$\mathcal{TDL}$–A Type Description Language for Unification-Based Grammars*

Hans-Ulrich Krieger, Ulrich Schäfer
{krieger, schaefer}@dfki.uni-sb.de

German Research Center for Artificial Intelligence (DFKI)
Universität des Saarlandes
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany

Abstract

Unification-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics. Their success stems from the fact that they can be seen as high-level declarative programming languages for linguists which allow them to express linguistic knowledge in a monotonic fashion. Moreover, such formalisms can be given a precise, set-theoretical semantics.
This paper presents $\mathcal{TDL}$, a typed feature-based language which is specifically designed to support highly lexicalized grammar theories like HPSG, FUG, or CUG. $\mathcal{TDL}$ 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. $\mathcal{TDL}$ 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. $\mathcal{TDL}$ is incremental, i.e., it allows the redefinition of types and the use of undefined types. Efficient reasoning is accomplished through specialized modules.

Topic Areas: Type and Feature Constraints, Disjunction and Negation, Type Hierarchies.

*We are grateful to the other *FoPra Brothers*, Stephan Diehl and Karsten Konrad, for their support and numerous help. This work has been supported by a research grant (ITW 9002 0) from the German Bundesministerium für Forschung und Technologie to the DISCO project of the DFKI.

1 Introduction

Over the last few years, unification-based (or more general: constraint-based) grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics.[1] Their success stems from the fact that they can be seen as a monotonic, high-level representation language for linguistic knowledge, where a dedicated parser/generator or a uniform type deduction mechanism acts as the inference engine. The main idea of representing as much linguistic knowledge as possible through a unique data type called *feature structures*, allows the integration of different description levels, starting with phonology and ending in pragmatics, therefore a feature structure directly serves as an *interface* between the different description stages, which can be accessed by a parser or a generator at the same time. In this context, *unification* is concerned with two different tasks: (i) *combining information* (unification is a structure-building operation), and (ii) *rejecting inconsistent knowledge* (unification determines the satisfiability).

While the first approaches relied on annotated phrase structure rules (for instance GPSG [Gazdar et al. 85] and PATR-II [Shieber et al. 83], as well as their successors CLE [Alshawi 92] and ELU [Russell et al. 92]), modern formalisms try to specify grammatical knowledge as well as lexicon entries merely through feature structures. In order to achieve this goal, one must enrich the expressive power of the first unification-based formalisms with different forms of *disjunctive descriptions* (atomic disjunctions, general disjunctions, distributed disjunctions etc.). Later, other operations came into play, viz., *(classical) negation* or *implication*. Full negation however can be seen as an input macro facility because it can be expressed through the use of disjunctions, negated coreferences, and negated atoms with the help of existential quantification as shown in [Smolka 88]. Other proposals considered the integration of *functional* and *relational dependencies* into the formalism which makes them Turing-complete in general.[2] However the most important extension to formalisms consists of the incorporation of *types*, for instance in modern systems like TFS [Zajac 92], CUF [Dörre & Eisele 91], or $\mathcal{TDL}$ [Krieger & Schäfer 93a]. Types are ordered *hierarchically* (via *subsumption*) as it is known from object-oriented programming languages. This leads to *multiple inheritance* in the description of linguistic entities (see [Daelemans et al. 92] for a comprehensive introduction). Finally, *recursive types* are necessary to describe at least phrase-structure recursion which is inherent in all grammar formalisms which are not provided with a *context-free backbone*.

Martin Kay was the first person who laid out a generalized linguistic framework, called *unification-based grammars*, by introducing the notions of *extension, unification*, and *generalization* into computational linguistics. Kays *Functional Grammar* [Kay 79] represents the first formalism in the unification paradigm and is the predecessor of strictly lexicalized approaches like FUG [Kay 85], HPSG [Pollard & Sag 87; Pollard & Sag 93] or UCG [Moens et al. 89]. Pereira and Shieber were the first to give a mathematical reconstruction of PATR-II in terms of a denotational semantics [Pereira & Shieber 84]. The work of Karttunen led to major extensions of PATR-II, concerning disjunction, atomic negation, and the use of cyclic structures [Karttunen 84]. Kasper and Rounds' seminal work is important in many respects: they clarified the connection between feature structures and finite automata, gave a logical characterization of the notion of disjunction, and presented for the first time complexity results (see [Kasper & Rounds 90] for a summary). Mark Johnson then enriched the descriptive apparatus with classical negation and showed that the feature calculus is a decidable subset of first-order predicate logic [Johnson 88]. Finally, Gert Smolka's work gave a fresh impetus to the whole field: his approach is distinguished from others in that he presents a sorted set-theoretical semantics for feature structures [Smolka 88; Smolka 89]. Moreover, Smolka gave solutions to problems concerning the complexity and decidability of feature structure descriptions. Paul King's work aims to reconstruct a special grammar theory, viz. HPSG, in mathematical terms [King 89], whereas the Backofen and Smolka's treatment is the

[1] [Shieber 86] and [Uszkoreit 88] give an excellent introduction to the field of unification-based grammar theories. [Pereira 87] makes the connection explicit between unification-based grammar formalisms and logic programming. [Knight 89] presents an overview to the different fields in computer science which make use of the notion of unification.

[2] For instance, Carpenter's ALE system [Carpenter 92a] gives a user the opportunity to define definite clauses, using disjunction, negation, and Prolog cut.

most general and complete one, bridging the gap between logic programming and unification-based grammar formalisms [Backofen & Smolka 92]. There exist only a few other proposals to feature structures nowadays which do not use standard first order logic directly, for instance Reape's approach, using a polymodal logic [Reape 91].

