The ALEP platform for language research and engineering

Report of the EAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken

Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, Hans Uszkoreit
editors

March 1-3, 1993

Contents

Part 1

Introduction

From the 1st through the 3rd of March 1993, the Working Group on Linguistic Formalisms of the EAGLES initiative held a workshop in Saarbrücken on implemented grammar formalisms.

Starting with some notes on the Working Group, we will describe objectives, organization, and results of the workshop. We also summarize some relevant general findings as they emerged from the final discussion.

The systems demonstrated are described in a detailed synopsis. In order to facilitate comparison a standardized questionnaire was used for the individual descriptions. The questionnaires were filled out by the developers. Since there might always be relevant pieces of information that do not fit well in such a questionnaire, the developers could also provide a short prose description. Most developers took advantage of this opportunity and attached a brief summary of their system.

1.1 The Working Group

The Working Group on Linguistic Formalisms of the EAGLES initiative brings together experts on the design and implementation of linguistic formalisms from academia and industry in order to :

- come to a consensus on the basic features and properties for NLP formalisms and indicate likely and needed future features;
- promote consensus with respect to the definition of de facto standards for grammar formalisms;
- exchange information about each other's projects and, as far as compatible with IPR, know-how and results, thus increasing the awareness of possible synergies;
- where appropriate, concretize potential synergies by promoting cooperative actions, thus furthering the definition of de-facto standards in the field;
- disseminate information about its activities, participate in and organize events aimed at make these activities better known (round-tables, workshops, conferences);

- coordinate and cooperate with national and international initiatives; and
- suggest actions needed for the creation of formal and computational prerequisites for the development of multilingual, reusable, grammatical resources.

The group is hosted by the DFKI in Saarbrücken.
The members of the WG are:

H. Ulrich Block - Siemens AG, Munich,
Ewan Klein - University of Edinburgh,
Jeremy Peckham - Logica Cambridge,
Steve Pulman - SRI Cambridge,
Christian Rohrer - University of Stuttgart,
Hans Uszkoreit - DFKI Saarbrücken (chair).

Many additional industrial and academic research institutions are represented by specialists in the three Subgroups of the Working Group:

Linguistic Adequacy,
Computability and Implementation,
Industrial Requirements.

1.2 Objectives of the Workshop

The main objective of the workshop was to obtain an urgently needed overview of existing software systems, including their development platforms, that implement state-of-the-art grammar formalisms. It is not possible to derive such an overview by surveying the literature. In their publications, the developers of such formalisms usually focus on certain selected aspects of their systems that constitute novel scientific approaches. The state of the implementation of a system and its robustness, performance and overall usability can never be judged from the literature.

Another motivation was the broadly-felt necessity to exchange experience gathered in implementing constraint-based grammar formalisms among the relevant developers.

Therefore the focus of this workshop was not on the linguistic, philosophical and semantic foundations of advanced typed feature-unification formalisms. The meeting concentrated on existing implementations.

One goal of the workshop was to obtain an overview of what is feasible and usable today. This overview served as the starting point for the Working Group's activities during the survey phase. It is an important part of the ongoing survey of existing implemented linguistic formalisms. It will also provide a good basis for characterizing the state of the art in this area.

To participants working in the area of formalism development, the workshop offered a unique opportunity to learn those facts about other researchers' formalisms that cannot be found in the literature.

1.3 Organization

At the workshop we made sure that the mutual review did not turn into a contest. The evaluation and assessment of NLP systems has not yet reached a state that permits an objective comparison of systems. The theoretical premises on which the systems were developed and the goals of their developers differ so much that it would be impossible to agree on a reasonable single ranking scheme. On the other hand, a comparison of implementations along several dimensions was undertaken because it is urgently needed by everyone working in this area. Such comparative knowledge is also very important for the further work of EAGLES. In order to arrive at such a synopsis, detailed questionnaires were designed and handed out to all participants. We had a return rate in excess of 100%, due to the fact that some participants copied the questionnaire to describe additional systems not shown at the workshop. These description are not considered in the current report on implemented formalisms but they have been included in the working material of the group.

Seventeen systems were invited: fourteen from European research sites, three from the USA. This imbalance not only reflects the focus of the EAGLES WG but demonstrates the growing role that European sites play in today's formalisms research. At the workshop fourteen systems were exhibited. Two sites turned down the invitation because their new systems were not yet in a state to be demonstrated; one American researcher was unable to attend because of a scheduling conflict.

All systems were first presented in brief talks. On the second day, six review teams, each of three researchers, were formed. Although in theory every participant could see every system, reviewers often focussed on the systems they were to report on. The results of this review were reported in a plenary session. After each report the developers of the respective systems received time for rebuttal. Most of the questions from the audience concerned implementation details. The discussion was very concrete and stayed on a high technical level. Although the review did not follow measurable evaluation criteria, the findings and subjective comments were generally accepted by the developers.

1.4 Results and Findings

For the discipline of computational linguistics as a whole, the workshop has resulted in this synopsis of relevant implemented formalisms. All participating developers have provided a standardized characterization of their system by filling out a very detailed questionnaire.

In a final session all participants contributed to a discussion of general developments and urgent problems. This discussion was very important for further WG activities.

The following summary lists the most relevant findings of the workshop.

Clear progress: All of the systems showed strong advantages over the formalisms that were used a decade ago.

European Role: European research in the area no longer trails behind the corresponding American research.

Lack of Industrial Strength: None of the systems has, as yet, the desired performance behaviour for broad industrial application: some could be used in limited applications, others have potential in this direction. The most pressing problem for research on formalisms is the need for adequate performance models.

Convergence: There are strong tendencies towards convergence: e.g., most formalisms are based on typed feature logics, most new systems use multiple inheritance hierarchies, and so on.

Connections to constraint logic programming: There is a close relationship with ongoing developments in programming language design, especially in CLP, that needs to be explored further.

It was pointed out that the systems demonstrated represented quite a spectrum of compromises between sophistication in linguistic description on the one hand and efficiency in processing on the other. The range of such decisions was noted and discussed.

In the subsequent discussion the concern was raised that improvements in the actual formalisms such as powerful type systems and additional data types have not brought the systems any closer to exploitation in industrial strength systems and, moreover, that efficient processing models might even have become less likely through such development. The strong move towards low-level processing methods and statistical approaches would indicate the dissatisfaction with the pure feature logic based approach.

Proponents of the current formalism research replied that the real challenge lies in combining the sophisticated high-level description approaches with powerful low-level processing methods. Low level processing methods and statistical approaches to grammar development alone have not been able so far to arrive at comprehensive, reusable grammatical resources. They also pointed at the attempts to come up with strategies for effectively controlling linguistic processing based on constraint-based grammars.

Another point of discussion was the utilization of results from constraint logic programming. A number of demonstrated systems were actually described in CLP terms as special instances of the Höhfeld/Smolka model. The question was raised whether one should not leave the search for more efficient processing models to the much larger community of CLP researchers. It was replied that the linguistic descriptions for NLP offer such a strongly structured and challenging domain to CLP that it would be in the mutual interest of both communities to work together. The CLP community was represented at the meeting by such eminent scientists as Hassan Aït-Kaci, Luis Damas, Andreas Podelski, and Gert Smolka. It was decided that the connections between CLP and NLP should be strengthened. The upcoming workshop of the Computability and Implementation Subgroup will focus on relevant developments in CLP and on the exchange between the two communities.

Several participants proposed to hold a follow-up meeting in a year's time. In the discussion it became clear that such an event would most likely not have the same status with respect to EAGLES work, but that the EAGLES working group might

serve as co-sponsor. Some very concrete suggestions concerning the organization of the next meeting were made. They were discussed and taken to the records.

Summing up it became obvious that the workshop not only delivered important empirical input for the program of the EAGLES Formalisms Working Group, it also provided the participants with a better overview of the state of the art and served as a forum for discussing new trends, burning issues and further collaboration.

Hans Uszkoreit

Part 2

Systems Exhbited

2.1 ALE: An Attribute Logic Engine

Bob Carpenter
Computational Linguistics Program, Philosophy Dept.
Carnegie Mellon University, Pittsburgh, PA 15213
Net: carp@lcl.cmu.edu Phone: (412) 268-8573 Fax: (412) 268-1440

ALE, a public domain system written in Prolog, integrates phrase structure parsing and constraint logic programming with typed feature structures as terms. This generalizes both the feature structures of PATR-II and the terms of Prolog II to allow type inheritance and appropriateness specifications for features and values. Grammars may also interleave unification steps with logic program goal calls (as can be done in DCGs), thus allowing parsing to be interleaved with other system components. While ALE was developed to handle HPSG grammars, it can also execute PATR-II grammars, DCG grammars, Prolog, Prolog-II, and LOGIN programs, etc.

Grammars and logic programs are specified using a typed version of Rounds-Kasper attribute-value logic, which includes variables and full disjunction. Programs are then compiled into low-level Prolog instructions corresponding to the basic operations of the typed Rounds-Kapser logic. There is a strong type discipline enforced on descriptions, allowing many errors to be detected at compile-time.

The logic programming and parsing systems may be used independently or together. Features of the logic programming system include negation, disjunction and cuts. It has last call optimization, but does not perform any argument indexing. On the "naive reverse" benchmark, it performed at 1000 LI/s on a DEC 5100 running SICStus 2.1, which is roughly 7% as fast as the SICStus interpreter and 0.7% as fast as the SICStus compiler.

The phrase structure system employs a bottom-up all-paths dynamic chart parser. A general lexical rule component is provided, including procedural attachment and general methods for orthographic transformations using pattern matching or Prolog. Empty categories are permitted in the grammar. Both the phrase structure and logic programming components of the system allow parametric macros to be defined and freely employed in descriptions. Parser performance is similar to that of the logic programming system. In an early HPSG grammar, where feature structures consisted of roughly 100–200 nodes each, a 10 word sentence producing 25 completed inactive edges parsed in roughly two seconds, using SICStus 2.1 on a DEC 5100.

Complete documentation (running to 100 pages, with examples of everything, programming advice, and sample grammars), is available as:

> Bob Carpenter (1992) ALE User's Guide. Carnegie Mellon University Laboratory for Computational Linguistics Technical Report. Pittsburgh.

ALE can be run in either SICStus or Quintus Prolog, and with other compatible compilers doing first-argument indexing and last-call optimization. The system and its documentation are available without charge for research purposes.

A future version of ALE should be available by Summer 1993 which contains a full implementation of inequations, extensionality, atoms, hooks to Prolog, general constraints on types and a number of optimizations.

System Name: ALE
Designed and Implemented by: Bob Carpenter

GENERAL DESCRIPTION I		
Inference Engine		
	unique engine vs. dedicated modules	single system with general modules for types, descriptions and unification, definite clauses, lex rules and chart parsing
	non-montonic devices	none
	control facilities	CFG: breadth-first, bottom-up definite clauses: Prolog style with cut, negation by failure and disjunction
	parser/generator?	parser (Generator under development)
	others	empty categories and lexical rules
Data Types		
	arity (fixed?)	each type has fixed arity, but subtypes can extend features (records in implementation)
	cyclic structures	yes—fully integrated with types
	lists/sets	lists as structures, using Prolog defs as macros
	functions/relations	general definite clause, definable functions and relations
	others	
Interaction FS $\Longleftrightarrow$ Types		
	type unification	yes—by hashing
	type expansion	
	at definition/ compile time	eager for lexical entries
	at run time: (delayed/partial/ recursive)	eager type inference for rules/edges—every unification that changes types causes inference
	others (templates...)	parametric macros—compile time expansion to descriptions

GENERAL DESCRIPTION II		
Interfaces to		
	morphology	general lexical rules transforming structures; template spelling changes and hooks to Prolog
	semantics/ knowledge repr.	general hooks to Prolog and definite clauses over feature logic
Implementational Issues		
	programming lang.	Sicstus/Quintus Prologs
	machine	any
	others (O/S, graphics...)	–
Applications		
	grammar theories?	HPSG, CG (simple), attribute-value phonology (general purpose)
	educational vs. commercial system	public domain
	used in projects/ other systems?	yes—but not by me
Grammar coded		
	size	HPSG vol. II—chapters 1–3 (ca. 150 nodes/lex entry)
	language	English/German
Tools		
Comments		Modular development, so following may be extracted: • type compiler • unification • description compiler • definite clause resolution

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	built on Prolog backtracking/copying for chart
non-destructive	
disjunction:	
atoms only	full atomic expressibility
full (DNF)	full-lexicon to DNF, chart to DNF/rules at runtime
distributed	
others	in definite clauses—full disjunction
negation	
atoms only	isa- and isnota-negation encodable in types
negated corefs	yes
full	
others	inequations (and unappropriateness in type system)
implication	
via negation	
others	type inference
Additional Operations	
subsumption	no
functional uncertainty	expressible with definite clauses
others	extensionality declarations—interact with inequations
Tools	
Comments	type system compiled out, type inference compiled and at run-time—interacts with description solver and unifier

TYPE SYSTEM		
Type Connectives		
	conjunction: single vs. multiple inheritance	multiple inheritance—interacts with types /appropriateness
	disjunction	implicit
	negation	implicit
	others	ISA/ISNOTA
Type Definitions		
	via feature structures	under development
	via appropriateness conditions	yes—declare appropriate features and value types (allows encoding of FS constraints)
	recursive?	no
	others	
Additional Operations		
	type inference/ classification	full inference—linear (infers value types and classification by appropriateness)
	GLB/LUB type subsumption	
	others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)		can be specified by 1. systemic /HPSG partitions compiles to → 2. ISA/ISNOTA declarations compiles to → 3. BCPO
Tools		compile-time type-checking
Comments		

2.2 The ALEP Platform for Language Research and Engineering[a]

N K Simpkins [(1)], **M Groenendijk** [(1)] **& P Meylemans** [(2)b]
[(1)] **P-E International,**
11b Boulevard Joseph II,
L-1840 Luxembourg
[(2)] **Commission of the European Communities,**
Bâtiment Jean Monnet B4/120A,
L-2920 Luxembourg

This paper describes some aspects of the Advanced Linguistic Engineering Platform (ALEP) prototype system (ALEP-0) [SIM-93a]. ALEP is an initiative of the Commission of the European Communities (CEC) to provide the natural language research and engineering community in Europe with a versatile and flexible general purpose development environment.

The linguistic formalism and tools of the current prototype, and development of a full more extensive and open environment are outlined. The architecture of the platform which is intended to support cooperation, exchange and re-use of results and resources, comparative evaluation and application prototyping, is also described.

Keywords: ALEP, Advanced Linguistic Engineering Platform
Linguistic Tools, Machine Translation
Natural Language Processing, European Initiatives

2.2.1 Introduction

Natural Language Processing (NLP) and application development projects currently lack a solid, commonly accepted and widely available platform for the development of large scale linguistic resources and applications. As a consequence, researchers and system designers are forced to build the tools and development aids they need from scratch, before undertaking the implementation of what matters most to them; linguistic resources or applications. This situation constitutes a major bottleneck for any serious attempt to build a strong and effective European NLP industry.

Within the Linguistic Research & Engineering (LRE) programme the CEC has invested in a generic, formal and computational environment, which can be put at the disposal of all Community and national R&D projects in relevant areas. This environment is called ALEP (Advanced Linguistic Engineering Platform).

In making widely available the ALEP system, the CEC aims to promote cooperation between different research centres and to progress towards portability and re-use of research results.

A typical user of the ALEP environment will be either a skilled researcher in computational linguistics or a team of researchers and application designers, who will be provided with a software environment enabling them to produce linguistic descriptions of different languages, for a number of different NLP application domains.

2.2.2 Development of ALEP

The development and distribution of ALEP is planned in a number of stages:

- Preparation and design (1991-1992)
- A prototyping stage (1992)
- A development stage (1992-1994)
- Phase-in stage (1994-1995)

[a] within the CEC's Linguistic Research and Engineering (LRE) Programme [CEC-91]

[b] The authors would like to thank Roberto Cencioni and Nino Varile of the CEC for their invaluable contributions.

Several research centres and universities, such as SRI-CRC, UMIST, IAI, CAP GEMINI, SNI, were involved in the preparatory stage ([ALS-91], [IAI-91], [DEV-91]). The development of the final platform has been contracted to BIM, with subcontractors SEMA Group, SRI-CRC and IAI. P-E International have been charged with the development of a prototype system as well as maintenance and support services for the final system.

2.2.3 Formalism

The basic ALEP formalisms were designed by SRI International Cambridge within the ET-6/1 rule formalism and virtual machine design study [ALS-91]. Several different formalisms are provided for:

- analysis of word form variation (two-level rules)
- syntactic and semantic analysis/synthesis (typed unification grammar)
- transfer-based MT (general transfer rule formalism)

The central analysis and synthesis formalism has a context-free skeleton but does not intend to embody any particular linguistic theory. The formalism was designed to be conservative, 'mainstream', efficient, expressive, declarative, reversible and monotonic. This typed unification grammar has a three level architecture:

- Level 1: simple 'PATR like' terms, constraints applied directly by unification.
- Level 2: notational enrichments which can be compiled into constructs of level 1.
- Level 3: notational enrichments which cannot be compiled into constructs of level 1 and which require additional machinery over and above unification.

The prototype ALEP-0 implements a large part of the ET-6/1 specification with a few restrictions and differences:

- The 'concrete' syntax is different (close to that of the ψ terms described in ET-6/1, p220).
- The user language (Level 2) is incomplete in not allowing some notations to be used (unordered elements) or not allowing their use at specific points (eg disjunction over sharing). Tuples and specifiers are also not supported. Most of these restrictions can be worked around. No preference mechanism is yet provided.
- The type system has been altered to impose a stricter typing. Attributes are associated with either a basic type (list, atom, boolean expression or term) or with the name of another user defined type, itself composed of typed attributes. All types are of fixed arity. The type system also allows for different attributes to have the same name when within different types and allows for simple compilable type hierarchies and inheritance.

The algorithms which perform analysis, refinement, transfer and synthesis have been isolated from the main virtual machine component. This allows selection of an appropriate algorithm for a specific grammar when the system is invoked, and for third party contribution of new algorithms.

The core of the formalism (Level 1) has a conservative design for potential efficiency. As such this is under-expressive for some users. The third level of the formalism allows for unprescribed extension with 'external' constraint systems which operate in parallel, after unification (Figure 2.1) [SIM-93b].

The ALEP-0 algorithms now include an experimental set of calls to such an external system. An illustrative sample negation solver is supplied with the system.

Figure 2.1: Interaction of core unification & external constraint solvers

Figure 2.2: Outline of ALEP-0 environment modules

Figure 2.3: Outline of full ALEP Environment

2.2.4 Environment

The ALEP-0 prototype software is intended for small to medium scale lingware development, debugging and testing. It provides a formalism and tools outlined in Figure 2.2.

The most important application is the **Virtual Machine** (VM) with either graphical or command-line oriented user interface. The VM is implemented using Quintus Prolog Version 3.1.1.

Xalep ([GRO-93b]) is an experimental, simple to use, graphical user interface to the VM. It consists of a number of 'toolboxes' (analysis, transfer, synthesis, text-, object- and lingware-handling). This is implemented in C using the OSF/Motif widget set and Quintus' Prolog Foreign Language Interface. **Xalep** can run on most displays with X-Windows version 11 and makes direct calls to the Virtual Machine.

ALEP-0 is not intended as open software as designed in ET-6/2 [IAI-91]. The software applications are monolithical and only make use of other applications, such as the graphical feature viewer, via simple Unix calls and ICCCM based communication. Integration of other applications requires source code changes, although the VM allows for user contributed algorithms as separate modules and extensions via the *Hooks* and *External Operations Library* mechanisms [SIM-93b].

ALEP-0 is distributed with the following additional tools:

- *XmInfo*: a graphical tool for browsing hierarchically structured on line documentation. All ALEP-0 documentation can be used as an on line reference manual for ALEP-0 users.
- *Xmfed*: a graphical feature viewer for viewing large (linguistic) structures, integrated with Xalep and also with the VM tracer to provide graphical feedback during tracing of linguistic operations [GRO-93a].
- *App*: a customized linguistic macro preprocessor.
- *alepemacs*: a grammar editing mode (elisp) for the GNU emacs editor with dynamic syntax checking and string completion.

2.2.5 Further Development

As outlined in the development plan, ALEP-0 is only the small-scale prototype of the full ALEP platform [MEY-93]. The first version of the full environment has been designed and implemented by BIM [BIM-92], [BIM-93]. This version, ALEP-1, is to undergo assessment before a second development cycle which will end in 1994. The environment of ALEP-1 (Figure 2.3) is formalism independent and open to customization and extension.

The default formalism available within this system is closely based on the ET-6.1 design, extended inline with developments under ALEP-0. The ALEP-1 formalisms are largely compatible with that of ALEP-0 such that lingware developed under the prototype can be reused with ALEP-1 at low cost [THE-93].

Bibliography

[ALS-91] Alshawi H, Arnold D J, Backofen R, Carter D M, Lindop J, Netter K, Pulman S G, Tsujii J & Uszkoreit H, *Eurotra ET6/1: Rule Formalism and Virtual Machine Design Study (Final Report)* , CEC 1991.

[BIM-92] BIM, *The Functional Specifications and High-Level Design of ALEP 1.0*, CEC, 1992.

[BIM-93] BIM, *ALEP Distributed Architecture; Integrating User-Interface Tools and Applications in ALEP*, Version 1.0, CEC, 1993.

[CEC-91] Commission of the European Communities, *Council Decision of 7 June 1991 adopting a specific programme of research and technological development in the field of telematic systems in areas of general interest (1990-1994) - (91/353/EEC)*, Official Journal of the European Communities, No. L-192, July 1991, pp18-28.

[DEV-91] Devillers C, Burnard L D, Hockey S M & Gibeaux G, *Eurotra-6/3 Text Handling Design Study (Final Report)*, CEC, 1991.

[GRO-93a] Groenendijk M, P-E International, *XMFED User Guide*, CEC, 1993.

[GRO-93b] Groenendijk M, P-E International, *ALEP-0 Graphical User Interface*, CEC, 1993.

[IAI-91] IAI, CAP Gemini & SNI, *ET6/2 Software Environment Study Vol2 Outline Design Study (Final Report)*, CEC, 1991.

[MEY-93] Meylemans P & Simpkins N, *Towards a Portable Platform for Language Research and Engineering*, Thirteenth International Conference on Artificial Intelligence, Expert Systems and Natural Language, Volume 3, EC2, 1993, pp161-170.

[SIM-93a] Simpkins N K & Cruickshank G, P-E International, *ALEP-0 Virtual Machine*, CEC, 1993.

[SIM-93b] Simpkins N K & Cruickshank G, P-E International, *ALEP-0 Virtual Machine Extensions*, CEC, 1993.

[THE-93] Theofilidis A, *ET-9/1 Lingware Report Phase 2*, IAI, 1993.

