., `lexicon_type/1` and `lexicon_style/3`. The former specifies the lexical type in the grammar, e.g., `lexicon_type(word).`. The latter determines the exact meaning of '`--->`', e.g., `lexicon_style(LHS,RHS,(RHS,phon:[LHS])).`. The first two arguments of `lexicon_style/3` (`LHS` and `RHS` in this example) should (normally) be variables representing the left- and right-hand sides of the '`--->`' statements, while the third argument (here `(RHS,phon:[LHS])`) specifies how such statements are translated into standard ConTroll. For example, if the `lexicon_type/1` and `lexicon_style/3` predicates are defined as above, the statement in (99a) will be translated into (99b).

(99) a. `john ---> cat:np.`

b. `word ==> cat:np, phon:[john].`

Lexical Rules Lexical rules are a very recent feature of ConTroll, still in the testing phase. They are based on the so-called Description-level Lexical Rules approach of Meurers (1995), as opposed to the more orthodox Meta-level Lexical Rules approach of Calcagno and Pollard (1995) and Calcagno (1995).

In order for the lexical rules to work, the '`--->`' format of the base lexicon has to be used and features `lr_class:string`[29] and `lr_status:list` have to be declared on the lexical type (*word*) in the signature. Unlike in ALE, the grammar writer has to specify only the information which changes across the entries. For example, in the `naive_celr` lexical rule (100) only the value of SYNSEM|LOC|CAT|COMPS changes:

(100)
```
naive_celr lex_rule
   synsem:loc:cat:comps:tl:X
**>
   synsem:loc:cat:comps:X
```

The reader is referred to Meurers (1995) for theoretical basis of the implementation and to Meurers and Minnen (1995) and Meurers and Minnen (1996) for a discussion of the technical issues.

[29] `string` is a special ConTroll type.

9.3 Computational Aspects

Like CUF and ALE, ConTroll provides an incremental compiler and an interpreter.

9.3.1 The Compiler

During compilation, the type system and constraints are compiled into definite clauses, and constraint inheritance and code optimization are performed (Götz and Meurers, 1995). All deterministic constraints are unfolded at compile time, although this behaviour can be prevented by the user.

ConTroll also uses off-line lazy evaluation techniques in order to compute (off-line) which nodes of the feature structure have to be checked (on-line) in order to guarantee that there is a solution (Götz and Meurers, 1996). For example, since each type is expected to be satisfiable (see below), nodes containing no information other than their type will not be unfolded at run-time.

Employing off-line lazy evaluation techniques results in an efficiency gain of up to 30% and better termination properties of queries, but the property of *persistence* (*subsumption monotonicity*) is lost: if ϕ is a solution and there are terms more specific than ϕ, then only some of these terms must be solutions, not necessarily all of them as the property of persistence would require.

Lazy evaluation also imposes an additional requirement on the grammars, i.e., *type consistency*. A grammar is type consistent iff it has a model where every type has a non-empty denotation. Hence, for type-consistent grammars, if there are no features defined on a node then models satisfying that node exist (this is guaranteed by type consistency which is assumed rather than checked by the compiler) and the compiler does not search any further. Unfortunately, it is in general undecidable if a theory is type consistent, so the responsibility of ensuring this rests on the grammar writer.

9.3.2 The Interpreter

ConTroll employs a corouting interpreter, i.e., the order of unfolding goals is based on the control statements (cf. section 9.3.3), and not on the order of clauses, etc.

Moreover, it employs a deterministic closure strategy, i.e., after unfolding the first undelayed goal, all resulting deterministic goals are unfolded. (If there are no more undelayed goals, computation terminates and the frozen goals are displayed on the screen.) This strategy, in general, increases efficiency, but termination properties may change (e.g., in some cases deterministic expansion may not terminate, i.e., when expanding a deterministic goal results in another deterministic goal, etc.). Run-time deterministic expansion of goals can be prevented with a `lazy/1` statement.

9.3.3 Control Statements

These statements let the grammar writer specify the order in which goals should be considered.

Compiler Directives There are three types of compiler directives. The first of them, '`<</2`', is used for specifying in which order subtypes of a given type should be tried, e.g., `word << phrase.` tells the compiler to try to analyze structures of type *sign* as *word*s, before analyzing them as *phrase*s.

The directive '`<<</2`' specifies the ordering of nodes, e.g., `dtr2 <<< dtr1.` says that the nodes under `dtr2` should be expanded before those under `dtr1`.

Finally, `goal_order/1` specifies in which order (everything else being equal) relational goals should be tried, e.g., `goal_order([np_sign, append]).`.

Corouting Statements These statements are, in general, more powerful than the compiler directives. ConTroll's `delay/2` corresponds to CUF's `wait/1`, e.g., `delay(append, (arg0:list ; arg1:list)).` The first argument of `delay/2` is the name of the relation, the second is a feature term[30] that has to be properly satisfied in order for the goal to be considered; here, either the result of `append/2` (`arg0`), or the first argument (`arg1`) has to be known as more specific than *list*.

`delay_deterministic/1` is another corouting statement. The argument must be a type or a relation, which is subsequently delayed until it can be deterministically executed, i.e., until a subtype of the type or a clause of the relation can be deterministically chosen.

Finally, the goals specified as `lazy/1` (analogous to CUF's `lazy_goal`) are not considered for deterministic evaluation at run-time (cf. section 9.3.2), e.g.: `lazy(phrase).`.

9.3.4 Interaction with the User

File Organization ConTroll has been created with the aim of implementing large HPSG grammars in mind; hence, it is equipped with mechanisms for modular organization of files. The grammars are compiled with the directive `compile_user_grammar/0`. The first thing it does is read the user-defined `config.pl` file in the local directory, if present. This file should contain ConTroll commands used for setting and changing file search paths associated with the various modules (signature, theory, lexicon) of the grammar, as well as commands defining dependencies between these files. If `config.pl` is not found, then the system assumes the default configuration, i.e., local files `signature` and `theory` containing, respectively, the type system and the constraints.[31]

Compiling Grammars Apart from the `compile_user_grammar/0` command, there are also some commands for incremental compilation, e.g., of the signature or the lexicon alone, though many of them are not documented.

Basic Commands Finding feature structures which satisfy a given description can be done with the `del_q/1` command, e.g.: `del_q(phon:[janek,biegnie]).`. Internal AVM representation of feature terms can be viewed with `show_mgs/1`, and the constraints associated with a given type or relation can be inspected with `show_constraint/1`.

Debugging ConTroll comes equipped with a sophisticated debugger, which allows the user to creep, choosing various goals to unfold, with or without deterministic expansion, view and/or retry previous goals, inspect clauses matching the current (or any other) goal, etc.