2 Motivation

Modern typed unification-based grammar formalisms (like TFS, CUF, or $\mathcal{TDL}$) differ from 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 (see [Aït-Kaci et al. 89]), where a GLB (greatest lower bound), LUB (least upper bound), or a $\preceq$ (type subsumption) computation reduces to low-level bit manipulation (see section 3.2). Moreover, types release untyped unification from expensive computation, e.g., 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. This can only be accomplished if the system makes a mechanism for type expansion available (see section 3.4).
- 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. Again, type expansion is necessary to determine the global consistency of a given description.
- 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* [Pereira 83] paradigm enforces a grammar writer to replace the context-free backbone through recursive types. Here, parameterized delayed type expansion is the ticket to the world of controlled linguistic deduction [Uszkoreit 91] (see section 3.4).

3 $\mathcal{TDL}$

$\mathcal{TDL}$ is a unification-based grammar development environment and run time system supporting HPSG-like grammars. Work on $\mathcal{TDL}$ has started at the end of 1988 in the DISCO project of the DFKI and led to $\mathcal{TDL}$*ExtraLight*, the predecessor of $\mathcal{TDL}$ [Krieger & Schäfer 93b]. The DISCO grammar currently consists of more than 700 type specifications written in $\mathcal{TDL}$ and is the largest HPSG grammar for German [Netter 93]. Grammars and lexicons written in $\mathcal{TDL}$ 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 [Kiefer 93]. The core machinery of DISCO consists of $\mathcal{TDL}$ (see below) and the feature constraint solver $\mathcal{UD}i\mathcal{N}e$ [Backofen & Weyers 93]. $\mathcal{UD}i\mathcal{N}e$ itself is a powerful untyped unification machinery which allows the use of distributed disjunctions, general

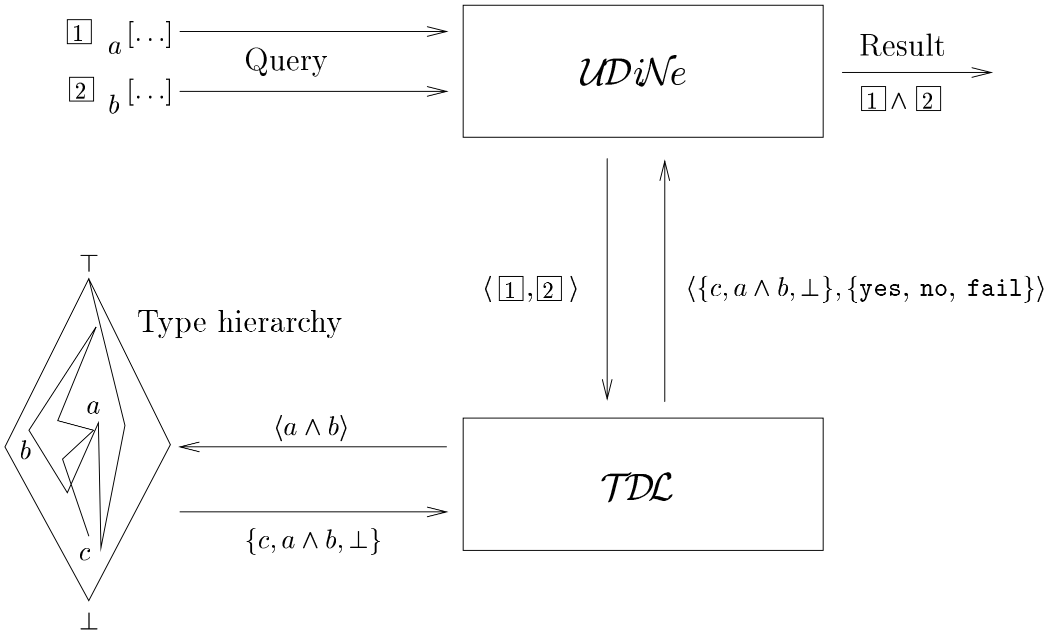

Figure 1: **Interface between $\mathcal{TDL}$ and $\mathcal{UDiNe}$.** *Depending on the type hierarchy and the type of* $\boxed{1}$ *and* $\boxed{2}$*,* $\mathcal{TDL}$ *either returns* c *(*c *is definitely the GLB of* a *and* b*) or* $a \wedge b$ *(open-world reasoning) resp.* $\bot$ *(closed-world reasoning) if there doesn't exist a single type which is equal to the GLB of* a *and* b*. In addition,* $\mathcal{TDL}$ *determines whether* $\mathcal{UDiNe}$ *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).*

negation, and functional dependencies. The modules communicate through an interface, and this communication mirrors exactly the way an abstract type 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 $\mathcal{UDiNe}$ handles over two typed feature structures to $\mathcal{TDL}$ which gives back a simplified form (plus additional information; see Fig. 1). 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 $\mathcal{TDL}$) 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.

We will now turn our focus to the main ingredients, $\mathcal{TDL}$ consists of (see Fig. 2). We start with a general overview of the language and then have a closer look on certain modules of the system.