(questionnaire not submitted)

2.3 CAT2

Randall Sharp
IAI
Martin-Luther-Straße 14
D-6600 Saarbrücken 3
Germany

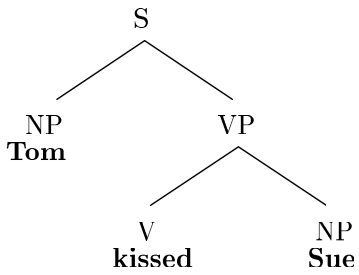
2.3.1 Overview of the CAT2 Formalism

The CAT2 formalism is used to describe (1) the grammar of a language defining the set of well-formed linguistic structures that belong to the language, and (2) a mapping relation between the linguistic structures of one language and those of another. To this end, the formalism consists of descriptive mechanisms for generating the linguistic structures and for translating from one structure to another. The former are called Generators, the latter Translators. (This terminology is taken from the original <C,A>,T specifications.)

Generators

Generators describe linguistic structures in terms of trees. We will assume here an intuitive understanding of the notion of a tree, i.e. that of a root (mother) node dominating zero or more subtrees (daughters). Each daughter has a unique mother node, except the topmost node, which has no mother node. Furthermore, we will assume that the daughters under a root are ordered, so we can speak of a left branch, right branch, middle branch, etc. Finally, a root with no daughters is a terminal node, a leaf in the tree.

An example of a typical tree structure is the following:



The tree above shows only the syntactic categories, but this is only one piece of linguistic information. Other properties, such as person, number, gender, tense, and many others, would be required to fully describe the actual properties pertaining to a given construction. As is now standard in modern computational linguistic theories these properties are described as **features**, i.e. attribute–value pairs; each node in the tree contains a set of such features, called a **feature bundle**. For example, the subject NP node above might be illustrated by the following feature bundle:

```
{ cat = np
  ortho = 'Tom'
  lex = tom
  agr = { per = 3
          num = sing
          gen = masc }
  case = nom }
```

Using trees whose nodes are feature bundles as representations of linguistic structure, we can now define them in the CAT2 formalism.

We define, first, a feature bundle FB as:

(1) FB ≡ { F+ }

where F+ is a list of one or more features F, enclosed in curly brackets.

A feature F is defined as:

(2) F ≡ <attribute> = <value>

where <attribute> is some atomic constant, a label, and <value> is either an atomic constant or a feature bundle. The former are called **simple features**, e.g. `cat`, `per`, and `case` above, and the latter **complex features**, e.g. `agr`.

An example of the above NP feature bundle would be written in CAT2 notation as:

```
{cat=np,ortho='Tom',lex=tom,agr={per=3,num=sing,gen=masc},case=nom}
```

Given this introduction to tree and feature structures, we now define a generator $\mathcal{G}$ for some language as a tuple:

(3) $\mathcal{G} \equiv < \mathcal{B}, \mathcal{F} >$

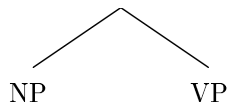
where $\mathcal{B}$ is a non-empty set of constructors, called **b-rules** ("building rules"), and $\mathcal{F}$ is a possibly empty set of well-formedness constraints, called **f-rules** ("feature rules") on the constructions produced by $\mathcal{B}$.

B-Rules The constructors, or b-rules, define partial tree structures, i.e. mother nodes and their immediate daughters. A constructor C, C $\in \mathcal{B}$, is defined as:

(4) C ≡ <rulename> = ROOT.BODY.

where <rulename> is a rule identifier, ROOT describes the root of the tree, and BODY is a list of the immediate daughters under the root. In the case of a terminal node, the body is the empty list; such constructors, called **atoms**, define extensionally the lexicon of the generator, i.e. those constituents which cannot be further decomposed into subconstituents. By restricting the tree description to the **immediate** constituents, we have the equivalent of a context-free grammar. That is, the structure shown in (**a**) corresponds to the context-free rewrite rule in (**b**), which is written in CAT2 notation as (**c**):

a.



b. S → NP VP

c. `{cat=s}.[{cat=np}, {cat=vp}].`

An example of an atom is given below for the verb **kissed**:

```
kiss1 = {cat=v,ortho=kissed,lex=kiss,tense=past}.[].
```

A more complete verb entry would include things such as argument structure, Aktionsart, aspect, inflection, possibly phonological information if relevant, etc. We will look at some of these in some detail in S2.

F-Rules F-rules operate on the partial trees generated by b-rules and are used to assign default feature values and enforce well-formedness of tree and feature structures. They have a form similar to b-rules, with the exception that they are not limited to describing just immediate daughters — they can map onto an arbitrarily deep tree structure.

F-rules come in three types: default, filter and strict. A default f-rule assigns feature values to a structure unless it already has different values, in which case the default rule does not apply. A filter f-rule also tries to assign values, and if it succeeds, the structure is deemed ill-formed and rejected from further analysis. The strict f-rule requires a structure to have or accept the given features; if the requirement cannot be fulfilled, the structure is rejected.

Default f-rules are most often used to assign features to the lexicon. For example, most English verbs cannot be treated as auxiliaries, i.e. they do not front in questions, they do not contract with **not**, etc. Rather than stating this fact in every verb entry, we can write the following default rule once to apply to all verbs:

```
default_aux = {cat=v,aux=no}.[].
```

The auxiliaries **have** and **be** and the modals would explicitly be marked with the feature `{aux=yes}` in the lexicon and therefore not be affected by the rule.

F-rules are used quite extensively in practice, since the set of phrase structure rules, i.e. b-rules, is rather minimal, given the implementation of X-bar theory as outlined in S2. Most well-formedness conditions, aside from pure structural well-formedness, are in fact controlled by way of f-rules.

Rule Application CAT2 belongs to the family of unification-based formalisms such as PATR (Shieber 1984). This means the basic operation is that of unification (Shieber 1986), both of tree and feature structures. Assume we have the feature bundle description in (**a**) and this is to be unified with the feature bundle in (**b**):

a. `{wh=no,agr={per=3,num=sing}}`
b. `{cat=n,ortho='Sue',agr={per=3,gen=fem},wh=X}`
c. `{cat=n,ortho='Sue',agr={per=3,gen=fem,num=sing},wh=no}`
d. `{cat=n,agr={per=3,num=plu}}`

The result of unification is the feature bundle in (**c**), where the variable X has been unified with the value `no`, and the value of `agr` is extended to include `{num=sing}`. (**a**) cannot unify with (**d**) because of conflicting values for `num`.

The notion of unification has been upgraded to that of **constraint satisfaction**, in which constraints on feature values are evaluated only when there is sufficient data to positively establish a value. The feature constraints provided in CAT2 are itemized below:

a.	positive constraint	`{case=nom}`
b.	negative constraint	`{case~=gen}`
c.	disjunctive constraint	`{case=(dat;acc)}`

The positive constraint assigns a value to a feature, or confirms its value if it is already present in the feature bundle under investigation. The negative constraint states what a given value is **not** permitted to unify with, and the disjunctive constraint states what values the given feature may unify with. In the process of unification, a feature may happen to not be assigned a positive value, in which case any negative or disjunctive constraints will be retained until a value is assigned by some other rule application, at which time the constraint(s) can be re-evaluated. Failure of the constraint to be satisfied at any time will cause the structure under evaluation to be rejected. Backtracking will take place in this case to some previous choice point, if any. In the current implementation of CAT2, all choice points will be exhaustively evaluated so that all possible paths in the grammar and lexicon are followed.

A further notational extension provides for optionality and alternation of constituent structures. We can define the contents of the body of a rule as a regular expression EXPR whose Backus-Naur Form notation is shown below:

a.	EXPR := FB	(basic feature bundle description)
b.	EXPR := ^EXPR	(optionality)
c.	EXPR := *EXPR	(zero or more expansions of EXPR)
d.	EXPR := +EXPR	(one or more expansions of EXPR)
e.	EXPR := (EXPR ; EXPR)	(alternation of expressions)
f.	EXPR := (EXPR , EXPR)	(sequence of expressions)

A somewhat artificial example is shown below for a noun phrase containing an optional determiner, zero or more adjective phrases, at least one noun, and optionally followed by either a prepositional phrase or a non-infinitive sentential phrase:

```
{cat=np}.[^{cat=det}, *{cat=ap}, +{cat=n}, ^ ({cat=pp};{cat=s,tense~=infin})].
```

The application of f-rules to a structure is carried out in the order they occur in the grammar. This ordered relation enables default rules to assign features to a structure, which subsequent filter and/or strict rules can verify. This adds a procedural element to an otherwise fully declarative grammar formalism, meaning the linguist must be cognizant of the correct order in which rules are

to be written. At the same time, the linguist has control over processing, and can, for example, order filter rules earlier in order to reject invalid structures before other rules are unnecessarily applied.

As to the parsing strategy, CAT2 employs a bottom-up chart parser with one symbol lookahead. It essentially implements the Earley algorithm, where the completion step has been generalized to include the scanning of terminal constituents (Kilbury 1985).

The reverse of parsing, synthesis, is carried out by the process of translation, discussed next.

Translators

Translators map one structure onto another by a recursive process of decomposition, transfer and recomposition. For example, a tree structure created by the parser is transformed into a new structure with possibly differing feature structures. The new structure may reflect another aspect of the language, e.g. a semantic or pragmatic representation, or it may be a representation appropriate to a target language. Schematically, we have the following situation where a source text T_s is translated to a target text T_t by transforming it through a series of intermediate structures L_i, where each L_i is defined by a generator G_i:

$$\begin{array}{ccccccccccccc} & & & {}_1T_2 & & {}_2T_3 & & {}_{n-2}T_{n-1} & & {}_{n-1}T_n & & & \\ & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & & \\ T_s & \Longrightarrow & L_1 & \Longrightarrow & L_2 & \Longrightarrow & \cdots & \Longrightarrow & L_{n-1} & \Longrightarrow & L_n & \Longrightarrow & T_t \\ & & \uparrow & & \uparrow & & & & \uparrow & & \uparrow & & \\ & & G_1 & & G_2 & & & & G_{n-1} & & G_n & & \end{array}$$

The parser uses G_1 to generate L_1 from T_s, and a series of translators ${}_iT_{i+1}$ transform this into $L_2, \ldots, L_n$. A simple yield function produces the target string T_t, selecting out the orthographic features in L_n.

A translator $\mathcal{T}$ is defined by the tuple:

(5) $\mathcal{T} \equiv < \mathcal{TB}, \mathcal{TF} >$

where $\mathcal{TB}$ is a non-empty set of structural translation rules ("t-rules"), and $\mathcal{TF}$ is a possibly empty set of feature translation rules ("tf-rules").

Syntactically, the t-rule has the form:

(6) <rulename> = ROOT.BODY ⇒ ROOT.BODY.

where the lefthand side of the "⇒" symbol is a partial description of a source structure and the righthand side is a partial description of a target object to be constructed. Unlike generator b-rules, the body of either side of the t-rule may specify a tree to an arbitrary depth of detail. Consequently, non-atoms may be mapped to atoms and vice versa. Semantically, the t-rule states: if the lefthand side unifies with the source object, then a target object unifying with the righthand side is created, provided that such object is licensed by the generator for the target level. An object is licensed in this sense if at least one rule in the generator unifies with the object, and each of the object's daughters unifies with at least one rule. Again, unification is the basic operation, including negation and disjunction as constraints.

The simplest t-rule expresses the relation holding between two atomic objects, as found, for example, in a French-to-English translator:

```
atomic_t_rule = {lex=aller}.[] => {lex=go}.[].
```

More complex t-rules are defined recursively over subobjects, permitting a compositional breakdown of a source object and construction of a target object, controlled by explicitly marking the subobjects to be recursively translated. For example, the following t-rule relates a constituent structure of a sentence to a lowered-governor dependency structure, in which the subject NP, the governing verb, and any following constituents are selected on the lefthand side for recursive translation, and repositioned in the target object such that (the translation of) the verb appears first under the root node, followed by (the translations of) the subject and the remaining constituents:

```
t = {cat=s}.[ 1:{cat=np}, {cat=vp}.[ 2:{cat=v}, *3 ]] => _.[ 2, 1, 3 ]
```

Besides constituent reordering, the VP node disappears, since it is unmarked on the lefthand side, illustrating how nodes are "deleted" from a structure; conversely, nodes are inserted into a structure by explicitly describing them on the righthand side of the t-rule. In synthesis we would have the reverse of the above rule, inserting a VP node in the target structure.

Translator f-rules are similar to generator f-rules, in that they do not affect structure, but rather affect the feature content of objects. The test for applicability is unification of the entire lefthand side of the rule with the source object. If applicable, the righthand side is unified with the target object; success or failure of unification has the same effect as for generator f-rules, and depends on whether the f-rule is typed default, strict, or filter. As an example, the following strict translator f-rule copies over the complex agreement feature:

```
tf = {agr=A}.[*] => {agr=A}.[*].
```

(questionnaire not submitted)

2.4 Constraint Categorial Grammars

Luis Damas, Nelma Moreira
Universidade do Porto, Rua do Campo Alegre 823, 4000 Porto, Portugal
Giovanni B. Varile
CEC , 2920 Luxembourg, Luxembourg

2.4.1 Introduction

Unification based formalisms show a clear inability to deal in a natural way with phenomena such as the semantics of coordination. As a matter of fact although unification can be used to implement a weak form of β-reduction it seems that this kind of phenomena is better handled by using some form of λ-calculus. One possibilty, which is at the heart of λ-prolog, is to extend both the notion of term, to include λ-abstraction and application, and the definition of unification to deal with λ-terms. For this extension to be technically sound it is neccessary to require λ-terms to be well typed.

On the other hand, it turns out that if instead of using terms we use complex feature descriptions (where conjunction replaces unification), we still can follow the same plan to produce a higher-order calculus of feature descriptions.

CCLG is a simple formalism, based on categorial grammars, designed to test the practical feasibility of such an approach.

The main reason for selecting a categorial framework for this experiment was that, due to the simplicity of the categorial framework, it allowed us to concentrate on the constraints calculus itself. Another reason was the close historical relationship between categorial grammars and semantic formalisms incorporating λ-abstraction.

CCLG extends categorial grammar by associating not only a category but also a higher-order feature description with each well-formed part of speech. The type of these feature descriptions are determined by the associated category. Note also that a derivation leading to an unsatisfiable feature description is illegal.

When compared with other formalisms one of the main distinguishing features of CCLG is the fact that it computes partial descriptions of feature structures and not the feature structures themselves.

In the next sections we briefly describe this formalism.

2.4.2 Feature Description Calculus

The feature description calculus $\Lambda_{\mathcal{FD}}$ at the heart of CCLG is inspired both on the λ-calculus and on Feature Logics. For technical reasons, namely that we want to insure the existence of normal forms, it is a typed calculus. Our base types are **bool** for truth values and **fs** for feature structures. Our types are described by

$$\tau ::= \mathbf{bool} \mid \mathbf{fs} \to \tau \mid \tau \to \tau'$$

Note that we exclude **fs** as the type of any feature description. This reflects our commitment to compute partial descriptions of feature structures rather than feature structures.

Now assume we are given a set of **atoms** a, b, ..., a set of **feature** symbols f, g, ..., a set of **variables for feature structure** x, y, ..., and, for each type τ, a set of **variables of type** τ x_τ, y_τ, ... Then the set of **feature descriptions** of type τ is described by

$$\begin{array}{lcl} e_\tau & ::= & \mathbf{true} \mid \mathbf{false} \mid x_\tau \mid e_\tau \wedge e_\tau \mid e_\tau \vee e_\tau \mid \neg e_\tau \mid e_{\mathbf{fs}\to\tau} x.p \mid e_{\tau'\to\tau} e_{\tau'} \\ e_{\mathbf{bool}} & ::= & t.p \doteq s \\ e_{\mathbf{fs}\to\tau} & ::= & \lambda x.e_\tau \\ e_{\tau'\to\tau} & ::= & \lambda x_{\tau'}.e_\tau \end{array}$$

where s and t denote either atoms or feature structure variables, and p is a, possibly empty, sequence of feature symbols denoting a path in a feature structure.

Note the languages thus defined includes both feature logics and a typed λ calculus. We import from both theories such notions as substitution, free and bounded occurrences of variables, α and β reductions and normal form.

To define a semantics for the calculus of feature descriptions we adopt the standard model $\mathcal{RT}$ of rational trees for feature structures and we associate with each type τ a semantic domain D_τ as follows

$$\begin{array}{lcl} D_{\mathbf{bool}} & = & \{0,1\} \\ D_{\mathbf{fs}_\tau} & = & \mathcal{RT} \rightarrow D_\tau \\ D_{\tau' \rightarrow \tau} & = & D_{\tau'} \rightarrow D_\tau \end{array}$$

From this point on a semantics for feature descriptions is defined in the same way as for feature logics and the typed λ-calculus by noting that the standard boolean operations can be extended to all the semantic domains involved in a component-wise fashion, e.g.

$$(\lambda x.e) \vee (\lambda x.e') =_{def} (\lambda x.e \vee e').$$

Similarly, for each type τ, **true** and **false** denote the obvious elements of $\mathcal{D}_\tau$.

An important property of the feature description calculus is the existence of normal form under β-reduction which is a simple consequence of well-typeness. Another important property is that for any closed feature description of type τ we can decide if it is equivalent to **false**. This last property is essentially an extention of the the satisfiability problem for a complete axiomatization of feature logics. For this reason we will say that a feature description of type τ is satisfiable iff its semantics is not that of **false**.

Our implementation of the feature description calculus is based on the reduction to normal form followed by the techniques used in CLG for resolving complex feature constraints.

2.4.3 Categorial Grammar

We use a basic (rigid) categorial grammar, consisting of a set of categories, a lexicon which assigns categories to words and a calculus which determines the set of admissible category combinations. Given a set of basic categories $\mathcal{C}at_0$ we define recursively the set of categories $\mathcal{C}at$ by: the elements of $\mathcal{C}at_0$ are categories; if A and B are categories then A/B and $A\backslash B$ are categories. The two combination rules are **left-application** (app$\backslash$) and **right-application** (app/):

$$(app/)\ A/B \ + \ B \ \rightarrow \ A$$

$$(app\backslash)\ B \ + \ A\backslash B \ \rightarrow \ A$$

The meaning of the resulting expression (A) is the application of the meaning of the functor expression (A/B or $A\backslash B$) to that of the argument expression (B).

Some unary (lexical) rules (lifting, division, etc) were added to provide a flexible *CG* which can cope with discontinuity and other linguistic phenomena. Semantically these rules allow functional abstraction over displaced or missing elements.

2.4.4 Constraint Categorial Grammar

A Constraint Categorial Grammar is a tuple $\langle Cat_0, \Upsilon, Lexicon, Rules \rangle$ where

1. Cat_0 is a set of base categories
2. Υ is a map which associates with each category C a type $\Upsilon(C)$ and satisfies
$$\Upsilon(A/B) = \Upsilon(A\backslash B) = \Upsilon(B) \rightarrow \Upsilon(A)$$
3. *Lexicon* is a set of triples $\langle w, A, c \rangle$, where w is a word, A a category and c is a feature description of type $\Upsilon(A)$

4. *Rules* is the set of inference rules to combine pairs $A - c$ of syntactic categories and semantic representation.

The inference rules used in the current grammars are:

$$(app/)\ \frac{A/B - c_f \quad B - c_b}{A - (c_f c_b)} \quad \text{if } c_f c_b \text{ is satisfiable}$$

$$(app\backslash)\ \frac{B - c_b \quad B\backslash A - c_f}{A - (c_f c_b)} \quad \text{if } c_f c_b \text{ is satisfiable}$$

2.4.5 A sample grammar

In this section we give a very small fragment of an English grammar. The `let` constructor allows the use of macros in the writting of the lexicon. The inference rules are build in the grammar processor. Note also that although the feature descriptions used in the grammar are untyped a type inference algorithm is used to infer types for each expression.

```
Base_Categories         % Define the set of base categories
     s = fs -> bool,    % and their types
     vi = fs -> fs -> bool,
     np = s/vi,
     vt = vi/np,
     n = fs -> bool,
     det = np/n,
     xnp = (s/np)/(vi/np),
     xthat=(n\s)\np;

transformation          % define a type raising rule
     np = (s/np)/(vi/np) :  \S \Vt \C. S (Vt C);

%%%%%%%% some useful abreviations

let 3RD_SG = \X. X.pers=p3 & X.nb=sg;

let NOT_3RD_SG = \X. X.pers\=p3 | X.nb\=sg;

let PN(W) = \P.\s.  s.quant=exists_one & s.arg.reln=naming &
     s.arg.arg1=W & 3RD_SG(s.arg) & P s.arg s.pred ;

let CN(W,AGR) = \s.  s.reln=W & AGR s;

let DET(Q,AGR) = \N \P \s.  s.quant=Q & AGR s.arg &
     N s.arg & P s.arg s.pred;

let VI(W,AGR) = \x\p p.reln=W & p.arg1=x & AGR x;

let VT(W,AGR) = \C. \x\p C (\y \q.  q.reln =W & q.arg2=y & q.arg1=x) p;

%%%%%%%%%%%%%% lexicon

lex a, det, DET(exists_one,3RD_SG);

lex book, n, CN(book,3RD_SG);

lex john, np, PN(john);
lex mary, np, PN(mary);
```

```
lex died, vi, VI(die,3RD_SG);

lex loves, vt, VT(love,3RD_SG);

% coordination

lex and, s\(s/s), \S1\S2\s.  s.type=coord & S1 s.arg1 & S2 s.arg2;

lex and, np\((vt\vi)/np),
     \NP1\NP2\VT. \subj\s.  s.type=coord &
     VT NP1 subj s.arg1 & VT NP2 subj s.arg2;

lex and, vi\(vi/vi), \V1\V2.  \subj.\s.
     s.type=coord & V1 subj s.arg1 & V2 subj s.arg2;
```

2.4.6 Conclusions

The current CCLG implementation shows the practical feasibilty of using higher order feature structure descriptions as semantic representations. This reflects the fact that the complexity of the satisfiability problem for higher order feature descriptions is essentially the same as for feature logics.

We should also point out that the good performance of the system results in part from its hybrid nature where a categorial grammar with atomic base categories is used to guide parsing.

System Name: CC(L)G Categorial Constraint Grammar
Designed and Implemented by: L. Damas

GENERAL DESCRIPTION I		
Inference Engine		
	unique engine vs. dedicated modules	Categorial grammar parser and Constraint solver
	non-montonic devices	
	control facilities	
	parser/generator?	parser only
	others	
Data Types		
	arity (fixed?)	Feature structure constraints
	cyclic structures	yes
	lists/sets	
	functions/relations	
	others	high order feature structure descriptions
Interaction FS $\Longleftrightarrow$ Types		
	type unification	
	type expansion	
	at definition/ compile time	compile time "P"
	at run time: (delayed/partial/ recursive)	
	others (templates...)	

GENERAL DESCRIPTION II	
Interfaces to	
morphology	
semantics/ knowledge repr.	
Implementational Issues	
programming lang.	Prolog and C
machine	Sparc, Mips, ...
others (O/S, graphics...)	
Applications	
grammar theories?	X
educational vs. commercial system	
used in projects/ other systems?	
Grammar coded	
size	Toy size
language	English
Tools	
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	no
non-destructive	yes
disjunction:	
atoms only	
full (DNF)	yes but not DNF in general
distributed	
others	
negation	
atoms only	
negated corefs	
full	yes over feature constraints
others	
implication	
via negation	yes
others	
Additional Operations	
subsumption	
functional uncertainty	
others	α-abstraction and β-reduction
Tools	
Comments	

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	
disjunction	
negation	
others	
Type Definitions	
via feature structures	
via appropriateness conditions	
recursive?	
others	
Additional Operations	
type inference/ classification	
GLB/LUB type subsumption	
others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	
Tools	
Comments	

2.5 CLARE (includes Core Language Engine)

SRI International, Cambridge

Introduction

The Core Language Engine is a general purpose, wide coverage, unification-based system for the analysis and generation of sentences. The following slides give a high level overview of the architecture of the system and some of the applications it has been used for. These applications include database query, transfer-based translation, and spoken language understanding.

