**Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH**

**Research
Report**

RR-94-37

TDL—A Type Description Language for HPSG Part 1: Overview

Hans-Ulrich Krieger, Ulrich Schäfer

November 1994

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland

Director

TDL
A Type Description Language for HPSG
Part 1: Overview

Hans-Ulrich Krieger, Ulrich Schäfer

DFKI-RR-94-37

Parts of this paper have been published in:
Proceedings of the Workshop on “Neuere Entwicklungen der deklarativen KI-Programmierung, KI-93, Berlin, 1993.
Proceedings of the 15th International Conference on Computational Linguistics (COLING 94), Kyoto, 1994.

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-9002 0).

ISSN 0946-008X

TDL
A Type Description Language for HPSG
Part 1: Overview

Hans-Ulrich Krieger, Ulrich Schäfer
{krieger,schaefer}@dfki.uni-sb.de
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany

Abstract

Unification-based grammar formalisms have become the predominant paradigm in natural language processing (NLP) and computational linguistics (CL). 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 *TDL*, a typed feature-based language and inference system, which is specifically designed to support highly lexicalized grammar theories like HPSG, FUG, or CUG.[1] *TDL* allows the user to define (possibly recursive) hierarchically-ordered types, consisting of type constraints and feature constraints over the boolean connectives $\wedge$, $\vee$, and $\neg$. *TDL* distinguishes between *avm types* (open-world reasoning), *sort types* (closed-world reasoning), *built-in types* and *atoms*, and allows the declaration of partitions and incompatible types. Working with partially as well as with fully expanded types is possible, both at definition time and at run time. *TDL* is incremental, i.e., it allows the redefinition of types and the use of undefined types. Efficient reasoning is accomplished through four specialized reasoners.

[1] Although the title might suggest that our formalism only suits the needs of HPSG-based grammars, it is of wider applicability in that it allows for annotated CF grammars in the PATR/GPSG tradition as well as purely feature-based, FUG-style grammars.

Acknowledgements. We are grateful to the other *FoPra Brothers*, Stephan *Don* Diehl and Karsten *KaKo* Konrad, for their support and ingenious programs. We have also benefited from two anonymous COLING referees and from our colleagues at the DFKI, especially Rolf Backofen, Klaus Netter, Stephan Oepen, and Christoph Weyers, and from the reactions of audiences where we presented different parts of it, in particular at the EAGLES workshop on *Implemented Formalisms*, Saarbrücken; the workshop on *Implementations of Attribute-Value Logics for Grammar Formalisms* at the European Summer School in Language, Logic, and Information, Lisbon; the workshop on *Neuere Entwicklungen der deklarativen KI-Programmierung*, KI-93, Berlin; and the International Conference on Computational Linguistics, *COLING-94*, Kyoto. We would also like to thank Elizabeth Hinkelman for carefully reading several drafts of this paper and for making detailed suggestions.

Contents

1 Introduction

Over the last few years, unification-based (or more generally, constraint-based) grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics.[2] Their success stems from the fact that they can be seen as a monotonic, high-level representation language for linguistic knowledge, for which a parser/generator or a uniform type deduction mechanism acts as the inference engine. The representation of as much linguistic knowledge as possible through a unique data type called *feature structures* allows the integration of different description levels, from phonology to pragmatics. Here, the feature structure itself serves as an *interface* between the different description stages which can therefore be accessed at the same time. In this context, *unification* is concerned with two different tasks:

1. COMBINING INFORMATION
   unification is a structure-building operation

2. REJECTING INCONSISTENT KNOWLEDGE
   unification determines the satisfiability of a description

1.1 A Short History

Martin Kay was the first person to lay out a generalized linguistic framework, called *unification-based grammars*, by introducing the notions of *extension, unification*, and *generalization* into computational linguistics (see overview in [Rupp et al. 94] for a good introduction). Kay's *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 94] and 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].[3] 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 [Kasper & Rounds 86; Rounds & Kasper 86] is important in many respects: it clarified the connection between feature structures and finite automata, gave a logical characterization of the notion of disjunction, and presented complexity results for the first time (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 oth-

[2] [Shieber 86] and [Uszkoreit 88] are excellent introductions to unification-based grammar theories. [Keller 93] investigates different characterizations of feature logics and compares them. [Pereira 87] makes the connection between unification-based grammar formalisms and logic programming explicit. [Knight 89] gives an overview of the different fields in computer science which make use of the notion of unification.

[3] Pereira and Shieber's work was novel in that they made a distinction between descriptions and described objects, which seems to date back to the early work in LFG. Moreover, they presented a fixpoint semantics for PATR-II (actually, they chose the least fixpoint) where PATR-II grammars are interpreted in the rational tree domain.

ers 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 descriptions. Work by Rounds and Manaster-Ramer, however, showed that a Kasper-Rounds logic enriched with types (type definitions) leads to the undecidability of the satisfiability problem [Rounds & Manaster-Ramer 87]. Later, [Smolka 89] explained that the undecidability result is due to the use of coreference constraints. Paul King's work aimed to reconstruct a special grammar theory, viz., HPSG, in mathematical terms [King 89], whereas Backofen and Smolka's treatment bridged the gap between logic programming and unification-based grammar formalisms [Backofen & Smolka 92]. New work by Backofen investigates a very general feature theory which incorporates nearly all extensions of feature descriptions that have been proposed in the literature. This language is of course undecidable w.r.t. satisfiability, but Backofen presents several fragments of the language that show more desirable properties [Backofen 94]. There exist only a few other proposals to feature descriptions nowadays which do not use standard first order logic directly, for instance Reape's approach, using a polymodal logic [Reape 91] (see [Blackburn 94] for an overview).

1.2 Expressivity of Formalisms

While the first unification-based 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 entirely through feature structures.

In order to achieve this goal, one must enrich the expressive power of the early 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 formalisms which makes them Turing-complete in general.[4]

However the most important extension to formalisms consists in the incorporation of *types*, for instance in modern systems like TFS [Emele & Zajac 90; Zajac 92], CUF [Dörre & Eisele 91; Dörre & Dorna 93], or $\mathcal{TDL}$ [Krieger & Schäfer 93a; Krieger & Schäfer 94a; Krieger & Schäfer 94b].[5] Types are ordered *hierarchically* (via *subsumption*) as in object-oriented programming languages. This leads to *multiple inheritance* in the description of linguistic entities.[6]

Finally, if a formalism is intended to be used as a stand-alone system, it must implement *recursive types* if it does not provide phrase-structure recursion directly (within the formalism)

[4]For instance, Bob Carpenter's ALE system [Carpenter & Penn 94] gives a user the option of defining definite clauses, using disjunction, negation, and Prolog cut.

[5]Cf. [Backofen et al. 93] for a comprehensive overview of modern systems, including a detailed description of their features.

[6]See [Daelemans et al. 92] for a general introduction.

or indirectly (via a parser/generator).[7] In addition, certain forms of relations (like *append*) or additional extensions of the formalism (like functional uncertainty) can be nicely modelled through recursive types.

1.3 Overview of the Paper

In the next section, we argue for the need and relevance of using types in CL and NLP. After that, we give an overview of $\mathcal{TDL}$ and its specialized inference modules. In particular, we have a closer look at the novel features of $\mathcal{TDL}$ and present the techniques we employ in implementing $\mathcal{TDL}$.[8] We then compare $\mathcal{TDL}$ with other grammatical formalisms. Finally, we specify the concrete syntax of $\mathcal{TDL}$ in BNF and present a small, linguistically-motivated example written in $\mathcal{TDL}$.

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 emphasize the notion of a *feature type*. Types can be arranged hierarchically, where a subtype monotonically *inherits* all the information from its supertypes and unification plays the role of the primary information-combining operation.

An abstract *type definition* $s := \langle t, \phi \rangle$ in $\mathcal{TDL}$ can be seen as an abbreviation for a complex expression, consisting of type constraints t (concerning the sub-/supertype relationship) and feature constraints ϕ (stating the necessary features and their values) over the standard connectives $\wedge$, $\vee$, and $\neg$.[9] Note however that a feature structure might have other attributes not mentioned in the type definition as well. Thus a "definition" only states which attributes (and values) are required for a certain type. Informally, if

$$s := \langle t, \phi \rangle$$

is a type definition, the intended meaning is roughly the following implication:

$$\forall x \,.\, s(x) \rightarrow t(x) \wedge \phi(x)$$

Types are thus a necessary requirement for a grammar development environment because they serve as abbreviations for lexicon entries, immediate dominance rule schemata, and universal as well as language-specific principles as is familiar from HPSG.

Types not only serve as a shorthand, like templates, but also yield other advantages as well which cannot however be accomplished by templates:

[7] For instance, ALE employs a bottom-up chart parser, whereas TFS relies entirely on type deduction. Note that recursive types can be substituted by definite relations (equivalences), as is the case for CUF, such that parsing/generation roughly corresponds to SLD resolution.

[8] A more practice-oriented introduction to $\mathcal{TDL}$ is [Krieger & Schäfer 94a]. This document investigates different tools of $\mathcal{TDL}$, describes internal software switches, focusses on each construct of syntax (plus examples), and describes other well-worth noting features of $\mathcal{TDL}$, e.g., non-monotonic overwriting, templates, instance definition facility, etc.

[9] The concrete syntax for type definitions and declarations is given in the Appendix.

- STRUCTURING OF LINGUISTIC KNOWLEDGE
  Types together with the possibility to order them hierarchically allow for a modular way to represent linguistic knowledge adequately. Moreover, generalizations can be made at the appropriate levels of representation.

- EFFICIENT PROCESSING
  Certain type constraints can be compiled into more efficient representations, for instance, [Aït-Kaci et al. 89] reduces GLB (greatest lower bound), LUB (least upper bound), and $\preceq$ (type subsumption) computation to low-level bit manipulations; see Section 3.2. Moreover, types can be used to eliminate expensive unification operations, for example, by explicit declaration of type incompatibility. 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 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 forces a grammar writer to replace the context-free backbone through recursive types. Here, parameterized delayed type expansion is the key to controlled linguistic deduction [Uszkoreit 91]; see Section 3.4.

3 TDL

TDL is a unification-based grammar development environment and run time system supporting HPSG-style grammars. Work on TDL started at the end of 1988 in the DISCO project of the DFKI and led to *TDL ExtraLight*, the predecessor of TDL [Krieger & Schäfer 93b]. The DISCO grammar currently consists of more than 900 type specifications written in TDL and is the largest HPSG grammar for German [Netter 93].
Grammars and lexicons written in TDL can be tested by using the DISCO parser. The parser is a bidirectional bottom-up chart parser, providing a user with parameterized parsing strategies as well as control over the processing of individual rules [Kiefer & Scherf 94].
The core machinery of DISCO consists of TDL (see below) and the feature constraint solver UDiNe [Backofen & Weyers 94]. UDiNe itself is a powerful untyped unification machine which allows the use of distributed disjunctions, general 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 known to be compatible. This is accomplished by the unifier

in that $\mathcal{UDiNe}$ hands 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 specialized modules (which again might consist of specialized components as is the case in $\mathcal{TDL}$) is twofold: (i) this strategy reduces the complexity of the whole system, thus making the architecture clear, and (ii) leads to faster processing because every module is designed to handle only a specialized task.