3.1 $\mathcal{TDL}$ 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

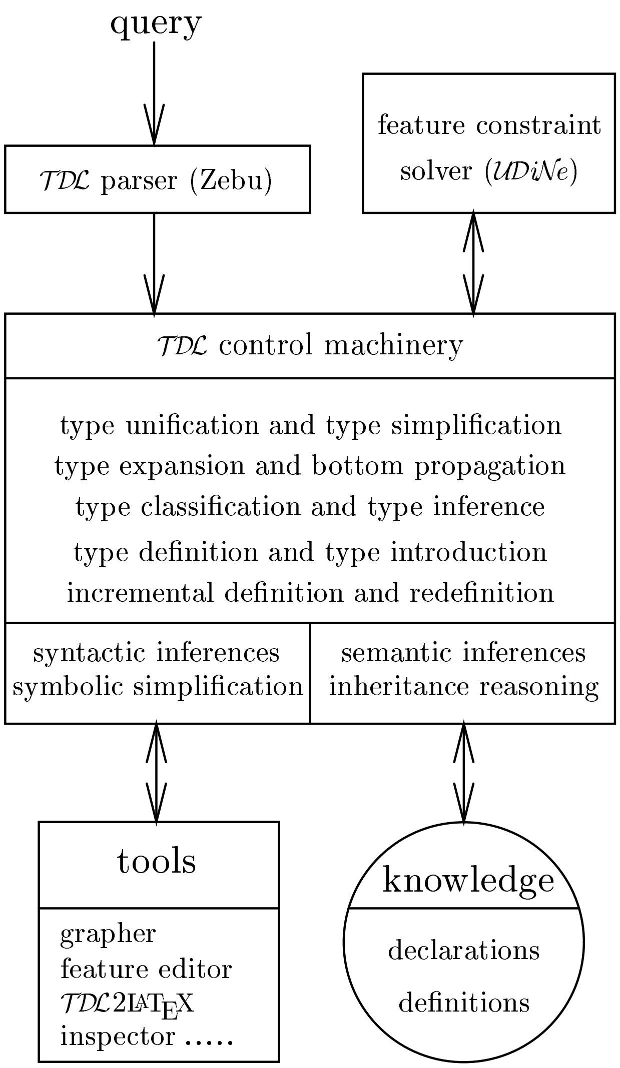

Figure 2: **Architecture of $\mathcal{TDL}$.** *The control machinery of $\mathcal{TDL}$ is called either by $\mathcal{UDiNe}$ during run time or by a user at definition time.*

sign. In addition, one can declare sets of types as *incompatible*, meaning that the conjunction of them yields $\perp$.

$\mathcal{TDL}$ 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 (see also [Pollard & Sag 87], p. 198). 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 [Laubsch 93] to allow for an intuitive, high-level input syntax and to abstract from uninteresting details imposed by the unifier and the underlying LISP system.

The kernel of $\mathcal{TDL}$ (and of most other monotonic systems) can be given a set-theoretical semantics along the lines of [Smolka 88]. It is easy to translate $\mathcal{TDL}$ statements into denotation-preserving expressions of Smolka's feature logic, thus viewing $\mathcal{TDL}$ only as syntactic sugar for a restricted (decidable) subset of first-order logic. Take for instance the following feature description ϕ written as an attribute-value matrix:

$$\phi = \begin{bmatrix} np & \\ \texttt{AGR} \ \boxed{\text{x}} & \begin{bmatrix} agreement \\ \texttt{NUM} \ sg \\ \texttt{PERS} \ \mathit{3rd} \end{bmatrix} \\ \texttt{SUBJ} \ \boxed{\text{x}} & \end{bmatrix}$$

It is not hard to rewrite this two-dimensional descriprtion to a flat first-order formula, where attributes/features (e.g., AGR) are interpreted as binary predicate symbols and sorts (e.g., *np*) as unary predicates:

$$\exists x \ . \ np(\phi) \wedge \texttt{AGR}(\phi, x) \wedge agreement(x) \wedge \texttt{NUM}(x, sg) \wedge \texttt{PERS}(x, \mathit{3rd}) \wedge \texttt{SUBJ}(\phi, x)$$

The corresponding $\mathcal{TDL}$ type definition of ϕ looks as follows (actually & is used on the keyboard instead of $\wedge$):

$$\begin{aligned} \phi := np \wedge [&\texttt{AGR} \ \#1 \wedge agreement \wedge [\texttt{NUM} \ sg, \texttt{PERS} \ \mathit{3rd}], \\ &\texttt{SUBJ} \ \#1]. \end{aligned}$$

3.2 Type Hierarchy

The type hierarchy is either called directly by the control machinery of $\mathcal{TDL}$ during the definition of a type (type classification) or indirectly via the simplifier both at definition and at run time (type unification).

3.2.1 Encoding Method

The implementation of the type hierarchy is based on Aït-Kaci's bit vector encoding technique for partial orders [Aït-Kaci et al. 89]. Every type t is assigned a code $\gamma(t)$ (represented through a bit vector) such that $\gamma(t)$ reflects the reflexive transitive closure of the subsumption relation with respect to t. Decoding a code c is realized either by a hash table look-up (iff $\exists t_c \ . \ \gamma^{-1}(c) = t_c$) or by computing the 'maximal restriction' of the set of types whose codes are less than c. Depending on the encoding method, the hierarchy occupies $O(n \log n)$ (compact encoding) resp. $O(n^2)$ (transitive closure encoding) bits. Here, GLB/LUB operations directly corresponds to bit-or/and instructions. GLB, LUB and $\preceq$ computations have the nice property that they can be carried out in this framework in $O(n)$ (resp. $O(1)$ on an ideal machine), where n is the number of types.[3]

The method has been modified to open-world reasoning over avm types in that potential GLB/LUB candidates (calculated from their codes), must be verified by inspecting the type hierarchy through a sophisticated graph search. Why so? Take the following example to see why this is necessary:

$$\begin{aligned} x &:= y \wedge z \\ x' &:= y' \wedge z' \wedge [a \ 1] \end{aligned}$$

During processing, one can definitely substitute $y \wedge z$ through x, but rewriting $y' \wedge z'$ to x' is not correct, because x' differ from $y' \wedge z'$—x' is more specific as a consequence of the feature constraint $a \doteq 1$. Therefore we made a distinction between the 'internal' greatest lower bound $\text{GLB}_{\preceq}$, concerning only the type subsumption relation by using Aït-Kaci's method alone (which is used for sort types) and the 'external' one $\text{GLB}_{\sqsubseteq}$ which takes the subsumption relation over feature structures into account. The same distinction is made for LUBs.

With $\text{GLB}_{\preceq}$ and $\text{GLB}_{\sqsubseteq}$ in mind, we can define a generalized GLB operation informally by the following table. This GLB operation is actually used during type unification.

[3]Actually, one can choose in $\mathcal{TDL}$ between the two encoding techniques and between bit vectors and bignums for the representation of the codes. Operations on bignums are a magnitude faster than the corresponding operations on bit vectors.