A more detailed description of the CLE, including a full specification of the formalism and descriptions of the associated processing algorithms can be found in the book edited by Hiyan Alshawi: 'The Core Language Engine', MIT Press, 1992.

Stephen Pulman,

July 1992.

THE CORE LANGUAGE ENGINE

SRI International
Cambridge Computer Science
Research Centre

* Language research at SRI-CCSRC
* Design themes
* How the themes are realised
* Performance evaluation

Key to some acronyms:

CLE: Core Language Engine
CLARE: CLE with reasoning and cooperative response
BCI: Bilingual Conversation Interpreter
VEX: Vocabulary EXpander
QLF: Quasi Logical Form
RQLF: Resolved QLF
TRL: Target reasoning language
SLT: Spoken Language Translation
SRI: SRI

BACKGROUND

* SRI International: 3,000 people worldwide, scientific & other consultancy

* Cambridge laboratory founded 1986. Now 5 natural language researchers, (4 hardware/software verification): NL: Stephen Pulman, David Carter, Manny Rayner, Ian Lewin, Dick Crouch.

* Earlier contributors: Bob Moore, Fernando Pereira, Doug Moran, Jan van Eijck, Hiyan Alshawi, Arnold Smith

DESIGN THEMES

1 Modular staged architecture

2 Well-defined intermediate representations

3 Local ambiguity packing

4 Declarative rules applied by (Prolog) unification

5 Compilation of rules and entries

6 Balance of user intervention and system preferences in making choices

7 Customization for applications

(1) MODULAR STAGED ARCHITECTURE

Advantages:

* Easier system development by a team

* Modules can be evaluated and debugged separately

* Modules can be reused in different combinations

Pitfalls to be avoided:

* Arbitrary boundaries between modules

* Risk of inefficiency, e.g. during parsing

(2) INTERMEDIATE REPRESENTATIONS

* These are well-defined and explicit at all levels. Some examples:

  Target Reasoning Language (TRL).
  -- First order logic augmented with limited lambda abstraction, sets, cardinality, time relations...to support reasoning.

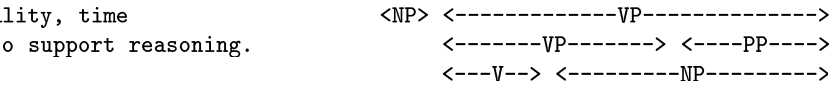
  Quasi Logical Form (QLF).
  -- Limit of compositionality
  -- TRL constructs augmented with generalized quantifiers; vague terms and formulas; category information.

  Mediating between QLF and TRL:
  Resolved Quasi Logical Form (RQLF).
  -- QLF with more variables instantiated to allow translation to TRL.
  -- conversion to TRL is purely syntactic & loses information.

(3) LOCAL AMBIGUITY PACKING

* Allows staged architecture to be efficient. Used up to semantic analysis stage.

"Wren designed a library in Cambridge."

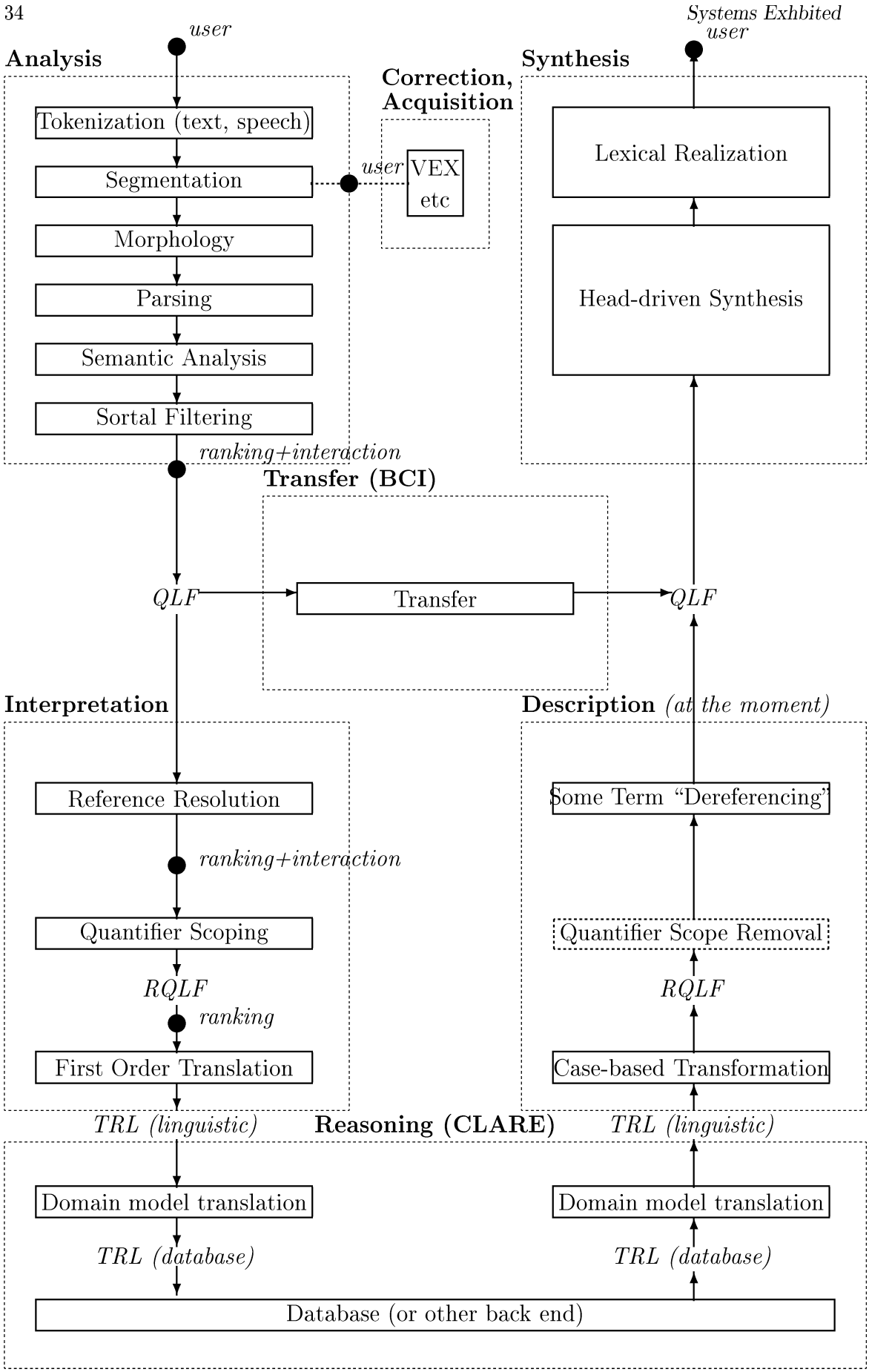


* No full parse trees are produced (except for inspection).

* Can support numerical weighting (iterative deepening, A* search).

* Avoids the need for true disjunction in rules.

* Lattice also used in tokenization (typos, slashes, hyphens; speech)


Systems Exhbited
user
user
Analysis
Correction,
Acquisition
Synthesis
Tokenization (text, speech)
Segmentation
user
VEX
etc
Morphology
Parsing
Semantic Analysis
Sortal Filtering
Lexical Realization
Head-driven Synthesis
ranking+interaction
Transfer (BCI)
QLF
Transfer
QLF
Interpretation
Description (at the moment)
Reference Resolution
ranking+interaction
Quantifier Scoping
RQLF
ranking
First Order Translation
Some Term "Dereferencing"
Quantifier Scope Removal
RQLF
Case-based Transformation
TRL (linguistic)
Reasoning (CLARE)
TRL (linguistic)
Domain model translation
TRL (database)
Domain model translation
TRL (database)
Database (or other back end)

(4) DECLARATIVE RULES AND UNIFICATION

* Reversability; monotonicity; debuggability; efficiency (Prolog).

* Major category symbols, complex syntactic and semantic features

* In semantics, QLFs built up like feature values

* Rule schemas: verb complement list in VP rule instantiated from verb entry

* Limited disjunction for e.g. agreement (compiled away)

* All rules are declarative: segmentation, morphology, syntax, semantics, sorts, reference, scoping, domain translation...

(5) COMPILATION OF RULES AND ENTRIES

* USERS need readability, flexibility, irredundancy. SYSTEM needs explicitness and fixed formats.

* Macros expanded out first. (Most of the lexicon is effectively macro calls).

* feature=value compiles to positional notation. Defaults used.

* Disjunction -> boolean vectors.

* Rules are compiled in same way for analysis & generation, but indexed differently.

* Sortal class lists compile to partial hierarchies.

(6) INTERACTION AND PREFERENCES

* Wide coverage -> many analyses:
  -- system cannot reliably choose
  -- user cannot cope with too much and may not know linguistics
  -- system/user balance changes as technology improves

* Numerical preferences on:
  -- word senses (from corpus)
  -- rule choices (NOT rules)
  -- word occurences (acoustic)
  -- syntactic/semantic properties
  -- scopes, references, etc

* System ORDERS readings numerically then ASKS user: paraphrases, bracketings. User can specify constraints (at QLF level).

* This happens after semantic analysis, reference resolution, (scoping), transfer.

INTERACTION IN ACTION

```
>> I met the man in the bank.
 ...
6 well-sorted semantic analyses.

Complete sentence with bracketing:

  "{I} met {{the man} in {the bank}}."

Word senses (unordered):

   meet: encounter (rather than "be
                    adjacent")
   man: male person
   bank: company (rather than "building"
                  or "edge")

Confirm this analysis? (y/n/c/p/?):
```

(7) CUSTOMIZATION FOR APPLICATIONS

* VEX (Vocabulary EXpander) allows non-linguist domain experts to add lexical entries for words and phrases. Based on "paradigms". Grammar can change under its feet.

* Sortal hierarchy can be extended (without rewriting lexicon).

* User-defined reference rules (e.g. for mouse pointing).

* Declarative domain model specifying logical equivalences between linguistic and domain predicates.

LINGUISTIC COVERAGE

* Major clause types: declaratives, imperatives, wh- and yes-no questions, relatives, passives, clefts, there-clauses.

* Verb phrases: complement subcategorization, control verbs, verb particles, auxiliaries, tense operators, some adverbials.

* Noun phrases: prenominal and postnominal modifiers, lexical and phrasal quantifiers/specifiers.

* Coordination: conjunctions and disjunctions of a wide class of noun phrases, verb phrases, and clauses; adjectival, nominal, and adverbial comparatives.

* Anaphoric expressions: definite descriptions, reflexive and nonreflexive pronouns, bound variable anaphora, implicit relations.

* Ellipsis: 'one'-anaphora, intrasentential and intersentential verb phrase ellipsis, follow-on questions.

* Morphology: inflectional morphology, simple productive cases of derivational morphology, special form tokens.

* Core lexicon: 1600 function words and content word stems, 2300 senses with associated selectional restrictions. External lexicon interface available.

EVALUATING COVERAGE

* LOB corpus: "legible" characters, limited length, core vocabulary or external lexicon. (Core vocabulary words are harder!).

* Can only evaluate as far as QLF level (no discourse context available).

* 'Good' means that the QLF ranked highest by the preferences is one that is correct in some reasonably plausible context. This is not the same as correctness in an application, which is often easier to achieve.

* NB this measure ignores the fact that lower ranked QLFs may be 'good'.

Measuring progress:
QLF rates for sentences up to 10 words over the last few years

Date	Unlim. vocab %Any	%Good
May 89	15	12
Oct 90	39	22
Oct 91	54	33
Oct 92	71	44

Comparison of random corpus sentences with those for which the system has been customised.
(ATIS = Air Travel Information Service corpus from US DARPA program)

October 1992, QLF Accuracy:

up to 15 words	ATIS	LOB
Got QLFs:	90%	57%
1st QLF good:	80%	62%
Accuracy:	72%	35%

System Name: CLARE (includes Core Language Engine)
Designed and Implemented by: SRI International, Cambridge

GENERAL DESCRIPTION I		
Inference Engine		
	unique engine vs. dedicated modules	dedicated modules
	non-montonic devices	no
	control facilities	in generation, via definition of "head"
	parser/generator?	both
	others	preference mechanisms, quantifier scoping, reference resolution
Data Types		
	arity (fixed?)	fixed
	cyclic structures	possible but not encouraged
	lists/sets	lists, no sets
	functions/relations	not in syntax
	others	terms etc.
Interaction FS $\Longleftrightarrow$ Types		
	type unification	via term unification
	type expansion	
	at definition/ compile time	compile time
	at run time: (delayed/partial/ recursive)	
	others (templates...)	parameterised macros, expanded at compile time

GENERAL DESCRIPTION II	
Interfaces to	
morphology	yes
semantics/ knowledge repr.	yes
Implementational Issues	
programming lang.	Quintus Prolog, Sicstus Prolog (not fully supported)
machine	any UNIX machine
others (O/S, graphics...)	ORACLE database
Applications	
grammar theories?	more in semantics than in syntax
educational vs. commercial system	used for both purposes
used in projects/ other systems?	yes (translation, database query, text processing, spoken language understanding)
Grammar coded	
size	1. ca. 150 rules, wide coverage 2. ca. 100 rules 3. ca. 30 rules
language	1. English, Swedish 2. Japanese 3. German
Tools	Stepper/debugger version control for grammar development
Comments	fully reversible system efficient, relatively wide coverage (as measured on red. corpus)

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	uses Prolog unification
non-destructive	
disjunction:	
atoms only	yes
full (DNF)	no
distributed	no
others	no
negation	
atoms only	yes
negated corefs	
full	
others	
implication	
via negation	yes
others	
Additional Operations	
subsumption	via term subsumption
functional uncertainty	no
others	none
Tools	compilation of feature structures into terms
Comments	deliberately conservative formalism for efficiency

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	flat type system
disjunction	
negation	
others	
Type Definitions	
via feature structures	yes
via appropriateness conditions	
recursive?	no
others	
Additional Operations	
type inference/ classification	no
GLB/LUB type subsumption	no
others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	
Tools	
Comments	

2.6 CLG: Constraint Logic Grammar

Luís Damas, Nelma Moreira
{luis,nam}@ncc.up.pt
LIACC, Univ. do Porto, Rua do Campo Alegre 823, 4100 Porto, Portugal

The goal of the **CLG** project was to see to what extent and with what benefits the techniques of Constraint Logic Programming [8] could be adapted for use in NLP.

The potential benefits expected from such an undertaking were:

- expressivity;
- efficiency;
- soundness.

These goals were met to a large extent as reported in [1, 2, 3, 4, 5, 6, 7].

Other benefits expected from the **CLG** approach are modularity and scalability, due in particular to the hybrid architecture inherited from **CLP**.

Rationale

The **CLG** framework assumes that any grammar is a particular first order theory with equality admitting complete models.

(Constraint) Logic Programming is an ideal paradigm for such a framework in that it supports a direct mapping between grammars (the first order logic theories of linguistic descriptions) and the first order theory of their implementation and at the same time provide a formally sound and efficient computational scheme.

The **CLG** programme is compatible with the unification grammar tradition and constitutes a simple framework for extending the notion of unification to complex constraint resolution. At the same time a high degree of declarativeness is achieved by avoiding any reference to an **operation** like **unification**.

In **CLG** substantial attention has been paid to a detailed formalization of the of the underlying processing model.

The main reasons derives from the fact that restricting the formal analysis to the **static** properties of formalisms does not do justice to the **computational** complexity of modern linguistic frameworks. A finer grained analysis of the formal **and computational** properties of formalisms than decidability, formal complexity and model theoretic properties, sheds a different light on the problem motivating choices which would otherwise appear to be arbitrary.

While sound denotational semantics and appropriate formal complexity characteristics are necessary conditions to be met by linguistic formalisms, they are not sufficient.

Rather, it is necessary to provide sound and adequate formal processing schemes for such formalisms, lacking which the main challenges facing modern grammatical formalism design are not addressed.

Taking this point further, we claim that the theory of a grammar formalism and its formal processing model constitute a homogeneous and integrated whole and that the practice of relegating processing issues to low level implementation decisions had, and has, negative consequences, not least preventing the right questions to be addressed.

The deductive process by which a fact is proven or an object computed must be the subject of theoretical inquiry just as, and together with, the fact or object and their descriptions.

The analogy with logic programming is paradigmatic: defining the syntax and (static) semantics of a logic programming scheme is an essential first step. But it also constitutes a relatively trivial task compared with definition of a formal processing scheme with the necessary computational characteristics.

Another aspect of central importance to **CLG** are the circumstances under which one can ensure a simple and natural relation between grammars with complex constraint expression and

the logic programming paradigm, in particular a version of Constraint Logic Programming over the domain of rational trees.

The details of the **CLG** project relating to these and other aspects are described in [1, 2, 3, 4, 5, 6, 7].

Bibliography

[1] Luis Damas and Giovanni B. Varile. CLG: A grammar formalism based on constraint resolution. In E.M.Morgado and J.P.Martins, editors, **EPIA 89**, volume 390 of **Lecture Notes in Artificial Intelligence**, pages 175–186. Springer Verlag, 1989.

[2] Sergio Balari, Luis Damas, Nelma Moreira and Giovanni B. Varile. CLG(n): Constraint Logic Grammars. In **Proceedings of the 13th International Conference on Computational Linguistics (COLING)**, Vol. 3, pages 7–12, Helsinki, Finland, 1990.

[3] Luis Damas, Nelma Moreira, and Giovanni B. Varile. The formal and processing models of CLG. In **Fifth Conference of the European Chapter of the Association for Computational Linguistics**, pages 173–178, Berlin, 1991.

[4] Luis Damas and Giovanni B. Varile. On the satisfiability of complex constraints. In **Proceedings of the 14th International Conference on Computational Linguistics (COLING)**, pages 108–112, Nantes, France, 19 92.

[5] Luis Damas, Nelma Moreira and Sabine Broda. Resolution of constraints in algebras of Rational Trees. In Miguel Filgueiras and Luís Damas , editors, **EPIA 93**, volume 727 of **Lecture Notes in Artificial Intelligence**, pages 61–76. Springer Verlag, 1993.

[6] Luis Damas, Nelma Moreira, and Giovanni B. Varile. The formal and computational theory of Constraint Logic Grammars. In C.J. Rupp, M. Rosner and R. Johnson, editors, **Constraints, Language and Computation**, Academic Press, 1994.

[7] Luís Damas, Nelma Moreira, and Giovanni B. Varile. Constraint Categorial Grammars Submitted to COLING94 [1].

[8] Jaffar, J., J-L. Lassez, 1987. Constraint Logic Programming. In: Symposium on Principles of Programming Languages, Munich.

[1] This is an extended version of the **CCLG** description in section 2.4 of the present volume.

System Name: CLG
Designed and Implemented by:

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	2, equation solver and non-equational constraint solver
non-montonic devices	no
control facilities	no
parser/generator?	depending on versions: only parser, parser & generator
others	
Data Types	
arity (fixed?)	fixed
cyclic structures	yes
lists/sets	yes/member-type with "discharge"
functions/relations	no/yes
others	–
Interaction FS $\Longleftrightarrow$ Types	
type unification	yes
type expansion	
at definition/ compile time	compile-time
at run time: (delayed/partial/ recursive)	
others (templates...)	templates (parametrized)

GENERAL DESCRIPTION II	
Interfaces to	
morphology	no
semantics/ knowledge repr.	no
Implementational Issues	
programming lang.	Prolog
machine	Sun Microsystem 3—Sparc, Dec station, Mac
others (O/S, graphics...)	UNIX, X-WINDOWS, Finder
Applications	
grammar theories?	augmented CFG; HPSG
educational vs. commercial system	no
used in projects/ other systems?	
Grammar coded	
size	various sizes, up to 100 KB
language	various: EN, DA, DE, CA, PT
Tools	debuggers, displayer
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	yes
non-destructive	no
disjunction:	
atoms only	
full (DNF)	full, not DNF
distributed	
others	–
negation	
atoms only	
negated corefs	
full	full
others	–
implication	
via negation	
others	
Additional Operations	
subsumption	no
functional uncertainty	no
others	
Tools	
Comments	

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	single inheritance
disjunction	yes
negation	yes
others	–
Type Definitions	
via feature structures	yes
via appropriateness conditions	yes
recursive?	yes
others	–
Additional Operations	
type inference/ classification	type expansion
GLB/LUB type subsumption	LUB
others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	tree
Tools	
Comments	

2.7 Comprehensive Unification Formalism (CUF)

Jochen Dörre, Michael Dorna
Institut für maschinelle Sprachverarbeitung
Universität Stuttgart, Germany

CUF is a theory-neutral universal grammar formalism like PATR-II which has been developed in the ESPRIT-Project DYANA (BRA 3175 and 6852). It is based on defining feature structures and relations over these as encodings of linguistic principles and data. However, it is radically more expressive than conventional grammar formalisms, since it allows the definition of arbitrary recursive relational dependencies without tying recursion to phrase structure rules. Hence, CUF provides the basis for highly integrated processing of linguistic descriptions of different linguistic research areas. A system implementing this formalism in PROLOG and C is freely available from our institute.

The language of CUF uses a syntax especially well suited for a direct description of feature structures similar to Kasper/Rounds logic (feature-matrix notation) combined with the possibility of stating definite clauses over feature terms. Moreover, feature structures are typed, with the types possibly being ordered in a hierarchy. The CUF type discipline allows for an axiomatic statement of global restrictions on the structures in which the program is to be interpreted providing enough redundancy in the descriptions to detect mistakes without burdening the grammar writer with tedious repetitions.

CUF does not predefine any grammar rule formats like PATR's contextfree-based rules or GPSG's ID/LP rules. Instead, the grammar writer is free to define her own rule formats or even grammar architecture. For instance, an architecture based on principles and rules can straightforwardly be implemented.

Fig. 2.4 presents an overview of all kinds of language constructs that can be used to compose a CUF program, including its control part.