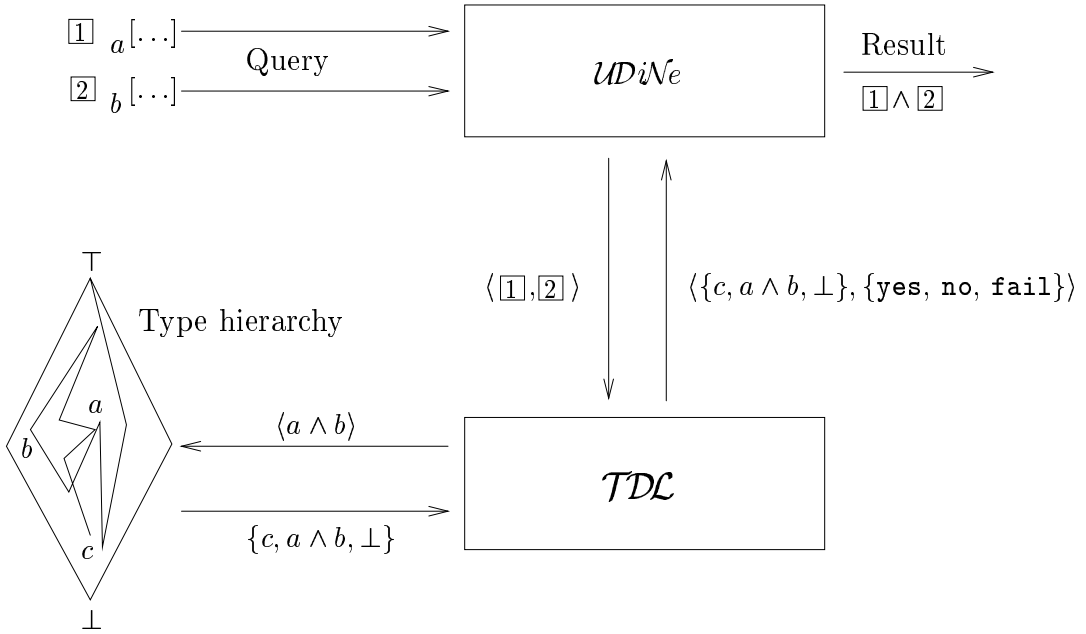

Figure 1: **Interface between $\mathcal{TDL}$ and $\mathcal{UDiNe}$.** *Depending on the type hierarchy and the type of 1 and 2, $\mathcal{TDL}$ either returns c (c is definitely the GLB of a and b) or $a \wedge b$ (open-world reasoning for GLB) or $\bot$ (closed-world reasoning for GLB) if a single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

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

3.1 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 to connect feature descriptions, coreference tags (logical variables) and types. $\mathcal{TDL}$ distinguishes between avm types (open-world semantics), sort types (closed-world semantics), built-in types (through Common Lisp), and atoms.
When asked 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 reasons for

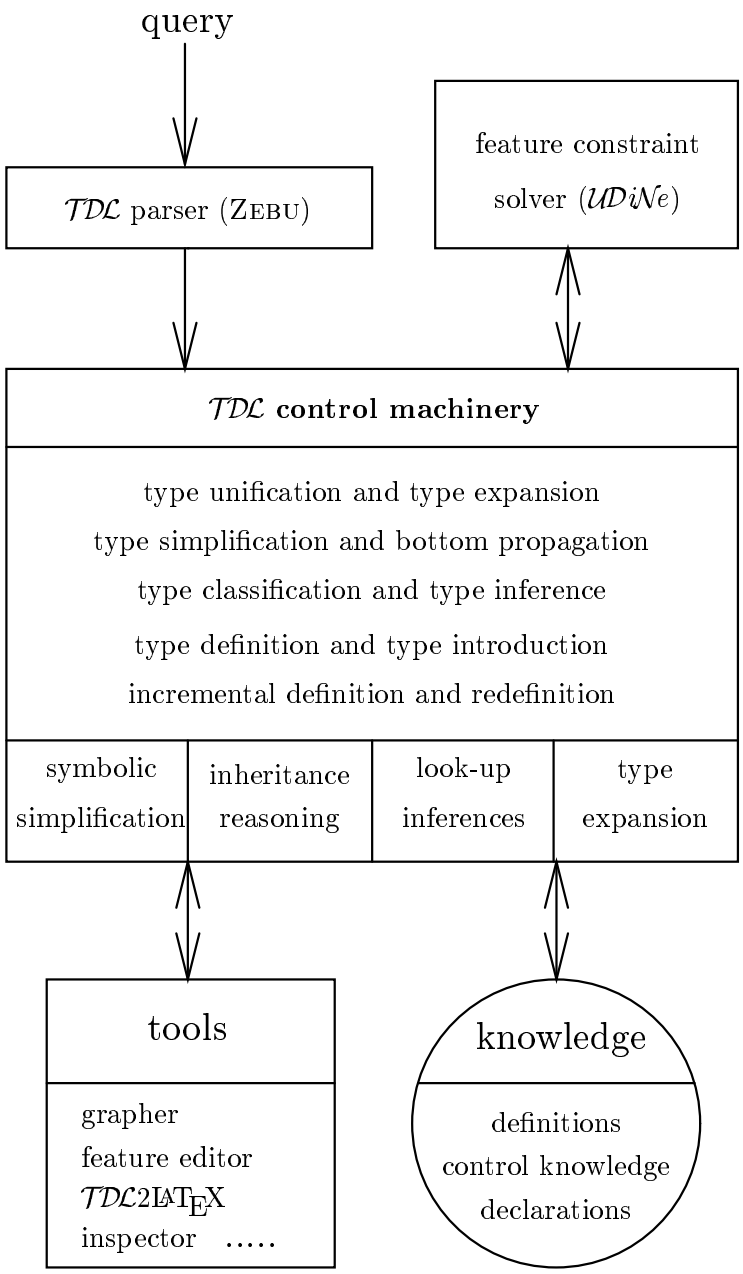

Figure 2: **Architecture of TDL.** *The control machinery of TDL is either called by the feature constraint solver at run time (typed unification) and during type expansion, or at definition time during incremental grammar/lexicon development. Moreover, the type expansion module can be called by other higher-level reasoners, e.g., a parser.*

assuming this are manifold:

1. partiality of our linguistic knowledge about a specific domain
2. the approach is in harmony with terminological (KL-ONE-like) languages which share a similar semantics
3. this view makes the stepwise refinement of grammars during the development process easier (which has been shown useful in our project)
4. we must not write superfluous type definitions to guarantee successful type unifications during processing

The opposite case holds for the GLB of sort types. Furthermore, sort types differ from avm types in that they are not further structured, as is the case for atoms.
Moreover, TDL allows the declaration of *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 their conjunction yields $\perp$.
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 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 TDL is parsed by a Zebu-generated LALR(1) parser [Laubsch 93] to allow for an intuitive, high-level input syntax and to abstract away from uninteresting details of the unifier and the underlying LISP system.

The kernel of TDL (and of most other monotonic systems) can be given a set-theoretical semantics along the lines of [Smolka 88; Smolka 89]. It is easy to translate TDL statements into denotation-preserving expressions of Smolka's feature logic (or into definite equivalences), thus viewing TDL as just syntactic sugar for first-order predicate logic.[10]
For instance, take the following feature description ϕ written as an attribute-value matrix:

$$\phi = \left[\begin{array}{ll} np & \\ \texttt{AGR}\ \boxed{\texttt{x}} & \left[\begin{array}{l} agreement \\ \texttt{NUM}\ sg \\ \texttt{PERS}\ 3rd \end{array}\right] \\ \texttt{SUBJ}\ \boxed{\texttt{x}} & \end{array}\right]$$

[10] Cf. [Krieger 95] for a precise description of the semantics of TDL, including a fixpoint characterization of recursive types.

It is not hard to rewrite this two-dimensional description 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:

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

The corresponding TDL type definition of ϕ looks as follows—actually & is used on the keyboard instead of $\wedge$, | replaces $\vee$, and $\neg$ is substituted by $\sim$):

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

3.2 Type Hierarchy

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

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. 85; 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) or $O(n^2)$ (transitive closure encoding) bits, resp. Here, GLB/LUB operations corresponds directly to bitwise or/and instructions. GLB, LUB and $\preceq$ computations have the pleasant property that they can be carried out in this framework in $O(n)$ (or $O(1)$ on an ideal machine), where n is the number of types.[11]

The method has been modified for an open-world reasoning over avm types, in that potential GLB/LUB candidates (calculated from their codes) are 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 $y \wedge z$ by x, but rewriting $y' \wedge z'$ to x' is not correct, because x' differs from $y' \wedge z'$—x' is more specific as a consequence of the feature constraint $[a\ 1]$. Thus the implementation distinguishes between

[11] Actually, one can choose in TDL between the two encoding techniques and between bit vectors and bignums (arbitrary long integers) for the representation of the codes. Operations on bignums are an order of magnitude faster than the corresponding operations on bit vectors.

- INTERNAL GREATEST LOWER BOUND $\mathrm{GLB}_{\preceq}$
  employ the type subsumption relation via Aït-Kaci's method (used in case of sort types)
- EXTERNAL GREATEST LOWER BOUND $\mathrm{GLB}_{\sqsubseteq}$
  take the subsumption relation over feature structures into account.

The same distinction is made for LUBs.

With $\mathrm{GLB}_{\preceq}$ and $\mathrm{GLB}_{\sqsubseteq}$ in mind, we can define a generalized GLB operation informally by the following table. This GLB operation is actually used during type unification (fc = feature constraint):

GLB	avm_1	$sort_1$	$atom_1$	fc_1
avm_2	see 1.	$\bot$	$\bot$	see 2.
$sort_2$	$\bot$	see 3.	see 4.	$\bot$
$atom_2$	$\bot$	see 4.	see 5.	$\bot$
fc_2	see 2.	$\bot$	$\bot$	see 6.

where

1. $\begin{cases} avm_3 \Longleftrightarrow \mathrm{GLB}_{\sqsubseteq}(avm_1, avm_2) = avm_3 \\ avm_1 \Longleftrightarrow avm_1 = avm_2 \\ \bot \Longleftrightarrow \mathrm{GLB}_{\preceq}(avm_1, avm_2) = \bot \text{ (via an explicit incompatibility declaration)} \\ avm_1 \wedge avm_2, \text{ otherwise (open world reasoning for GLB)} \end{cases}$

2. $\begin{cases} avm_{1,2} \Longleftrightarrow \mathit{expand\text{–}tfs}(\langle avm_{1,2}, fc_{2,1}\rangle) \neq \bot \text{ (type expansion switched on)} \\ avm_{1,2} \Longleftrightarrow \text{type expansion is switched off} \\ \bot, \text{ otherwise} \end{cases}$

3. $\begin{cases} sort_3 \Longleftrightarrow \mathrm{GLB}_{\preceq}(sort_1, sort_2) = sort_3 \\ sort_1 \Longleftrightarrow sort_1 = sort_2 \\ \bot, \text{ otherwise (closed world reasoning for GLB)} \end{cases}$

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

5. $\begin{cases} atom_1 \Longleftrightarrow atom_1 = atom_2 \\ \bot, \text{ otherwise} \end{cases}$

6. $\begin{cases} \top \Longleftrightarrow fc_1 \wedge fc_2 \neq \bot \\ \bot, \text{ otherwise} \end{cases}$

The encoding algorithm has been 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 not only means to make changes local to this type. Rather, 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.[12]

3.2.2 Decomposing Type Definitions

Conjunctive type specifications (e.g., $x := y \wedge z$) and disjunctive ones (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 disjunction alternatives y' and z'.[13] This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunction elements, see Fig. 4 and 5). 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.

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 (plus type definitions of the form $s := \neg t$). Intermediate type names are enclosed in vertical bars (cf. the intermediate types $|u \wedge v|$ and $|u \wedge v \wedge w|$ in Fig. 3).