GLB	avm_1	$sort_1$	$atom_1$	$feat\_constr_1$
avm_2	see 1.	$\bot$	$\bot$	see 2.
$sort_2$	$\bot$	see 3.	see 4.	$\bot$
$atom_2$	$\bot$	see 4.	see 5.	$\bot$
$feat\_constr_2$	see 2.	$\bot$	$\bot$	see 6.

where

$$1.\ \begin{cases} avm_3 \iff avm_3 = \mathrm{GLB}_{\sqsubseteq}(avm_1, avm_2) \\ avm_1 \iff avm_1 \doteq avm_2 \\ \bot \iff \bot = \mathrm{GLB}_{\preceq}(avm_1, avm_2) \ \text{(through an explicit incompatibility declaration)} \\ avm_1 \wedge avm_2, \ \text{otherwise (open-world semantics)} \end{cases}$$

$$2.\ \begin{cases} avm_{1,2} \iff \text{expand-type}(avm_{1,2}) \sqcap feat\_constr_{2,1} \neq \bot \\ \bot, \ \text{otherwise} \end{cases}$$

$$3.\ \begin{cases} sort_3 \iff sort_3 = \mathrm{GLB}_{\preceq}(sort_1, sort_2) \\ sort_1 \iff sort_1 \doteq sort_2 \\ \bot, \ \text{otherwise (closed-world semantics)} \end{cases}$$

$$4.\ \begin{cases} atom_{1,2} \iff \text{type-of}(atom_{1,2}) \preceq sort_{2,1}, \ \ (sort_{2,1} \text{ is a built-in type}) \\ \bot, \ \text{otherwise} \end{cases}$$

$$5.\ \begin{cases} atom_1 \iff atom_1 \doteq atom_2 \\ \bot, \ \text{otherwise} \end{cases}$$

$$6.\ \begin{cases} \top \iff feat\_constr_1 \sqcap feat\_constr_2 \neq \bot \\ \bot, \ \text{otherwise} \end{cases}$$

The encoding algorithm is extended to cope with the redefinition of types and the use of undefined 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* (SCC) of t with respect to the dependency relation. It is important to redefine the dependents in the 'right' order to obtain a new consistent type hierarchy.[4]

3.2.2 Decomposing Type Definitions

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'.[5] This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunction elements, see Fig. 4). But it is worth noting that both of them express subsumption ($x \preceq y$ and $x' \succeq y'$ in the above example) and that the GLB/LUB operations must work properly over 'conjunctive' as well as 'disjunctive' subsumption links.

$\mathcal{TDL}$ 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 of type symbols. Intermediate type names are enclosed in vertical bars (cf. the intermediate types $|u \wedge v|$ and $|u \wedge v \wedge w|$ in Fig. 3).

[4]Enriching the type hierarchy with dependency links leads in general no longer to a cycle-free graph. So it is not obvious how to establish a topological order on the set of types. However, one can topologically sort the SCCs of the hierarchy without dependency links (which leads to a total order with respect to a certain SCC) and then implode the SCCs of the hierarchy into nodes which ultimately leads to a DAG which itself can be totally ordered.

[5]So one can see conjunctive types as *top-down specialization* of their supertypes and disjunctive ones as *bottom-up generalization* of their disjunction elements.

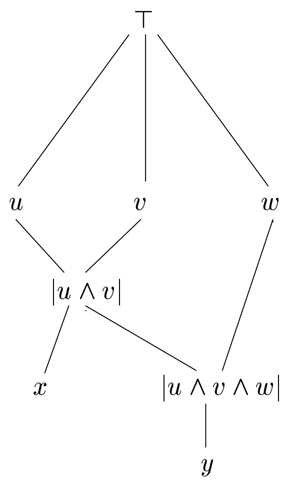

Figure 3: *The intermediate types* $|u{\wedge}v|$ *and* $|u{\wedge}v{\wedge}w|$ *are introduced during the type definitions* $x := u \wedge v \wedge [a\ 0]$ *and* $y := w \wedge v \wedge u \wedge [a\ 1]$.

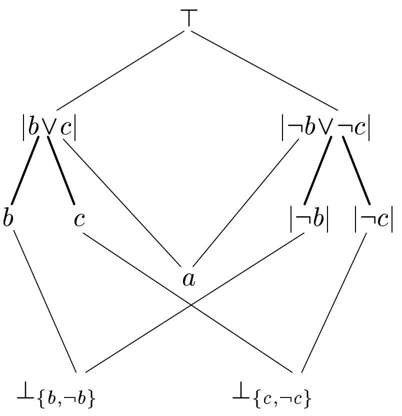

Figure 4: *Decomposing* $a := b \oplus c$*, such that* a *inherits from the intermediates* $|b{\vee}c|$ *and* $|\neg b{\vee}\neg c|$.

The same technique is applied when using $\oplus$ (see Fig. 4). $\oplus$ will be decomposed into $\wedge$, $\vee$ and $\neg$, plus additional intermediates. For each negated type $\neg t$, $\mathcal{TDL}$ introduces a new intermediate type symbol $|\neg t|$ with the definition $\neg t$ and declares it incompatible with t (see section 3.2.3). In addition, if t is not already present, $\mathcal{TDL}$ will add t as a new type to the hierarchy (see types $|\neg b|$ and $|\neg c|$ in Fig. 4).

Let's consider the example $a := b \oplus c$. The decomposition performed by $\mathcal{TDL}$ can then be stated informally by the following rewrite steps (assuming that CNF is switched on):

$$\begin{array}{c}
a := b \oplus c \\ \hline
a := (b \wedge \neg c) \vee (\neg b \wedge c) \\ \hline
a := (b \vee \neg b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg c \vee c) \\ \hline
a := (b \vee c) \wedge (\neg b \vee \neg c) \\ \hline
a := |b{\vee}c| \wedge |\neg b{\vee}\neg c|
\end{array}$$

where $|b{\vee}c| := b \vee c$, $|\neg b{\vee}\neg c| := |\neg b| \vee |\neg c|$, $|\neg b| := \neg b$, $|\neg c| := \neg c$, $\perp_{\{b,\neg b\}} := b \wedge |\neg b|$, *and* $\perp_{\{c,\neg c\}} := c \wedge |\neg c|$.