Graphical User Interface ConTroll can be coupled with XTroll, which provides an elegant windows interface. All the commands described above (and more) can be accessed with menus, partial results can be stored and manipulated in separate windows, feature values can be hidden or expanded, etc. At the moment XTroll is maintained only on Solaris machines.

[30] This feature term cannot contain variables or relations.

[31] The signature and the theory have to be kept separate since the latter consists of Prolog-readable terms, while the former, due to its peculiar indentation format, does not.

9.4 Development Stage

ConTroll has been developed in Quintus Prolog and runs under Unix. The current version number of ConTroll is 0, although the system is extensively used in an effort to implement a large-scale German grammar (subprojects B4 and B8 of Sonderforschungsbereich 340 *Linguistic Foundations for Computational Linguistics*). Unfortunately, ConTroll is currently not publicly available.

Since ConTroll is a constraint based system, it is, various optimization techniques employed notwithstanding, not very efficient. However, because of its uncompromising truthfulness to the HPSG semantics, it should be strongly recommended as a workbench for testing HPSG theories.

At the moment no documentation is available.

10 Comparisons

This final section is devoted to a comparison of the systems described above. We were trying to get the necessary information both from existing documentation and through some hands-on experience.[32]

10.1 Type System

In this section, we compare the systems considered in this report with respect to the type organization they impose.[33] Note that, since CL-ONE inherits its type system characteristics from ProFIT, we do not include it explicitly in the considerations below.

10.1.1 Type Algebra

The most restrictive type hierarchy is that of ALEP: multiple inheritance is not supported (Schütz, 1994b, p. 55), i.e., the type hierarchy constitutes a finite right-linear order (FRLO, a forest-like structure).

Less restrictive, and essentially the same, are the type hierarchies of ALE and ConTroll: they are finite semi-lattices. More specifically, in ALE, $\sigma \sqsubseteq \tau$ means that σ is a *supersort* of τ (Carpenter, 1992, p. 11) and, hence, the types of ALE are organized in a finite *meet* semi-lattice (FMSL), while in ConTroll $\sigma \preceq \tau$ means that σ is a *subsort* of τ (Götz, 1996, p. 1), so the types constitute a finite *join* semi-lattice (FJSL).

In contrast, the type hierarchies of TFS and ProFIT are weaker than a semi-lattice, as types in these systems are organized in a (finite) partial order (FPO, cf. Emele (1993) and Erbach (1994a) respectively). It seems that also in PAGE types constitute in general FPO.

Finally, the type algebra of CUF is, in a sense, weakest (most general) of all considered here. Type axioms involving conjunction, disjunction and negation can be used to define any partial order on types.

[32] Due to the fact that we did not manage to install and test PAGE, parts of this section may be lacking relevant information regarding this system. We will do our best to do justice to PAGE in subsequent versions of the report.

[33] In this section, by *type hierarchy* we mean a kind of order the system imposes on types, and by *type system* we mean both the type hierarchy and appropriateness conditions.

10.1.2 Intensional and Extensional Types

The two systems which explicitly mention the intensional vs. extensional type distinction are ALE and ProFIT, though the exact definitions of these notions differ a little in both systems. In ALE, "two feature structures of the same extensional type are token-identical if and only if, for every feature appropriate to that type, their respective values on that feature are token identical" (Carpenter and Penn, 1994, p. 17).[34] On the other hand, in ProFIT "two terms which are of an extensional sort are only identical if... all features are instantiated to ground terms" (Erbach, 1994b, sec. 2.1).

Among the other systems, ALEP, ConTroll and TFS treat all types as intensional, i.e., two objects of a given type are assumed to be identical only if they are explicitly equated.[35] Also PAGE seems not to distinguish between intensional and extensional types. On the other hand, in CUF ordinary types are understood intensionally, but constants, which also belong to the type hierarchy, have only single objects in their denotations, i.e., they are extensional.

As far as we can see, intensional vs. extensional type distinction does not play any crucial role in HPSG, although it can be used, e.g., to specify uniqueness of the empty list (*elist*) type.[36]

10.1.3 Open vs. Closed World

A type hierarchy adheres to the closed world philosophy if any object of the domain is assumed to be in the denotation of exactly one *maximally specific type* (called also *species* (King, 1994) or *variety*). In other words, two closed-world assumptions have to be satisfied (Gerdemann, 1995): *partition condition* (if an object is of type t then it is of at least one species subsumed by t) and *disjoint species condition* (the object of type t is of no more than one species subsumed by t).[37]

Two of the systems discussed here employ the closed world interpretation of type hierarchy: TFS and ConTroll. On the other hand, ALE violates the partition condition, while ProFIT violates the disjoint species condition, so these systems have open-world semantics (but see a discussion of ProFIT below). CUF, thanks to its general type axioms, can be used to specify a type hierarchy interpreted either in open-world manner, or in closed-world manner.[38]

PAGE, in contrast, is heterogeneous in this respect: avm types adhere to the open-world reasoning, while sort types are subject to closed-world interpretation (Krieger and Schäfer, 1994a. 9–11). It should be, however, noted, that PAGE, similarly to CUF, has a type system powerful enough to simulate closed-world approach also on avm types (Krieger and Schäfer, 1994a, p. 11).

As argued in Gerdemann (1995), the distinction between open and closed world approaches to type hierarchies is to a large extent a matter of user interface. This means that, *in principle*, both approaches are interdefinable. More specifically, any closed world type hierarchy can be translated into an open world type hierarchy, but in order to make the translation in the other

[34] Cf. also fn. 10 in section 4.2.1.

[35] In TFS, however, unique identifiers can be assigned to objects to force extensional behaviour (Emele, 1994, p. 3).

[36] This observation is due to Tilman Höhle (p.c.).

[37] The notion of open and closed world interpretation is a source of some confusion. Some authors call any system which violates disjoint species condition as "open world system." Others (e.g., Gerald Penn, p.c.) argue that "closed world" simply means that types which are not declared to have subtypes are regarded as maximal. (In this sense ALE can be claimed to have closed world semantics; see, e.g., Erbach (1994a).) This clearly shows the need to work out generally acceptable definitions of these notions.