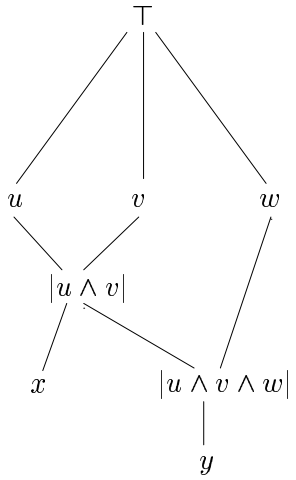

Figure 3: *The intermediate types* $|u \wedge v|$ *and* $|u \wedge v \wedge w|$ *are introduced during the definition of the types* $x := u \wedge v \wedge [a\ 0]$ *and* $y := w \wedge v \wedge u \wedge [a\ 1]$.

[12] In general, enriching the type hierarchy with dependency links no longer leads 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).

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

The same technique is applied when using $\oplus$ (see Fig. 4 and 5). $\oplus$ 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 and 5).

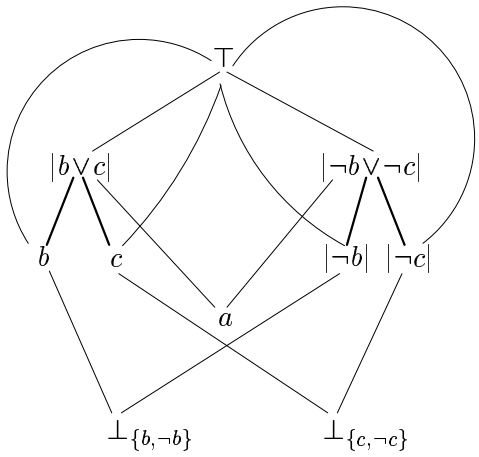

Figure 4: *Decomposing $a := b \oplus c$ into conjunctive normal form, such that a inherits from the intermediates $|b \vee c|$ and $|\neg b \vee \neg c|$.*

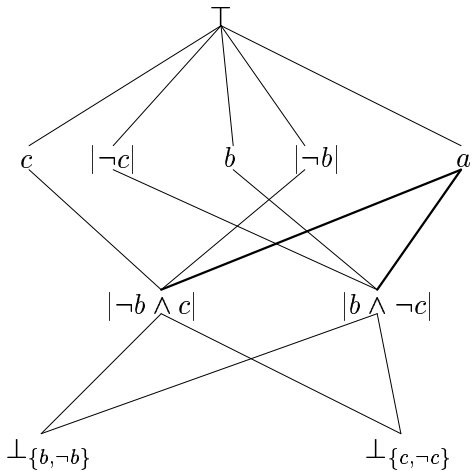

Figure 5: *Decomposing $a := b \oplus c$ into disjunctive normal form, such that a is defined through its disjunction alternatives $|b \wedge \neg c|$ and $|\neg b \wedge c|$.*

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 mode is switched on; see Fig. 4):

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

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

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 (Fig. 5):

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

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

3.2.3 Incompatible Types and Bottom Propagation

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 4, 5 and 6) which are, however, identified in the underlying logic (this identification is somewhat related to the notion of a *coalesced sum*, known from domain theory). I.e., these symbols are always interpreted as representing inconsistent information, thus they denote the empty set. Bottom symbols must be propagated downwards by a mechanism called *bottom propagation* which takes place at definition time (see Fig. 6). Note that it is important to take not only subtypes of incompatible types into account but also disjunction elements, as the following example shows:

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

It is worth noting that because we employ an explicitly represented type hierarchy within GLB, LUB and $\preceq$ computations, a single bottom symbol that is a subtype of every other type, would lead to false inferences. Consider the following example. Assume that we declare a and b, as well as c and d as incompatible. If only a single bottom symbol $\bot$ is used, we would deduce that $a \wedge c$ is $\bot$ which however is not necessarily the case. However, introducing two bottom symbols $\bot_{\{a,b\}}$ and $\bot_{\{c,d\}}$ is the right way to guarantee proper results.

One might expect that incompatibility statements together with feature term unification no longer lead 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

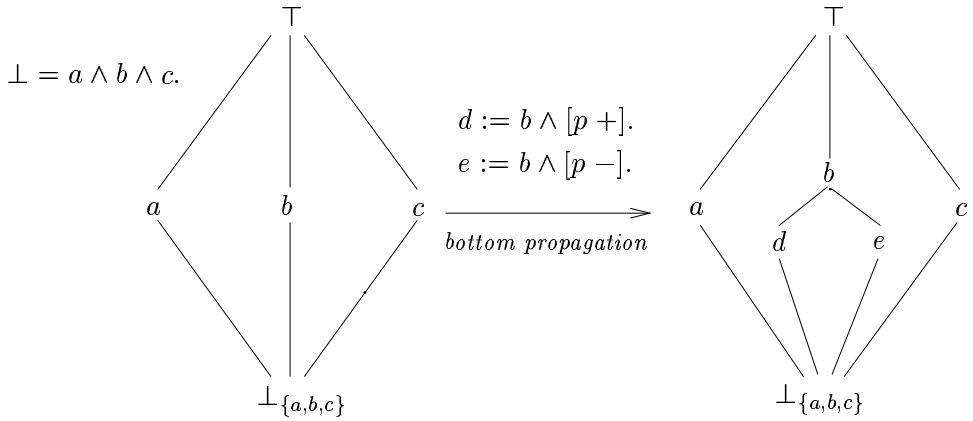

Figure 6: *Bottom propagation triggered through the subtypes d and e of b, so that $a \wedge d \wedge c$ as well as $a \wedge e \wedge c$ will simplify to $\bot$ during processing.*

interpretation is determined by the intersection of the two sets (cf. [Krieger 95] for a thorough investigation). Take for instance the type definitions $A := [a\ 1]$ and $B := [b\ 1]$, plus the user declaration $\bot = A \wedge B$, viz., that A and B are incompatible. Then $A \wedge B$ will simplify to $\bot$ 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 or type expansion 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 $\bot$ 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. (cf. Fig. 7). 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.
Confluence and termination are guaranteed by imposing a *total generalized lexicographic order* $<_{NF}$ on complex type expressions (either CNF or DNF). In addition, this order has the nice effect of neglecting the commutativity schemata (C) in Fig. 7 (which are expensive and might lead to termination problems): there is only one representative for a given formula. Therefore, *memoization* of type expressions is cheap (see Section 3.3.5) and is employed in $\mathcal{TDL}$ to reuse precomputed results of simplified formulae (one must not cover all permutations of a formula). Consider the conjunction $t_1 \wedge \cdots \wedge t_n$ for which $n!$ permutations exist. Now let π be a permutation, such that $t_{\pi(1)} <_{NF} \ldots <_{NF} t_{\pi(n)}$ is the case. Then $t_{\pi(1)} \wedge \cdots \wedge t_{\pi(n)}$ is the unique representative for all $n!$ permutations of $t_1 \wedge \cdots \wedge t_n$ (the exact definition of $<_{NF}$

is given below).
Additional reduction rules are applied at run time using "semantic" information from the type hierarchy (cf. Fig. 8 and 9).

3.3.1 Type Expressions

Formally, a signature for $\mathcal{TDL}$ contains disjoint sets for atoms (or constants) $\mathcal{A}$ and types $\mathcal{T}$, where $\mathcal{T} := \mathcal{T}_s \uplus \mathcal{T}_a \uplus \{\top, \bot\}$. $\mathcal{T}_s$ denotes the set of sort types and $\mathcal{T}_a$ the set of avm types. Furthermore, $\mathcal{T}_s := \mathcal{T}_b \uplus \mathcal{T}_u$ is subdivided into built-in sorts $\mathcal{T}_b$ and user defined sorts $\mathcal{T}_u$. We will use these abbreviations in the simplification schemata depicted in Fig. 7, 8, and 9.

We can then define the set $\mathcal{T}^*$ of (complex) type expressions inductively as follows:

- 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, the *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". One can either 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 obtained similarly. 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 and avoid the application of the commutativity rule (C) (see Section 3.3.4).

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

- COMPARABILITY
  This property is a consequence of the two other properties. Unique and linear expressions make it easy to find or compare (sub)expressions. This is important for the memoization technique described in Section 3.3.5.

3.3.3 Reduction Rules

The current implementation of the simplifier uses the hard-wired reduction rules as shown in Fig. 7. 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 for double negation (DN), De Morgan's laws (DM) and the schemata for distributivity (D). However, 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, additional rules are employed: idempotence, identity, absorption etc. If they can be applied, they always reduce the length of the (sub)expressions.
Especially at run time, but also at definition time, it is useful to exploit information from the type hierarchy. Further simplifications are possible by employing the schemata of Fig. 8 and 9 (it is possible to switch off the use of type hierarchy information at any time).
The recursive simplification algorithm *simplify–type* that implements the simplification schemata is given in pseudo-code in Fig. 10.

3.3.4 Lexicographic Order

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

$\downarrow x \quad y \rightarrow$	type	neg. type	conjunction	disjunction
type	$x <_{lex} y$	*true*	*true*	*true*
neg. type	*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 (or symbol names), e.g., the predicate `STRING<` in COMMON LISP, for example:

$$a <_{NF} b <_{NF} bb <_{NF} \neg a <_{NF} a \wedge b <_{NF} a \wedge \neg a <_{NF} a \vee b <_{NF} a \vee b \vee c <_{NF} a \vee 1$$

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

(DN)	$\dfrac{\neg\,\neg\, s}{s}$	
(C)	$\dfrac{s \wedge t}{t \wedge s}$	$\dfrac{s \vee t}{t \vee s}$
(DM)	$\dfrac{\neg\,(s \wedge t)}{\neg\, s \vee \neg\, t}$	$\dfrac{\neg\,(s \vee t)}{\neg\, s \wedge \neg\, t}$
(D)	$\dfrac{s \wedge (t \vee u)}{(s \wedge t) \vee (s \wedge u)}$	$\dfrac{s \vee (t \wedge u)}{(s \vee t) \wedge (s \vee u)}$
(F)	$\dfrac{s \wedge (t \wedge u)}{s \wedge t \wedge u}$	$\dfrac{s \vee (t \vee u)}{s \vee t \vee u}$
(I)	$\dfrac{s \wedge s}{s}$	$\dfrac{s \vee s}{s}$
(A)	$\dfrac{s \wedge (s \vee t)}{s}$	$\dfrac{s \vee (s \wedge t)}{s}$
(B1)	$\dfrac{s \wedge \neg\, s}{\bot}$	$\dfrac{s \vee \neg\, s}{\top}$
(B2)	$\dfrac{s \wedge \bot}{\bot}$	$\dfrac{s \vee \top}{\top}$
(B3)	$\dfrac{\neg\, \top}{\bot}$	$\dfrac{\neg\, \bot}{\top}$
(NE)	$\dfrac{s \wedge \top}{s}$	$\dfrac{s \vee \bot}{s}$
(T)	$\dfrac{\bigwedge_{i=1}^{1} s_i}{s_1}$	$\dfrac{\bigvee_{i=1}^{1} s_i}{s_1}$
(E)	$\dfrac{\bigwedge_{i=1}^{0} s_i}{\top}$	$\dfrac{\bigvee_{i=1}^{0} s_i}{\bot}$

Figure 7: **Syntactic simplification schemata employed in $\mathcal{TDL}$ ($s, s_i, t, u \in \mathcal{T}^*$).** *Note that the schemata for commutativity* (C) *must not be tested explicitly because $\mathcal{TDL}$ impose a total order on type expressions.*