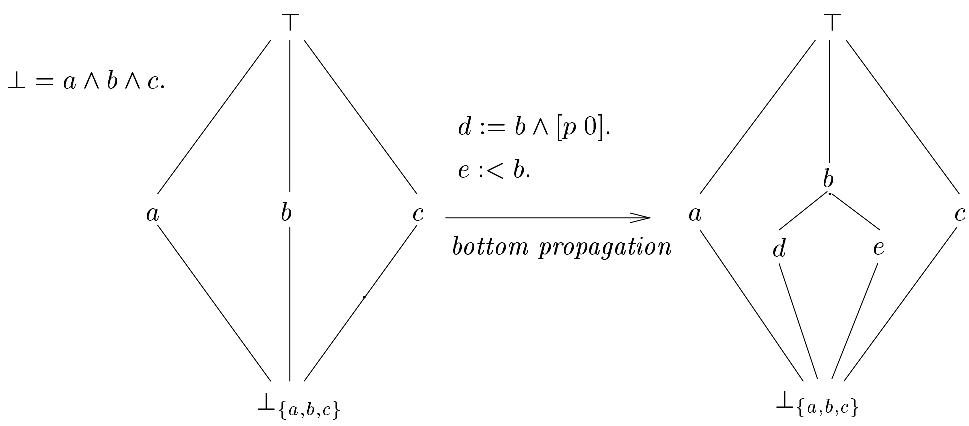

Figure 5: *Bottom propagation triggered through the subtypes d and e of b, so that $a \wedge d \wedge c$ as well as $a \wedge e \wedge c$ will simplify to $\perp$ during processing.*

If disjunctive normal form instead is enforced by the user, the decomposition of $a := b \oplus c$ leads of course to a different type hierarchy:

$$\frac{\dfrac{a := b \oplus c}{a := (b \wedge \neg c) \vee (\neg b \wedge c)}}{a := |b \wedge \neg c| \vee |\neg b \wedge c|}$$

where $|b \wedge \neg c| := b \wedge |\neg c|$, $|\neg b \wedge c| := |\neg b| \wedge c$, $|\neg c| := \neg c$, $|\neg b| := \neg b$, $\perp_{\{b, \neg b\}} := b \wedge |\neg b|$, *and* $\perp_{\{c, \neg c\}} := c \wedge |\neg c|$.

3.2.3 Incompatible Types and Bottom Propagation

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 4 and 5) which are however identified in the underlying logic (this is related to the construction of a *separated sum* in domain theory). These bottom symbols must be propagated downwards by a mechanism called *bottom propagation* which takes place at definition time (see Fig. 5). Note that it is important to take not only subtypes of incompatible types into account but also disjunction elements as the following example shows:

$$\left\{ \begin{array}{c} \perp = a \wedge b. \\ b := b_1 \vee b_2. \end{array} \right\} \xrightarrow{\textit{bottom propagation}} a \wedge b_1 = \perp \textit{ and } a \wedge b_2 = \perp$$

One might expect that incompatibility statements together with feature term unification lead no longer to a monotonic, set-theoretical semantics. But this is not the case. To preserve monotonicity, one must assume a *2-level interpretation* of typed feature structures, where feature constraints and type constraints can denote different sets of objects and the global interpretation is determined by the intersection of the two sets. Take for instance the type definitions $A := [a\ 1]$ and $B := [b\ 1]$, plus the user declaration $\perp = A \wedge B$ that A and B are incompatible. Then $A \wedge B$ will simplify to $\perp$ although the corresponding feature structures of A and B successfully unify to $[a\ 1,\ b\ 1]$.

3.3 Symbolic Simplifier

The simplifier operates on arbitrary $\mathcal{TDL}$ expressions. Simplification is done at definition time as well as at run time when typed unification takes place (cf. Fig. 1).

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. Consider an expression like $x_1 \wedge x_2 \ldots \wedge x_i \ldots \wedge \neg x_i \ldots \wedge x_n$. Symbolic simplification will detect $\perp$ 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 to speed up computation. Formally, type simplification in $\mathcal{TDL}$ can be characterized as a term rewriting system. A set of reduction rules is applied until a *normal form* is reached. Confluency and termination is guaranteed by imposing a *total generalized lexicographic order* on terms (either CNF or DNF). In addition, this order has the nice effects of neglecting 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 $\mathcal{TDL}$ to reuse precomputed results of simplified 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 $\preceq$).

3.3.1 Type Expressions

$\mathcal{TDL}$ type expressions are recursively defined as follows:[6]

- any type symbol is a valid type expression,
- any atom (a quoted symbol, a string or a number) is a valid type expression,
- if $t_1, \ldots, t_n$ are valid type expressions, the *conjunction* $t_1 \wedge \cdots \wedge t_n$ is a valid type expression ($n \geq 0$),
- if $t_1, \ldots, t_n$ are valid type expressions, the *disjunction* $t_1 \vee \cdots \vee t_n$ is a valid type expression ($n \geq 0$),
- if t is a valid type expression, its negation $\neg t$ is a valid type expression,
- nothing else is a type expression.

Symbols and negated symbols are also called *literals*.

3.3.2 Normal Form

In order to reduce an arbitrary type expression to a simpler expression, simplification rules must be applied. So we have to define what it means for an expression to be 'simple'. We choose the conjunctive (or disjunctive) normal form. A type expression is in *conjunctive normal form* (CNF), if it is a literal, or a conjunction of literals, or a conjunction of disjunctions of literals. The definition of *disjunctive normal form* (DNF) is the dual counterpart. In $\mathcal{TDL}$, the user may choose between CNF and DNF. The advantages of CNF/DNF are:

- UNIQUENESS
  Type expressions in normal form are unique modulo commutativity. Sorting type expressions according to a total lexicographic order will lead to a total uniqueness of type expressions (see section 3.3.4).