[38] Actually, CUF *by default* violates the disjoint species condition, i.e., if types are not explicitly specified as disjoint, they are interpreted as consistent. It also *by default* violates the partition condition: if subtypes of a type are not explicitly defined to partition the denotation of this type, then there might be objects denoted by this type and not belonging to any of the subtypes at the same time. This, in a way, validates the claim in Gerdemann (1995) that CUF is an open-world system.

direction possible, the expressiveness of the type system has to be powerful enough.[39] According to Gerdemann (1995), such expressiveness is provided by CUF, but not by ALE.[40]

The situation with ProFIT is actually more complicated because, even though its author calls it an open world system in Erbach (1994a), it seems to have an essentially closed world semantics with some additional notation provided for expressing open-world-like behaviour (dimensions). In this sense, ProFIT actually adduces some evidence to the claim in Gerdemann (1995) that open vs. closed world interpretation of a type hierarchy is to some extent a matter of user interface.[41]

10.1.4 Disjointness

In the previous subsection, we noted that some systems having open world semantics are expressive enough to define a closed world type inheritance hierarchy in them. One of the most important features of this expressiveness is whether it is possible to state in the system that two types have disjoint denotations. Of course, in any "closed world system," any two types which are not explicitly defined to have a common subtype, are disjoint. Also in ALE and ProFIT, any two types (which belong to the same dimension, in the case of ProFIT) are disjoint. On the other hand, in CUF and in PAGE types (avm types, in case of PAGE) are assumed to be consistent unless specified otherwise (cf. (Dörre and Eisele, 1991, p. 11) and (Krieger and Schäfer, 1994a, p. 11), respectively), but an axiom can be posited to the extent that types be disjoint. In short, all the systems we consider here either satisfy, or can be forced to satisfy the disjoint species condition.

10.1.5 Partition Condition

Another important expressiveness criterion related to open world semantics is whether it is possible to state in the type hierarchy that any object of a given type belongs to the denotation of some of its subtypes. This does not seem to be the case in ALE (cf., however, fn. 40, section 10.1.3), while ProFIT, again, behaves like a closed world system, i.e., partition condition holds. In CUF, on the other hand, although simply specifying, say, types *f1* and *f2* as subtypes of *f* (e.g., with the statement `f1 | f2 < f.`) does not guarantee that any object of type *f* belongs to the denotation of *f1* or *f2*, this can be enforced by the grammar writer, e.g., via statement `f = f1 | f2.`. Similarly, PAGE's default interpretation of any two avm types as, in principle, consistent, can be overridden with the help of the exclusive or operator, `^`.

10.1.6 Inheritance, Multiple Inheritance

All of the systems we consider here support the notation of inheritance, i.e., the attribute-value specifications of a type hold for its subtypes. Also, if types are allowed to have common subtypes (all systems apart from ALEP), multiple inheritance is supported.

10.1.7 Further Specifications

One of the main differences between various inheritance systems is whether they allow further constraining the value of a feature on a type if it has already been specified on its supertype. For

[39] As shown by Carpenter and King (1995), there is complexity price to be paid for the closed world expressiveness.

[40] Note that we consider only *type hierarchies* here. Of course, as we were reminded by Thilo Götz (p.c.), ALE can be made to behave like a closed-world system with the help of constraints attached to types (e.g., `sign sub [word, phrase] cons (word;phrase).`)

[41] We hope that this discussion explains an apparent contradiction resulting from classifying ProFIT as in a way violating disjoint species condition (when types belong to different dimensions) on one hand and as having disjoint types "by default" (when these types belong to the same dimension) on the other hand.

example, one might imagine the following fragment of a type system:

(101)
$$\begin{tikzcd} & \begin{bmatrix} \textit{sign} \\ \text{LEX bool} \end{bmatrix} \arrow[dl, dash] \arrow[dr, dash] & \\ \begin{bmatrix} \textit{word} \\ \text{LEX } + \end{bmatrix} & & \begin{bmatrix} \textit{phrase} \\ \text{LEX } - \end{bmatrix} \end{tikzcd}$$

Such a type system can be represented in all systems, apart from ProFIT. However, in ALEP, only a very restricted version of this functionality is present, namely, if a type specifies the value of its feature as boolean (i.e., denumerable), then a subtype of this type can further specify this value (as one of the listed atoms).

10.1.8 Generality

By generality of a type system we mean, after (Manandhar, 1993, p. 46), the possibility of defining recursive constraints on types, consisting of at least any combination of type symbols, feature selections and variables. Neither CUF nor ProFIT are general since only appropriateness conditions can be imposed on types in these systems. On the other hand, ALE, ALEP, ConTroll, PAGE and TFS are general.[42]

10.1.9 Feature Introduction Condition

The Feature Introduction Condition (FIC), a requirement that for each feature there be a most general type on which this feature is defined, has been introduced by (Carpenter, 1992, p. 86). Its usefulness has been, however, questioned by King and Götz (1993), who also modify Carpenter's specification of type inference to make it suited for formalisms not satisfying FIC.

Of the systems considered here, only ALE (Carpenter and Penn, 1994, p. 15) enforces FIC,[43] although some systems may give warning messages if FIC is violated (cf. e.g. Krieger and Schäfer, 1994b, p. 37).

10.1.10 Well-typedness

Notions such as well-typedness (WT) and total well-typedness (TWT) introduced in (Carpenter, 1992, pp. 88–95) describe to what extent feature structures of a certain type adhere to appropriateness specifications of the type system. If all features present on a feature structure of type t have been defined in the type system, and the actual values are of the type specified, the feature structure is well-typed. If, additionally, all the features declared in the type system are present, the feature structure is totally well-typed.

Of the systems considered, only CUF and PAGE have a 'loose' approach to type declarations in that they allow feature structures with attributes which have not been defined in the hierarchy (CUF issues a warning then, PAGE can be forced to do so as well). In PAGE this behaviour can be controlled by setting appropriate variables (Krieger and Schäfer, 1994b, p. 37), so the system can be made to admit only well-typed feature structures.[44] All the other systems allow only totally well-typed feature structures.

[42]Manandhar (1993) considers ALE to have a restricted type system, but this judgement is based on version β of ALE.

[43](Erbach, 1994b, sec. 2.2) mentions that ProFIT imposes FIC, but both the manual Erbach (1995) and our experience with the system contradict this information.

[44]It is not clear to us if they have to be *totally* well typed, though.

10.2 Syntax of Feature Terms

10.2.1 Conjunction

All the systems allow conjunction in descriptions.

10.2.2 Disjunction

The only system which does not have a special syntax for disjunction is TFS. This, however, does not limit the expressive power of the system in any way: most other systems simply compile disjunction out into clauses anyway, cf. e.g. (Simpkins, 1994a, p. 22) and (Erbach, 1994b, p. 7). An interesting feature of ALEP and PAGE is that they support the concept of named disjunctions (cf. (Simpkins, 1994a, p. 23) and (Krieger and Schäfer, 1994b, sec. 4.6.10), respectively).