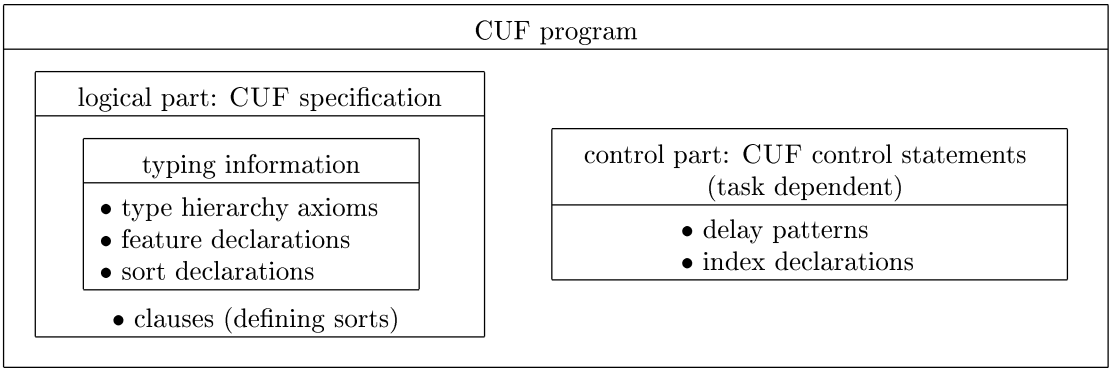


Figure 2.4: Parts of a CUF Program

CUF is an instance of constraint-logic programming (CLP) of the very general Höhfeld/Smolka scheme [HS88]. This provides us not only with a sound and complete proof procedure, but also equips us with the right paradigm to attack the efficiency problems associated with highly modular specifications, as for instance proposed by GB theory. For a more complete description of the CUF language, please refer to [DD93].

The current CUF system (Version 2.28) consists of a compiler, an runtime evaluator and an ASCII and a graphical user interface (GUI)[1] with several development tools like debugger, data base inspector, and feature structure browser. The implementation runs under Quintus and SICStus PROLOG under UNIX and X11.

The incremental compiler is used to translate the CUF desriptions into an interpretable format. Type checking and inference is used to eliminate errors very early in the development phase of a

[1]Currently, the GUI is still under development and not delivered yet. However, the ASCII interface provides main functionalities of the GUI.

description. The most distinguishing features of CUF's type system are:

- type interdependencies can be stated in full propositional logic, allowing to state all kinds of type hierarchies
- features may be fully polymorphic (no restrictions on multiple feature declarations)
- complete type checking during compilation
- runtime type checking is reduced to a minimum

CUF makes a clear distinction between the purely declarative logical specification and the control statements which are used to guide the proof procedure without compromising the logical semantics of the specification. The runtime evaluator is an SLD-resolution engine whose selection strategy can be customized by the user. By default the strategy selects deterministicly expandable literals first, or else the leftmost (nondeterministic) literal. By use of delay statements the user can change this behaviour. Another type of control statement is the declaration of predicates for which the system should build an index.

The system CUF is freely available. Just fetch it via anonymous ftp from

ftp.ims.uni-stuttgart.de:/pub/cuf

or write to cuf-request@ims.uni-stuttgart.de or to:

Jochen Dörre
Institut für maschinelle Sprachverarbeitung
Universität Stuttgart
Azenbergstr. 12
D-70174 Stuttgart, GERMANY

email: Jochen.Doerre@ims.uni-stuttgart.de

Bibliography

[DD93] Jochen Dörre and Michael Dorna. CUF — a formalism for linguistic knowledge representation. In Jochen Dörre, editor, **Computational Aspects of Constraint-Based Linguistic Description I, DYANA-2 deliverable R1.2.A**. ESPRIT, Basic Research Project 6852, July 1993.

[HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, October 1988.

System Name: CUF System
Designed and Implemented by: Jochen Dörre, Michael Dorna

GENERAL DESCRIPTION I		
Inference Engine		
	unique engine vs. dedicated modules	Modules for Feature Constraint Solving and Propositional Constraint Solving
	non-montonic devices	0
	control facilities	DELAY declarations and INDEX declarations
	parser/generator?	System is good for both; however dedicated control statements needed
	others	system is a universal deduction system for definite clauses over 'primitively' typed feature constraints
Data Types		
	arity (fixed?)	arity of types can be deduced from hierarchy
	cyclic structures	currently not supported, planned
	lists/sets	built in/definable
	functions/relations	functional or relational constraints are supported
	others	atoms, strings and list are builtin. We differentiate between primitive types (only propositionally definable) and general predicates
Interaction FS $\Longleftrightarrow$ Types		
	type unification	unification takes care of primitive types (see above)
	type expansion	
	at definition/ compile time	partial evaluation (can be switched off), compile time
	at run time: (delayed/partial/ recursive)	deterministic closure over goals, then nondet. choose first non-delayed goal, search by backtrack (delayed, partial, recursive)
	others (templates...)	enhanced Earley engine, which can be parametrized by the goals to store

GENERAL DESCRIPTION II		
Interfaces to		
	morphology	interface not needed full integration possible
	semantics/ knowledge repr.	see above
Implementational Issues		
	programming lang.	Quintus+C
	machine	any
	others (O/S, graphics...)	UNIX for G.U.I.: X11
Applications		
	grammar theories?	any
	educational vs. commercial system	public domain
	used in projects/ other systems?	used in DYANA, SFB 340, Uni Bielefeld, Uni Tübingen
Grammar coded		
	size	basic fragments of German, English/declarative phonology of German, small experimental grammars for hard linguistic problems
	language	
Tools		• Interactive Proof Tree Stepper for Debugging with Retry Skip, Creep and Clause-Selection Option • Browser for Result-Feature-Structures and Argument Bindings
Comments		

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	
non-destructive	X
disjunction:	
atoms only	
full (DNF)	
distributed	
others	delayed (compiled out); disjunctions between primitive types are handled by constraint solver
negation	
atoms only	
negated corefs	
full	X
others	
implication	
via negation	X
others	
Additional Operations	
subsumption	
functional uncertainty	encodable
others	
Tools	
Comments	

TYPE SYSTEM		
Type Connectives		
	conjunction: single vs. multiple inheritance	for general pred.: multiple inheritance for primitive types: multiple inheritance
	disjunction	for general pred.: yes for primitive types: yes
	negation	for general pred.: yes for primitive types: yes
	others	for primitive types: disjointness
Type Definitions		
	via feature structures	for general pred.: any complex typed f. str. (with variables) types (predicates) may have arguments
	via appropriateness conditions	for general pred.: predicates can be also typed for primitive types: yes
	recursive?	
	others	
Additional Operations		
	type inference/ classification	for primitive types: yes
	GLB/LUB type subsumption	for primitive types: yes
	others	for general pred.: determinism checking, evaluation modes: 'deterministic only', 'undelayed only', 'all'
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)		for general pred.: definite clauses for primitive types: none
Tools		
Comments		The distinction between a decidable 'primitive' typed constraint language and predicate definitions as clauses ensures that the potential existence of models of the whole specification is decidable! Moreover the "unification" component is independent of the firing of goals.

2.8 ELU: Environnement linguistique d'unification

ISSCO, Geneva

2.8.1 General

ELU is a unification-based linguistic programming environment designed for research and teaching purposes. It is implemented in Common Lisp, the compiler for data files being written in yacc and flex. It shares a common origin with the UD system developed at IDSIA (see section 2.13); the unifier, parser and finite-state lexicon have remained largely unchanged since 1989, with the generator and inheritance lexicon being added in 1990, and the transfer mechanism in 1991. See also Johnson and Rosner (1989) and Estival (1990).

2.8.2 Unifier

ELU employs a polymorphic, structure-sharing unifier.

Data Types

Atom	As in PATR-II; implicit conversion to string by 'concatenate' built-in.
Disjunction	Defined over atoms: unification interpreted as set intersection. `a/b/c ⊔ b = b`, `a/b/c ⊔ b/c/d = b/c`
Negation	Defined over atoms and disjunction: unification interpreted as intersection with complement. `~a ⊔ b = b`, `~a/b/c ⊔ c/d/e = d/e`
List	As in Prolog.
Tree	Like Prolog compound terms, except that the root ('principle functor') may be named by a variable.
Typing	A typing facility permitting complex FSs with specified contents to be named. Two typed FSs unify only if they are of the same type; a FS of type T unifies with an untyped FS only if the features in the result have been declared as appropriate for FSs of type T. While there is no built-in support for type subsumption or inheritance of information between types, these may be implemented by means of relational abstractions. `name = (f1, f2, f3)`: declares the content of a FS of type `name` to be the features `f1`, `f2` and `f3`. `VAR == name`: constrains the instantiated value of `VAR` to be a FS of type `name`.

Relational Abstractions

An extension of the PATR-II 'template' facility to form a constraint language closely resembling Prolog, but without the extralogical devices of that language ('cut', negation, conditionals, 'assert', 'var', etc.).

```
Proc(A,B,C)          three arguments
    <A f> = [B|D]    B head of a list value
    !Proc2(C,D)      call to another R.A.
```

Relational abstractions may be defined recursively and/or in terms of multiple subclauses; in the latter case evaluation involves breadth-first expansion of all possibilities, while in the former evaluation is suspended as long as insufficient information is available to identify the boundary case.

Built-in Relations A number of useful three-place relations are provided, using a specialized notation:

Append	`A ++ B = C` `C` is the result of appending the lists `A` and `B`.
Extract	`A -- B = C` `C` is the result of extracting the element `B` from the list `A`.
Concatenate	`A && B = C` `C` is the result of concatenating the strings `A` and `B`.

Restrictors

Users may declare certain features (normally those with the most distinctive values) as 'restrictors'; these are used in the prediction step of the parser, indexing the inheritance lexicon, and preprocessing the grammar for generation, and generally as a prefilter for unification. See Shieber (1985).

2.8.3 Finite-State Lexicon

One of ELU's lexicon systems takes the form of a finite-state machine in which states are associated with various types of information (equations, calls to abstractions, etc.) and arcs are labelled with segments of words (stems, suffixes, etc.). This is similar to the continuation-class approach to lexical organization taken by Koskenniemi (1983), but does not make use of the two-level 'spelling rule' mechanism. Looking up a word involves traversing the automaton, concatenating arc labels to instantiate the user-declared 'form' feature, and unifying information in the states to build a FS associated with that word-form.

`stemvariant`	form of word-segment
`<cat> = x`	some information
`@stem\main`	merge info. from `stem` entry in lexicon `main`
`+e/t`	insert `e` if suffix begins with `t`
`$sufs1`	name of continuation class for suffixes.

2.8.4 Default Inheritance Lexicon

As an alternative to the finite-state lexicon, ELU also provides for lexicons in the form of a restricted multiple inheritance hierarchy combining strict and defeasible unification. The lexicon below associates with the `example` class the two FSs shown:

$$\begin{bmatrix} \text{f1} & \text{val1} \\ \text{b} & \text{abc} \\ \text{f2} & \text{val3} \end{bmatrix} \qquad \begin{bmatrix} \text{f1} & \text{val1} \\ \text{b} & \text{abcxyz} \\ \text{f2} & \text{val4} \end{bmatrix}$$

`#Word example (Super)`	inherits from one superclass
`<f1> = val1`	overrides defeasible information in `Super`
`<b> = abc`	
`#Class Super ()`	no superclasses
`<f1> = val2`	`example` an exception
`|`	non-defeasible below here
`<f2> = val3`	first variant
`<f0> = <b>`	
`|`	
`<f2> = val4`	second variant
`<f0> = <b> && xyz`	concatenation

See Russell et al. (1992) for a fuller description.

2.8.5 Grammar Rules

Essentially as in PATR-II, modulo the extended inventory of data types: each rule consists of a 'rewrite' and an 'information' section. The generator requires one right-hand side item to be marked (with a prefixed `H`) as the head.

```
A -> H_B C          rule involves three FSs – H_B is semantic head
    <A cat> = c     atomic value for feature cat in A
    !Head(A,B)      call to relational abstraction
    C == sign       type of FS C
```

2.8.6 Parser

A two-pass approach: first a chart parser using Earley prediction builds structures on the basis of the 'rewrite' section of the grammar rules and user-declared restrictors, then constraints are solved in order to instantiate FSs and possibly eliminate some analyses.

2.8.7 Generator

The Shieber et al. (1990) algorithm, modified to complete the bottom-up attachment phase before initiating top-down treatment of non-head constituents. A 'semantic head' is marked in each grammar rule, rather than being derived automatically as in the standard algorithm, thus allowing grammar writers to force a rule to be interpreted top-down when it would otherwise be interpreted bottom-up. In non-chaining rules, top-down generation begins with this item.

2.8.8 Transfer Rules

Transfer-based machine translation is supported by a facility permitting users to define what amounts to a grammar capable of analysing a FS and building another based on its contents. The following mapping between FSs is established by the transfer rules shown below:

$$\begin{bmatrix} \text{f0} & \text{f-val} \\ \text{f1} & \begin{bmatrix} \text{f2} & \text{f-val-a} \\ \text{f3} & \text{f-val-b} \end{bmatrix} \end{bmatrix} \iff \begin{bmatrix} \text{g0} & \text{g-val} \\ \text{g1} & \text{g-val-b} \\ \text{g2} & \text{g-val-a} \end{bmatrix}$$

```
:T:  example                transfer rule name
:L1: <f0> = f-val
     <f1> = A
     <A f2> = X1            complex value of f1
     <A f3> = X2
:L2: <g0> = g-val
     <g2> = Y2
     <g3> = Y1
:X:  X1 = Y1                recursive transfer through variables
     X2 = Y2
:TA: f-val-a  g-val-a       atomic transfer rules
:TA: f-val-b  g-val-b
```

See Russell at al. (1991) for a fuller description.

2.8.9 Compiler

The user language illustrated in the examples given here is compiled into Lisp expressions by an independent program, **eluc**. Dependencies between files may be managed by means of an 'include' directive.

2.8.10 User Environment

From the user's point of view, a typical session with ELU involves editing, compiling and loading a number of files containing one or more linguistic descriptions; each of these descriptions is installed in a named 'setup', normally corresponding to one of the languages between which translation is to be performed. The ELU top level provides commands for:

- compiling and loading data files into a given setup
- switching between setups
- analysing and generating words
- parsing and generating sentences
- applying transfer rules
- saving and reusing results of computation
- escapes to the shell or an editor
- tracing, debugging, inspecting the chart

Preferred settings for various options may be placed in an initialization file.

ELU output is character-based rather than graphical.

2.8.11 Responsibility

ELU is the result of collaborative work over a number of years involving: Rod Johnson and Mike Rosner (IDSIA), John Carroll (Cambridge University Computer Laboratory), Amy Winarske (Lucid Inc.), Afzal Ballim, Graham Russell, Dominique Estival and Susan Armstrong (ISSCO).

References

D. Estival (1990) **ELU User Manual**. Technical Report 1, ISSCO, Geneva.

Johnson, R. and M. Rosner (1989) "A Rich Environment for Experimentation with Unification Grammars", **Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics**, 182–189.

Koskenniemi, K. (1983) **Two-level Morphology: a General Computational Model for Word-form Recognition and Production**. Publication 11, Department of General Linguistics, University of Helsinki.

Russell, G., A. Ballim, J. Carroll and S. Warwick-Armstrong (1992) "A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons", **Computational Linguistics** 18(3), 311–337.

Russell, G., A. Ballim, D. Estival and S. Warwick-Armstrong (1991) "A Language for the Statement of Binary Relations over Feature Structures", **Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics**, 287–292

Shieber, S.M. (1985) "Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms", **Proceedings of the Eighteenth Annual Meeting of the Association for Computational Linguistics**, 145–152.

Shieber, S.M., G. van Noord, R.C. Moore and F.C.N. Pereira (1990) "Semantic-Head-Driven Generation", **Computational Linguistics** 16(1), 30–42.

System Name: ELU
Designed and Implemented by: Rod Johnson, Mike Rosner (IDSIA), John Carroll (Cambridge), Afzal Ballim, Dominique Estival,Graham Russell, Susan Warwick (ISSCO)

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	unique
non-montonic devices	Default inheritance in lexicon—standard unifier under different control
control facilities	"restrictor"—declared f-values to be unified first for early failure. For generator—RHS item to generate first is marked
parser/generator?	Parser—2 pass, earley CF and unif. constraint solver Generator—Shieber-van Noord Head Driven
others	Transfer rules—non-monotonic bidirectional mappings between feature structures
Data Types	
arity (fixed?)	any, except for typed FSs
cyclic structures	no checking, but not supported by parser, generator, transfer
lists/sets	lists (+ "append", "element" ops.)
functions/relations	"Relational Abstractions"—like pure Prolog some built-in (append, element, concatenation)
others	trees, string (+ concatenation op.)
Interaction FS $\Longleftrightarrow$ Types	
type unification	identity only
type expansion	
at definition/ compile time	–
at run time: (delayed/partial/ recursive)	–
others (templates...)	

GENERAL DESCRIPTION II		
Interfaces to		
	morphology	• finite-state continuation-class model • hierarchical lexicon
	semantics/ knowledge repr.	no extra-sentential processing: no facilities for clean interface
Implementational Issues		
	programming lang.	Common Lisp—Allegro 4+, compiler: yacc & flex
	machine	Sun
	others (O/S, graphics...)	SunOS 4+ no graphics, etc.
Applications		
	grammar theories?	experimental grammars in style of LFG, GPSG, Categorial G., GB, HPSG—varying degrees of fidelity
	educational vs. commercial system	educational—research and teaching
	used in projects/ other systems?	main practical application is continuing project to make system for translating Swiss avalanche warning bulletins
Grammar coded		
	size	variable—up to Lexicons 250–60,000 words
	language	French, German, Italian, English
Tools		• Debugger—print internal objects in external form at several levels of detail. • Tracer—focus on named rule & rel. abstractions when debugging. • Tree display (ASCII) • FS display (ASCII) • Lexicon dump → indexed disk file • Display time & other statistics
Comments		a partial port to Macintosh CL exists

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	
non-destructive	x
disjunction:	
atoms only	x
full (DNF)	
distributed	
others	
negation	
atoms only	
negated corefs	
full	
others	atoms and disjunctions of atoms
implication	
via negation	–
others	–
Additional Operations	
subsumption	–
functional uncertainty	via recursive Rel. Abstractions & path variables
others	–
Tools	
Comments	

TYPE SYSTEM		
Type Connectives		
	conjunction: single vs. multiple inheritance	–
	disjunction	–
	negation	–
	others	–
Type Definitions		
	via feature structures	no
	via appropriateness conditions	Type $t_1(f_1 \ldots f_n)$: "Type t_1 has features $f_1 \ldots f_n$, & only these"—no value typing
	recursive?	no
	others	
Additional Operations		
	type inference/ classification	no
	GLB/LUB type subsumption	no
	others	–
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)		trivial—distinct types don't unify.
Tools		–
Comments		Typing is optional Full typing not enforced More sophisticated systems may be simulated with Relational Abstractions.

2.9 Pleuk

Jo Calder[a], Kevin Humphreys[b]

This section is a lightly edited version of a paper written by Jo Calder and Kevin Humphreys. It describes Pleuk—a shell within which interpreters for grammatical formalisms can be embedded. Its design is intended to allow the encoding of a wide range of grammatical formalisms, while providing sophisticated facilities for interacting with such formalisms. A number of currently popular formalisms have been implemented within Pleuk. The result is a system with applications in the fields of grammar development, education and elsewhere. The version of Pleuk described in the paper does not differ significantly from the one demonstrated by Chris Brew at the EAGLES workshop on implemented grammar formalisms. Since Pleuk is not a formalism, not all the formalisms mentioned in the paper were actually used in the demonstration. An updated version of Pleuk is scheduled to be available by ftp from the DFKI's server. The same distribution is also available from Michael Covington's archive at the University of Georgia.

2.9.1 Introduction

A current concern within computational linguistics is with the reusability of resources, in particular corpora and lexical databases. The same concern arises, of course, with respect to resources of other kinds, such as computer implementations of grammatical formalisms, but has yet to be addressed in any substantive way. The system discussed in this section attempts to improve upon this situation. We present a system called *Pleuk* which is intended as a "formalism-neutral" shell within which to embed computational interpretations of grammatical formalisms.

The organization of this section is as follows. We first discuss the design of the system, and the distinctions we make between the various tasks such systems have to perform. These fall into three basic classes, which we gloss as the "functional", "interface" and "grammatical" parts of the system. We then discuss briefly the grammatical formalisms which Pleuk currently supports. We assess the system's future potential and close with a brief description of its current implementation.[1]

The system as a whole is complex,so the current description is considerably simplified. Further information is available in the form of a printed/on-line manual, from which some of this section is derived.

2.9.2 The Tasks of Grammar Development

We view the tasks any grammar development system must perform as dividing into three categories:

- maintaining an accurate picture of the grammar currently being worked on and interaction with the host operating system;
- performing the operations required by some grammatical formalism under the control of the user and
- allowing the user to control those operations via a reasonable interface.

Accordingly, we divide the tasks that the system performs into three parts:

- the *functional backbone* (FB);

[a]Jo Calder, School of Computing Science, Simon Fraser University, Burnaby BC, CANADA V5A 1S6, Phone: (604) 291 3012, Fax: (604) 291 3045, Email: `jcalder@cs.sfu.ca`

[b]Kevin Humphreys, University of Edinburgh, Centre for Cognitive Science, 2 Buccleuch Place, Edinburgh EH8 9LW, Scotland, email: `kwh@cogsci.ed.ac.uk`

[1]In this section, the following trademarks are used: X Windows is a trademark of MIT. PostScript is a trademark of Adobe Systems Inc. SPARCstation is a trademark of Sun Microsystems Inc.

- the *specialization* and
- the *user interface*

We discuss these in turn. As the system is implemented in Prolog, certain terminology (for example *database*) should be interpreted in that light. The term *definition* should be construed as referring to any element which may form part of a grammatical description.

Functional Backbone

The FB provides support for the following operations:

- Compute files to be loaded to produce an up-to-date image of the current grammar.
- Call appropriate functions to read and compile definitions from files.
- Register definitions in or delete definitions from the database.
- Report current status of files and definitions.
- Retrieve definitions from database by name, by kind or by file.
- Call appropriate functions to format definitions.
- Call appropriate functions to parse or generate or otherwise interact with the information in the grammar.
- Interact with the host environment in appropriate ways (e.g. interpret command line switches, determine availability of graphics systems).
- Invoke an editor on the file containing a particular definition, to allow the addition or deletion of definitions.
- Maintain a set of variables and values controlling aspects of the system's behaviour.
- Provide low-level support for interaction with the user.

While this list certainly does not include all that one might expect from such a system, it represents at least sufficient functionality for a workable system.

Note that in no cases do we assume a particular format for definitions, either in the form they take on when constructed by the user, or in their internal database representation. This is essential if the FB is to operate with grammatical formalisms whose syntax and semantics may vary in arbitrary ways.

Specializations

A *specialization* is some collection of code which

- determines possible definitions of some grammatical formalism,
- defines some way of turning a definition in a file into the required internal format,
- provides a parser, generator or other means of interacting with the formalism in question and
- provides mappings between internal representations and a Pleuk-defined *printing format* to be discussed below.

The key idea here is then that the FB is entirely unconcerned with the internal representation of definitions. The specialization must define appropriate routines for reading definitions in files (often this is just Prolog's `read/1`) and for translating such definitions into the form that, for example, a parser might expect. Details of those specializations that currently exist are given in Section 2.9.3.

A specialization is defined by providing routines with the above functionality. In particular, the different kinds of definitions that a particular formalism uses must be stated, together with their distribution in different files, routines for formatting definitions or derivations to be printed, and an indication of the kinds of processing facilities (i.e. parsers, generators, ...) that the specialization offers. An order for loading the files that define different aspects of some grammar may be stated. This allows definitions which are necessary for the interpretation of later definitions to be loaded earlier. An example of this might be the definition of a mapping from attribute names to term positions and the later use of attribute names.