- LINEARITY
  Type expressions in normal form are linear. Any arbitrarily nested expression may be transformed into a 'flat' expression. This may reduce the complexity of later simplifications, e.g., at run time.

[6]For the sake of simplicity, we do not distinguish between *sort* and *avm types* here, both are subsumed by the notion of *type* in this section.

- COMPARABILITY
  This property is a consequence of the two properties mentioned before. Unique and linear expressions make it easy to find or compare (sub)expressions. This is important for the memoization technique described in section 3.3.6.

3.3.3 Reduction Rules

The current implementation of the simplifier uses the following hard-wired reduction rules (only the 'conjunctive' schemata are depicted—the dual set of disjunctive rules are applied as well):

dbl_neg
$$\frac{\neg\neg f}{f}$$

demorgan
$$\frac{\neg(f_1 \wedge \cdots \wedge f_n)}{\neg f_1 \vee \cdots \vee \neg f_n}$$

distrib
$$\frac{f_1 \wedge \cdots \wedge (g_1 \vee \cdots \vee g_m) \wedge \cdots \wedge f_n}{(g_1 \wedge f_1 \wedge \cdots \wedge f_n) \vee \cdots \vee (g_m \wedge f_1 \wedge \cdots \wedge f_n)}$$

flatten
$$\frac{f_1 \wedge \cdots \wedge (g_1 \wedge \cdots \wedge g_m) \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g_1 \wedge \cdots \wedge g_m \wedge \cdots \wedge f_n}$$

idempot
$$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge g \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge f_n}$$

absorpt
$$\frac{f_1 \wedge \cdots \wedge h \wedge \cdots \wedge (g_1 \vee \cdots \vee h \vee \cdots \vee g_m) \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge h \wedge \cdots \wedge f_n}$$

inverse1
$$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge \neg g \wedge \cdots \wedge f_n}{\bot}$$

inverse2
$$\frac{f_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge f_n}{\bot}$$

inverse3
$$\frac{\neg\top}{\bot}$$

neutr_el
$$\frac{f_1 \wedge \cdots \wedge \top \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge f_n}$$

identity
$$\frac{\bigwedge_{i=1}^{1} f_i}{f_1}$$

empty
$$\frac{\bigwedge_{i=1}^{0} f_i}{\top}$$

Note that only one of the two distributivity rules is applied depending on the chosen normal form (CNF or DNF). Otherwise simplification might not terminate.
In order to reach a normal form, it would suffice to apply only the rules dbl_neg, demorgan and distrib. But in the worst case, the application of these three rules would blow up the length of the normal form to exponential size (compared with the number of literals in the original expression). To avoid this, the other rules are used intermediately. If they can be applied, they always reduce the length of the (sub)expressions.

3.3.4 Lexicographic Order

In order to avoid the application of the commutativity rule, we introduce a lexicographic order on type expressions. Together with DNF/CNF, we get a unique sorted normal form for an arbitrary type expression. This guarantees confluency and fast comparability of type expressions.
First of all, we define the order $x <_{NF} y$ on n-ary normal forms by the following table, with symbol $<_{NF}$ neg_symbol $<_{NF}$ conjunction $<_{NF}$ disjunction:

$\downarrow x \quad y \rightarrow$	symbol	neg_symbol	conjunction	disjunction
symbol	$x <_{lex} y$	*true*	*true*	*true*
neg_symbol	*false*	$x_1 <_{lex} y_1$	*true*	*true*
conjunction	*false*	*false*	$\forall i : x_i <_{NF} y_i$	*true*
disjunction	*false*	*false*	*false*	$\forall i : x_i <_{NF} y_i$

where $1 \leq i \leq \max(|x|, |y|)$ and $<_{lex}$ is a total lexicographic order on strings (resp. symbol names), e.g. the predicate `STRING<` in COMMON LISP, for example:

$$a <_{NF} b <_{NF} bb <_{NF} \neg a <_{NF} a \wedge b <_{NF} a \wedge \neg a <_{NF} a \vee b <_{NF} a \vee b \vee c$$

We then extend $<_{NF}$ for atomic values, such that disjunction $<_{NF}$ atomic_symbol $<_{NF}$ string $<_{NF}$ number. The following matrix is the continuation of the table above at its lower right corner ($1 \leq i \leq \max(|x|, |y|)$):

$\downarrow x \quad y \rightarrow$	disjunction	atomic_symbol	string	number
disjunction	$\forall i : x_i <_{NF} y_i$	*true*	*true*	*true*
atomic_symbol	*false*	$x <_{lex} y$	*true*	*true*
string	*false*	*false*	$x_1 <_{lex} y_1$	*true*
number	*false*	*false*	*false*	$x < y$

3.3.5 Using Information from the Type Hierarchy

Especially at run time, but also at definition time, it is useful to exploit information from the type hierarchy. Further simplifications are possible by using the following rules (it is possible to switch off the use of type hierarchy information at any time). Again, the two dual disjunctive schemata are used as well (extension/LUB).

subsumption $$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge h \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge f_n}, \quad \textit{where } g \preceq h$$

GLB $$\frac{f_1 \wedge \cdots \wedge f_n}{g}, \quad \textit{where } g = \mathrm{GLB}(f_1, \ldots, f_n)$$

3.3.6 Memoization

The memoization technique described by [Norvig 91] has been adapted in order to reuse precomputed results of type simplification. There are four memoization hash tables for each $\mathcal{TDL}$ type domain: for CNF with/without hierarchy and DNF with/without hierarchy. The lexicographically sorted normal form described in section 3.3.4 guarantees fast access to precomputed type simplifications. Memoization results are also used by the recursive simplification algorithm to exploit precomputed results for subexpressions.
Some empirical results show the usefulness of memoization. The DISCO grammar for German consists of 750 types, 35 templates, and a toy lexicon of 170 instances/entries. After a full type

expansion of all instances, the memoization hashtables contain 4413 entries (literals are not memoized). 194 849 results have been reused at least once (some up to 1000 times) of which 84.6 % are proper simplifications (i.e., the simplified formulae are really shorter than the unsimplified formulae).