10.2.3 Negation

The systems dealt with in this report differ widely when it comes to allowing negation in descriptions. Two of them, ALE and TFS do not allow negation, although some of its effect can be achieved by combining definite descriptions and path inequality, e.g., TFS for saying that the value of case is non-nominative could be:[45]

(102)
```
agr [CASE #X] :-
  #X =/= nom.
```

On the other hand, ProFIT and ALEP allow a very restricted use of negation, whose arguments can be only types defined by enumeration, i.e., types from finite domains in ProFIT or boolean types in ALEP. CL-ONE, again, behaves like ProFIT in this respect, although, it seems, negation-like behaviour can be simulated via guarded constraints.

Only CUF, ConTroll and PAGE provide more general syntax for descriptions with negation, though neither in CUF, nor in ConTroll can relations be in the scope of negation.[46] Moreover, in the current version 0 of ConTroll, variables are not allowed in the scope of negation, either.

10.2.4 Variables

All systems use variables in the standard way to define structure sharing. The variables denote single objects (i.e., they are first order variables) and are local to clauses or constraints.

10.2.5 Path Inequality

The ability to express path inequality is crucial, e.g., for expressing Linear Precedence statements with structure sharing across descriptions, cf. Kasper *et al.* (1995); Kathol (1995). For example, the LP rule (103) means that any domain elements X and Y such that X < Y in some domain have to satisfy the constraint in (104):

(103) $[\text{SYNSEM } \boxed{1}] \prec [\text{SYNSEM}|\text{SUBJ } \langle\boxed{1}\rangle]$

[45]Actually, the authors of ALE argue in Carpenter and Penn (1993) that general negation is not necessary for linguistic purposes, and that inequations and a system of types suffice.

[46]We are not sure whether functional constraints can be in the scope of negation in PAGE.

(104) $[\text{X}|\text{SYNSEM}|\text{SUBJ}|\text{FIRST} \neq \text{Y}|\text{SYNSEM}] \vee$

$[\text{X}|\text{SYNSEM}|\text{SUBJ}\ \text{elist}] \vee$

$[\text{X}|\text{SYNSEM}|\text{SUBJ} \begin{bmatrix} \textit{nelist} \\ \text{REST nelist} \end{bmatrix}]$

Of the systems considered here, only ALE, PAGE and TFS allow expressing path inequalities in a straightforward manner (cf. (Carpenter and Penn, 1994, p. 19), (Krieger and Schäfer, 1994b, p.18) and (Emele, 1993, p. 2), respectively). In systems such as ProFIT and CL-ONE path inequality can be expressed with the use of (Prolog) definite clauses (predicate '=/='), while in CUF it can be expressed using negation (Dörre and Dorna, 1993, p. 8, ex. 2). However, this option is at the moment not available in ConTroll because variables are not allowed in the scope of negation in that system, neither is it available in ALEP, which does not allow general negation or Prolog calls.

10.2.6 Cyclicity

All the underlying logic formalisms for HPSG starting with Pollard (1989) and King (1989) allow cyclic structures as the one in (105), so it seems desirable that any computational system for implementing HPSG should be able to deal with such structures.[47]

(105) $\boxed{1}\begin{bmatrix} \textit{type} \\ \text{ATTR}\ \boxed{1} \end{bmatrix}$

The only systems that do not allow cyclic structures are ALEP (Genabith *et al.*, 1994, p. 98) and CUF.[48] Most other systems explicitly mention this possibility, e.g., (Carpenter and Penn, 1994, p. 9), (Erbach, 1994a, p. 8) or (Emele and Zajac, 1990a).

10.2.7 Definite Relations and Macros

For the purpose of this subsection, we take recursiveness as the main difference between definite clauses and macros, i.e., macros, which do not allow recursive calls, can be compiled out during the compile time, while definite clauses generally have to be interpreted at run-time.

ALE, CL-ONE, ProFIT and TFS have both macro and definite relations mechanisms defined over typed feature structure terms. The main difference between these systems is that in ALE and TFS definite clauses can be attached to feature terms (and, of course, they take feature terms as arguments), i.e., they are part of the description language, while in ProFIT and CL-ONE definite clauses can be defined only *over* feature terms. Also, in ProFIT and CL-ONE definite clauses are compiled into Prolog and only Prolog unification is used.

On the other hand, CUF and ConTroll provide only definite relations, which, because of their pseudofunctional syntax, are fully integrated into the description language and can be used just as macros would be used. In fact, ConTroll performs some optimization in that it recognizes when a predicate is used as a macro (no recursion) rather than a relation, and executes this predicate at compile time.

The functionality of ALEP and PAGE is weaker than that of the above systems: only a macro facility is provided.[49]

[47] They were not allowed in Pollard and Sag (1987), though.

[48] In the case of CUF, documentation does not mention cyclic structures at all, but our experience with the system suggests that they are not supported.

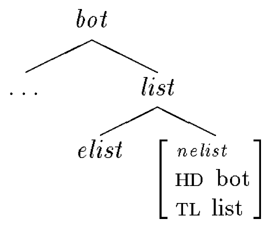
[49] PAGE allows also *functional constraints*, i.e., calls to LISP functions (Krieger and Schäfer, 1994b, sec. 4.6.15).

At the moment, none of the systems sides with Dörre *et al.* (1996b), who recommend that only macros and *a small set* of pre-defined definite relations (such as `append` and `reverse`) be used on the grounds of restricting the power of the formalism. We partly agree with this recommendation.

10.2.8 Lists, Sets

Of course, in none of these systems is a special notation for lists necessary, since lists can be defined in the type system in the standard way, cf. (106).

(106)



However, a Prolog-like syntactic sugar for list notation is very convenient, so it is provided by many of the systems: it is present in CUF (Dörre and Dorna, 1993, p. 12), ALE (Carpenter and Penn, 1994, p. 28), ALEP[50], PAGE (Krieger and Schäfer, 1994b, sec. 4.6.11) and ConTroll.[51] A LISP-like notation for lists is also easily definable in TFS (Kuhn, 1993, p. 5). However, ProFIT and CL-ONE do not allow such notation for lists of feature terms because of possible ambiguities (Prolog and ProFIT terms can be mixed in these systems).

The situation is quite different when it comes to special notation for sets. Even though sets are widely used in HPSG, only CL-ONE provides mechanisms for set description, they have to be approximated by lists in other systems.

10.3 Lexicon

10.3.1 Lexical Entries

It is natural that systems based on phrase structure rules should employ a special notation for lexical entries, while pure constraint-based systems might, but do not have to provide such a notation. Indeed, ALE and ALEP, as well as ProFIT and CL-ONE, which can make use of ALE's parser, employ a special notation for lexical entries. In the case of ALE (ProFIT, CL-ONE), lexical entries are specified as rewriting rules connecting phonology (a string) with a description being an ALE (resp., ProFIT) term (Carpenter and Penn, 1994, p. 42). Moreover, ALE supports operations on large lexicons by using Prolog's hashing mechanism. ALEP employs special mechanism for increasing retrieving efficiency, i.e., indexing keys (Simpkins, 1994a, p. 30), as well as tools for browsing and sorting lexical entries. In all of these systems macros are heavily used as shorthands for lexicon entries.