The specialization may also ask for particular routines to be run each time a file containing certain definitions is loaded. This allows definitions whose interpretation may be dependent on other definitions to be computed after all such definitions are loaded. (An example of this would be a system where certain axioms must be obeyed by all definitions; the routine could in this case compute simplifications that may thereby result.)

It will be noted that it is also the specialization's responsibility to define appropriate parsing and/or generation routines. The provision of generic routines, while feasible to some extent, has not been investigated.

User Interface

The third part of the system covers interaction with the user. The main functions here are:

- Interpret menu definitions provided by the FB and the specialization.
- Allow for choice of menu options or user input where appropriate.
- Manage the display of definitions and the results of computations.
- Provide on-line help.

The definitions of menus is independent from the manner in which the user interacts with them. In particular, we have interpreters for the menu system which allow interaction via a dumb terminal or graphically under X Windows.

Menus may be defined either by the FB or by a specialization, with the latter given priority in the case of multiple definitions. Menus are constructed dynamically, and so may be adjusted in order that options reflect the current state of the system.

Standard Printing Format The one area in which Pleuk makes assumptions about the format of terms manipulated by some specialization is in the output of routines that compute representations to be printed. In this case, the term is assumed to be in *Standard Printing Format* (SPF). SPF has a formal definition as a set of Prolog terms, and a graphical interpreter for this format is available on-line in Prolog under X Windows. A PostScript interpreter for a closely related format is also available, as well as a character-based approximation suitable for non-graphics devices.

Terms in SPF may be written out in PostScript format and the result included within printed documents (in the manner of Figure 2.5). This figure shows an example SPF term, with some internal structure suppressed, together with its graphical interpretation via PostScript.

In addition to supporting the attribute-value diagrams and sequences shown in the figure, SPF provides facilities for representing trees, tags indicating shared structure, symbols, including logical connectives, italics, sets, relations, infixes and a number of other diagram types. We are aware of respects in which the current facilities are deficient—for instance, it is currently impossible to represent derivations in the style preferred by many categorial grammarians (e.g. Steedman 1987) where the use of combination rules is expressed by a a solid line beneath the elements involved

```
avm([phon=sequence([atomic(loves)]),
     synsem=avm([local=avm([cat=avm([head=avm([v=atomic(+),
                                               n=atomic(-),
                                               vform=atomic(fin)]),
                                     subcat=sequence([..., ...],
                                     lex=atomic(+)]
```

Figure 2.5: A term in Standard Printing Format and its graphical interpretation

in that combination. In this case, the specification of SPF requires extension. Also, the current tree drawing algorithm produces results which are not aesthetically pleasing. The advantage of a formal specification for SPF is that the interpreters may be improved in any number of ways without affecting the formalism-specific routines that compute SPF.

One of the spin-offs of the use of SPF is that, given appropriate graphics facilities, it is simple to implement a *Derivation Checker* in which the user is provided with a point-and-click interface, where trees (or derivations, as appropriate for the grammar in question) can be constructed out of other trees or lexical elements. This seems to be a useful model for debugging and educational purposes, and also to be applicable to a wide variety of grammatical formalisms (Calder, 1993).

2.9.3 Specializations

To date, specializations defining the following formalisms have been incorporated into Pleuk, roughly in order of implementation:

Term A term-based unification grammar system, originally developed for the support of Unification Categorial Grammar (Zeevat **et al** 1991).

Mike A simple graph-based unification system, enhanced with additional operations for the treatment of free word order presented in Reape (1989).

Cfg A simple context-free grammar system, intended for demonstration purposes.

SLE A graph-based formalism enhanced with arbitrary relations in the manner of Johnson & Rosner (1989) and Dörre & Eisele (1991). Delayed evaluation is used to compute infinite relations. This system has been used for the development of several HPSG-style grammars (Pollard & Sag 1987, forthcoming).

Sdg The system described in Dahl *et al* (1991) for the implementation of principles-and-parameters grammars in terms of Dahl's Static Discontinuity Grammars (Dahl & Popowich 1990). (Incomplete and currently under revision)

HPSG-PL An HPSG system developed at Simon Fraser University by Popowich, Vogel and Kudric.

The core of Pleuk (i.e. the FB and user interface) stabilized after the first two specializations mentioned above were implemented. Various other formalisms are being implemented, in particular Carpenter's system for typed feature structures (Carpenter 1992).

In general, the cost of porting a particular grammatical formalism for which a Prolog implementation exists has not been found to be particularly high, of the order of a few hours of work, although we should emphasize that such work has to date been carried out by people familiar both with Pleuk and with the target formalisms. The bulk of the effort, unsurprisingly, has to do with generation of SPF terms from definitions.

2.9.4 Assessment

Pleuk is currently in use at a number of research laboratories in various countries. To date, most of its use has been concerned with the development of grammars either for demonstration purposes, or for the examination of particular grammatical phenomena, rather than with the development of large coverage grammars. In these tasks, Pleuk seems to have been adequate. The ability to produce high-quality output both on-line and for published documents will become of greater importance as the grammars developed within Pleuk become more complex.

Pleuk also seems to be suitable for use in educational settings. In particular, the ability to construct derivations graphically (see Section 2.9.2) offers interaction with a particular grammar which is at once more detailed and more directly under the control of the user than is possible when parsers and/or generators are the only means of constructing a derivation for some grammar.

There are, of course, limitations to what Pleuk can do. At the very least, as it is implemented in Prolog, interpreters for formalisms implemented in other languages cannot be immediately embedded within Pleuk. Less generally, implementation of some formalism within Pleuk is easiest when there is a straightforward relationship between a definition stated by the user and its internal representation. The simplest relationship is where each definition gives rise to one and only one internal representation. Other cases, for example, where a particular definition gives rise to more than one internal definition are also catered for. A more problematic case is that mentioned at the end of Section 2.9.2 where some set of definitions has global interpretation. A further example of this is a set of statements defining a mapping from attribute names to term positions to take advantage of term, rather than graph, unification. In this case, certain implementations may decide to compute terms directly for the representation of other definitions. A problem will arise if the set of statements changes—Pleuk has no way of determining whether such a change means that the interpretation of other definitions has to be revised.

One facility which is of use in educational and machine-translation settings (as well as in grammar development more generally) is the possibility of manipulating several grammars at once. Pleuk does not provide such a facility directly. However, in an extension of Pleuk completed for an industrial research laboratory, definitions containing grammatical information are organized into a hierarchy. At each node in the hierarchy, all and only definitions from the current node and from dominating nodes are available. Evaluation of this approach is continuing.

2.9.5 Implementation

Pleuk is currently implemented in SICStus Prolog, version 2.1 (Carlsson *et al* 1991), with a small number of functions defined in C, running on Sun SPARCstations. Menus for user input and output windows, including the output of graphical interpretations of SPF, make use of the Graphics Manager supplied with SICStus. We have endeavoured to maintain portability of the FB—no Prolog system predicates are called directly—but this is currently compromised by dependencies on the user interface side of the system. Certain specializations also make use of SICStus-specific facilities, such as the Boolean Constraint Solver.

System documentation is written in the Free Software Foundation's Texinfo format and on-line help is provided by the XInfo system by Jordan K. Hubbard. Documentation is currently incomplete in the case of some specializations. On-line interpretation of PostScript output is possible via Aladdin Enterprises' Ghostscript.

2.9.6 Conclusions

Pleuk is a shell for the implementation of grammatical formalisms. The system has been used successfully to encode a variety of grammatical formalisms, and for the development of a number of grammars. The system allows the on-line display of high-quality graphical representations of definitions and derivations, and their inclusion within other documents.

Bibliography

Calder, J. (1993) Graphical interaction with constraint-based grammars, Proceedings of PACLING, 1993, Vancouver, Canada.

Carlsson, M., J. Widén, J. Andersson, S. Andersson, K, Boortz, H. Nilsson and T. Sjöland (1991) SICStus Prolog User's Manual. Swedish Institute of Computer Science, technical report T91:11B.

Carpenter, B. (1992) **The Logic of Typed Feature Structures: with applications to unification grammars, logic programs, and constraint resolution**, Cambridge Tracts in Theoretical Computer Science, no. 32. Cambridge, New York: Cambridge University Press.

Dahl, V. and F. Popowich (1990) Parsing and Generation with Static Discontinuity Grammars. *New Generation Computing*, **8**, pp245–274.

Dahl, V., F. Popowich and M. Rochemont (1991) A Principled Characterization of Dislocated Phrases: Capturing Barriers with Static Discontinuity Grammars. Technical report CPMT TR 91-09, Centre for Systems Science, Simon Fraser University.

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

Johnson, R. and M. Rosner (1989) A rich environment for experimentation with unification grammars. In EACL4, pp182–189.

Pollard, C. and I. A. Sag (1987) **Information-Based Syntax and Semantics, Volume 1: Fundamentals**. CSLI Lecture Notes No 13. Stanford, Ca.: Center for the Study of Language and Information.

Pollard, C. and I. A. Sag (forthcoming) **Information-Based Syntax and Semantics, Volume 2**. CSLI Lecture Notes. Stanford, Ca.: Center for the Study of Language and Information.

Reape, M. (1989) A logical treatment of semi-free word order and bounded discontinuous constituency. In EACL4, pp103–110.

Steedman, M. (1987) Combinatory grammars and parasitic gaps. Natural Language and Linguistic Theory, **5**.3, pp403–439.

Zeevat, H., Klein, E. and Calder, J. (1991) Unification Categorial Grammar, **Lingua e stile**, **17**.4, pp499–527.

System Name: SLE formalism (just part of Pleuk. Pleuk itself is a generic framework, not a formalism)

Designed and Implemented by: Jonathan Calder, Kevin Humphreys, Mike Reape

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	unique engine
non-montonic devices	no
control facilities	control statements in template definitions allow grammar writer to specify when it is safe to expand recursive templates
parser/generator?	1. chart parser 2. non-deterministic bidirectional parser/generator
others	user controlled interactive derivation checker
Data Types	
arity (fixed?)	open-ended feature graphs
cyclic structures	no
lists/sets	no
functions/relations	Horn-clause relational dependencies via template definitions
others	–
Interaction FS $\Longleftrightarrow$ Types	
type unification	no
type expansion	
at definition/ compile time	no
at run time: (delayed/partial/ recursive)	no
others (templates...)	Template system uses Horn-clause definitions and control statements to delay/control evaluation.

GENERAL DESCRIPTION II	
Interfaces to	
morphology	no
semantics/ knowledge repr.	no
Implementational Issues	
programming lang.	Prolog
machine	sun-4, sparcstation etc.
others (O/S, graphics...)	Derivation checker uses Sicstus Prolog graphics manager
Applications	
grammar theories?	HPSG, UCG
educational vs. commercial system	mostly educational but larger grammars under development by industrial partner
used in projects/ other systems?	
Grammar coded	
size	small demos
language	English, French
Tools	High level interface language for printing of linguistic objects. Interface to editor. Interface to grammar displayer.
Comments	The tools are generic facilities of Pleuk, not specific to SLE formalism.

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	no
non-destructive	yes
disjunction:	
atoms only	yes
full (DNF)	no
distributed	no
others	no
negation	
atoms only	yes
negated corefs	no
full	no
others	additional sound special case $\neg(f : \text{Top})$
implication	
via negation	
others	implicational constraints
Additional Operations	
subsumption	no
functional uncertainty	encodable
others	none
Tools	Limited user control of evaluation strategy/depth bound via run-time switches
Comments	none

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	n/a
disjunction	n/a
negation	n/a
others	–
Type Definitions	
via feature structures	n/a
via appropriateness conditions	n/a
recursive?	n/a
others	–
Additional Operations	
type inference/ classification	n/a
GLB/LUB type subsumption	n/a
others	–
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	n/a
Tools	–
Comments	Direct specification of features appropriate at a node.

2.10 The *TDL/UDiNe* System

Rolf Backofen, Stefan Diehl, Bernd Kiefer, Karsten Konrad, Hans-Ulrich Krieger, Ulrich Schäfer, Christoph Weyers

TDL is a typed feature-based language specifically designed to support highly lexicalized grammar theories like HPSG, FUG, or CUG. *TDL* offers the possibility to define (possibly recursive) types, consisting of type constraints and feature constraints over the standard connectives $\wedge$, $\vee$, and $\neg$, where the types are arranged in a subsumption hierarchy. *TDL* distinguishes between *avm types* (open-world reasoning) and *sort types* (closed-world reasoning) and allows the declaration of partitions and incompatible types. Working with partially as well as with fully expanded types is possible, both at definition and at run time. *TDL* is incremental, i.e., it allows the redefinition of types and the use of undefined types.

TDL is based on *UDiNe*, a sophisticated feature constraint solver. *UDiNe* incorporates most of the advanced means that have been described in literature or used in practical system, e.g., distributed disjunctions, negative coreferences, full negation as well as functional and relational constraints.

TDL and *UDiNe* together provide both a grammar definition environment and a typed run time system which supports lazy type expansion. Efficient reasoning in the system is accomplished through specialized modules.

2.10.1 Motivation

Modern typed unification-based grammar formalisms (like TFS, CUF, or *TDL*) differ from the early untyped systems like PATR-II in that they highlight the notion of a *feature type*. Types can be arranged hierarchically, where a subtype *inherits* monotonically all the information from its supertypes and unification plays the role of the primary information-combining operation. A *type definition* can be seen as an abbreviation for a complex expression, consisting of type constraints (concerning the sub-/supertype relationship) and feature constraints (stating the appropriate values of attributes) over the standard connectives $\wedge$, $\vee$, and $\neg$. Types can therefore lay foundations for a grammar development environment because they might serve as abbreviations for lexicon entries, ID rule schemata, and universal as well as language-specific principles as is familiar from HPSG. Besides using types as a referential mean as templates are, there are other advantages as well which however cannot be accomplished by templates:

- EFFICIENT PROCESSING. Certain type constraints can be compiled into more efficient representations like bit vectors, where a GLB (greatest lower bound), LUB (least upper bound), or a $\preceq$ (type subsumption) computation reduces to low-level bit manipulation. Moreover, types release untyped unification from expensive computation through the possibility of declaring them incompatible. In addition, working with type names only or with partially expanded types, minimizes the costs of copying structures during processing.
- TYPE CHECKING. Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore disallowing to write inconsistent feature structures.
- RECURSIVE TYPES. Recursive types give a grammar writer the opportunity to formulate certain functions or relations as recursive type specifications. Working in the *Parsing as Deduction* paradigm enforces a grammar writer to replace the CF backbone through recursive types.

2.10.2 The DISCO Core Engine

The core machinery of DISCO consists of *TDL* and the feature constraint solver *UDiNe*. The *TDL* system is a unification-based grammar development environment and run-time system to support HPSG-like grammars. The DISCO grammar currently consists of more than 700 type

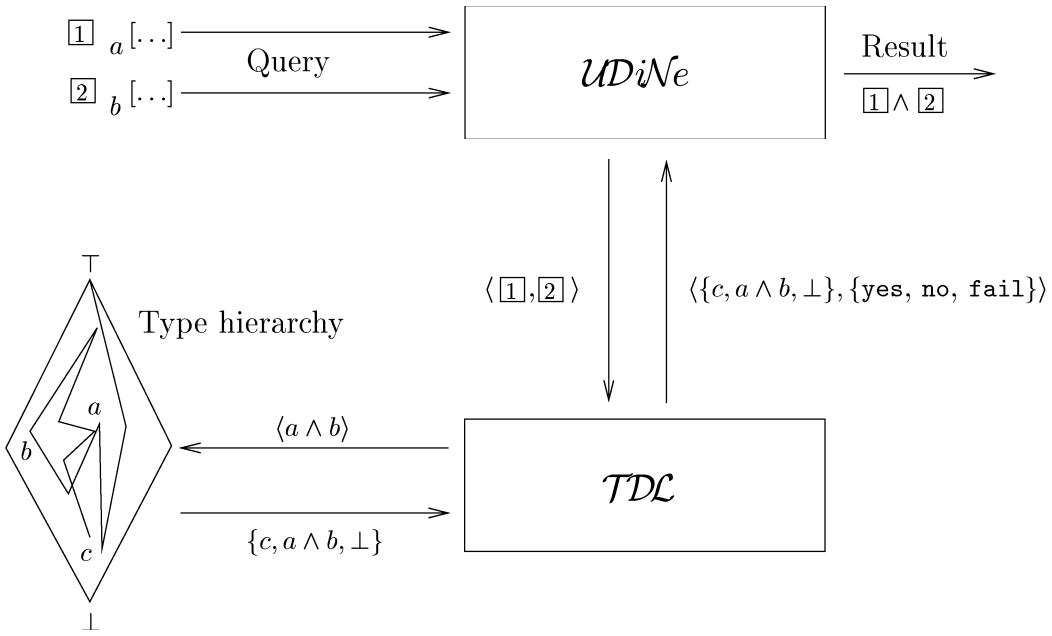


Figure 2.6: **Interface between *TDL* and *UDiNe*.** Depending on the type hierarchy and the type of 1 and 2, *TDL* either returns c (c is definitely the GLB of a and b) or $a \wedge b$ (open-world reasoning) resp. $\bot$ (closed-world reasoning) if there doesn't exist a single type which is equal to the GLB of a and b. In addition, *TDL* determines whether *UDiNe* must carry out feature term unification (`yes`) or not (`no`), i.e., the return type contains all the information one needs to work on properly (`fail` signals a global unification failure).

specifications written in *TDL* and is the largest HPSG grammar for German. The *UDiNe* feature constraint solver is the main processing machinery of DISCO, which has been well-tested in the DISCO environment over years. Typical size of the processed structures reaches more than 1000 nodes and 140 coreferences (which would need up to 185 000 nodes in a PROLOG tree notation).

Both modules communicate through an interface, and this communication mirrors exactly the way an abstract typed unification algorithm works: two typed feature structures can only be unified if the attached types are definitely compatible. This is accomplished by the unifier in that *UDiNe* handles over two typed feature structures to *TDL* which gives back a simplified form (plus additional information; see Fig. 2.6). The motivation for separating type and feature constraints and processing them in dedicated modules (which again might consist of specialized components as is the case in *TDL*) is twofold: (i) it reduces the complexity of the whole system, thus making the architecture much clearer, and (ii) leads to a faster system performance because every dedicated module is designed to cover only a specialized task.

Grammars and lexicons can be tested by using the parser of the DISCO system. The parser is a bidirectional bottom-up chart parser, providing a user with parameterized parsing strategies as well as giving him control over the processing of individual rules.

2.10.3 The *UDiNe* Feature Constraint Solver

UDiNe is a modern feature constraint solver that provides distributed disjunctions over arbitrary structures, negative coreferences, full negation and functional constraints. It is the first (and to our knowledge the only) implemented feature constraint solver that integrates both full negation and distributed disjunctions. A relational extension has been implemented, but not yet integrated into the system.

UDiNe works on an internal representation of feature structures, where coreferences are represented using structure sharing. The connection between the internal representation and the

(good readable) external one is established by input/output functions. There exists an advanced window-based feature editor called FEGRAMED allowing to define, print and manage feature structures.

During the translation of the external representation into the internal one, several normalization steps are performed. One of this steps is the elimination of full negation in the input structure. We use the method of Smolka (1988), which introduces implicit existential quantification. Using this method, negation can be eliminated if the feature systems provides disjunction, negative coreferences and negated atoms/types.

UDiNe uses distributed disjunctions not only as a tool for efficient processing. They are also part of the input syntax, which allows for a very compact representation of the input data. In contrast to other systems using distributed disjunctions, we do not restrict disjunctions to length 2 (neither in input nor during processing). This reduces the size of the representation of a feature structure massively.

UDiNe is a dedicated feature constraints solver that can be connected with different type systems. Unification is done destructively using the lazy copying technique introduced by Aït-Kaci, where only the affected structure must be copied. Non-destructive unification is performed using copy functions. *UDiNe* has been successfully used for several tasks in the DISCO project, viz. for parsing, generation, extended two-level morphology and surface oriented speech act processing.

The functionality of *UDiNe* is completed by several auxiliary functions. It is possible to remove inconsistent alternatives, to simplify structures, to extract subterms or to evaluate functional constraints. A general visiting function can be used for constructing user-oriented extensions. Furthermore, one can build the disjunctive normal form of a feature structure. This is needed by other tools used in the application system if they cannot handle distributed disjunction.

2.10.4 Intelligent Backtracking

Uszkoreit introduced in 1991 a new strategy for linguistic processing called **controlled linguistic deduction**. The evaluation of both conjunctive and disjunctive constraints can be controlled in this framework. For conjunctive constraints, the one with the highest failure probability should be evaluated first. For disjunctive ones, a success probability is used instead. The alternative with the highest success probability is used until a unification fails, in which case one has to backtrack to the next best alternative. Besides more complex ones, Uszkoreit also proposed a strategy that uses static values for the success probabilities (called preferences). In the following, we will call unifier that control the evaluation of disjunctions in this way unifier with intelligent backtracking.

Because of similarities between this control method and the mechanism of intelligent backtracking in PROLOG, we can formulate the following properties that a unifier with intelligent backtracking should fulfill:

- INDEPENDENCE. Backtracking must be independent from the computation history, i.e. backtracking should not be restricted to the last processed disjunction.
- CONFLICT DETECTION. It must be possible to determine the disjunctive structures that are involved in a unification failure. This is necessary in order to restrict the set of candidates for backtracking.
- CONFLICT DEDUCTION. The conflict information of several unification errors can be used for further restricting the conflicting set of disjunctions. This avoids unnecessary backtracking.
- COMPLETENESS. It must be guaranteed that consistent combination of disjunction alternatives will be detected.

The most promising candidates for implementing intelligent backtracking are unifiers that use distributed disjunctions, since they provide most of the concepts mentioned above. Hereby, the notion of context common to all of these unifiers plays an important role. A **context** is partial function mapping disjunctions to corresponding alternatives. Every node has a unique context that describes under which disjunctions and which alternatives this node can be found. If a

unification fails, the context of the node where the inconsistent information has been found is called **inconsistent context**. The inconsistent contexts are stored in order to deduce minimal inconsistent contexts and to detect a global inconsistency. Thus, inconsistent contexts can be used for conflict detection. The calculation of minimal inconsistent contexts corresponds to conflict deduction. The check for global inconsistency can be used for guaranteeing completeness.

We have implemented a prototypical extension of $\mathcal{UDiNe}$ that incorporates intelligent backtracking and provides the independence property mentioned above. The implementation works as follows. If a disjunction is encountered, the alternative with the highest preference is chosen, and only this alternative is used for later unifications. If a unification fails, the involved disjunctions are determined by the inconsistent context. Now one of the involved disjunctions has to be selected for backtracking. There are two possibilities: (i) one can use the static preference for this selection; and (ii) the unifier calls a user program in order to select a disjunction. The idea is to use the second selection mechanism for implementing more complex control methods. E.g. to achieve better selection criteria, we can provide the user program with the definition of the disjunction and the conjunctive part the disjunction has be unified with.

The backtracking of the selected disjunction first undoes the unification with the previously chosen alternative. We have modified the existing method for undoing destructive unification in order to guarantee a local undo. Second, the cancelled unifications are redone using the new selected alternative. The algorithm guarantees, that unification is restricted to the substructure starting with the disjunction.