3.4 Type Expansion and Control

As we noted earlier, types allow us to refer to complex constraints through the use of a symbol name. In order to reconstruct the constraints which determine a type (represented as a feature structure), we require a complex operation called *type expansion*. This operation is comparable to Carpenter's *well-typedness* [Carpenter 92b].

3.4.1 Motivation

In $\mathcal{TDL}$, the motivation for type expansion is manifold:

- CONSISTENCY
  The global consistency/satisfiability of a type expression can in general only be decided through type expansion. At definition time, type expansion determines whether a type definition is consistent. At run time, type expansion is involved in checking the consistency of the unification of two typed feature structures.

- ECONOMY
  From the standpoint of efficiency, it does make sense to work only with small, partially expanded structures (if possible) to speed up feature term unification and to reduce the amount of copying. At last however, at the end of processing, one has to make the result (the constraints) explicit.

- RECURSION
  Recursive types are inherently present in modern constraint-based grammar formalisms like HPSG which are not provided with a context-free backbone. Moreover, if the formalism does not allow functional or relational constraints, one must specify certain functions/relations like *append* through recursive types. Take for instance Aït-Kaci's version of *append* [Aït-Kaci 86] which can be stated in $\mathcal{TDL}$ as follows:

$$
\begin{array}{lll}
\textit{append}_0 & := & [\,\mathtt{FRONT}\ <\,>, \\
 & & \;\;\mathtt{BACK}\ \#1 \wedge \textit{list}, \\
 & & \;\;\mathtt{WHOLE}\ \#1\,]. \\
\textit{append}_1 & := & [\,\mathtt{FRONT}\ <\#\textit{first}\,.\,\#\textit{rest1}>, \\
 & & \;\;\mathtt{BACK}\ \#\textit{back} \wedge \textit{list}, \\
 & & \;\;\mathtt{WHOLE}\ <\#\textit{first}\,.\,\#\textit{rest2}>, \\
 & & \;\;\mathtt{PATCH}\ \textit{append} \wedge [\,\mathtt{FRONT}\ \#\textit{rest}1, \\
 & & \qquad\qquad\qquad\quad \mathtt{BACK}\ \#\textit{back}, \\
 & & \qquad\qquad\qquad\quad \mathtt{WHOLE}\ \#\textit{rest}2\,]\,]. \\
\textit{append} & := & \textit{append}_0 \vee \textit{append}_1\,.
\end{array}
$$

- TYPE DEDUCTION
  Parsing and generation can be seen in the light of type deduction as a uniform process, where only the phonology (for parsing) or the semantics (for generation) must be given as

the following simplified example illustrates:

$$\textbf{Parsing:} \quad \begin{bmatrix} \textit{phrase} \\ \texttt{PHON} \ \langle \ \textit{“John”} \ \textit{“likes”} \ \textit{“bagels”} \ \rangle \end{bmatrix}$$

$$\textbf{Generation:} \quad \begin{bmatrix} \textit{phrase} & \\ \texttt{SEM} & \begin{bmatrix} \texttt{RELN} \ \textit{like} \\ \texttt{ARG1|IND|RESTR|NAME} \ \textit{john} \\ \texttt{ARG2|IND|RESTR|RELN} \ \textit{bagel} \end{bmatrix} \end{bmatrix}$$

Type expansion (together with a sufficient specified grammar) then is responsible (in both cases) for constructing a fully specified feature structure which is maximal informative and compatible with the input structure. However, the system TFS has shown that type expansion without sophisticated control strategies is hopelessly inefficient and moreover does not guarantee termination.

3.4.2 Controlled Type Expansion

Uszkoreit introduced in [Uszkoreit 91] a new strategy for linguistic processing called *controlled linguistic deduction*. His approach permits the specification of linguistic performance models without giving up the declarative basis of linguistic competence, especially monotonicity and completeness. 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.

$\mathcal{TDL}$ will support this strategy in that every feature structure is associated with its success/failure potential such that type expansion can be sensitive to these settings. Moreover, one can make other decisions as well during type expansion.

- use the failure/success probabilities or not
- stick to breadth-first or depth-first type expansion
- only regard structures which are subsumed by a given type
- take into account only structures under certain paths
- set the depth of type expansion for a given type

Note that we are not restricted to apply only one of these settings—they can be used in combination and can be changed dynamically during processing. Some of these software switches have been realized in the current implementation of $\mathcal{TDL}$'s type expansion mechanism. The next version will incorporate all of them and will be integrated into a declarative specification language which allows linguists to define *control knowledge* that can be used during processing.

References

[Aït-Kaci et al. 89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. *Efficient Implementation of Lattice Operations*. ACM Transactions on Programming Languages and Systems, 11(1):115–146, January 1989.

[Aït-Kaci 86] Hassan Aït-Kaci. *An Algebraic Semantics Approach to the Effective Resolution of Type Equations*. Theoretical Computer Science, 45:293–351, 1986.

[Alshawi 92] Hiyan Alshawi (ed.). *The Core Language Engine*. ACL-MIT Press Series in Natural Language Processing. MIT Press, 1992.

[Backofen & Smolka 92] Rolf Backofen and Gert Smolka. *A Complete and Recursive Feature Theory.* Technical Report RR-92-30, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1992.

[Backofen & Weyers 93] Rolf Backofen and Christoph Weyers. *UDiNe—A Feature Constraint Solver with Distributed Disjunction and Classical Negation.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Carpenter 92a] Bob Carpenter. *ALE—The Attribute Logic Engine User's Guide. Version β.* Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, December 1992.

[Carpenter 92b] Bob Carpenter. *The Logic of Typed Feature Structures.* Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press, 1992.