Of the other, constraint-based systems, CUF and TFS do not envisage special format for lexical entries, while in ConTroll such a format is provided as syntactic sugar for full-blown constraints (but it has to be employed if lexical rules are used). Finally, PAGE provides *instances*, which can facilitate specifying lexical entries.

[50]However, in ALEP lists are not very useful because of the lack of a definite relation extension in this system. For example, the append function can be defined only on difference lists (Pulman, 1994, p. 51).

[51]PAGE provides also syntactic sugar for difference lists (Krieger and Schäfer, 1994b, sec. 4.6.12).

10.3.2 Lexical Rules

Five of the systems described in this report allow the encoding of lexical rules, which can be used for deriving new lexical entries from those already existing in the lexicon. In ALE (Carpenter and Penn, 1994, p. 42), lexical rules are employed at compile time, i.e., there is no difference at run time between the original and derived lexical entries. Since lexical rules can be, in principle, iterated infinitely many times, it is possible to specify the maximal number of iterations in order to prevent infinite loops.

ProFIT and CL-ONE allow lexical rules when coupled with the ALE parser, so the above remarks carry over to these systems.

In the other two systems which employ lexical rules, ALEP (*morph* rules; Bredenkamp and Hentze, 1995, p. 2) and ConTroll, they are applied at compile time. It is worth mentioning that the approach to lexical rules as implemented in ConTroll and in ALE essentially follows Description-level Lexical Rules approach as advocated by Meurers (1995), and differs from the more orthodox Meta-level Lexical Rules approach of Calcagno and Pollard (1995) and Calcagno (1995).

Although no special format for lexical rules is provided in CUF, PAGE and TFS, they can be encoded by means of relational clauses in a way similar to the actual encoding in ConTroll (cf. e.g. Kuhn, 1993, p. 14).

10.4 Computational Aspects

10.4.1 Phrase Structure Rules vs. Constraints

The systems included in this report are based on two very different procedural models: phrase structure rules or constraints. The obvious advantage of the latter is that it allows implementing constraint-based grammars, such as HPSG, in a straightforward manner, although, on the other hand, the former usually are much more efficient.

CUF, TFS, PAGE and ConTroll are pure constraint-based formalisms and they do not use phrase structure rules.[52] On the other hand, both ALE and ALEP are based on PS rules, though in the former, constraints can play an important role as well. All of these systems contrast with ProFIT and CL-ONE, which are, in principle, stand-alone modules with no processing components of their own, but which can be integrated with processing modules of other systems, e.g., ALE's parser. CL-ONE, moreover, can be regarded as a constraint-based extension of ProFIT in that it provides specialized constraint solvers for set descriptions, linear precedence rules and guarded constraints.

10.4.2 Base Language

Most of the systems described in this report are based on Prolog (SICStus, Quintus or both; in the case of ALE and ConTroll also SWI Prolog),[53] the only exceptions being TFS and PAGE, which are implemented in COMMON LISP.

10.4.3 Calls

The possibility to call the base language from a system is useful for implementing additional components of the grammar, such as, for example, a graphical user interface or a morphological interface in systems which do not provide a morphological analyzer. ALEP, ConTroll and TFS

[52] In case of PAGE, however, cf. (Krieger and Schäfer, 1994b, sec. 4.6.18).

[53] Note that the base language of CL-ONE is ProFIT which in turn is Prolog-based.

do not allow calls to their base languages, while the other systems do. Since the base language of most of these systems (i.e., apart from PAGE) is Prolog (via ProFIT in the case of CL-ONE), calls to other languages are possible using Prolog's foreign language interface mechanism.

10.4.4 Delays

Delays are one of the most useful mechanisms for guiding the control in constraint-based systems, hence they are present in ConTroll (`delay/2`, `delay_deterministic/1` and `lazy_goal`), CUF (`wait/1` and `lazy_goal/1`), PAGE (e.g., `:delay`) and CL-ONE (guarded constraints). Unfortunately, no such mechanism is available in TFS, which results in low efficiency of the system, nor is it available in ALE.

10.4.5 Parser

Apart from the purely constraint-based systems, CUF, ConTroll and TFS, all of the systems considered here have at least one built-in (or coupled) parser. ALE uses a bottom-up active chart parser, (Carpenter and Penn, 1995, p. 1) whereas in ALEP, "a non-deterministic bottom-up head-out parser... and a structure sharing algorithm" (Schütz, 1994b, p. 63) are used. Moreover, "[g]rammars and lexicon written in $\mathcal{TDL}$ can be tested by using the DISCO parser," a "bidirectional bottom-up chart parser..." (Krieger and Schäfer, 1994a, p. 8).

As mentioned in section 10.4.1, ProFIT and CL-ONE do not provide built-in processing components but they may be int egrated with modules of other systems. Thus, ProFIT uses ALE's parser for processing HPSG grammars (cf. `README` file available from the system site), while there are two parsers in CL-ONE: ALE's chart parser and a left-corner parser (`cl1.doc` file included in the system package).

10.4.6 Additional Components

ALEP provides a head-driven generator (Schütz, 1994b, p. 63) and a transfer module (Schütz, 1994b, p. 11). CL-ONE is distributed with a head-driven generator module.

10.4.7 Tracing, Debugger

Good debuggers which allow one to inspect programs interactively are available in TFS (see the list of debugging commands in (Kuhn, 1993, p. 34)), CUF (cf. Dörre *et al.* (1996a)) and ConTroll. Also ALE provides reasonable tools for testing the signature, evaluation of descriptions and definite clauses, (Carpenter and Penn, 1995, ch. 7), but no tracer (apart from that provided by Prolog). On the other hand, in ProFIT and CL-ONE only the Prolog debugger is available, while ALEP allows only for a post-mortem inspection of the processing information (Simpkins, 1994b, p. 49).

10.4.8 Graphical User Interface

ALEP and PAGE have probably the most friendly menu-based working environments; in the case of the former it is based on MOTIF and Emacs (Cruickshank, 1995, p. 2), in the case of the latter it is FEGRAMED, Grapher and an Emacs mode (Krieger and Schäfer, 1994b. 43–60). Also, ConTroll uses a windows interface, XTroll. For ALE and CUF, no special GUI is provided but integration with other environments is possible (Pleuk or HDRUG, and FEGRAMED respectively). An elegant windows-based interface for TFS is available only for some LISP dialects (Emele, 1993,

p. 4). ProFIT and CL-ONE have no GUI; only the Prolog environment can be used. For ProFIT, integration with FEGRAMED is foreseen, (Erbach, 1994b, p. 10).