(GLB1)	$\dfrac{s \wedge t}{s}$	if $s \preceq t$ and $s, t \in \mathcal{T}$
(GLB2)	$\dfrac{s_1 \wedge \cdots \wedge s_n}{t}$	if $t = glb(s_1, \ldots, s_n)$ and $s_1, \ldots, s_n \in \mathcal{T}, t \in \mathcal{T}^*$
(GLB3)	$\dfrac{a \wedge t}{a}$	if $type\text{–}of\,(a) = s$ such that $s \preceq t$ and $a \in \mathcal{A}, t \in \mathcal{T}_b$
(GLB4)	$\dfrac{\neg s \wedge t}{\bot}$	if $t \preceq s$
(GLB5)	$\dfrac{\neg s \wedge \neg t}{\neg t}$	if $s \preceq t$
(GLB6)	$\dfrac{s \wedge \neg t}{s}$	if $glb(s, t) = \bot$
(GLB7)	$\dfrac{s \wedge t}{\bot}$	if $s \in \mathcal{T}_a, t \in \mathcal{T}_s$
(GLB8)	$\dfrac{s \wedge t}{\bot}$	if $s \in \mathcal{T}_u, t \in \mathcal{T}_b$
(GLB9)	$\dfrac{s_1 \wedge \cdots \wedge s_n}{\bot}$	if $s_1, \ldots, s_n \in \mathcal{T}_s$ and $\nexists t \forall i \,.\, t \preceq s_i$
(GLB10)	$\dfrac{a \wedge t}{\bot}$	if $type\text{–}of\,(a) = s$ such that $s \not\preceq t$ and $a \in \mathcal{A}, t \in \mathcal{T}$
(GLB11)	$\dfrac{a \wedge b}{\bot}$	if $a \neq b$ and $a, b \in \mathcal{A}$
(GLB12)	$\dfrac{a \wedge \neg b}{a}$	if $a \neq b$ and $a, b \in \mathcal{A}$
(GLB13)	$\dfrac{s_1 \wedge \cdots \wedge s_n}{\bot}$	if $\begin{cases} \exists S \subseteq \{s_1, \ldots, s_n\} \\ \exists\, T = \{t_1, \ldots, t_m\} \subseteq \mathcal{T}, \\ \exists \text{ surjection } \iota : S \longmapsto T, \\ \exists \text{ declaration } \bot \doteq t_1 \wedge \cdots \wedge t_m, \\ \forall s \in S \,.\, s \preceq \iota(s) \end{cases}$
(GLB14)	$\dfrac{s \wedge \mathrm{U}}{\bot}$	if $s \in \mathcal{T}^* \setminus \{\top, \mathrm{U}\}$

Figure 8: **Semantic simplification schemata employed in $\mathcal{TDL}$ concerning only the greatest lower bound.**

(LUB1)	$\frac{s \vee t}{s}$	if $t \preceq s$ and $s, t \in \mathcal{T}$
(LUB2)	$\frac{s_1 \vee \cdots \vee s_n}{t}$	if $t = lub(s_1, \ldots, s_n)$ and $s_1, \ldots, s_n \in \mathcal{T}, t \in \mathcal{T}^*$
(LUB3)	$\frac{a \vee t}{t}$	if *type–of* $(a) = s$ such that $s \preceq t$ and $a \in \mathcal{A}, t \in \mathcal{T}_b$
(LUB4)	$\frac{s \vee \neg t}{\top}$	if $t \preceq s$
(LUB5)	$\frac{\neg s \vee \neg t}{\neg s}$	if $s \preceq t$
(LUB6)	$\frac{s \vee \neg t}{\neg t}$	if $lub(s, t) = \bot$
(LUB7)	$\frac{t_1 \vee \cdots \vee t_n}{p}$	if $p \doteq \bigvee_{i=1}^{n} t_i$

Figure 9: **Semantic simplification schemata employed in $\mathcal{TDL}$ concerning only the least upper bound.**

$\downarrow x \quad y \rightarrow$	disjunction	symbol	string	number
disjunction	$\forall i : x_i <_{NF} y_i$	*true*	*true*	*true*
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 Memoization

The memoization technique described in [Norvig 91b; Norvig 91a] has been adapted in order to reuse precomputed results of type simplification. There are four memoization tables for each $\mathcal{TDL}$ type domain: for CNF with/without hierarchy and DNF with/without hierarchy.[14] 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.
Note that it can be dangerous during definition time, to memoize results that depend on the type hierarchy. This is because redefinitions will make previous inferences invalid. Clearly,

[14] We have implemented the memoization tables via COMMON LISP hash tables. The average access time in case of the generalized lexicographic normal form for an `EQUAL` hash table (Allegro CL 4.2, Sun SPARC SS10) is fast: 0.05 ms for a hash table containing about 4000 entries. Hash tables seem to be good candidates for memoization because they can be implemented with constant access time and linear space complexity.

simplify–type(x):
  x :=**apply** (x, DN) ; /* double negation */
  x :=**apply** (x, B3) ; /* $\neg\top = \bot$ etc. */
  if literalp(x) **then return** x ;
  if $x = \neg y$ **then return** *simplify–type* (**apply** (x,DM)) ; /* DeMorgan */
  /* now either $x = x_1 \wedge \ldots \wedge x_n$ or $x = x_1 \vee \ldots \vee x_n$ */
  for all $0 \leq i \leq n$ **do** x_i := *simplify–type*(x_i) ;
  x := **apply** (x, F) ; /* flatten */
  x := **apply** (x, D) ; /* distributivity */
  /* now x is in normal form */
  x := **apply** (x, I, GLB11, B1, B2, GLB4, LUB4) ;
  x := **apply** (x, NE, A, GLB1/LUB1, GLB3/LUB3, GLB5/LUB5, GLB6/LUB6) ;
  x := **apply** (x, GLB7, GLB8, GLB9, GLB10, GLB12, GLB14) ;
  x := **apply** (x, GLB2/LUB2, GLB13, LUB7, T, E) ;
  return x.

Figure 10: **The recursive simplification algorithm** *simplify–type*. *If* **apply** *has more than two arguments, i.e., more than one rule can be chosen, these rules will be applied to every conjunct/disjunct in parallel.*

deleting the hash table before a redefiniton takes place is a first solution, however, a more appropriate strategy would be to impose a reason maintenance system on top of the simplifier. Some empirical results show the usefulness of memoization. The current DISCO grammar for German consists of 885 types and 27 templates. After a full type expansion of a toy lexicon of 244 instances/entries, the memoization hash tables contain 3185 entries (literals are not memoized). 40629 results have been reused at least once (some up to 250 times) of which 93 % are proper simplifications (i.e., the simplified formulae are really shorter than the unsimplified formulae).

3.4 Type Expansion and Control

We noted earlier that types allow us to refer to complex constraints through the use of symbol names. Reconstructing the constraints which determine a type (idiosyncratic plus inherited constraints) requires a complex operation called *type expansion*. This operation is comparable to Carpenter's *total well-typedness* [Carpenter 92] or Aït-Kaci's sort unfolding [Aït-Kaci et al. 93].
Thus type expansion is faced with two main tasks:

1. making all or particular feature constraints explicit (type expansion is a structure-building operation)

2. determining the global consistency of a type, or more generally, of a typed feature structure (if possible; see below)

3.4.1 Motivation

In TDL, the motivation for type expansion is manifold:

- CONSISTENCY
  At definition time, type expansion determines whether the set of type definitions (grammar and lexicon) is consistent. At run time, type expansion is involved in checking the satisfiability of the unification of two partially expanded typed feature structures, e.g., during parsing.

- 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 the end of processing however, one has to make the result/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 TDL as follows:

$$
\begin{aligned}
append_0 := [\,&\texttt{FRONT} < >,\\
&\texttt{BACK}\ \#1 \wedge list,\\
&\texttt{WHOLE}\ \#1\,].\\
append_1 := [\,&\texttt{FRONT} < \#first\,.\,\#rest1 >,\\
&\texttt{BACK}\ \#back \wedge list,\\
&\texttt{WHOLE} < \#first\,.\,\#rest2 >,\\
&\texttt{PATCH}\ append \wedge [\,\texttt{FRONT}\ \#rest1,\\
&\qquad\qquad\qquad\qquad \texttt{BACK}\ \#back,\\
&\qquad\qquad\qquad\qquad \texttt{WHOLE}\ \#rest2\,]\,].\\
append := &\ append_0 \vee append_1\,.
\end{aligned}
$$

- 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:

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

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

Type expansion together with a sufficiently 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, [Zajac 92] has shown that type expansion without sophisticated control strategies is in many cases 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.

$\mathcal{TDL}$ (together with $\mathcal{UDiNe}$) supports this strategy in that every feature structure can be 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:

- only regard structures which are subsumed by a given type, or conversely, only those which are not (e.g., expand the type *subcat-list* always or never expand the type *daughters*)
- take into account only structures under certain paths, or conversely, all structures except those under the specified paths (e.g., always expand the value under path `SYNSEM|LOC|CAT`; in addition, it is possible to employ path patterns in the sense of pattern matching[15])
- set the depth of type expansion for a given type

Note that we are not restricted to applying only one of these settings—they can be used in combination and can be changed dynamically during processing. It does make sense, for instance, to expand the (partial) information obtained so far at certain well-defined points during parsing. If this will not result in a failure, one can throw away (or store) this fully expanded feature structure, working on with the older (and smaller) one. However, if the information is inconsistent, we must backtrack to older stages in computation. Going this way which of course assumes heuristic knowledge (language as well as grammar-specific knowledge) results in faster processing and copying. Moreover, the inference engine must be able to handle possibly inconsistent knowledge, e.g., in case of a chart parser to allow for a third kind of edge (besides active and passive ones).

[15] This is different from functional uncertainty.

3.4.3 Preliminaries

In order to describe our algorithm, we need only a small inventory to abstract from the concrete implementation in $\mathcal{TDL}$ and to make the approach comparable to others. First of all, we assume pairwise disjoint sets of *features* (attributes) $\mathcal{F}$, *atoms* (constants) $\mathcal{A}$, logical *variables* $\mathcal{V}$, and *types* $\mathcal{T}$.
In the following, we refer to a *type hierarchy* $\mathcal{I}$ by a pair $\langle \mathcal{T}, \preceq \rangle$, such that $\preceq \subseteq \mathcal{T} \times \mathcal{T}$ is a decidable partial order, i.e., $\preceq$ is reflexive, antisymmetric, and transitive.
A *typed feature structure* (TFS) θ is essentially either a ψ-term or an ϵ-term [Aït-Kaci 86], i.e.,

$$\theta ::= \langle x, \tau, \Phi \rangle \mid \langle x, \tau, \Theta \rangle$$

such that $x \in \mathcal{V}$, $\tau \in \mathcal{T}$, $\Phi = \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}$, and $\Theta = \{\theta_1, \ldots, \theta_n\}$, where each $f_i \in \mathcal{F}$ and θ_i is again a TFS.
We will call the equation $f \doteq \theta$ a *feature constraint* (or an attribute-value pair).[16] Φ is interpreted conjunctively, whereas Θ represents a disjunction. Variables are used to indicate structure sharing.
Let us give a small example to see the correspondences. The typed feature structure

$$\langle x, \mathit{cyc\text{–}list}, \{\textsc{first} \doteq 1, \textsc{rest} \doteq x\} \rangle$$

should denote the same set of objects than the following two-dimensional attribute-value matrix (AVM) notation:

$$\boxed{x} \begin{bmatrix} \mathit{cyc\text{–}list} \\ \textsc{first}\ \ 1 \\ \textsc{rest}\ \ \boxed{x} \end{bmatrix}$$

It is worth noting that for the purpose of simplicity and clarity, we restrict TFS to the above two cases. Actually, our algorithm is more powerful in that it handles other cases, for instance conjunction, disjunction, and negation of types and feature constraints.
A *type system* Ω is a pair $\langle \Theta, \mathcal{I} \rangle$, where Θ is a finite set of typed feature structures and $\mathcal{I}$ an inheritance hierarchy. Given Ω, we call $\theta \in \Theta$ a *type definition*.
Our algorithm is independent of the underlying deduction system—we are not interested in the normalization of feature constraints (i.e., how unification of feature structures is actually done) nor are we interested in the logic of types, e.g., whether the existence of a greatest lower bound is obligatory (TFS [Zajac 92]; ALE [Carpenter & Penn 94]) or optional as in $\mathcal{TDL}$. We assume here that *typed unification* is simply a black box and can be accessed through an interface function (say *unify–tfs*). From this perspective, our expansion mechanism can be either used as a stand-alone system or as an integrated part of the typed unification machinery.