2.10.5 The $\mathcal{TDL}$ language

$\mathcal{TDL}$ supports type definitions consisting of type constraints and feature constraints over the standard operators $\wedge$, $\vee$, $\neg$, and $\oplus$ (xor). The operators are generalized in that they can connect feature descriptions, coreference tags (logical variables) as well as types. $\mathcal{TDL}$ distinguishes between avm types (open-world semantics), sort types (closed-world semantics), and built-in types. In asking for the greatest lower bound of two avm types a and b which share no common subtype, $\mathcal{TDL}$ always returns $a \wedge b$ (open-world reasoning), and not $\bot$. The opposite case holds for sort types. Furthermore, sort types differ in another point from avm types in that they are not further structured, like atoms are. Moreover, $\mathcal{TDL}$ offers the possibility to declare *exhaustive and disjoint partitions* of types, for example $sign = word \oplus phrase$ which expresses the fact that (i) there are no other subtypes of *sign* than *word* and *phrase*, (ii) the sets of objects denoted by these types are disjoint, and (iii) the disjunction of *word* and *phrase* can be rewritten (during processing) to *sign*. In addition, one can declare sets of types as *incompatible*, meaning that the conjunction of them yields $\bot$.

$\mathcal{TDL}$ allows a grammarian to define and use parameterized templates (macros). There exists a special instance definition facility to ease the writing of lexicon entries which differ from normal types in that they are not entered into the type hierarchy. Strictly speaking, lexicon entries can be seen as the leaves in the type hierarchy which do not admit further subtypes. This dichotomy is the analogue to the distinction between classes and instances in object-oriented programming languages. Input given to $\mathcal{TDL}$ is parsed by a Zebu-generated LALR(1) parser to allow for an intuitive, high-level input syntax and to abstract from uninteresting details imposed by the unifier and the underlying Lisp system.

2.10.6 Type Hierarchy

The implementation of the type hierarchy is based on Aït-Kaci's bit vector encoding technique for boolean lattices (a bit-and/or operation corresponds to a LUB/GLB computation). The method has been modified to open-world reasoning over avm types in that potential GLB/LUB candidates must be verified by inspecting the type hierarchy through a sophisticated graph search. GLB, LUB and $\preceq$ computations have the nice property that they can be carried out in $O(n)$, where n is the number of types. Depending on the encoding method, the hierarchy occupies $O(n \log n)$ (compact encoding) resp. $O(n^2)$ (transitive closure encoding) bits.

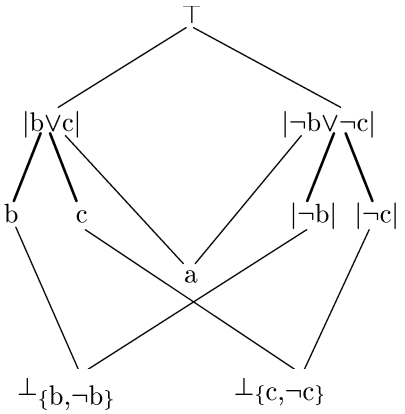


Figure 2.7: Decomposing $a := b \oplus c$, so that a inherits from the intermediates $|b \vee c|$ and $|\neg b \vee \neg c|$.

The encoding algorithm is extended to cope with the redefinition of types, an essential part of an incremental grammar/lexicon development system. Redefining a type means not only to make changes local to this type. Instead, one has to redefine all dependents of this type—all subtypes, in case of a conjunctive type definition and all disjunction elements for a disjunctive type specification plus, in both cases, all types which mention these types in their definition. The dependent types of a type t can be characterized graph-theoretically via the *strongly connected components* of t.

Conjunctive, e.g., $x := y \wedge z$ and disjunctive type specifications, e.g., $x' := y' \vee z'$ are entered differently into the hierarchy: x inherits from its supertypes y and z, whereas x' defines itself through its elements y' and z'. This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunctive elements, see Fig. 2.7).

TDL decomposes complex definitions consisting of $\wedge$, $\vee$, and $\neg$ by introducing *intermediate types*, so that the resulting expression is either a pure conjunction or a disjunction. The same technique is applied when using $\oplus$ (see Fig. 2.7). $\oplus$ will be decomposed into $\wedge$, $\vee$ and $\neg$, plus additional intermediates. For each negated type $\neg t$, *TDL* introduces a new intermediate type symbol $|\neg t|$ with the definition $\neg t$ and declares it incompatible with t.

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 2.7) which are however identified in the underlying logic. These bottom symbols must be propagated downwards by a mechanism called *bottom propagation* which takes place at definition time.

2.10.7 Symbolic Simplifier

The simplifier operates on arbitrary *TDL* expressions. Simplification is done at definition time as well as at run time when typed unification takes place (cf. Figure 2.6). The main issue of symbolic simplification is to avoid (i) unnecessary feature constraint unification and (ii) queries to the type hierarchy by simply applying 'syntactic' reduction rules.

The simplification schemata are well known from the propositional calculus, e.g., De Morgan's laws, idempotence, identity, absorption, etc. They are hard-wired in Common Lisp in order to speed up computation. Formally, type simplification in *TDL* can be characterized as a term rewriting system. Confluency and termination is guaranteed by imposing a *generalized lexicographically ordered normal form* on terms (either CNF or DNF). In addition, this order has the nice effects of neglecting the law of commutativity (which is expensive and might lead to termination problems): there is only one representative for a given formula. Therefore, *memoization* is cheap and is employed in *TDL* to reuse precomputed results of simplified (sub)expressions (one must not cover all permutations of a formula). Additional reduction rules are applied at run time using 'semantic' information of the type hierarchy (GLB, LUB, and $\preceq$).

System Name: TDL/UDINE
Designed and Implemented by: Rolf Backofen, Stefan Diehl, Bernd Kiefer, Karsten Konrad, Hans-Ulrich Krieger, Ulrich Schäfer, Christoph Weyers

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	dedicated modules: • feature constraint solver • sophisticated type system: – symbolic simplification – inheritance reasoning
non-montonic devices	special form of inheritance and unification; some theoretical results available
control facilities	• intelligent backtracking (weighted disjuncts) • type expansion
parser/generator?	• advanced chart parser • semantic head-driven generator
others	future version of the type expansion mechanism can be parametrized for different expanding strategies
Data Types	
arity (fixed?)	free
cyclic structures	yes
lists/sets	only list via FIRST/REST encoding special constructs in the specification language
functions/relations	functions with residuation relations
others	built-ins: integer, strings, symbols
Interaction FS $\Longleftrightarrow$ Types	
type unification	yes
type expansion	
at definition/ compile time	yes—work with partially and fully expanded types and recursive ones
at run time: (delayed/partial/ recursive)	yes—work with partially and fully expanded types and recursive ones
others (templates...)	parametrized templates

GENERAL DESCRIPTION II	
Interfaces to	
morphology	extended 2-level morphology X2 MorF
semantics/ knowledge repr.	semantic representation language $\mathcal{NLL}$
Implementational Issues	
programming lang.	Common Lisp (Franz Inc., Allegro)
machine	SUN/SPARC compatible machines
others (O/S, graphics...)	UNIX CLIM (Common Lisp Interface Manager)
Applications	
grammar theories?	HPSG
educational vs. commercial system	educational
used in projects/ other systems?	DISCO (DFKI), BiLD (Dep. of Comp. Ling.), PRACMA (Comp. Science Saarbrücken
Grammar coded	
size	650 type specification 250 lexicon entries
language	
Tools	
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	yes
non-destructive	non-destructive via COPY & RESET
disjunction:	
atoms only	
full (DNF)	yes (via MAKE-DNF)
distributed	yes; both in implementation and specification language
others	
negation	
atoms only	
negated corefs	yes
full	yes with restrictions to distributed disjunctions (negation over non-distributed disj. only)
others	
implication	
via negation	yes
others	
Additional Operations	
subsumption	on DNF
functional uncertainty	via recursive type specifications
others	RESET, COPY, MAKE-DNF
Tools	• generic traversing function on feature structures • print feature structure
Comments	prototype version of "intelligent backtracking" (weighted disjuncts)

TYPE SYSTEM		
Type Connectives		
	conjunction: single vs. multiple inheritance	multiple inheritance
	disjunction	yes
	negation	yes
	others	XOR via OR and NOT
Type Definitions		
	via feature structures	yes
	via appropriateness conditions	
	recursive?	yes
	others	type declarations: incompatibility and partitions
Additional Operations		
	type inference/ classification	type inference via type unification type classification during type definition
	GLB/LUB type subsumption	GLB, LUB, type subsumption
	others	GLB/LUB behave differently when applied to sorts or avm types
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)		unrestricted input of type specs, some transformation are performed on the input by the system
Tools		type grapher tdl2 Latex software switches which changes the behaviour of the whole system
Comments		making a distinction between sort types (closed-world reasoning) and avm types (open-world)

2.11 The Typed Feature Structure Representation Formalism

Martin C. Emele
Institut für maschinelle Sprachverarbeitung
Universität Stuttgart
emele@ims.uni-stuttgart.de

The Typed Feature Structure (TFS) representation formalism is an attempt to provide a synthesis of several of the key concepts of unification-based grammar formalisms (feature structures), knowledge representation languages (inheritance) and logic programming (logical variables and declarativity). The inheritance-based constraint 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. The interpreter of the formalism is described as a term rewriting system based on type unfolding where unification of typed feature structures is used to detect inconsistencies between a query and the constraints imposed by the feature type system.

The grammar writer organizes unification grammars as inheritance networks of typed feature structures. Complex linguistic structures are described by means of recursive type constraints which correspond to class definitions in object-oriented formalisms. The use of an object-oriented methodology with inheritance is very attractive for natural language processing and offers a number of advantages such as abstraction and generalization, information sharing, modularity and reusability of descriptions. Through the development of a wide variety of different applications it has been demonstrated that the formalism is flexible enough and well suited to represent the principles and parameters approach of modern computational linguistic theories. In particular, it has been successfully used for the encoding of large grammar fragments as described in the HPSG grammar representation formalism.

2.11.1 Type Constraint System

Type Hierarchy

One of the design criteria behind the implementation of TFS was to minimize the amount of information the user has to explicitly provide while specifying a grammar description. Hence we do not have to specify separate appropriateness conditions instead they will be inferred from the type constraints. The following assumptions about types hold:

- types prestructure the domain of discourse and are interpreted as unary predicates which denote subsets of the universe.
- we assume that all minimal types exhaustively partition the domain. Hence **negative** information, which shows up as inconsistency between types, is represented only implicitly. Such an approach is motivated by the fact that for most of the linguistic applications only a few types interact and thus we have to express only the positive statements for those types and not all the negative ones which are implied by the partitioning assumption.
- all non-minimal types are equivalent to the union of the minimal types which they subsume.
- types not defined in the hierarchy are unconstrained and assumed to be pairwise incomparable.

The set of types *Type* is ordered by a subtype relation where the user may specify an arbitrary finite partial order (po) $\langle \mathit{Type}, \leq \rangle$ for defining the type hierarchy without any further conditions like unary branching or being a bounded complete partial order (bcpo) as it is assumed in other

formalisms. The user specified po will be embedded into the restricted powerset $2^{\lfloor Type \rfloor}$ which is constructed by taking all non-empty subsets of pairwise incomparable elements and hence defines a cochain. The resulting construction forms a distributive lattice which preserves the original ordering and already existing meets.

Typed Feature Structures

Typed feature structures are very similar to structured objects of object-oriented languages and act as the lingua franca for computational linguists. They are defined over a finite set of features *Feat* and over a finite type hierarchy $\langle Type, \leq \rangle$.

Type Definitions

A collection of recursive type definitions which associates typed feature structure constraints to types forms a feature type system and offers the functionality of class definitions imposing constraints on objects. Possible constraints involve:

- **structural** constraints which define for each type the set of appropriate attributes and for each attribute the attribute-value restriction,
- **(in)equality** constraints expressed over a set of variables *Var* and a special equality predicate,
- **relational** constraints attached to types as further conditions which must be true and hence further restrict the denotation of the constrained type.

Formally, type definitions may be seen as axioms forming a theory. Satisfiability of feature structure descriptions is checked with respect to this theory. Type unfolding is used to enforce the constraints imposed on types by their definitions. In contrast to other systems TFS supports not only inheritance of appropriateness information but also inheritance of equational and relational constraints. Whereas the satisfiability of the structural constraints is decidable, adding equality and relational constraints leads in general to undecidability. By checking the decidable structural constraints at compile-time we gain the same advantages as in a system with an explicit type discipline but without having to duplicate the typing information into a redundant appropriateness specification. The appropriateness information is extracted from the structural constraints and used for inferring missing type information and for doing actual type checking by testing for satisfiability.

2.11.2 Summary

The TFS system has been developed to provide a computational environment for the design and the implementation of formal models of natural language. It does not offer means of defining control information that would make execution more efficient (but less general), as it would be needed if it were envisaged to use the system in an application-oriented environment (e.g., as a parser in a natural language interface to a database system). As such, the TFS formalism is not designed as a programming language, but as a specification language that can be used to design, implement, and test formal linguistic models. From these formal models, it could be envisaged to develop programs, i.e., parsers or generators, that would implement efficiently the declarative knowledge contained in the formal specifications.

The TFS system is implemented using rewriting techniques in a constraint-based architecture adapted to typed feature structures:

- The language is a logical language directly based on typed feature structure constraints, and supports an object-oriented style based on multiple inheritance.

- Grammars are expressed as inheritance networks of typed feature structure descriptions. They define constraints on the set of acceptable linguistic structures. As a consequence we have a truly multi-directional architecture and there is no formal distinction between "input" and "output".
- A unique general constraint solving mechanism is used. Specific mapping properties, based on constituency, linear precedence or functional composition, are not part of the formalism itself, but can be encoded explicitly using the formalism.

Although the current implementation is very much at the level of an experimental prototype, and is still evolving, it has allowed to validate the basic concepts of the language and of the implementation, and we have been able to show that TFS can adequately model a wide variety of descriptive paradigms in computational linguistics: descriptive work and migration from existing resources was carried out in the frameworks of LFG, DCG, SFG and HPSG grammars. From these various experimentations, we have defined extensions and improvements, both on the language and on the implementation, that are needed for scaling up the system.

On the formal language side, more expressivity is needed. For example, sets of feature structures are necessary to formalize non trivial semantic structures. Feature structure encodable types like lists and strings could be conveniently added to the system as libraries of built-in types together with a specific syntax and associated operations, like concatenation, etc., for which specialized constraint-solvers could be provided to improve the termination behaviour and the performance of these operations.

On the implementation side, the use of implementation techniques adapted from Prolog implementations, especially the compilation of feature constraints into an abstract machine like the WAM together with more sophisticated control strategies would greatly enhance the efficiency of the constraint solver. For specific application domains, like natural language parsing, compilation of string concatenation into a standard chart-parsing module, where the linguistic description allows such a compilation, might turn out to be a viable strategy for improving the efficiency.

Implementation

TFS was designed and implemented by Martin C. Emele und Rémi Zajac within the German Polygloss Project (BMFT Project 08 B 3116 3 / 08 B 3120 6) and is implemented in Common Lisp (e.g. MCL, Allegro, Lucid, CMU CL, CLISP, AKCL). It runs on virtually any architecture and software system for which an appropriate Common Lisp dialect is available. TFS owes much of its portability to the fact that it does not include any graphic displays for its basic version. For some architectures plus Common Lisp dialects (currently LISP-machines and Mac IIs) the implementation of TFS supports graphic output and a menu-based window interface. A CLIM-based (Common Lisp/ X Window System / Interface Manager) graphical interface has been written by Oliver Christ.

TFS runs as a stand-alone system for the Macintosh II family, and requires System 7. For other architectures it requires the appropriate Common Lisp license.

Availability and Maintenance

TFS is copyright Institut für maschinelle Sprachverarbeitung, Universität Stuttgart. It is distributed in a binary form only, and is available without charge for academic research for non-commercial use. The main TFS files for different LISP dialects and architectures can be obtained by anonymous ftp from the address `ftp.ims.uni-stuttgart.de` (141.58.127.8) in the directory TFS. There are also subdirectories 'demo' and 'HPSG' containing further sample grammars and programs.

References

For more information about the theoretical motivations for TFS, see the following articles, and references therein:

Emele, Martin and Rémi Zajac (1990). Typed Unification Grammars. In: Hans Karlgreen (ed.), Proceedings of the 13th International Conference on Computational Linguistics (CoLing90), Helsinki, August 1990.

Zajac, Rémi (1992). Inheritance and Constraint-Based Grammar Formalisms. Computational Linguistics 18, pp159–180.

System Name: TFS
Designed and Implemented by: Martin C. Emele, Rémi Zajac

GENERAL DESCRIPTION I		
Inference Engine		
	unique engine vs. dedicated modules	unique engine (constraint solver)
	non-montonic devices	no
	control facilities	no explicit control clause-indexing over lexicon
	parser/generator?	not necessary since using the engine the same grammar description successfully used for parsing and generation
	others	
Data Types		
	arity (fixed?)	no fixed arity terms
	cyclic structures	yes
	lists/sets	lists, set operations encodable
	functions/relations	• using macro syntax • definite clause compiler available (e)
	others	
Interaction FS $\Longleftrightarrow$ Types		
	type unification	yes, uses precomputed type lattice
	type expansion	
	at definition/ compile time	deterministic expansion at compile time
	at run time: (delayed/partial/ recursive)	evaluation of recursive type definitions with delayed expansion
	others (templates...)	parametric macros expanded at compile time

GENERAL DESCRIPTION II	
Interfaces to	
morphology	not necessary since morphology can be encoded within the system as well
semantics/ knowledge repr.	not necessary since morphology can be encoded within the system as well
Implementational Issues	
programming lang.	Common Lisp
machine	any on which a suitable CL runs
others (O/S, graphics...)	Macintosh System 7, UNIX, X11, CLIM (Common Lisp Interface Manager)
Applications	
grammar theories?	HPSG, LFG, GB, SFG
educational vs. commercial system	public domain
used in projects/ other systems?	used widely about 20 installations worldwide
Grammar coded	
size	covers large parts of the II Vol. of Pollard & Sag
language	EN, GE, FR, JA sample grammars of HPSG
Tools	• Graphical Feature Structure/Tree Displayer • Browser and Editor for type hierarchy
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	ombination of destructive and non-destructive
non-destructive	unification with structure-sharing and lazy copying
disjunction:	
atoms only	atomic disjunctions of types expanded at run time
full (DNF)	full disjunction possible within definite clauses (e)
distributed	no
others	disjunctive type definition
negation	
atoms only	
negated corefs	inequations
full	
others	
implication	
via negation	
others	
Additional Operations	
subsumption	
functional uncertainty	encodable with recursive types
others	
Tools	
Comments	

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	multiple inheritance
disjunction	yes
negation	expressed via closed world assumption
others	
Type Definitions	
via feature structures	inferred from the type definitions
via appropriateness conditions	
recursive?	yes
others	
Additional Operations	
type inference/ classification	full type inference (infers missing type info and classifies feature term descriptions according to type definitions)
GLB/LUB type subsumption	GLB/LUB
others	compile-time type-checking
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	unrestricted partial order which is embedded into a distributive lattice
Tools	
Comments	

2.12 Short Description of Trace & Unification Grammar (TUG)

Hans Ulrich Block
Siemens AG, Corporate Research, ZFE ST SN 74
Otto Hahn-Ring 6
D-81730 München
Germany
block@zfe.siemens.de

ABSTRACT

This paper presents Trace & Unification Grammar (TUG), a declarative and reversible grammar formalism that brings together Unification Grammar (UG) and ideas of Government & Binding Theory (GB). The TUG system is part of a polyfunctional linguistic processor for German called LINGUISTIC KERNEL PROCESSOR (LKP). The LKP contains a grammar of German with broad coverage. The grammar describes the relation between a subset of German and a subset of QLF, the intermediate semantic form that is used in the *Core Language Engine* of SRI Cambridge (Alshawi 1990). The LKP has been implemented in PROLOG. Parsing and Generation of a sentence up to 15 words normally takes between 1 and 10 seconds, with a strong tendency to the lower bound.

2 FORMALISM

The design of Trace and Unification Grammar has been guided by the following goals:

- **Perspicuity**. We are convinced that the generality, coverage, reliability and development speed of a grammar are a direct function of its perspicuity, just as programming in Pascal is less errorprone than programming in assembler. In the optimal case, the grammar writer should be freed of reflections on how to code things best for processing but should only be guided by linguistic criteria. These goals led for example to the introduction of unrestricted disjunction into the TUG formalism.
- **Compatibility to GB Theory**. It was a major objective of the LKP to base the grammar on well understood and motivated grounds. As most of the newer linguistic descriptions on German are in the framework of GB theory, TUG was designed to be somehow compatible with this theory though it was not our goal to "hardwire" every GB principle.
- **Efficiency**. As the LKP is supposed to be the basis of products for interactive usage of natural language, efficiency is a very important goal. Making efficiency a design goal of the formalism led e.g. to the introduction of feature types and the separation of the movement rules into head movement and argument movement.

The basis of TUG is formed by a context free grammar that is augmented by PATR II-style feature equations. Besides this basis, the main features of TUG are feature typing, mixing of attribute-value-pair and (PROLOG-) term unification, flexible macros, unrestricted disjunction and special rule types for argument and head movement.

2.1 BASIC FEATURES

As a very simple example we will look at the TUG version of the example grammar in Shieber (1984).

```
% type definition

s      => f.
```

```
np     => f(agr:agrmnt).
vp     => f(agr:agrmnt).
v      => f(agr:agrmnt).

agrmnt => f(number:number,person:person).

number => {singular,plural}.
person => {first,second,third}.

% rules

s ---> np, vp |
     np:agr = vp:agr.

vp ---> v, np |
     vp:agr = v:agr.

% lexicon

lexicon('Uther',np) |
     agr:number = singular,
     agr:person = third.
lexicon('Arthur',np) |
     agr:number = singular,
     agr:person = third.
lexicon(knights,v) |
     agr:number = singular,
     agr:person = third.
lexicon(knight,v) |
     ( agr:number = singular,
       ( agr:person = first
       ; agr:person = second
       )
     ;
       agr:number = plural
     ).
```

The two main differences to PATR II in the basic framwork are that first, TUG is less flexible in that it has a "hard" contextfree backbone, whereas in PATR II categories of the context free part are placeholders for feature structures, their names beeing taken as the value of the `cat` feature in the structure. Second, TUG has a strict typing. For a feature path to be well defined, each of its attributes has to be declared in the type definition.

Besides defined attribute-value-pairs, TUG allows for the mixing of attribute-value-pair unification with arbitrary structures like PROLOG terms using a back-quote notation. This can be regarded as the unificational variant of the `BUILDQ` operation known from ATNs. As an example consider the following lexicon entry of *each* that constructs a predicate logic notation out of `det:base`, `det:scope` and `det:var`.

```
lexicon(each,det) |
     det:sem =
          `all(det:var,det:base ->
                         det:scope)
```

During our work on the German grammar we found that this feature was very useful for the construction of semantic forms.