[Daelemans et al. 92] Walter Daelemans, Koenraad De Smedt, and Gerald Gazdar. *Inheritance in Natural Language Processing.* Computational Linguistics, 18(2):205–218, 1992.

[Dörre & Eisele 91] Jochen Dörre and Andreas Eisele. *A Comprehensive Unification-Based Grammar Formalism.* Technical Report Deliverable R3.1.B, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Gazdar et al. 85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, 1985.

[Johnson 88] Mark Johnson. *Attribute Value Logic and the Theory of Grammar.* CSLI Lecture Notes, Number 16. Stanford: Center for the Study of Language and Information, 1988.

[Karttunen 84] Lauri Karttunen. *Features and Values.* In: Proceedings of the 10th International Conference on Computational Linguistics, COLING-84, pp. 28–33, 1984.

[Kasper & Rounds 90] Robert T. Kasper and William C. Rounds. *The Logic of Unification in Grammar.* Linguistics and Philosophy, 13:35–58, 1990.

[Kay 79] Martin Kay. *Functional Grammar.* In: C. Chiarello et al. (ed.), Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society, pp. 142–158, Berkeley, Cal, 1979.

[Kay 85] Martin Kay. *Parsing in Functional Unification Grammar.* In: David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky (eds.), Natural Language Parsing. Psychological, Computational, and Theoretical Perspectives, chapter 7, pp. 251–278. Cambridge: Cambridge University Press, 1985.

[Kiefer 93] Bernd Kiefer. *Gimmie more HQ Parsers.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[King 89] Paul J. King. *A Logical Formalism for Head-Driven Phrase Structure Grammar.* PhD thesis, University of Manchester, Department of Mathematics, 1989.

[Knight 89] Kevin Knight. *Unification: A Multidisciplinary Survey.* ACM Computing Surveys, 21(1):93–124, March 1989.

[Krieger & Schäfer 93a] Hans-Ulrich Krieger and Ulrich Schäfer. *$\mathcal{TDL}$—A Type Description Language for HPSG. Part 2: System Description.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Krieger & Schäfer 93b] Hans-Ulrich Krieger and Ulrich Schäfer. *$\mathcal{TDL}$ExtraLight User's Guide.* Technical Report D-93-09, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993.

[Laubsch 93] Joachim Laubsch. *Zebu: A Tool for Specifying Reversible LALR(1) Parsers.* Technical report, Hewlett-Packard, 1993.

[Moens et al. 89] Marc Moens, Jo Calder, Ewan Klein, Mike Reape, and Henk Zeevat. *Expressing generalizations in unification-based grammar formalisms.* In: Proceedings of the 4th EACL, pp. 174–181, 1989.

[Netter 93] Klaus Netter. *Architecture and Coverage of the DISCO Grammar.* In: S. Busemann and Karin Harbusch (eds.), Proceedings of the DFKI Workshop on Natural Language Systems: Modularity and Re-Usability, 1993.

[Norvig 91] Peter Norvig. *Techniques for Automatic Memoization with Applications to Context-Free Parsing.* Computational Linguistics, 17(1):91–98, 1991.

[Pereira & Shieber 84] Fernando C.N. Pereira and Stuart M. Shieber. *The Semantics of Grammar Formalisms Seen as Computer Languages.* In: Proceedings of the 10th International Conference on Computational Linguistics, pp. 123–129, 1984.

[Pereira 83] Fernando C.N. Pereira. *Parsing as Deduction.* In: Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, pp. 137–144, 1983.

[Pereira 87] Fernando C.N. Pereira. *Grammars and Logics of Partial Information.* In: J.-L. Lassez (ed.), Proceedings of the 4th International Conference on Logic Programming, Vol. 2, pp. 989–1013, 1987.

[Pollard & Sag 87] Carl Pollard and Ivan Sag. *Information-Based Syntax and Semantics. Vol. I: Fundamentals.* CSLI Lecture Notes, Number 13. Stanford: Center for the Study of Language and Information, 1987.

[Pollard & Sag 93] Carl Pollard and Ivan Sag. *Head-Driven Phrase Structure Grammar.* CSLI Lecture Notes. Stanford: Center for the Study of Language and Information, 1993.

[Reape 91] Mike Reape. *An Introduction to the Semantics of Unification-Based Grammar Formalisms.* Technical Report Deliverable R3.2.A, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Russell et al. 92] Graham Russell, Afzal Ballim, John Carroll, and Susan Warwick-Armstrong. *A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons.* Computational Linguistics, 18(3):311–337, 1992.

[Shieber et al. 83] Stuart Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson. *The Formalism and Implementation of PATR-II.* In: Barbara J. Grosz and Mark E. Stickel (eds.), Research on Interactive Acquisition and Use of Knowledge, pp. 39–79. Menlo Park, Cal.: AI Center, SRI International, 1983.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes, Number 4. Stanford: Center for the Study of Language and Information, 1986.

[Smolka 88] Gert Smolka. *A Feature Logic with Subsorts.* LILOG Report 33, WT LILOG–IBM Germany, Stuttgart, Mai 1988.

[Smolka 89] Gert Smolka. *Feature Constraint Logic for Unification Grammars.* IWBS Report 93, IWBS–IBM Germany, Stuttgart, November 1989.

[Uszkoreit 88] Hans Uszkoreit. *From Feature Bundles to Abstract Data Types: New Directions in the Representation and Processing of Linguistic Knowledge.* In: A. Blaser (ed.), Natural Language at the Computer—Contributions to Syntax and Semantics for Text Processing and Man-Machine Translation, pp. 31–64. Berlin: Springer, 1988.

[Uszkoreit 91] Hans Uszkoreit. *Strategies for Adding Control Information to Declarative Grammars.* In: Proceedings of the 29th Meeting of the ACL, pp. 237–245, 1991.

[Zajac 92] Rémi Zajac. *Inheritance and Constraint-Based Grammar Formalisms.* Computational Linguistics, 18(2):159–182, 1992.