[16] It should be noted that we define TFS to have a nested structure and not to be flat (in contrast to feature clauses in a more logic-oriented approach, e.g., [Aït-Kaci et al. 93]) in order to make the connection to the implementation clear and to come close to the structured attribute-value matrix notation.

We only have to say a few words on the semantic foundations of type expansion at the end of this section. This is because we could either choose extensions of *feature logic* [Smolka 89] or directly interpret our structures within the paradigm of (constraint) logic programming [Lloyd 87].

3.4.4 Algorithm

In this section, we explain the basic structure of our algorithm, extend it by a technique called indexed prototype memoization, describe the syntax of control information and (informally) the integration into the algorithm, and finally give an example.[17]

The overall design of our TE algorithm was inspired by the following requirements:

- support a *complete* expansion strategy
- allow *lazy expansion* of recursive types
- minimize the number of unifications
- make expansion parameterized for delay and preference information

Before we describe the algorithm, we modify the syntax of TFS to get rid of unimportant details. First, we simplify TFS in that we omit variables. This can be done without loss of generality if variables are directly implemented through structure-sharing (which is the case for our system). Hence conjunctive TFS have the form $\langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$, whereas disjunctive TFS are of the form $\langle \tau, \{\theta_1, \ldots, \theta_n\}\rangle$.
Given a TFS θ, *type–of*(θ) returns the type of θ, whereas *typedef*(τ) obtains the type definition without inherited constraints as given by the type system $\Omega = \langle \Theta, \mathcal{I}\rangle$. We call this TFS a *skeleton*. It is either $\langle \sigma, \{\theta_1, \ldots, \theta_n\}\rangle$ or $\langle \sigma, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$, where σ are the direct supertype(s) of τ.
Because the algorithm should support partially expanded (delayed) types, we enrich each TFS θ by two flags:

1. Δ-*expanded*(θ)=true, iff *typedef*(*type–of*(θ)) and the definitions of all its supertypes have been unified with θ, and false otherwise.
2. *expanded*(θ)=true, iff Δ-*expanded*(θ)=true **and** *expanded*(θ_i)=true for all elements θ_i of TFS θ.

Hence Δ-*expanded* is a local property of a TFS that tells whether the *definition* of its type is already present, while *expanded* is a global property which indicates that all substructures of a TFS are Δ-expanded. Clearly, atoms and types that possess no features are always expanded. The exploitation of these flags lead to a drastic reduction of the search space in the expansion algorithm.

[17] A thorough description of the algorithm, its realization, and other related subjects are presented in [Schäfer 95].

3.4.4.1 Basic Structure

The following functions briefly sketch the basic algorithm. It is a destructive depth-first algorithm with a special treatment of recursive types that will be explained in Section **??**. *expand–tfs* is the main function that initiates type expansion. The while loop is executed until the TFS θ is expanded or resolved (see below). Several passes may be necessary for recursive TFS.

expand–tfs(θ) :=
  while not (*expanded*(θ) **or**
    resolved(θ) **or**
    no unification occurred in the last pass)
   depth–first–expand(θ). /* or *types–first–expand*(θ), resp. */

depth–first–expand and *types–first–expand* recursively traverse a TFS. The visited check is done by comparing variables (actually, structure-sharing in the implementation makes variables obsolete). *types–first–expand* is defined analogously by interchanging the last two lines.

depth–first–expand(θ) :=
  if θ *has been already visited in this pass*
   then return
   else if $\theta = \langle \tau, \{\theta_1, \ldots, \theta_n\}\rangle$
      then for every $\theta \in \{\theta_1, \ldots, \theta_n\}$: *depth–first–expand*(θ)
      else /* $\theta = \langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$ */
        for every $\theta \in \{\theta_1, \ldots, \theta_n\}$: *depth–first–expand*(θ)
        if not Δ-*expanded*(θ) **then** *unify–type–and–node*(τ, θ).

unify–type–and–node destructively unifies θ with the expanded TFS of τ.

unify–type–and–node(τ, θ) :=
  if $\tau = \neg\sigma$
   then *unify–tfs* (*negate–fs* (*expand–type*(σ,*index*)), θ)
   else *unify–tfs* (*expand–type*(τ,*index*), θ);
  Δ-*expanded*(θ) := true.

We adapt Smolka's treatment of negation to our TFS [Smolka 89]. Note that we only depict the conjunctive case here.

negate–fs($\theta = \langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$) :=
  return $\langle \top, \{\langle \neg\tau, \{\}\rangle, \langle \top, \{f_1 \uparrow\}\rangle, \langle \top, \{f_1 \doteq$ *negate–fs*$(\theta_1)\}\rangle, \ldots,$
    $\langle \top, \{f_n \uparrow\}\rangle, \langle \top, \{f_n \doteq$ *negate–fs*$(\theta_n)\}\rangle\}\rangle$.

3.4.4.2 Indexed Prototype Memoization

The basic idea of *memoization* [Michie 68] is to tabulate results of function applications in order to prevent wasted calculations. The more expensive the computation of a value is,

the bigger the efficiency gain will be. To memoize a function, it must meet the following requirements: (i) it must be a proper function with no side effects, because side effects might cause wrong results and (ii) the function should be called more than once with the same argument, the more often, the better. Recursive functions meet these two requirements and hence serve as good examples for the efficiency of memoization.[18]
We apply this technique to the type expansion function. The argument of our memoized expansion function is a pair consisting of a type name (or a name of an lexicon entry or a rule) and an arbitrary index that allows to access different TFS of the same type which may be expanded in different ways (e.g., partially or fully). Such feature structures are called *prototypes*.
Once a prototype has been expanded according to the attached control information, its expanded version is recorded and all future calls return a copy of it, instead of repeating once again the same unifications:

expand–type$(\tau, index)$:=
  if *protomemo*$(\tau, index)$ **undefined**
   then θ := *expand–tfs*(*typedef* (τ));
        protomemo$(\tau, index)$:= θ;
        return *copy–tfs*(θ)
   else return *copy–tfs*(*protomemo*$(\tau, index)$).

Most of these computations can be done at compile time (*partial evaluation*), and hence speed up unification at run time. The prototypes can serve as *basic blocks* for building a partially expanded grammar.

Some empirical results show the usefulness of indexed prototype memoization. The following table (Figure 11) contains statistical information about the expansion of an HPSG grammar with approx. 900 type definitions (excluding lexicon entries). About 250 lexicon entries and rules have been expanded from scratch, i.e., all instances are unexpanded (*skeletons*) at the beginning. The type and instance skeletons together consist of about 9000 nodes. No preference or delay information was given. Note that the algorithm without memoization inserts only the unexpanded skeletons of a type definition while the memoized version expands each complex type once and afterwards returns only copies of it. The resulting structures consist of about 50000 nodes (27000 in type prototypes, 23800 in instance prototypes).
The measurements show that memoization speeds up expansion by a factor of 5/10 for this grammar (this factor is directly related to the number of unifications). The time difference between the memoized and non-memoized algorithm may be even bigger if disjunctions are involved. The sample grammar contains only a few disjunctions.

[18] One of the most impressing examples is the memoized *fib* function [Norvig 91b] (*fib* n returns the n-th Fibonacci number) which reduces exponential run-time to a simple table look-up for n once the value for a number $\geq n$ has been computed.

algorithm	*depth–1st–expand*		*types–1st–expand*		*depth–1st–expand*		*types–1st–expand*	
memoization	yes		yes		no		no	
time (secs)	45 | 23*		46 | 23*		216		218	
unifications	27221 | 14495*		27207 | 14481*		155888		155876	
number of	853	*cons*	260	*cons*	8330	*avm*	8454	*avm*
calls to	316	cat-type	147	*diff-list*	2392	sem-expr	2503	sem-expr
expand-type	269	*diff-list*	143	morph-type	1379	term-type	1420	term-type
	243	morph-type	94	nmorph-head	1161	*cons*	1196	*cons*
*: with types	208	atomic-wff	83	sort-expr	1003	wff-type	1073	wff-type
pre-expanded	202	rp-type	71	atomic-wff	933	agr-feat	951	agr-feat
	146	conj-wff-type	62	rp-type	880	semantics	747	semantics
	120	var-type	53	subwff-inst	823	indexed-wff	730	indexed-wff
	63	indexed-wff	53	cat-type	669	var-type	697	rp-type
	59	nmorph-head	46	sign-type	662	rp-type	690	var-type
	53	subwff-inst	42	mas-noun	589	*diff-list*	589	*diff-list*
	53	term-type	35	count-noun-lex	459	major-feat	447	head-feat
	51	semantics-type	35	semantics-type	447	head-feat	430	local-type
	50	sign-type	27	indexed-wff	444	local-type	427	case-type
	48	sort-expr	26	empty-quant	438	cat-type	426	head-val
	42	mas-noun	23	*avm*	426	head-val	423	subcat-type
	35	count-noun-lex	19	conj-wff-type	423	subcat-type	423	local-feat
	26	empty-quant	18	var-type	423	local-feat	423	head-type
	23	*avm*	18	trans-verb-lex	423	head-type	422	subj-type
	20	identity-wff	15	noun-type	422	subj-type	422	mod-type
	18	trans-verb-lex	14	agr-st-type	422	mod-type	422	minor-type
	17	proper-name	14	proper-noun	422	minor-type	422	major-type
	15	noun-type	13	adj-lex	422	major-type	421	gender-type
	15	phead-type	13	amorph-head	420	v-feat	418	cat-type
	14	agr-st-type	13	omorph-head	417	n-feat	416	local
	14	proper-noun	12	fem-noun	416	local	416	syntax
	13	adj-lex	12	sg-count-noun	416	syntax	416	morphology
	13	amorph-head	12	lex-sign-type	416	morphology	414	non-local
	13	omorph-head	12	major-val	414	non-local	414	syntax-type
	13	infl-val	12	verb-type	414	syntax-type	411	major-feat
	12	fem-noun	11	nbar-type	407	number-type	402	v-feat
	12	sg-count-noun	11	neu-noun	398	non-loc-type	399	n-feat
	12	lex-sign-type	10	dat	392	case-type	398	non-loc-type
	12	major-val	10	sg-agr	352	atomic-wff	387	atomic-wff
	12	verb-type	10	non-que-sign	346	gender-type	354	number-type
	12	local-type	10	wff-type	344	agr-val	339	semantics-type

Figure 11: Comparing the efficiency of depth-first vs. types-first expansion with/without prototype memoization. The run time on a Sparc 10 is stated in seconds. The left values in the run time and unifications rows are for expansion of all instances from scratch, the right ones when all types are already expanded.