10.4.9 Efficiency

ProFIT is claimed by the authors to be extremely efficient since the Prolog unification procedure can be used directly, (Erbach, 1994b, p. 10). The efficiency of ProFIT does not entirely carry over to the current implementation of CL-ONE (the algorithm for set unification is memory consuming) but the next release should be free from these processing difficulties (Wojciech Skut, p.c.).

ALE's efficiency was estimated as "...15% of the speed of the SICStus 2.1 interpreter" (Carpenter and Penn, 1995, p. 1) which, in practice, gives reasonable run times. In ALEP, the efficiency is, in principle, acceptable, but it drastically decreases if highly lexicalized grammars are processed (Genabith *et al.*, 1994, p. 114). CUF is relatively (i.e., for a constraint-based system) efficient, and so is ConTroll (both have been used for implementation of large grammars in scientific projects, Verbmobil and SFB 340 (B8), respectively). The efficiency of TFS, (Emele, 1993, p. 3), is low.

It should be strongly emphasized that the efficiency judgements above are based on our experience and authors' claims, rather than on any formal tests.

10.4.10 Incremental Compilation

Incremental compilation of parts of grammars is very useful when large grammars are processed. ProFIT, CL-ONE, PAGE and TFS do not allow incremental compilation, while ALE (Carpenter and Penn, 1995, ch. 6), ALEP (Simpkins, 1994b, ch. 2), ConTroll and CUF have a variety of incremental compilation mechanisms.

10.5 Etc.

10.5.1 Availability, Maintenance and Stability

All the systems except ConTroll and ALEP are freely available for scientific purposes. ALEP and PAGE are available on request, while ConTroll should be available in 1997. All systems apart from TFS are maintained, and most of them are stable.[54] Only CL-ONE and ConTroll are still in the development stage.

10.5.2 Documentation

Systems considered here vary greatly with respect to the documentation. One extreme is ALEP, for which thick manuals and user's guides (Groenendijk and Simpkins (1994), Schütz (1994a), Simpkins (1994b), Simpkins (1994a)), as well as various other documents (Bredenkamp and Hentze (1995), Genabith *et al.* (1994)) exist, while the other is ConTroll and CL-ONE, for which hardly any documentation exists (Götz (1995) and Manandhar (1994b), respectively).

The systems which are probably best documented are ALE and CUF. Both have up-to-date and reasonably sized manuals (Carpenter and Penn (1995) and Dörre *et al.* (1996a)), as well as good documents on the systems' theoretical foundations (Carpenter (1992) and Dörre and Dorna (1993)).

[54] Apparently, CUF is not maintained any longer (Gerald Penn, p.c.).

As far as TFS is concerned, the manual (Zajac, 1991) seems to be out-of-date, although other documents can help start writing HPSG grammars in the system (Kuhn (1993), Keller (1993), Emele (1993), Emele (1994)). The theoretical background can be found in Zajac (1992).

In the case of PAGE, only its main component, $\mathcal{TDL}$ is relatively well documented: a theoretical overwiev can be found in Krieger and Schäfer (1994a), and a more practical guide in Krieger and Schäfer (1994b). More general information on PAGE can be found in the web in: `http://cl-www.dfki.uni-sb.de/cl/systems/page/page.html`.

Finally, a reasonable ProFIT manual (Erbach, 1995) exists, along with a more theoretical squib (Erbach, 1994b) and a paper on multi-dimensional inheritance (Erbach, 1994a), though the three happen to contradict each other (e.g., when it comes to the Feature Introduction Condition).

10.5.3 HPSG Grammars

Various HPSG grammars have been implemented in the systems described here. Probably the biggest of them are the two German HPSG grammars implemented in CUF (in the Verbmobil project) and in ConTroll (in the Sonderforschungsbereich 340 project, subprojects B8 and B4), as well as the DISCO German grammar and the CSLI English grammar, both implemented in PAGE (i.e., $\mathcal{TDL}$). Some 'HPSG-inspired' grammars have been implemented also in ALEP: a German grammar (Schmidt, 1994), an English grammar (Genabith *et al.*, 1994) and a small Spanish grammar (Badia, 1994).

As far as the other systems are concerned, the biggest implemented HPSG grammar seems to be (large parts of) the English HPSG grammar of Pollard and Sag (1994). An implementation of chapters 1–5 and 8 comes with the distribution of ALE and it has been also translated into ProFIT, while so-called HPSG-II (chapters 1–8) and HPSG-III (chapter 9) versions of the grammar have been implemented in TFS (cf. Keller (1993) and Kuhn (1993)). In the case of CL-ONE, only a small grammar dealing with German *Mittelfeld* has been implemented.

10.6 Summary

Type System

Feature	**ALE**	**ALEP**	**ConTroll**	**CUF**	**PAGE**	**ProFIT**	**TFS**
type algebra	FMSL	FRLO	FJSL	(FPO)	FPO	FPO	FPO
intensional and extensional	+	−	−	+	−	+	−
open vs. closed world	open		closed	both	open/ closed	open (/closed)	closed
disjointness	default		default	possible	possible	default	default
partition condition	difficult		yes	possible	possible	yes	yes
inheritance	+	+	+	+	+	+	+
multiple inheritance	+	−	+	+	+	+	+
further specifications	+	−/+	+	+	+	−	+
generality	+	+	+	−	+	−	+
FIC	+	−	−	−	−	−(/+)	−
well-typedness	TWT	TWT	TWT	not WT	(WT)	TWT	TWT

Synatx and Feature Terms

Feature	ALE	ALEP	CL-ONE	ConTroll	CUF	PAGE	ProFIT	TFS
conjunction	+	+	+	+	+	+	+	+
disjunction	+	+	+	+	+	+	+	−
negation	−/+	−/+	−/+	+	+	+	−/+	−/+
variables	+	+	+	+	+	+	+	+
path inequality	+	−	+/−	−	+/−	+	+/−	+
cyclicity	+	−	+	+	+	+	+	+
definite relations	+	−	−	+	+	−	−	+
macros	+	+	+	−	−	+	+	+
lists	+	+/−	+/−	+	+	+	+/−	+/−
sets	−	−	+	−	−	−	−	−

Lexicon

Feature	ALE	ALEP	CL-ONE	ConTroll	CUF	PAGE	ProFIT	TFS
lexical entries	+	+	−/+	+	−	−/+	−/+	−
lexical rules	+	+	−/+	+	−	−	−/+	−

Computational Aspects

Feature	ALE	ALEP	CL-ONE	ConTroll	CUF	PAGE	ProFIT	TFS
PS rules vs. constraints	PS/cons	PS	cons/(PS)	cons	cons	cons	(cons/PS)	cons
base language	Prolog	Prolog	ProFIT	Prolog	Prolog	Lisp	Prolog	Lisp
calls allowed	+	−	+	−	+	+	+	−
delays	−	−	+	+	+	+	−	−
parser	+	+	+	−	−	+	−/+	−
additional components	−	gen, tr	gen	−	−	−	−	−
tracing	−	+	+	+	+	− (?)	+	+
debugger	ok	poor	Prolog	good	good	− (?)	Prolog	good
GUI	−/+	+	−	+	−/+	+	−	−/+
efficiency	good	good	good	low/ok	ok/low		very good	very low
incremental compilation	+	+	−	+	+	−	−	−

Etc.