TUG provides templates for a clearer organization of the grammar. The agreement in the above mentioned grammar might have been formulated like this:

```
agree(X,Y) short_for
     X:agr = Y:agr.

...

s ---> np, vp |
     agree(np,vp).
```

TUG allows for arbitrary disjunction of feature equations. Disjunctions and Conjunction may be mixed freely. Besides well known cases as in the entry for *knight* above, we found many cases where disjunctions of path equations are useful, e.g. for the description of the extraposed relative clauses[1].

2.2 MOVEMENT RULES

Further to these more standard UG-features, TUG provides special rule formats for the description of discontinuous dependencies, so called "movement rules". Two main types of movement are distinguished: argument movement and head movement. The format and processing of argument movement rules is greatly inspired by Chen *e.a.* (1988) and Chen (1990), the processing of head movement is based on GPSG like slash features.

Head Movement

A head movement rule defines a relation between two positions in a parse tree, one is the landing site, the other the trace position. Head movement is constrained by the condition that the trace is the head of a specified sister (the root node) of the landing site[2]. Trace and Antecedent are identical with the exception that the landing site contains overt material, the trace doesn't. Suppose, that `v` is the head of `vk`, `vk` the head of `vp` and `vp` the head of `s`, then only the first of the following structures is a correct head movement, the second is excluded because `np` is not head of `vp`, the third because antecedent and trace are unequal.

```
[s' v_i [s ... [vp ...
   [vk ... trace(v)_i ...]...]...]...]
[s' np_i [s ... [vp trace(np)_i ...
   [vk ... v ...]...]...]
[s' np_i [s ... [vp ...
   [vk ... trace(v)_i ...]...]...]...]
```

To formulate head movement in TUG the following format is used. First, a head definition defines which category is the head of which other.

```
v is_head_of vk.
vk is_head_of vp.
vp is_head_of s.
```

Second, the landing site is defined by a rule like

```
s' ---> v+s | ...
```

To include recursive rules in the head path, heads are defined by the following head definitions. In a structure `[M` D_1 `...` D_n`]` D_i is the head of `M` if either D_i `is_head_of M` is defined or D_i has the same category as `M` and either D_i `is_head_of X` or `X is_head_of` D_i is defined for any category `X`.

[1] Block/Schmid (1992) describes our processing technique for disjunctions.

[2] Here, "head of" is a transitive relation s.t. if x is head of y and y is head of z then x is head of z.

Head movement rules are very well suited for a concise description of the positions of the finite verb in German (sentence initial, second and final) as in

Hat$_i$ der Mann der Frau das Buch gegeben t$_i$?
Has$_i$ the man the woman the book given t$_i$
Der Mann hat$_i$ der Frau das Buch gegeben t$_i$
The man has$_i$ the woman the book given t$_i$
... daß der Mann der Frau das Buch gegeben hat
... that the man the woman the book given has

All that is needed are the head definitions and the rule that introduces the landing site[3].

Argument Movement

Argument movement rules describe a relation between a landing site and a trace. The trace is always c-commanded by the landing site, its antecedent. Two different traces are distinguished, anaphoric traces and variable traces. Anaphoric traces must find their antecedent within the same bounding node, variable trace binding is constrained by subjacency, i.e. the binding of the trace to its antecedent must not cross two bounding nodes. Anaphoric traces are found for example in English passive constructions [`s` [`np` The book of this author]$_i$ was read t$_i$] whereas variable traces are usually found in wh-constructions and topicalization. Similar to the proposal in Chen *e.a.* (1988), argument movement is coded in TUG by a rule that describes the landing site, as for example in

```
s2 ---> np:ante<trace(var,np:trace), s1 |
     ante:fx = trace:fx,
     ...
```

This rule states that `np:ante`[4] is the antecedent of an np-trace that is dominated by s1. This rule describes a leftward movement. Following Chen's proposal, TUG also provides for rightward movement rules, though these are not needed in the German grammar. A rightward movement rule might look like this.

```
s2 ---> s1, trace(var,np:trace)>np:ante |
     ante:fx = trace:fx,
     ...
```

The first argument in the trace-term indicates whether the landing site is for a variable (`var`) or for an anaphoric (`ana`) trace. Other than head movement, where trace and antecedent are by definition identical, the feature sharing of argument traces with their antecedents has to be defined in the grammar by feature equations (`ante:fx = trace:fx, ...`). Furthermore, it is not necessary that the antecedent and the trace have the same syntactic category. A rule for pronoun fronting in German might e.g. look like this:

```
spr ---> pron<trace(ana,np), s | ...
```

The current version of the formalisms requires that the grammar contains a declaration on which categories are possible traces. In such a declaration it is possible to assign features to a trace, for example marking it as empty:

```
trace(np) | np:empty = yes.
```

[3]Even though verb movement is not supposed to be a topic for English grammar, one might think of describing English Subj-Aux inversion in terms of head movement.
Peter has been reading a book
Has$_i$ Peter t$_i$ been reading a book

[4]The notation `Cat:Index` is used to distinguish two or more occurrences of the same category in the same rule in the equation part. `:ante` and `:trace` are arbitrary names used as index to refer to the two different `np`s.

Bounding nodes have to be declared as such in the grammar by statements of the form

```
bounding_node(np).
bounding_node(s) | s:tense = yes.
```

As in the second case, bounding nodes may be defined in terms of category symbols and features[5]. Typical long distance movement phenomena are described within this formalism as in GB by trace hopping. Below is a grammar fragment to describe the sentence *Which books$_i$ do you think t_i John knows t_i Mary didn't understand t_i*:

```
bounding_node(s).
bounding_node(np).

s1 ---> np<trace(var,np), s | ...
s ---> np, vp | ...
s ---> aux, np, vp | ...
np ---> propernoun | ...
np ---> det, n |
vp ---> v, s1 | ...
vp ---> v, np | ...

trace(np).
```

The main difference of argument movement to other approaches for the description of discontinuities like extraposition grammars (Pereira 1981) is that argument movement is not restricted to nested rule application. This makes the approach especially atractive for a scrambling analysis of the relative free word order in the German *Mittelfeld* as in

Ihm$_i$ hat$_j$ das Buch$_k$ keiner t_i t_k gegeben t_j.

3 COMPILATION OF TUG

For efficient processing TUG is compiled to two different forms, one for parsing and one for generation. Prior to both compilations a TUG is transformed to DCG format.

For parsing, this format is then transformed for processing with a Tomita parser (Tomita 1986) in several steps:

- expansion of head movement rules
- transformation of argument movement rules
- elimination of empty productions
- conversion to LR(K) format
- computation of LR tables

After these compilation steps the context free rules are transformed to YACC format and YACC is used to compute the LR parsing table. Finally, YACC's `y.output` file is transformed to PROLOG.

For generation with TUG an improved version of the semantic-head-driven generator (SHDG) (see Shieber *e.a.* 1990) is used. Before being useful for generation, the grammar is transformed in the following steps:

- expansion of head movement rules
- transformation to the semantic head driven generator format

[5] Currently, only conjunction of equations is allowed in the definition of bounding nodes.

- expansion of movement rules
- elimination of nonchainrules with uninstantiated semantics
- goal reordering and transformation to executable prolog code

4 CONCLUSION

We have presented Trace & Unification Grammar, a grammar formalism that tries to bridge the gap between UG and GB theory. TUG comes with a parser generator and a generator generator that lead to effient runtime code of the grammar both for parsing and for generation.

The presented grammar formalism has been used to describe a relevant subset of German language and smaller subsets of Chinese and Japanese. The grammars describe a mapping between German, Chinese and Japanese and QLF expressions. TUG has been used in the German ASL-project for speech understanding, in the CSTAR project for spoken language translation from German to Japanese and in the German part of the SUNDIAL project.

ACKNOWLEDGEMENTS

The TUG-system and the grammars for German have been developed by Manfred Gehrke, Rudi Hunze, Steffi Schachtl, Ludwig Schmid and Christine Zünkler. The Chinese grammar has been written by Ping Peng. The Japanese grammar has been written by Juunko Hosaka.

REFERENCES

Alshawi, H. (1990) "Resolving Quasi Logical Forms", *Computational Linguistics*, Vol. 16, pp. 133-144.

Block, H. U. (1991) "Compiling Trace and Unification Grammar for Parsing and Generation", *Proc. of the ACL-Workshop on Reversible Grammar in Natural Language Processing*, pp. 100-108.

Block, H. U. and S. Schachtl (1992) "Trace and Unification Gramar", *Proc. 14th International Conference on Computational Linguistics* (COLING-92), pp. 87-93.

Block, H. U. and L. A. Schmid (1992) "Using Disjunctive Constraints in a Bottom-Up Parser" *Konferenz "Verarbeitung natürlicher Sprache"* (KONVENS 92), pp. 169-177.

Chen, H.-H., I-P. Lin and C.-P. Wu (1988) "A new design of Prolog-based bottom-up Parsing System with Government-Binding Theory", *Proc. 12th International Conference on Computational Linguistics* (COLING-88), pp. 112-116.

Chen, H.-H. (1990) "A Logic-Based Government-Binding Parser for Mandarin Chinese", *Proc. 13th International Conference on Computational Linguistics* (COLING-90), pp. 1-6.

Pereira, F. (1981) "Extraposition Grammar" *Computational Linguistics* Vol. 7, pp. 243-256.

Shieber, S.M. (1984) "The design of a Computer Language for Linguistic Information" *Proc. 10th International Conference on Computational Linguistics* (COLING-84), pp. 362-366.

Shieber, S.M. (1988) "A Uniform Architecture for Parsing and Generation", *Proc. 12th International Conference on Computational Linguistics* (COLING-88), pp. 614-619.

Shieber, S.M., G. van Noord, F.C.N. Pereira and R.C. Moore (1990). "Semantic-Head-Driven Generation". *Computational Linguistics*, Vol. 16, pp. 30-43.

Tomita, M. (1986). *Efficient Parsing for Natural Language: A fast Algorithm for Practical Systems*. Boston: Kluwer Academic Publishers.

System Name: Trace & Unification Grammar (TUG)
Designed and Implemented by: Siemens AG

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	
non-montonic devices	no
control facilities	no
parser/generator?	parser and generator
others	
Data Types	
arity (fixed?)	fixed
cyclic structures	no
lists/sets	lists
functions/relations	no
others	"movement rules"
Interaction FS $\Longleftrightarrow$ Types	
type unification	Prolog term unification
type expansion	
at definition/ compile time	type elimination at compile time
at run time: (delayed/partial/ recursive)	
others (templates...)	

GENERAL DESCRIPTION II	
Interfaces to	
morphology	no
semantics/ knowledge repr.	yes
Implementational Issues	
programming lang.	Prolog (Quintus-, Sixtus-, SNI)
machine	Sun Sparc WS
others (O/S, graphics...)	UNIX
Applications	
grammar theories?	trace theory
educational vs. commercial system	commercial
used in projects/ other systems?	ASL, SUNDIAL, CSTAR, MCI
Grammar coded	
size	ca. 370 rules, 160 type defs.
language	German, Chinese, Japanese
Tools	Lexicon Tool
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	
non-destructive	x
disjunction:	
atoms only	
full (DNF)	
distributed	
others	full, not DNF
negation	
atoms only	no
negated corefs	no
full	no
others	no
implication	
via negation	no
others	no
Additional Operations	
subsumption	no
functional uncertainty	no
others	movement rules
Tools	
Comments	

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	no
disjunction	no
negation	no
others	no
Type Definitions	
via feature structures	yes
via appropriateness conditions	
recursive?	yes
others	
Additional Operations	
type inference/ classification	no
GLB/LUB type subsumption	no
others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	
Tools	
Comments	

2.13 UD: A Unification Device

C.J. Rupp and Rod Johnson
IDSIA, Corso Elvezia 36
CH-6900 Lugano, Switzerland
email: {cj,rod}@idsia.chRupp, Mike Rosner, Paolo Cattaneo

2.13.1 Origins and Motivations

The current version of UD is a direct descendent of an earlier version of the system that was also the progenitor of the ELU system developed at ISSCO. The original version of UD was also developed at ISSCO and is reported on in Johnson & Rosner (1989) and Rupp *et al.* (1992). Subsequent development of UD since 1988, has been carried out at IDSIA and has focussed on improvements to the morphology and on the efficiency of the implementation.

The nature of the UD formalism still bears many of the influences of the original motivations for developing such a system. The original purpose of building UD was to support the development of a prototype MT system. We realised from the outset that there would be certain technical, logistical and theoretical difficulties in developing extensive linguistic descriptions and in making use of these descriptions in an MT system with adequate performance. We therefore considered it important to distinguish between a grammar development environment and the intended ultimate implementation. A major distinction between these two systems would be that the the former allowed a considerable degree of "theory prototyping", while the latter would "hard code" the major theoretical constructs of the descriptions in order to optimise performance. At the time that this project was being planned it was clear that none of the available constraint-based linguistic theories was fully adequate for the construction of extensive linguistic descriptions; so we were especially careful to permit the linguists constructing the descriptions to develop their own set of theoretical constructs which could be optimised for performance as, and when, they became stabilised. This is essentially a pragmatic approach in that, unlike, say, the LFG system, it does not assume *a priori* the adoption of a particular grammatical (meta)theory. Our approach also gives the grammar writers some control over the development of the formalism and as a result the design of the UD formalism has been largely "demand driven". This philosophy also has its weaknesses: in particular there is a risk of losing homogeneity and of excessive notational redundancy. Nevertheless the capability to support theory prototyping is a major characteristic of the UD formalism, and in this respect its expressive power is equivalent to other current formalisms of much later design, such as the CUF (Dörre & Eisele, 1991).

2.13.2 The UD Formalism

The formal notation of the UD system was designed in the latter part of the eighties to be a generic constraint-based formalism, hence it takes as its starting point the equational notation popularised by PATR-II. The two main extensions to the equational framework that UD offers are the ability to state linguistic generalisations in the form of definite relations and the provision of a number of additional data types, most notably lists, and certain key operations over these data types. The *relational abstraction* mechanism, used to state principles and other linguistic generalisations, may be seen as a direct extension of PATR-II templates, but with a considerably increased expressive power. Indeed it has been demonstrated (Reape 1991, 1993) that relations can also be used to define additional data types, but our purpose in providing an explicit representation of constructs such as lists is more than just a syntactic sweetener: if constructs of a known structure can be identified as early as possible then their structure can be exploited in the implementation.[1]

[1] There is an important pragmatic difference between a theoretically elegant minimalist view of the semantics of a formalism and the practical necessities of using that formalism to construct useful artefacts. The definition of pure Lisp, as we all know, requires just five basic functions and a constant; but the market for Lisp systems which provide only five functions and one constant is vanishingly small.

As a direct consequence of the incorporation of additional data types and of relational rather than predicative abstractions the UD notation makes use of variables in order to denote the internal structure of objects and argument positions. This further implies that paths must be rooted to determine their starting node, rather than systematically referring back to the root of the feature structure (cf. PATR-II and LFG). These key extensions over earlier formalisms offered by UD raise the question of what notational conventions to adopt. Here again our solution has been relatively pragmatic: given that most students of computational linguistics are exposed to the Edinburgh syntax notation of Prolog this provides a ready source of notational forms for constructs such as lists, variables, anonymous variables, terms and relations themselves. The adoption of a number of Prolog notational conventions was intended to diminish the learning time for novice users; but we have also found it to be a source of confusion in that novice users and audiences at demos often assumed that the underlying implementation must also be in Prolog and even inherit Prolog procedural semantics – which is emphatically not the case.

The combination of PATR-II and Prolog styles of notation covers most of the syntactic needs of the UD formalism; the remainder can be found in other well-known sources, such as the various instantiations of LFG.

The basic UD syntax can be sketched by a few schematic examples. Constraint information in UD is expressed by two basic types of construct: equations and relational abstractions. The basic form of equations is predictable from our decision to use a generic PATR-style syntax, plus the need to root paths.

$$< \texttt{Root attr1...attrn} > = \texttt{value}$$

The form of relational constraints is also unsurprising, except for the preceding exclamation mark (`!`) denoting invocation.

$$\texttt{!Relation(Arg1, ..., Arg2)}$$

Most of the other extensions to the language to account for additional data structures and built in operations over these can most easily be accounted for by considering them as additional terms which slot into the two basic forms of literal, i.e. they are equated or occur in argument positions in a relation.

`X`	variables
`[Head|Tail]`	lists
`Mother(Dtr1,...,Dtrn)`	trees
`{X,Y,Z}`	disjunctions
`List1 ++ List2`	two lists appended together
`List -- Element`	the remainder of a list after an element is removed
`<Root attr1...Y...attrn>`	a variable path
`Atom1 && Atom2`	the concatenation of two atoms
`atom1/.../atomn`	a disjunction of atoms
`~atom`	an atomic negation

The use of Prolog-style variable notation in the above table is intended to be suggestive of the fact that the syntax allows variables over any of the above syntactic types, subject to the obvious semantic constraints. For example, the variable at the root of a path must evaluate to a feature structure; and within a path a variable should evaluate to either an atom or a list of atoms. Similar constraints are obviously implied by the argument structure of other built in operators.

UD feature structures are, in general, open structures with no restrictions on the range features that may appear. It is, however, possible to impose such restrictions using a rudimentary form of typing.

Typing	Definition
`X == closed-type`	`closed-type = (first,next,...,last)`

A typing statement requires the presence of all and only those features that appear in its definition; hence there is no possibility to combine types, nor do they impose any conditions on feature values. Such statements are of limited applicability, but have found two quite independent uses that may be of interest: in preserving the syntactic integrity of semantic representations, which are in effect representations of statements in another language, and in the emulation of HPSG grammars where closed types are essential in constraining the potential proliferation of relation invocations.

A linguistic description in UD has a number of components consisting, respectively, of grammar rules, lexical entries, morphology state transitions and relation definitions.

```
Mother -> Daughter1 ...  Daughtern      grammar rule
word                                    lexical entry
lexical-string -> actual-string         morphology transition
                  $successor-state
                  $succcessor-state ...
```

Constraint information, consisting of equations and relation invocations, is associated with each of these constructs.[2]

The definition of a relation is given by a sequence of clauses, consisting of a head, which is like an invocation without the exclamation mark prefix, and a body made up of constraint information.

```
Relation(X,Y,Z) :   !Predicate(X)
                    <X path> = value
                    Z = Y -- X
                    !Macro
```

The clauses of a definition are treated disjunctively and may be recursively defined.

2.13.3 Discussion

Given the emphasis in the design of UD on the ability to build new theoretical constructs, we expect that the fabric of a linguistic description will largely consist of the invocation of relational abstractions and that the raw equational notation will largely be confined to relation definitions. Hence the dependencies between relations also form the main organisational principles of the description. This is in many ways similar to the definition of type hierarchies as in HPSG, but somewhat more flexible. This reliance on relational constructs to provide the fabric of a linguistic description is in no way diminished by the presence of additional notations representing grammar rules, lexica and morphological rules, since the constraint language remains the same throughout every component of the description. Indeed the level of modularity provided by the different components of the description can be further enhanced by the use of relational constructs to define "communication protocols" between the different representational domains of the description. The technique has been demonstrated to be effective by the continued use of the same extensive morphology and lexicon over a number of years despite considerable variations in the associated syntactic and semantic representations. This level of modularity is an obvious asset in descriptions that are constructed by more than one linguist. The general need for "declarative modularity" within constraint-based formalism is further discussed in Rupp (1993).

Given that UD was designed from the outset as a development tool for real systems, we were forced to come to terms very early with the need to maintain a very degree of expressivity without sacrificing performance. In a serious development context it is not sufficient to allow the user to say the kind of thing that you think they are going to want to say; the environment in which the formalism is implemented must equally provide an efficient testbed for the description. There is

[2]Note that this includes constructs in the morphology, which in UD is seen as a three-place relation mapping textual strings, lexical forms and feature structures. In this regard, although the underlying morphological engine is still a nondeterministic finite state transducer, UD morphology differs from the two-level orthodoxy; it is more comparable to the X2MorF used in TDL.

an obvious interaction between concision and expressivity in formalisms and their complexity, and yet most current formalisms have broadly the same expressive power. In general the key issue is non-determinism, which is usually expressed in disjunctive statements of one form or another. As has been seen above UD contains two main forms of disjunction. By far the most significant is the presence of multiple clauses in relation definitions, since these definitions are the primary constructs of a description, in both theoretical and practical terms. A more direct notation for disjunctive terms also exists, but this is mainly employed to describe alternations which though systematic do not play a significant role in the structure of the descriptions.

We have devoted a great deal of effort in UD to optimisation of the treatment of non-determinism, which is based on breadth-first expansion with heuristically driven delayed evaluation. Details of the strategy used and the reasoning behind it can be found in Johnson & Rupp (1993). The technique has proved very effective in processing descriptions which are, theoretically at least, highly complex. It is also notable that the same approach is applied to many different forms of non-determinism which occur in UD descriptions, including disjunctive and recursive relation definitions, disjunctive terms and operators over specific data types. UD has been used successfully to define numerous descriptions, including: an extensive fragment of German; complete Italian and French verbal morphologies; substantial fragments of Italian and French, including good coverage of clitic constructions; transcodings of many toy fragments of English used to illustrate other constraint-based formalisms.

Bibliography

[1] J. Dörre and A. Eisele. A comprehensive unification-based grammar formalism. DYANA deliverable R3.1.B, Centre for Cognitive Science, University of Edinburgh, Scotland, January 1991.

[2] R. Johnson and M. Rosner. A rich environment for experimentation with unification grammars. In **Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics**, pages 182–189, Manchester, 1989.

[3] R. Johnson and C. Rupp. Evaluating complex constraints in linguistic formalisms. In H. Trost, editor, **Feature Formalisms and Linguistic Ambiguity**, Chichester, 1993. Ellis Horwoood. To appear.

[4] M. Reape. An introduction to the semantics of unification-based grammar formalisms. DYANA deliverable R3.2.A, Centre for Cognitive Science, University of Edinburgh, Scotland, 1991.

[5] M. Reape. A feature value logic with intensionality, nonwellfoundedness and functional and relational dependencies. In C. Rupp, M. Rosner, and R. Johnson, editors, **Constraints, Language, and Computation**. Academic Press, London, 1993.

[6] C. Rupp, R. Johnson, and M. Rosner. Situation schemata and linguistic representation. In M. Rosner and R. Johnson, editors, **Computational Linguistics and Formal Semantics**. Cambridge University Press, Cambridge, 1992.

[7] C. J. Rupp. Constraint propagation and semantic representation. In C. Rupp, M. Rosner, and R. Johnson, editors, **Constraints, Language, and Computation**. Academic Press, London, 1993.

System Name: UD
Designed and Implemented by: Rod Johnson with contributions from C.J. Rupp, Mike Rosner, Paolo Cattaneo

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	dedicated parsing & morphology
non-montonic devices	none
control facilities	Preference ordering of outputs of rule reductions
parser/generator?	tabular parser with Earley prediction over finite restrictors; invertible fst morphology
others	
Data Types	
arity (fixed?)	open: F-structures, lists closed: F-structures, terms
cyclic structures	F-structures, lists, terms
lists/sets	lists
functions/relations	relations
others	strings
Interaction FS $\Longleftrightarrow$ Types	
type unification	N.A.
type expansion	
at definition/ compile time	deterministic expansion at load time
at run time: (delayed/partial/ recursive)	lazy evaluation at run time
others (templates...)	