3.4.4.3 Detecting Recursion

The memoization technique is also employed in detecting recursive types. This is important in order to ensure termination. We use the so-called "call stack" of *expand–type* to check whether a type is recursive or not (see Section 3.4.4.4). Each call of *expand–type*$(\tau, index)$ will push τ onto the call stack. This stack then is passed to *expand–tfs*.
If the type τ on top of the call stack also occurs below in the stack

$$(\tau, \sigma_n, \ldots, \sigma_1, \tau, \rho_m, \ldots, \rho_1)$$

we immediate know that the types $\tau, \sigma_n, \ldots, \sigma_1$ are recursive. Furthermore,these types form a *strongly connected component* (scc) of the type dependency (or occurrence) graph, i.e., each type in the scc is reachable from every other type in the scc. Examples for such sccs are (*cons list*) and (*state1*) in the trace of the example below (Section 3.4.4.4).
Testing whether a type is recursive or not thus reduces to a simple *find* operation in a global list that contains all sccs. The expansion algorithm uses this information in *expand–tfs* to delay recursive types if the call stack contains more than one element. Otherwise, prototype memoization would loop.
If a recursive type occurs in a TFS and this type has already been expanded under a subpath, and no features or other types are specified at this node, then this type will be delayed, since it would expand forever (we call this *lazy expansion*). An instance of such a recursive type, where expansion will terminate, is the recursive version of *list*, as defined below.

3.4.4.4 Example

In the following, we define a finite state machine [Krieger et al. 93] with two states that accepts the language $\mathtt{a}^*(\mathtt{a}+\mathtt{b})$. The input is specified through a list under path INPUT ; cf. the definition of type *ab* below. The distributed (or named) disjunction [Eisele & Dörre 90] headed by \$1 in type *state1* is used to map input symbols to state types (and vice versa). The encoding of this finite state machine in the concrete syntax of $\mathcal{TDL}$ is given in Appendix B.2.

$$\mathit{list} \Rightarrow \{\mathit{cons}, \langle\,\rangle\}$$

$$\mathit{cons} \Rightarrow \begin{bmatrix} \textsc{first} & \top \\ \textsc{rest} & \mathit{list} \end{bmatrix} \quad \text{we abbreviate } \mathit{cons} \text{ via } \langle \ldots \rangle$$

$$\mathit{non\text{-}final\text{-}config} \Rightarrow \begin{bmatrix} \textsc{input} & \langle \boxed{1} . \boxed{2} \rangle \\ \textsc{edge} & \boxed{1} \\ \textsc{next} & \left[\textsc{input}\ \boxed{2}\right] \end{bmatrix}$$

$$\textit{final-config} \Rightarrow \begin{bmatrix} \text{INPUT} & \langle\,\rangle \\ \text{EDGE} & \textit{undef} \\ \text{NEXT} & \textit{undef} \end{bmatrix}$$

$$\textit{state1} \Rightarrow \begin{bmatrix} \textit{non-final-config} \\ \text{EDGE} & \$1\,\{\mathtt{a}, \{\mathtt{a}, \mathtt{b}\}\} \\ \text{NEXT} & \$1\,\{\textit{state1}, \textit{final-config}\} \end{bmatrix}$$

$$\textit{ab} \Rightarrow \begin{bmatrix} \textit{state1} \\ \text{INPUT} & \langle\,\mathtt{a}, \mathtt{b}\,\rangle \end{bmatrix}$$

Let us give a trace of the expansion of type *ab*—the algorithm is *depth–first–expand* without any delay or preference information. In this trace, we assume that it was not known before that the types *cons* (abbreviated as $\langle \ldots \rangle$), *list*, and *state1* are recursive, hence the sccs will be computed on the fly.

step	*expand–type*	in type	under path	call stack
1	*cons*	*ab*	INPUT.REST	(*ab*)
2	*list*	*cons*	REST	(*cons ab*)
3	*cons*	*list*	ϵ	(*list cons ab*)
	$\boxed{\rightarrow$ (*cons list*) is new scc, delay *cons* here$}$			
4	*cons*	*ab*	INPUT	(*ab*)
5	*state1*	*ab*	ϵ	(*ab*)
6	*state1*	*state1*	NEXT	(*state1 ab*)
	$\boxed{\rightarrow$ (*state1*) is new scc, delay *state1* here$}$			
7	*final-config*	*state1*	NEXT	(*state1 ab*)
8	*non-final-config*	*state1*	ϵ	(*state1 ab*)
9	*cons*	*non-final-config*	INPUT	(*non-final-config state1 ab*)
10	*state1*	*ab*	NEXT	(*ab*)

The result of *expand–type*(*ab*) is the following feature structure:

$$\textit{expand–type}(\textit{ab}) \Rightarrow \begin{bmatrix} \textit{ab} \\ \text{INPUT} & \langle\,\boxed{1}\mathtt{a}\,.\,\boxed{2}\langle\,\boxed{3}\mathtt{b}\,.\,\boxed{4}\langle\,\rangle\,\rangle\,\rangle \\ \text{EDGE} & \boxed{1} \\ \text{NEXT} & \begin{bmatrix} \textit{state1} \\ \text{INPUT} & \boxed{2} \\ \text{EDGE} & \boxed{3} \\ \text{NEXT} & \begin{bmatrix} \textit{final-config} \\ \text{INPUT} & \boxed{4} \\ \text{EDGE} & \textit{undef} \\ \text{NEXT} & \textit{undef} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

If we ran our automaton on the input `abb`,

$$abb \Rightarrow \begin{bmatrix} state1 & \\ \text{INPUT} & \langle\, \texttt{a}, \texttt{b}, \texttt{b} \,\rangle \end{bmatrix}$$

it would be rejected: *expand-type*(*abb*) $\Rightarrow$ fail.

3.4.4.5 Declarative Specification of Control Information

Control information for the expansion algorithm can be specified globally, locally for each *prototype*, or for a specific *expand–tfs* call. The following control keywords have been implemented so far.

- `:expand-function` {*depth* | *types*}*–first–expand*
  Specifies the basic expansion algorithm. *depth–first–expand* (default) is a proper depth first algorithm, while *types–first–expand* expands types first and then goes down the feature graph (and first delays the expansion of recursive types to prevent infinite loops).

- `:delay` { `(` {*type* | `(`*type* [*pred*]`)`} {*path*}$^+$ `)` }*
  Specifies which types should be delayed during expansion. *path* may be a feature path or a complex path pattern with wildcard symbols `*`, `+`, `?`, feature and segment variables. *pred* is a test predicate to compare types, e.g., $=$, $\preceq$, $\succeq$, as well as user-defined predicates are supported. The `:delay` information overrides the `:expand` and `:expand-only` slots and will be checked in function *unify–type–and–node*.

- {`:expand` | `:expand-only`} { `(` {*type* | `(`*type* [*index* [*pred*]]`)`} {*path*}$^+$ `)` }*
  There are two mutually exclusive modes concerning expansion of types. If the `:expand-only` list is specified, only types in this list will be expanded with the specified prototype *index*; all others will be delayed. If the `:expand` list is specified, all types will be expanded. Types not mentioned in the list will be expanded using the default prototype index `nil`, i.e., fully, if not specified otherwise. Path patterns and type predicates are supported as in the `:delay` list and will be checked in function *unify–type–and–node*.

- `:maxdepth` *integer*
  Specifies that all types at paths longer than *integer* will be delayed anyway (checked in function *unify–type–and–node*).

- `:attribute-preference` {*attribute*}*
  Defines a partial order on attributes that will be considered in the functions *depth–first–expand* and *types–first–expand*. The (sub) feature structures at the attributes leftmost in the list will be expanded first. This non-numerical preference may speed up expansion if no numerical heuristics are known.

- `:ask-disj-preference` {`t` | `nil`}
  If this flag is set to `t`, the expansion algorithm interactively asks for the order in which disjunction alternatives should be expanded (checked in *depth–first–expand*). Example:

```
Ask-Disj-Preference in G under path X
The following disjunctions are unexpanded:
Alternative 1:
  (:Type A :Expanded NIL) []
Alternative 2:
  (:Type B :Expanded NIL) []
Which alternative in G under path X should be expanded next (1, 2, or
0 to leave them unexpanded, or :all to expand all alternatives in this
order, or :quiet to continue without asking again in G) ? _
```

- `:use-{conj | disj}-heuristics` {`t` | `nil`}
  [Uszkoreit 91] suggested the exploitation of numerical preference information for features and disjunctions to speed up unification. Both slots control the use of this information in functions *depth–first–expand* and *types–first–expand*.

- `:resolved-predicate` {`resolved-p` | `always-false` | ...}
  This slot specifies a user definable predicate that may be used to stop recursion (see function *expand–tfs*). The default predicate is `always-false` which will lead to a complete expansion algorithm if no other delay information is specified.

- `:ignore-global-control` {`t` | `nil`}
  If this flag has value `t`, the values of the three globally specified lists `:expand-only`, `:expand`, `:delay` will be ignored. If `nil`, locally and globally specified lists will be taken into account.

Let us give an example to show how control information can be employed. Note that we formulate this example in the concrete syntax of TDL.

```
defcontrol verb
  ((:delay ((sign Subsumes) SYNSEM.NONLOCAL.?.SLASH))
   ;; ? matches INHERITED and TO-BIND
   (:attribute-preference SYNSEM DTRS SUBCAT HEAD)
   (:use-disj-heuristics T)
   (:ignore-global-control T)
   (:expand ((local initial) *)))
   ;; * matches all paths in type local
  :index 1.
```

Here, we specify control information for the type `verb`. However, not for all prototypes, but only for the one with index `1` (see Section 3.4.4.2 on indexed prototype memoization). The idea is to delay `sign` and all its subtypes (`Subsumes`) under all paths that start with `SYNSEM.NONLOCAL`, followed by an arbitrary attribute (`?`), and ending in `SLASH`. The prefered attribute preference during expansion is (highest priority first): (i) `SYNSEM`, (ii) `DTRS`, (iii)

SUBCAT, (iv) HEAD. The other attributes are not ordered, thus we expand their values depending on the traversing strategy of our expansion algorithm. We use the disjunction heuristics (if specified), and ignore globally specified control information that might conflict with this locally specified ones. In addition, the prototype of type `local` with index/name `initial` must be expanded under all its path (= *; including the empty path).

3.4.4.6 How to Stop Recursion

Type expansion with recursive type definition is undecidable in general, i.e., there is no complete algorithm that halts on arbitrary input (TFS) and decides whether a description is satisfiable or not (see Section 3.5). However, there are several ways to stop infinite expansion in our framework:

- The first method is part of the expansion algorithm (lazy expansion) as described before.
- The second way is brute force: use the `:maxdepth` slot to cut expansion at a suitable path depth.
- The third method is to define `:delay` patterns or to select the `:expand-only` mode with appropriate type and path patterns.
- The fourth method is to use the `:attribute-preference` list to define the "right" order for expansion.
- Finally, one can define an appropriate `:resolved-predicate` that is suitable for a class of recursive types.

3.5 Theoretical Results

It is worth noting that testing for the satisfiability of feature descriptions admitting recursive type equations/definitions is in general undecidable. [Rounds & Manaster-Ramer 87] were the first to have shown that a Kasper-Rounds logic enriched with recursive types allows one to encode a Turing machine—hence, deciding satisafiability would imply that the Halting problem is decidable (which is obviously not). Later, [Smolka 89] argued that the undecidability result is due to the use of coreference constraints. He demonstrated his claim by encoding the word problem of Thue systems. Hence, our expansion mechanism is faced with the same result, viz., that expansion might not terminate.
However, we conjecture that non-satisfiability and thus failure of type expansion is, in general, semi-decidable. The intuitive argument is as follows: given an arbitrary recursive TFS and assuming a fair type unfolding strategy, the only event under which TE terminates in finite time follows from a local unification failure which then leads to a global one. In every other case, the unfolding process goes on by substituting types through their definitions. Recently, [Aït-Kaci et al. 93] have formally shown a similar result by using the compactness theorem of first-order logic. However, their proof assumes the existence of an infinite OSF clause (generated by unfolding a ψ-term).