Feature	ALE	ALEP	CL-ONE	ConTroll	CUF	PAGE	ProFIT	TFS
availability	+	+/−	+	−	+	+/−	+	+
maintenance	+	+	+	+	+(?)	+	+	−
stability	+	+	−	−	+	+	+	+
documentation	(very) good	huge	poor	none	(very) good	some/ good	some	some/ good
HPSG grammars	+	+/−	−/+	++	++	++	+	+

References

Alshawi, H., Arnold, D., Backofen, R., Carter, D., Lindpo, J., Netter, K., Pulman, S., Tsujii, J., and Uszkoreit, H. (1991). Eurotra ET6/1: Rule formalism and virtual machine design study (final report). Technical report, Commission of the European Communities.

Backofen, R., Krieger, H.-U., Spackman, S. P., and Uszkoreit, H. (1993). Report of the EAGLES workshop on implemented formalisms at DFKI, Saarbrücken. Research Report D-93-27, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.

Badia, T. (1994). The Spanish grammar. In S. Markantonatou and L. Sadler, editors, *Grammatical Formalisms: Issues in Migration and Expessivity*, pages 139–166. Commission of the European Communities, Luxembourg.

Bredenkamp, A. and Hentze, R. (1995). Some aspects of HPSG implementation in the ALEP formalism. Technical report, University of Essex, Barcelona.

Calcagno, M. (1995). Interpreting lexical rules. In *Proceedings of ESSLLI7 Conference on Formal Grammar*. Universidad Politecnica de Catalunya, Barcelona.

Calcagno, M. and Pollard, C. (1995). Lexical rules as meta-descriptions. ACQUILEX workshop on lexical rules, Cambridge, UK.

Carpenter, B. (1992). *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Carpenter, B. and King, P. J. (1995). The complexity of closed world reasoning in constraint-based grammar theories. Presented to 4th Conference on Mathematics of Language. Philadelphia.

Carpenter, B. and Penn, G. (1993). Negation vs. inequation and typing for linguistic applications. Unpublished manuscript.

Carpenter, B. and Penn, G. (1994). *The Attribute Logic Engine (Version 2.0). User's Guide*. Carnegie Mellon University, Pittsburgh.

Carpenter, B. and Penn, G. (1995). *The Attribute Logic Engine (Version 2.0.1). User's Guide*. Carnegie Mellon University, Pittsburgh.

Cruickshank, G. (1995). *Introduction to the Environment*. Cray Systems.

Dörre, J. and Dorna, M. (1993). CUF — a formalism for linguistic knowledge representation. In J. Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I*. DYANA.

Dörre, J. and Eisele, A. (1991). A comprehensive unification-based grammar formalism. Technical Report R3.1.B, DYANA, Centre for Cognitive Science, University of Edinburgh, Edinburgh.

Dörre, J., Dorna, M., and Junger, J. (1996a). *The CUF User's Manual*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 1.7 edition.

Dörre, J., Erbach, G., Manandhar, S., Skut, W., and Uszkoreit, H. (1996b). A report on the draft EAGLES Encoding Standard for HPSG. In *Proceedings of Traitement Automatique de Langage Natural (TALN)*, Marseille.

Emele, M. C. (1993). TFS: The typed feature structure representation formalism. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart.

Emele, M. C. (1994). TFS: The typed feature structure representation formalism. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart.

Emele, M. C. and Zajac, R. (1990a). A fixed-point semantics for feature type systems. In *Proceedings, 2nd Workshop on Conditional and Typed Rewriting Systems*, Montreal, Quebec.

Emele, M. C. and Zajac, R. (1990b). Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, COLING-90, pages 293–298.

Erbach, G. (1994a). Multi-dimensional inheritance. In H. Trost, editor, *KONVENS '94*, pages 102–111, Vienna.

Erbach, G. (1994b). ProFIT: Prolog with features, inheritance and templates. In *32nd Annual Meeting of the Association for Computational Linguistics*, New Mexico State University, Las Cruces, New Mexico, USA. ACL.

Erbach, G. (1995). *ProFIT 1.54 user's manual.* Universität des Saarlandes, Saarbrücken.

Erbach, G., van der Kraan, M., Manandhar, S., Ruessink, H., Skut, W., and Thiersch, C. (1995). Extending unification formalisms. In *Constraint based formalisms and grammar writing.* Seventh European Summer School in Logic, Language and Information, Barcelona.

Gardent, C. (1993). Computational tools for discourse. Intermediate Report LRE 61-062 [C.0].

Geißler, S. (1994). Lexikalische Regeln in der IBM-Basisgrammatik. Verbmobil Report 20, IBM Informationssysteme GmbH.

Geißler, S. and Kiss, T. (1994). Erläuterungen zur Umsetzung einer HPSG im Basisformalismus STUF III. Verbmobil Report 19, IBM Informationssysteme GmbH.

Genabith, J., Markanonatou, S., Sadler, L., and Verhagen, M. (1994). English HPSG in ALEPH_0. In S. Markantonatou and L. Sadler, editors, *Grammatical Formalisms: Issues in Migration and Expessivity*, pages 91–116. Commission of the European Communities, Luxembourg.

Gerdemann, D. (1993). *Troll: Type Resolution System – Fundamental Principles and User's Guide.* Seminar für Sprachwissenschaft, Universität Tübingen, Tübingen.

Gerdemann, D. (1995). Open and closed world types in NLP. In J. Kilbury and R. Wiese, editors, *Integrative Ansätze in der Computerlinguistik*, pages 25–30, Düsseldorf.

Götz, T. (1995). *The* ConTroll *User's Guide and Manual: First Draft.* Seminar für Sprachwissenschaft, Universität Tübingen.

Götz, T. (1996). ConTroll: theoretical and practical aspects. Handout of a talk presented in Oberjoch, March 1996.

Götz, T. and Meurers, W. D. (1995). Compiling HPSG type constraints into definite clause programs. In *Proceedings of the Thirty-Third Annual Meeting of the ACL*, pages 85–91, Cambridge, MA. ACL.