GENERAL DESCRIPTION II	
Interfaces to	
morphology	integrated morphology
semantics/ knowledge repr.	no external semantics
Implementational Issues	
programming lang.	Allegro Common Lisp, compiler in C (via yacc)
machine	Sun Sparc, Sun3, Macintosh (no compiler), porting to i486 and windows
others (O/S, graphics...)	Unix ideally (ported to Macintosh or i486 and windows)
Applications	
grammar theories?	HPSG, UCG, LFG encodable
educational vs. commercial system	educational
used in projects/ other systems?	Nat+Lab (Delta D1016)
Grammar coded	
size	maximum (German) has 2,000 word lexicon
language	German, French, Italian, English (in that order)
Tools	Stepper. Compiler from external PATR-style notation to S-expression based internal representation.
Comments	

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	
non-destructive	non-destructive, structure-sharing
disjunction:	
atoms only	
full (DNF)	
distributed	
others	generalised disjunction over any data types, handled by lazy evaluation
negation	
atoms only	yes
negated corefs	no
full	no
others	
implication	
via negation	no
others	
Additional Operations	
subsumption	no
functional uncertainty	encodable easily (e.g. using path variables)
others	
Tools	
Comments	

TYPE SYSTEM		
Type Connectives		
	conjunction: single vs. multiple inheritance	yes
	disjunction	yes
	negation	no
	others	
Type Definitions		
	via feature structures	
	via appropriateness conditions	
	recursive?	yes
	others	via global conditions on feature structures
Additional Operations		
	type inference/ classification	N.A.
	GLB/LUB type subsumption	N.A.
	others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)		N.A.
Tools		
Comments		These responses refer to relations rather than types, so do not fit the questions particularly well.

Part 3

Unexhibited Systems

3.1 STUF-II

IBM Germany, Institute for Knowledge Based Systems, Roland Seiffert

(no description submitted)

System Name: STUF-II
Designed and Implemented by: IBM Germany, Institute for Knowledge Based Systems, Roland Seiffert

GENERAL DESCRIPTION I	
Inference Engine	
unique engine vs. dedicated modules	dedicated, exchangeable modules for • parsing • feature unification • sort lattice & GLB proc.
non-montonic devices	–
control facilities	tools for development & testing & evaluation of parsing strategies integrated in parser modules
parser/generator?	parser: adaptable bottom-up chart parser, generator under development
others	–
Data Types	
arity (fixed?)	free
cyclic structures	yes
lists/sets	not built-in but a set of list macros is provided sets: no
functions/relations	–
others	arbitrary sort lattices with associated GLB procedure may be integrated (one GLB proc. is built-in)
Interaction FS $\Longleftrightarrow$ Types	
type unification	FS unification calls sort GLB procedure on leaf nodes in a "sort domain"
type expansion	never (in principle, sorts can be defined arbitrarily if a GLB procedure is provided (e.g., sorts could be fs again))
at definition/ compile time	
at run time: (delayed/partial/ recursive)	
others (templates...)	templates: • parametrized • recursively definable if compile-time evaluable to a disjunction (finite)

GENERAL DESCRIPTION II		
Interfaces to		
	morphology	yes, a "lexicon manger" is integrated that allows to exploit variuous lexical resources at runtime (e.g. full-form lexicon, morpholoy, synonyms, ...)
	semantics/ knowledge repr.	yes, but very specialized (for LILOG project)
Implementational Issues		
	programming lang.	Quintus Prolog 3.2
	machine	IBM RS/6000 runing AIX3.1 + X , SUN 4, IBM PS/2 runing AIX1.1 + X
	others (O/S, graphics...)	there is a runtime version on RS/6000 that does not need Quintus Prolog
Applications		
	grammar theories?	open (we've done CUG and HPSG with it)
	educational vs. commercial system	
	used in projects/ other systems?	LILOG
Grammar coded		
	size	HPSG grammar covering a substantial fragment of German
	language	German
Tools		rather sophisticated window-oriented grammar development environment
Comments		available free of charge under licence from IBM Germany for non-profit organizations

FEATURE CONSTRAINT SOLVER	
Boolean Connectives	
unification:	
destructive	x
non-destructive	x
disjunction:	
atoms only	–
full (DNF)	–
distributed	X, Dörre/Eisele algorithm (but no distributed disjunctions in the grammar specification—only implementation technique!)
others	–
negation	
atoms only	x
negated corefs	–
full	–
others	–
implication	
via negation	–
others	–
Additional Operations	
subsumption	x
functional uncertainty	–
others	feature structures form an ADT with a whole bunch of possible operations
Tools	
Comments	

TYPE SYSTEM	
Type Connectives	
conjunction: single vs. multiple inheritance	
disjunction	
negation	
others	
Type Definitions	
via feature structures	
via appropriateness conditions	
recursive?	
others	
Additional Operations	
type inference/ classification	
GLB/LUB type subsumption	
others	
Restriction on Hierarchy (unrestricted partial order, bounded complete p.o., distributive lattice...)	
Tools	
Comments	

Part 4

Related Systems

4.1 An Informal Introduction to LIFE

Hassan Aït-Kaci Andreas Podelski Peter Van Roy
{hak,podelski,vanroy}@prl.dec.com
Digital Equipment Corporation
Paris Research Laboratory
85 Avenue Victor Hugo
92500 Rueil-Malmaison, France

LIFE is an experimental programming language proposing to integrate three orthogonal programming paradigms proven useful for symbolic computation. From the programmer's standpoint, it may be perceived as a language taking after logic programming, functional programming, and object-oriented programming. From a formal perspective, it may be seen as an instance (or rather, a composition of three instances) of a Constraint Logic Programming scheme. Here, we give an informal overview demonstrating LIFE as a programming language, illustrating how its primitives offer rather unusual, and perhaps (pleasantly) startling, conveniences.

As an acronym, 'LIFE' means **L**ogic, **I**nheritance, **F**unctions, and **E**quations. LIFE also designates an experimental programming language designed after these four precepts for specifying structures and computations.

In this document, we give an informal tour of some of LIFE's unusual programming conveniences. We hope by this to illustrate for the reader that some original functionality is available to a LIFE user. We do this by way of small yet (pleasantly) startling examples.

LIFE is a trinity. The function-oriented component of LIFE is directly derived from functional programming languages with higher-order functions as first-class objects, data constructors, and algebraic pattern-matching for parameter-passing. The convenience offered by this style of programming is one in which expressions of any order are first-class objects and computation is determinate. The relation-oriented component of LIFE is essentially one inspired by the Prolog language. Unification of first-order patterns used as the argument-passing operation turns out to be the key of a quite unique and hitherto unusual *generative* behavior of programs, which can construct missing information as needed to accommodate success. Finally, the most original part of LIFE is the structure-oriented component which consists of a calculus of type structures—the ψ-calculus [1, 2]—and accounts for some of the (multiple) inheritance convenience typically found in so-called object-oriented languages.

ψ-Calculus

In this section, we give an informal introduction of the notation, operations, and terminology of the data structures of LIFE. It is necessary to understand the programming examples to follow.

The ψ-calculus consists of a syntax of structured types called ψ-terms together with subtyping and type intersection operations. Intuitively, as expounded in [7], the ψ-calculus is a convenience for representing record-like data structures in logic and functional programming more adequately than first-order terms do, without loss of the well-appreciated instantiation ordering and unification operation.

Let us take an example to illustrate. Let us say that one has in mind to express syntactically a type structure for a *person* with the property, as expressed for the underlined symbol in Figure 4.1, that a certain functional diagram commutes.

The syntax of ψ-terms is one simply tailored to express as a term this kind of approximate description. Thus, in the ψ-calculus, the information of Figure 4.1 is unambiguously encoded into a formula, perspicuously expressed as the ψ-term:

$$\begin{array}{l} X : person(name \Rightarrow id(first \Rightarrow string, \\ \qquad\qquad\qquad\qquad last \Rightarrow S : string), \\ \qquad\qquad spouse \Rightarrow person(name \Rightarrow id(last \Rightarrow S), \\ \qquad\qquad\qquad\qquad spouse \Rightarrow X)). \end{array}$$

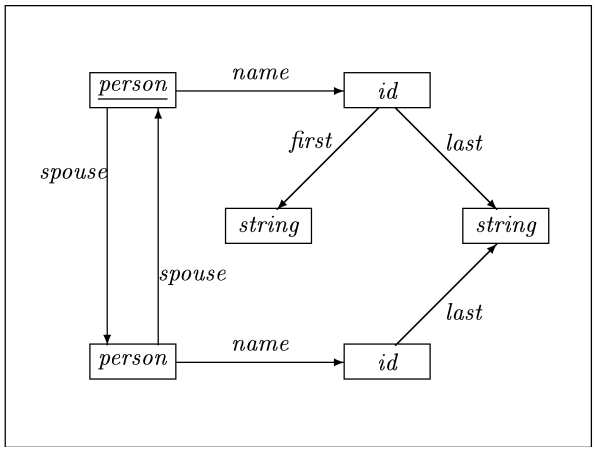


Figure 4.1: A commutative functional diagram

It is important to distinguish among the three kinds of symbols participating in a ψ-term. We assume given a set $\mathcal{S}$ of sorts or *type constructor symbols*, a set $\mathcal{F}$ of *features, or attributes* symbols, and a set V of *variables* (or *coreference tags*). In the ψ-term above, for example, the symbols *person*, *id*, *string* are drawn from $\mathcal{S}$, the symbols *name*, *first*, *last*, *spouse* from $\mathcal{F}$, and the symbols X, S from V. (We capitalize variables, as in Prolog.)

A ψ-term is either *tagged* or *untagged*. A tagged ψ-term is either a variable in V or an expression of the form $X : t$ where $X \in V$ is called the term's **root variable** and t is an untagged ψ-term. An untagged ψ-term is either *atomic* or *attributed*. An atomic ψ-term is a sort symbol in $\mathcal{S}$. An attributed ψ-term is an expression of the form $s(\ell_1 \Rightarrow t_1, \ldots, \ell_n \Rightarrow t_n)$ where the root variable's sort symbol $s \in \mathcal{S}$ and is called the ψ-term's *principal* type, the ℓ_i's are mutually distinct attribute symbols in $\mathcal{F}$, and the t_i's are ψ-terms ($n \geq 0$).

Variables capture coreference in a precise sense. They are coreference tags and may be viewed as typed variables where the type expressions are untagged ψ-terms. Hence, as a condition to be *well-formed*, a ψ-term must have all occurrences of each coreference tag consistently refer to the same structure. For example, the variable X in:

$$\begin{aligned} &\mathit{person}(\mathit{id} \Rightarrow \mathit{name}(\mathit{first} \Rightarrow \mathit{string}, \\ &\qquad\qquad \mathit{last} \Rightarrow X : \mathit{string}), \\ &\quad \mathit{father} \Rightarrow \mathit{person}(\mathit{id} \Rightarrow \mathit{name}(\mathit{last} \Rightarrow X : \mathit{string}))) \end{aligned}$$

refers consistently to the atomic ψ-term *string*. To simplify matters and avoid redundancy, we shall obey a simple convention of specifying the sort of a variable at most once and understand that other occurrences are equally referring to the same structure, as in:

$$\begin{aligned} &\mathit{person}(\mathit{id} \Rightarrow \mathit{name}(\mathit{first} \Rightarrow \mathit{string}, \\ &\qquad\qquad \mathit{last} \Rightarrow X : \mathit{string}), \\ &\quad \mathit{father} \Rightarrow \mathit{person}(\mathit{id} \Rightarrow \mathit{name}(\mathit{last} \Rightarrow X))) \end{aligned}$$

In fact, since there may be circular references as in $X : \mathit{person}(\mathit{spouse} \Rightarrow \mathit{person}(\mathit{spouse} \Rightarrow X))$, this convention is necessary. Finally, a variable appearing nowhere typed, as in $\mathit{junk}(\mathit{kind} \Rightarrow X)$ is implicitly typed by a special greatest initial sort symbol $\top$ always present in $\mathcal{S}$. This symbol will be left invisible and not written explicitly as in $(\mathit{age} \Rightarrow \mathit{integer}, \mathit{name} \Rightarrow \mathit{string})$, or written as the symbol @ as in $@(\mathit{age} \Rightarrow \mathit{integer}, \mathit{name} \Rightarrow \mathit{string})$. In the sequel, by ψ-term we shall always mean well-formed ψ-term and call such a form a *(ψ)-normal form*.

Generalizing first-order terms,[1] ψ-terms are ordered up to variable renaming. Given that the set $\mathcal{S}$ is partially-ordered (with a greatest element $\top$), its partial ordering is extended to the set of

[1] In fact, if a first-order term is written $f(t_1, \ldots, t_n)$, it is nothing other than syntactic sugar for the ψ-term

attributed ψ-terms. Informally, a ψ-term t_1 is subsumed by a ψ-term t_2 if (1) the principal type of t_1 is a subtype in $\mathcal{S}$ of the principal type of t_2; (2) all attributes of t_2 are also attributes of t_1 with ψ-terms which subsume their homologues in t_1; and, (3) all coreference constraints binding in t_2 must also be binding in t_1.

For example, if $student < person$ and $paris < cityname$ in $\mathcal{S}$ then the ψ-term:

$$\begin{aligned} &student(id \Rightarrow name(first \Rightarrow string,\\ &\qquad\qquad\qquad last \Rightarrow X : string),\\ &\quad lives\_at \Rightarrow Y : address(city \Rightarrow paris),\\ &\quad father \Rightarrow person(id \Rightarrow name(last \Rightarrow X),\\ &\qquad\qquad\qquad lives\_at \Rightarrow Y)) \end{aligned}$$

is subsumed by the ψ-term:

$$\begin{aligned} &person(id \Rightarrow name(last \Rightarrow X : string),\\ &\quad lives\_at \Rightarrow address(city \Rightarrow cityname),\\ &\quad father \Rightarrow person(id \Rightarrow name(last \Rightarrow X))). \end{aligned}$$

In fact, if the set $\mathcal{S}$ is such that *greatest lower bounds* (GLB's) exist for any pair of type symbols, then the subsumption ordering on ψ-term is also such that GLB's exist. Such are defined as the *unification* of two ψ-terms. A detailed unification algorithm for ψ-terms is given in [7].

Last in this brief introduction to the ψ-calculus, we explain type definitions. The concept is analogous to what a global store of constant definitions is in a practical functional programming language based on λ-calculus. The idea is that types in the signature may be specified to have attributes in addition to being partially-ordered. Inheritance of attributes from all supertypes to a subtype is done in accordance with ψ-term subsumption and unification. For example, given a simple signature for the specification of linear lists $\mathcal{S} = \{list, cons, nil\}$ with $nil < list$ and $cons < list$, it is yet possible to specify that *cons* has an attribute $tail \Rightarrow list$. We shall specify this as:

$$list := \{nil; cons(tail \Rightarrow list)\}.$$

From which the appropriate partial-ordering is inferred.

As in this *list* example, such type definitions may be recursive. Then, ψ-unification *modulo* such a type specification proceeds by unfolding type symbols according to their definitions. This is done by need as no expansion of symbols need be done in case of (1) failures due to order-theoretic clashes (**e.g.**, $cons(tail \Rightarrow list)$ unified with *nil* fails; **i.e.**, gives $\bot$); (2) symbol subsumption (**e.g.**, *cons* unified with *list* gives just *cons*), and (3) absence of attribute (**e.g.**, $cons(tail \Rightarrow cons)$ unified with *cons* gives $cons(tail \Rightarrow cons)$). Thus, attribute inheritance may be done "lazily," saving much unnecessary expansions [14].

In LIFE, a basic ψ-term denotes a functional application in FOOL's sense if its root symbol is a defined function. Thus, a *functional expression* is either a ψ-term or a conjunction of ψ-terms denoted by $t_1 : t_2 : \ldots : t_n$.[2] An example of such is $append(list, L) : list$, where *append* is the FOOL function defined as:

$$\begin{aligned} &list := \{[]; [@|list]\}.\\ &append([], L : list) \rightarrow L.\\ &append([H|T : list], L : list) \rightarrow [H|append(T, L)]. \end{aligned}$$

This is how functional dependency constraints are expressed in a ψ-term in LIFE. For example, in LIFE the ψ-term $foo(bar \Rightarrow X : list, baz \Rightarrow Y : list, fuz \Rightarrow append(X, Y) : list)$ is one in which the attribute *fuz* is derived as a list-valued function of the attributes *bar* and *baz*. Unifying such

$f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$.

[2] In fact, we propose to see the notation $\_ : \_$ simply as a dyadic operation resulting in the GLB of its arguments since, for example, the notation $X : t_1 : t_2$ is shorthand for $X : t_1, X : t_2$. Where the variable X is not necessary, (**i.e.**, not otherwise shared in the context), we may thus simply write $t_1 : t_2$.

ψ-terms proceeds as before modulo suspension of functional expressions whose arguments are not sufficiently refined to be provably subsumed by patterns of function definitions.

As for relational constraints on objects in LIFE, a ψ-term t may be followed by a *such-that* clause consisting of the logical conjunction of (relational) literals $C_1, \ldots, C_n$, possibly containing functional terms. It is written as $t \mid C_1, \ldots, C_n$. Unification of such relationally constrained terms is done modulo proving the conjoined constraints. In effect, this allows specifying *daemonic constraints* to be attached to objects. Such a (renamed) "daemon-constrained" object's specified relational and (equational) functional formula is normalized by LIFE, its proof being triggered by unification at the object's creation time.

Bibliography

[1] Hassan Aït-Kaci. **A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Types**. PhD thesis, University of Pennsylvania, Philadelphia, PA (1984).

[2] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. **Theoretical Computer Science**, 45:293–351 (1986).

[3] Hassan Aït-Kaci. **Warren's Abstract Machine, A Tutorial Reconstruction**. MIT Press, Cambridge, MA, 1991.

[4] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison (1993).

[5] Hassan Aït-Kaci and Patrick Lincoln. LIFE—a Natural Language for Natural Language. **T.A. Informations**, 30(1-2):37-67. Association pour le Traitement Automatique des Langues, Paris, France (1989).

[6] Hassan Aït-Kaci, Richard Meyer, and Peter Van Roy. Wild_LIFE, a user manual. PRL Technical Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

[7] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. **Journal of Logic Programming**, 3:185–215 (1986).

[8] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. **Lisp and Symbolic Computation**, 2:51–89 (1989).

[9] Hassan Aït-Kaci, Roger Nasr, and Patrick Lincoln. Le Fun: Logic, equations, and Functions. In **Proceedings of the Symposium on Logic Programming (San Francisco, CA)**, pages 17–23, Washington, DC (1987). IEEE, Computer Society Press.

[10] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1991). (Revised, November 1992).

[11] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of life. In Jan Maluszyński and Martin Wirsing, editors, **Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)**, pages 255–274. Springer-Verlag, LNCS 528 (August 1991). (Full paper to appear in the *Journal of Logic Programming*).

[12] Hassan Aït-Kaci and Andreas Podelski. Entailment and Disentailment of Order-Sorted Feature Constraints. In Andrei Voronkov, editor, **Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning (St. Petersburg, Russia)**. Springer-Verlag, LNCS (1993, to appear).

[13] Hassan Aït-Kaci and Andreas Podelski. Logic Programming with Functions over Order-Sorted Feature Terms. In E. Lamma and P. Mello, editors, **Proceedings of the 3rd International Workshop on Extensions of Logic Programming (Bologna, Italy)**. Springer-Verlag, LNAI 660 (1992).

[14] Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

[15] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In **Proceedings of the 5th International Conference on Fifth Generation Computer Systems**, pages 1012–1022, Tokyo, Japan (June 1992). ICOT. (Full paper to appear in **Theoretical Computer Science**).

[16] Joachim Niehren and Andreas Podelski. Feature automata and recognizable sets of feature trees. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, **Proceedings of the 4th International Joint Conference TAPSOFT'93: Theory and Practice of Software Development (Orsay, France)**, pages 356–375, Springer-Verlag LNCS 668 (1993).

[17] Joachim Niehren and Andreas Podelski and Ralf Treinen. Equational and Membership Constraints for Infinite Trees. In **Proceedings of the International Conference RTA'93: Rewriting Techniques and Applications** (1993, to appear).

[18] Andreas Podelski and Peter Van Roy. The Beauty and the Beast Algorithm: Testing Entailement and Disentailement Incrementally. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

[19] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. In **IEEE Computer** 25 (1), pages 54-68, Jan. 1992.

(questionnaire not submitted)

4.2 A Survey of Oz—A Higher-order Concurrent Constraint Language

Gert Smolka
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
smolka@dfki.uni-sb.de

Oz [7, 2] is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming. It builds on previous work in concurrent constraint programming [4, 5, 3] and advances the state of the art along the following directions:

- Oz is a higher-order language: there is no hard-wired distinction between program and query, every expression (possibly containing abstractions) can be abstracted, abstractions are first-class citizens. Oz's abstraction mechanism is novel in that it is fully compatible with first order constraints.
- Oz is a deep guard language. There is no restriction on the form of the guards appearing in Oz's choice combinators (conditional and disjunction). As a byproduct, Oz can express logically sound negation.
- Oz has a new asynchronous communication primitive, called constraint communication, avoiding the clumsiness of stream-based communication; constraint communication provides a minimal notion of state that is fully compatible with constraints.
- Oz provides records as logical data structure [1, 8].
- Oz comes with a powerful object system providing for the dynamic creation of concurrent objects. Object creation may involve multiple inheritance from already existing objects. Objects have persistent identity and encapsulated state. Objects are first-class citizens. The object system is written in Oz.
- Oz comes with a powerful window system providing for the interactive creation of graphical user interfaces. The window system is written in Oz's object system.

Oz is based on a simple calculus [6, 7] providing an operational semantics. The calculus fixes a class of expressions, a congruence on expressions, and a set of rules for rewriting expressions. An expression corresponds to a computation state, and rewriting with a rule corresponds to an abstract computation step. The choice of a rewriting step is don't care (that is, there is no backtracking). Termination of computation is defined as termination of rewriting. The calculus is parameterized with respect to a first-order constraint system. The congruence of the calculus is defined as a conservative extension of the logical equivalence on constraints. The calculus does not require an operational semantics for constraints (an implementation does, of course). The calculus accounts compositionally for dynamic creation of new and unique names.

Oz is implemented by means of a compiler and an abstract machine, written in C++. The programming environment is based on Emacs. It is interactive; at any time a new expression can be entered and will be executed concurrently with the already existing computations. We plan to have the implementation ready for ftp-based distribution in October 1993.

Bibliography

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1012–1021, Tokyo, Japan, 1992. ICOT. Full version will appear in Theoretical Computer Science.

[2] Martin Henz, Gert Smolka, and Jörg Würtz. Oz—a programming language for multi-agent systems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.

[3] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[4] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[5] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990.

[6] Gert Smolka. A calculus for higher-order concurrent constraint programming. Research report, DFKI, Saarbrücken, Germany, 1993. Forthcoming.

[7] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, DFKI, Saarbrücken, Germany, April 1993.

[8] Gert Smolka and Ralf Treinen. Records for logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254, Washington, USA, 1992. The MIT Press. Full version has appeared as Research Report RR-92-23, DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany.

(questionnaire not submitted)