Thus our algorithm might not terminate if we choose the complete expansion strategy. However, we noted above that we can even parameterize the complete version of our algorithm to ensure termination, for instance to restrict the depth of expansion (analogous to the off-line parsability constraint). The non-complete version always guarantees termination and might suffice in practice.

Semantically, we can formally account for such recursive feature descriptions (with respect to a type system) in different ways: either directly on the descriptions, or indirectly through a transformational approach into (first-order) logic. Both approaches rely on the construction of a fixpoint over a particular continuous function.[19] The first approach is in general closer to an implementation (and thus to our algorithm) in that the function which is involved in the fixpoint construction corresponds more or less to the unification/substitution of TFS (see for instance [Aït-Kaci 86] or [Pollard & Moshier 90]). The latter approach is based on the assumption that TFS are only syntactic sugar for first-order formulae. If we transform these descriptions into an equivalent set of definite clauses, we can employ techniques that are fairly common in logic programming, viz. characterizing the models of a definite program through a fixpoint. Take, for instance, our *cyc–list* example from the beginning to see the outcome of such a transformation (assume that *cyc–list* is a subtype of *list*):

$$\forall x.\mathit{cyc\text{–}list}(x) \leftrightarrow \exists y, z.\mathit{list}(x) \wedge \textsc{first}(x, y) \wedge \textsc{rest}(x, z) \wedge y \doteq 1 \wedge z \doteq x$$

3.6 Other Approaches

In this section, we will describe closely related approaches and compare our algorithm to them. To the best of our knowledge, the problem of type expansion within a typed feature-based environment was first addressed by [Aït-Kaci 86]. The language he described was called KBL and shared great similarities with LOGIN; see [Aït-Kaci & Nasr 86]. However, the expansion mechanism he described was order dependent in that it replaced types by their definition instead of unifying the information. Moreover, it was non-lazy, thus it will fail to terminate for recursive types and performs type expansion only at definition time as is the case for ALE [Carpenter & Penn 94]. However, ALE provides recursion through a built-in bottom-up chart parser and through definite clauses. Allowing type expansion only at definition time is in general space consuming, thus unification and copying is expensive at run time.

Another way one might pursue is to integrate type expansion into the typed unification process so that it can take place at run time. Systems that explore this strategy are TFS [Zajac 92] and LIFE [Aït-Kaci 93]. However, both implementations are not lazy, thus hard to control and moreover, might not terminate. In addition, if prototype memoization is not available, type expansion at run time is inefficient; cf. the results of our grammar example in Table (11). A system that employs a lazy strategy on demand at run time is CUF [Dörre & Dorna 93]. Laziness can be achieved by specifying delay patterns as is familiar from PROLOG. This means to delay the evaluation of a relation until the specified parameters are instantiated.

[19] In both cases, there is, in general, more than one fixpoint, but it seems desirable to choose the greatest one; see [Krieger 95].

Our approach, which has been fully implemented as a stand-alone module, is novel in that it combines the benefits of these systems plus much more:

- freely choose time of TE, e.g., during unification, parsing etc.
- local as well as global control is possible
- delayed expansion
- recursive types are treated specially
- preference information can be employed
- prototype memoization speeds up processing

4 Comparison to other Systems

$\mathcal{TDL}$ is unique in that it implements many novel features not found in other systems like ALE [Carpenter & Penn 94], LIFE [Aït-Kaci et al. 93], or TFS [Zajac 92]. Of course, these systems provide other features which are not present in our formalism.[20]
What makes $\mathcal{TDL}$ unique in comparison to them is the distinction open vs. closed world, the availability of the full boolean connectives and distributed disjunctions (via $\mathcal{UDiNe}$), as well as an implemented lazy type expansion mechanism for recursive types (as compared with LIFE). ALE, for instance, neither allows disjunctive nor recursive types and enforces the type hierarchy to be a BCPO. However, it makes recursion available through definite relations and incorporates special mechanisms for empty categories and lexical rules.
TFS is based on a closed world, the unavailability of negative information (only implicitly present) and only a poor form of disjunctive information but performs parsing and generation entirely through type deduction (in fact, it was the first system).
LIFE comes closest to us but provides a semantics for types that is similar to TFS. Moreover the lack of negative information and distributed disjunctions makes it again comparable with TFS. LIFE as a whole can be seen as an extension of PROLOG (as was the case for its predecessor LOGIN), where first-order terms are replaced by ψ-terms. In this sense, LIFE is richer than our formalism in that it offers a full relational calculus.

5 Summary

In this paper, we have presented $\mathcal{TDL}$, a typed feature formalism that integrates a powerful feature constraint solver and type system. Both of them provide the boolean connectives $\wedge$, $\vee$, and $\neg$, where a complex expression is decomposed by employing intermediate types. Moreover, recursive types are supported as well. In $\mathcal{TDL}$, a grammar writer decides whether types live in an open or a closed world. This effects GLB and LUB computations.

[20] [Backofen et al. 93] gives an overview of implemented formalisms.

The type system itself consists of several inference components, each designed to cover a specific task efficiently: (i) a bit vector encoding of the hierarchy, (ii) a fast symbolic simplifier for complex type expressions, (iii) memoization to cache precomputed results, and (iv) a sophisticated type expansion mechanism.
The system as described in this paper has been fully implemented in COMMON LISP and runs on various software/hardware platforms (Allegro CL, Lucid CL, Macintosh CL, CLisp). It has been integrated successfully into the DISCO environment [Uszkoreit et al. 94] and is used at several places outside (e.g., CSLI, Stanford currently uses $\mathcal{TDL}$ for writing a large English HPSG grammar).

The next major version of $\mathcal{TDL}$ will make certain forms of knowledge compilation available, e.g., extracting syntactic incompatibilities between types from a given grammar.
Other extensions of the system will concern the type expansion mechanism. We are planning to provide additional expansion strategies and to realize the expansion mechanism as a true anytime module [Wahlster 93] (implemented as a separate process), so that it can be interrupted and restarted from the outside.
We also plan to extend the grammar development environment with other useful tools, e.g., a classifier (cf. [Krieger & Schäfer 94a] for a description of the current status). Moreover, providing a classifier allow us to incorporate $\mathcal{TDL}$ in other areas of knowledge representation which are currently handled exclusively by terminological/KL-ONE-like languages.

A *TDL* BNF

The *TDL* syntax is given in extended BNF (Backus-Naur Form). Terminal symbols (characters to be typed in) are printed in `bold` style. Nonterminal symbols are printed in *italic* style. The grammar starts with the *start* production. It is case insensitive (except for strings). The following table explains the meanings of the metasymbols used in extended BNF.

metasymbols	**meaning**
... | ...	alternative expressions
[...]	one optional expression
[... | ... | ...]	one or none of the expressions
{ ... | ... | ... }	exactly one of the expressions
{ ... }*	n successive expressions, where $n \in \{0, 1, \ldots\}$
{ ... }+	n successive expressions, where $n \in \{1, 2, \ldots\}$

A.1 Type Definitions

type-def → *type* { *avm-def* | *subtype-def* } .
type → *identifier*
avm-def → `:=` *body* { `,` *option*}* |
`!=` *nonmonotonic* [`where (` *constraint* { `,` *constraint*}* `)`] { `,` *option*}*
body → *disjunction* [`-->`*list*] [`where (` *constraint* { `,` *constraint*}* `)`]
disjunction → *conjunction* { {`|` | `^`} *conjunction* }*
conjunction → *term* { `&` *term* }*
term → *type* | *atom* | *feature-term* | *diff-list* | *list* | *coreference* |
distributed-disj | *templ-par* | *templ-call* | `~`*term* | `(` *disjunction* `)`
atom → *string* | *integer* | `'`*identifier*
feature-term → `[` [*attr-val* { `,` *attr-val*}*] `]`
attr-val → *attribute* [`:`*restriction*] { `.` *attribute* [`:`*restriction*] [*disjunction*]}*
attribute → *identifier* | *templ-par*
restriction → *conj-restriction* { {`|` | `^`} *conj-restriction* }*
conj-restriction → *basic-restriction* { `&` *basic-restriction* }*
basic-restriction → *type* | `~`*basic-restriction* | *templ-par* | `(` *restriction* `)`
diff-list → `<!` [*disjunction* { `,` *disjunction*}*] `!>` [`:` *type*]
list → `<>` | `<` *nonempty-list* `>` [*list-restriction*]
nonempty-list → [*disjunction* { `,` *disjunction*}* `,`] `...` |
disjunction { `,` *disjunction*}* [`.` *disjunction*]
list-restriction → `:` `(` *restriction* `)` | `:` *type* [`:` `(`*integer*`,` *integer*`)` | `:` *integer*]
coreference → `#`*coref-name* | `~#(` *coref-name* { `,` *coref-name*}* `)`
coref-name → *identifier* | *integer*
distributed-disj → `%`*disj-name* `(` *disjunction* { `,` *disjunction*}+ `)`
disj-name → *identifier* | *integer*

templ-call → `@`*templ-name* `(` [*templ-par* {`,` *templ-par*}*] `)`
templ-name → *identifier*
templ-par → `$`*templ-var* [`=` *disjunction*]
templ-var → *identifier* | *integer*
constraint → `#`*coref-name* `=` { *function-call* | *disjunction* }
function-call → *function-name* `(` *disjunction* {`,` *disjunction*}* `)`
function-name → *identifier*
nonmonotonic → *type* `&` `[` *overwrite-path* {`,` *overwrite-path*}* `]`
overwrite-path → *identifier* { `.` *identifier* }* *disjunction*
subtype-def → { `:<` *type* }+ {`,` *option*}*
option → `status:` *identifier* | `author:` *string* | `date:` *string* | `doc:` *string* |
`expand-control:` *expand-control*
expand-control → `(` [`(:expand` { `(` {*type* | `(`*type* [*index* [*pred*]]`)`} {*path*}+ `)` }* `)` |
`(:expand-only` { `(` {*type* | `(`*type* [*index* [*pred*]]`)`} {*path*}+ `)` }* `)`] |
[`(:delay` { `(` {*type* | `(`*type* [*pred*]`)`} {*path*}+ `)` }* `)`] |
[`(:maxdepth` *integer* `)`] |
[`(:attribute-preference` {*identifier*}* `)`] |
[`(:ask-disj-preference` {`t` | `nil`} `)`] |
[`(:use-conj-heuristics` {`t` | `nil`} `)`] |
[`(:use-disj-heuristics` {`t` | `nil`} `)`] |
[`(:expand-function` {`depth-first-expand` | `types-first-expand`} `)`] |
[`(:resolved-predicate` {`resolved-p` | `always-false` | `...`} `)`] |
[`(:ignore-global-control` {`t` | `nil`} `)`] `)`
path → {*identifier* | *pattern*} {`.`{*identifier* | *pattern*}}*
pattern → `?` | `*` | `+` | `?`[*identifier*][`?`|`*`|`+`]
pred → `eq` | `subsumes` | `extends` | ...
index → *integer* for instances
integer | *identifier string* for avm types
integer → {`0`|`1`|`2`|`3`|`4`|`5`|`6`|`7`|`8`|`9`}+
identifier → {`a`–`z`|`A`–`Z`|`0`–`9`|`_`|`+`|`-`|`*`|`?`}+
string → `"`{*any character*}*`"`

A.2 Instance Definitions

instance-def → *instance avm-def* `.`
instance → *identifier*

A.3 Template Definitions

template-def → *templ-name* `(` [*templ-par* {`,` *templ-par*}*] `)` `:=` *body* {`,` *option*}* `.`

A.4 Declarations

declaration → *partition* | *incompatible* | *sort-def* | *built-in-def* |
hide-attributes | *hide-values* | *export-symbols*

partition → *type* `=` *type* `{` `{` `|` | `^` `}` *type* `}`* .

incompatible → `nil` `=` *type* {`&` *type*}+ .

sort-def → `sort`[`s`] : *type* {`,` *type*}* .

built-in-def → `built-in`[`s`] : *type* {`,` *type*}* .

hide-attributes → `hide-attribute`[`s`] : *identifier* {`,` *identifier*}* .

hide-values → `hide-value`[`s`] : *identifier* {`,` *identifier*}* .

export-symbols → `export-symbol`[`s`] : *identifier* {`,` *identifier*}* .