Götz, T. and Meurers, W. D. (1996). The importance of being lazy: Using lazy evaluation to process queries to HPSG grammars. In *Proceedings of TALN 96 (joint session with the Third International Conference on HPSG)*, Marseille, France.

Groenendijk, M. and Simpkins, N. (1994). *ALEP Reference Guide.* Cray Systems.

Hegner, S. (1994). Distributivity in incompletely specified type hierarchies. theory and computational complexity. In J. Dörre, editor, *Computational Aspects of Constraint-based Linguistic Description II*, number R1.2.B in DYANA Deliverables.

Kasper, R. and Rounds, W. (1986). A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of thhe Association for Computational Linguistics*, Morrisontown, N.J. Association for Computational Linguistics.

Kasper, R. T., Kathol, A., and Pollard, C. (1995). Linear precedence constraints and reentrancy. In J. Kilbury and R. Wiese, editors, *Integrative Ansätze in der Computerlinguistik*, pages 49–54, Düsseldorf.

Kathol, A. (1995). *Linearization-Based German Syntax.* PhD dissertation, Ohio State University.

Keller, F. (1993). Encoding HPSG grammars in TFS. Part II. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart.

King, P. J. (1989). *A logical formalism for head-driven phrase structure grammar.* PhD dissertation, Manchester University, Manchester, England.

King, P. J. (1994). An expanded logical formalism for Head-driven Phrase Structure Grammar. Sonderforschungsbereich 340 (Sprachtheoretische Grundlagen für die Computerlinguistik) Bericht Nr. 59, Seminar für Sprachwissenschaft, Universität Tübingen, Tübingen.

King, P. J. and Götz, T. (1993). Eliminating the feature introduction condition by modifying type inference. Sonderforschungsbereich 340 (Sprachtheoretische Grundlagen für die Computerlinguistik) Bericht Nr. 31, Seminar für Sprachwissenschaft, Universität Tübingen.

König, E. (1995). A CUF tutorial. http://www.ims.uni-stuttgart.de/cuf.

Krieger, H.-U. and Schäfer, U. (1994a). TDL — a type description language for HPSG. Part 1: Overview. Technical Report RR-94-37, DFKI, Saarbrücken.

Krieger, H.-U. and Schäfer, U. (1994b). TDL — a type description language for HPSG. Part 2: User guide. Technical Report D-94-14, DFKI.

Kuhn, J. (1993). Encoding HPSG grammars in TFS. Part I. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart.

Kupść, A. (1995). Kryteria oceny systemów do implementacji HPSG. IPI MMGroup Internal Document.

Manandhar, S. (1993). CUF in context. In J. Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I*, volume R1.2.A of *DYANA-2*, pages 45–53, Centre for Cognitive Science, University of Edinburgh.

Manandhar, S. (1994a). An attributive logic of set descriptions and set operations. In *Proceedings of ACL94*.

Manandhar, S. (1994b). *User's Guide for CL-ONE.* Centre for Cognitive Science, University of Edinburgh, Scotland.

Manandhar, S. (1995). Deterministic constituency checking of LP constraints. In *Poceedings of EACL95*, Dublin, Ireland.

Meurers, W. D. (1994). On implementing an HPSG theory. In E. Hinrichs, D. Meurers, and T. Nakazawa, editors, *Partial VP and Split NP Topicalization in German: An HPSG Analysis*, volume 58 of *Arbeitspapiere des SFB 340.*

Meurers, W. D. (1995). Towards a semantics for lexical rules as used in HPSG. In *Proceedings of the Formal Grammar Conference*, Barcelona, Spain.

Meurers, W. D. (1996). Some notes on writing grammars in ConTroll. Handout of a talk presented in Oberjoch, March 1996.

Meurers, W. D. and Minnen, G. (1995). A computational treatment of HPSG lexical rules as covariation in lexical entries. In *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Portugal.

Meurers, W. D. and Minnen, G. (1996). Off-line constraint propagation for efficient HPSG processing. In *HPSG/TALN Proceedings*, Marseille, France.

Pollard, C. (1989). Sorts in unification-based grammar and what they mean. Unpublished manuscript.

Pollard, C. and Sag, I. A. (1987). *Information-Based Syntax and Semantics, Volume 1: Fundamentals.* Center for the Study of Language and Information.

Pollard, C. and Sag, I. A. (1994). *Head-driven Phrase Structure Grammar.* Chicago University Press.

Przepiórkowski, A. (1995). Wstępny przegląd systemów przydatnych do implementacji HPSG. IPI MMGroup Internal Document.

Przepiórkowski, A. (1996). Case assignment in Polish: Towards an HPSG analysis. In C. Grover and E. Vallduví, editors, *Edinburgh Working Papers in Cognitive Science, Vol. 12: Studies in HPSG*, pages 191–228. Centre for Cognitive Science, University of Edinburgh.

Pulman, S. (1994). Expressivity of lean formalisms. In S. Markantonatou and L. Sadler, editors, *Grammatical Formalisms: Issues in Migration*, volume 4 of *Studies in Machine Translation and Natural Language Processing.* European Commission, Brussels.

Schmidt, P. (1994). The German grammar. In S. Markantonatou and L. Sadler, editors, *Grammatical Formalisms: Issues in Migration and Expessivity*, pages 167–188. Commission of the European Communities, Luxembourg.

Schütz, J. (1994a). The ALEP Formalism in a Nutshell. Technical report, IAI Saarbrücken.

Schütz, J. (1994b). *Terminological Knowledge in Multilingual Language Processing*, volume 5 of *Studies in Machine Translation and Natural Language Processing.* European Commission, Brussels.

Simpkins, N. (1994a). *ET-6/1 Linguistic Formalism.* Cray Systems.

Simpkins, N. (1994b). *Linguistic Development and Processing. User Guide.* Cray Systems.

Stolzenburg, F., Höhne, S., Koch, U., and Volk, M. (1996). Constraint logic programming for computational linguistics. In C. Retoré, editor, *Proceedings of the Conference on Logical Aspects of Computational Linguistics*, pages 19–23, Nancy, France. INRIA Lorraine and CRIN-C.N.R.S.

Uszkoreit, H., Backofen, R., Busemann, S., Diagne, A. K., Hinkelman, E. A., Kasper, W., Kiefer, B., Krieger, H.-U., Netter, K., Neumann, G., Oepen, S., and Spackman, S. P. (1994). DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of COLING-94, Kyoto, Japan.*

Zajac, R. (1991). Notes on the typed feature system. Project POLYGLOSS.

Zajac, R. (1992). Inheritance and constraint-based grammar formalisms. *Computational Linguistics*, **18**(2), 159–182.