B Sample Sessions

In the following, we present two sample sessions. The first one makes heavy use of Aït-Kaci's `append` encoding through types, the second one defines finite automata directly within $\mathcal{TDL}$ (see [Krieger et al. 93]).

B.1 Extracting List Elements

```
defdomain :less :load-built-ins-p nil.
begin :domain :less.
begin :declare.
  sorts: *built-in*, *null*.      ;; *null* represents the empty list < >
  NIL = *avm* & *built-in*.       ;; incompatibility declaration
end :declare.

begin :type.
  *avm* := [ ].                   ;; the top avm type
  *null* :< *built-in*.
  *list* := *null* | *cons*.
  *cons* := *avm* & [FIRST,REST *list*].
  append0 := *avm* & [FRONT < >,
                      BACK  #1 & *list*,
                      WHOLE #1].
  append1 := *avm* & [FRONT <#first . #rest1>,
                      BACK  #back & *list*,
                      WHOLE <#first . #rest2>,
                      PATCH append & [FRONT #rest1,
                                      BACK  #back,
                                      WHOLE #rest2]].
  append := append0 | append1.
  less := *avm* & [ELT #elt,
                   SET #set,
                   AUX append & [FRONT #front,
                                 BACK  < #elt . #rest >,
                                 WHOLE #set],
                   RES append & [FRONT #front,
                                 BACK  #rest]].
   w:= less & [ELT [E 0],
               SET <[A 1],[B 2],[C 3]>],
       doc: "Because [E 0] successfully unifies with every
             element of SET, RES will contain a disjunction
             of three lists, each of length 2.".
   expand-type 'w.
```

By using the type grapher of $\mathcal{TDL}$, we can depict the type hierarchy for this special type system (recall that thick lines indicates a disjunctive specification):

Expanding `w` (see sample session above) leads to the following feature structure—notice that we choose the feature editor FEGRAMED [Kiefer & Fettig 94] as the visualization tool (certain attributes are hidden). Another way to have access to this structure would be to employ the $\mathcal{TDL}$2LaTeX tool of $\mathcal{TDL}$ (see [Krieger & Schäfer 94a]).

B.2 Defining Finite Automata

```
defdomain :automata :load-built-ins-p NIL.
begin :domain :automata.
begin :declare.
  sorts:  *built-in*, *null*, *undef*.
  built-ins:  string, symbol, number.
  nil = *undef* & *built-in*.
  nil = *undef* & *avm*.
end :declare.
begin :type.
  symbol :< *built-in*.
  *null* :< *built-in*.
  string :< *built-in*.
  number :< *built-in*.
  *avm* := [ ].
  *cons* := *avm* & [FIRST, REST].
  *list* := *null* | *cons*.
  list-of-symbols := <...>:symbol.
  proto-config := *avm* &
    [EDGE:(symbol | *undef*),
     NEXT:(config | *undef*),
     INPUT:list-of-symbols].
  non-final-config := proto-config &
    [EDGE  #first,
     NEXT.INPUT  #rest,
     INPUT  <#first . #rest>].
  final-config := proto-config &
    [INPUT  < >,
     EDGE  *undef*,
     NEXT  *undef*].
  config := non-final-config | final-config.
;;; consider the two regular expressions R1=(a+b)^*c and R2=a(b^+)(c^*)
;;; the intersection of R1 and R2 is: R1&R2 = a(b^+)c
  U := non-final-config &
         [EDGE  %covary('a | 'b, 'c),
          NEXT  %covary(   U   ,  V)].
  V :< final-config.
  X := non-final-config &
         [EDGE  'a,
          NEXT   Y].     ;; expand-control: ((:delay (z next.*) (y next.*))).
  Y := non-final-config &
         [EDGE  'b,
```

```
        NEXT  Y | Z].  ;; expand-control: ((:delay (z next.*))).
Z := config &
       [EDGE  %covary( 'c, *undef*),
        NEXT  %covary(  Z, *undef*)].
test1 := U & X & [INPUT <'a,'b,'c>].
test2 := U & X & [INPUT <'a,'b,'b,'c>].
test3 := U & X & [INPUT <'b,'c>].
test4 := U & X & [INPUT <'a,'b,'c,'d>].
```

Expanding `test2` yields the following structure:

The type hierarchy is given by the following DAG:

References

[Aït-Kaci & Nasr 86] Hassan Aït-Kaci and Roger Nasr. *LOGIN: A Logic Programming Language with Built-In Inheritance.* Journal of Logic Programming, 3:185–215, 1986.

[Aït-Kaci et al. 85] Hassan Aït-Kaci, Robert Boyer, and Roger Nasr. *An Encoding Technique for the Efficient Implementation of Type Inheritance.* Technical Report AI-109-85, MCC, Austin, TX, 1985.

[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 et al. 93] Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. *Order-Sorted Feature Theory Unification.* Technical Report 32, Digital Equipment Corporation, DEC Paris Research Laboratory, France, May 1993. Also in Proceedings of the International Symposium on Logic Programming, Oct. 1993, MIT Press.

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

[Aït-Kaci 93] Hassan Aït-Kaci. *An Introduction to LIFE—Programming with Logic, Inheritance, Functions, and Equations.* In: Proceedings of the International Symposium on Logic Programming, pp. 52–68, 1993.

[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 (DFKI), Saarbrücken, Germany, 1992.

[Backofen & Weyers 94] Rolf Backofen and Christoph Weyers. *UDiNe—A Feature Constraint Solver with Distributed Disjunction and Classical Negation.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1994(95). Forthcoming, hopefully ;-}.

[Backofen et al. 93] Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, and Hans Uszkoreit (eds.). *Report of the EAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken.* Technical Report D-93-27, DFKI, Saarbrücken, 1993.

[Backofen 94] Rolf Backofen. *Expressivity and Decidability of First-Order Languages over Feature Trees.* PhD thesis, Universität des Saarlandes, Department of Computer Science, 1994. To appear.

[Blackburn 94] Patrick Blackburn. *Structures, Languages and Translations: the Structural Approach to Feature Logic.* In: C.J. Rupp, M.A. Rosner, and R.L. Johnson (eds.), Constraints, Language and Computation. Academic Press, 1994.

[Brew 93] Chris Brew. *Adding Preferences to CUF.* In: Jochen Dörre (ed.), Computational Aspects of Constraint-Based Linguistic Description I, pp. 54–69. ILLC/Department of Philosophy, University of Amsterdam, 1993. DYANA-2 Deliverable R1.2.A.

[Carpenter & Penn 94] Bob Carpenter and Gerald Penn. *ALE—The Attribute Logic Engine User's Guide. Version 2.0.* Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, August 1994.

[Carpenter 92] 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 & Dorna 93] Jochen Dörre and Michael Dorna. *CUF—A Formalism for Linguistic Knowledge Representation.* In: Jochen Dörre (ed.), Computational Aspects of Constraint-Based Linguistic Description I. DYANA, 1993.

[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.

[Eisele & Dörre 90] Andreas Eisele and Jochen Dörre. *Disjunctive Unification.* IWBS Report 124, IWBS, IBM Germany, Stuttgart, 1990.

[Emele & Zajac 90] Martin Emele and Rémi Zajac. *Typed Unification Grammars.* In: Proceedings of the 13th International Conference on Computational Linguistics, COLING-90, pp. 293–298, 1990.

[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 86] Robert T. Kasper and William C. Rounds. *A Logical Semantics for Feature Structures.* In: Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, pp. 257–266, 1986.

[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.

[Keller 93] Bill Keller. *Feature Logics, Infinitary Descriptions and Grammar.* CSLI Lecture Notes, Number 44. Stanford: Center for the Study of Language and Information, 1993.

[Kiefer & Fettig 94] Bernd Kiefer and Thomas Fettig. *FEGRAMED—An Interactive Graphics Editor for Feature Structures.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1994. Forthcoming.

[Kiefer & Scherf 94] Bernd Kiefer and Oliver Scherf. *Gimme more HQ Parsers. The Generic Parser Class of DISCO.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1994. 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. *TDL—A Type Description Language for Unification-Based Grammars.* In: Proceedings of the Workshop on "Neuere Entwicklungen der Deklarativen KI-Programmierung", KI-93, Berlin, 1993.

[Krieger & Schäfer 93b] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL ExtraLight User Guide.* Technical Report D-93-09, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1993.

[Krieger & Schäfer 94a] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for HPSG. Part 2: User Guide.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1994. Forthcoming.

[Krieger & Schäfer 94b] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for Constraint-Based Grammars.* In: Proceedings of the 15th International Conference on Computational Linguistics, COLING-94, Kyoto, Japan, pp. 893–899, 1994.

[Krieger et al. 93] Hans-Ulrich Krieger, John Nerbonne, and Hannes Pirker. *Feature-Based Allomorphy.* In: Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics, 1993. A version of this paper is available as DFKI Research Report RR-93-28.

[Krieger 95] Hans-Ulrich Krieger. *TDL—A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications.* PhD thesis, Universität des Saarlandes, Department of Computer Science, 1995. Forthcoming.

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

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd edition, 1987.

[Michie 68] Donald Michie. *Memo Functions and Machine Learning.* Nature, 218(1):19–22, 1968.

[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, DFKI, D-93-03, 1993.

[Norvig 91a] Peter Norvig. *Paradigms of Artificial Intelligence Programming.* San Mateo, CA: Morgan Kaufmann, 1991.

[Norvig 91b] 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 & Moshier 90] Carl J. Pollard and M. Drew Moshier. *Unifying Partial Descriptions of Sets.* In: P. Hanson (ed.), Information, Language, and Cognition. Vol. 1 of Vancouver Studies in Cognitive Science, pp. 285–322. University of British Columbia Press, 1990.

[Pollard & Sag 87] Carl Pollard and Ivan A. 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 94] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar.* Studies in Contemporary Linguistics. Chicago: University of Chicago Press, 1994.

[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.

[Rounds & Kasper 86] William C. Rounds and Robert T. Kasper. *A Complete Logical Calculus for Record Structures Representing Linguistic Information.* In: Proceedings of the 15th Annual Symposium of the IEEE on Logic in Computer Science, 1986.

[Rounds & Manaster-Ramer 87] William C. Rounds and Alexis Manaster-Ramer. *A Logical Version of Functional Grammar.* In: Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, pp. 89–96, 1987.

[Rupp et al. 94] C.J. Rupp, M.A. Rosner, and R.L. Johnson (eds.). *Constraints, Language and Computation.* Computation in Cognitive Science. Academic Press, 1994.

[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.

[Schäfer 95] Ulrich Schäfer. *Parametrizable Type Expansion for TDL.* Master's thesis, Universität des Saarlandes, Department of Computer Science, 1995. Forthcoming.

[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, May 1988. Also in J. Wedekind and C. Rohrer (eds.), Unification in Grammar, MIT Press, 1991.

[Smolka 89] Gert Smolka. *Feature Constraint Logic for Unification Grammars.* IWBS Report 93, IWBS, IBM Germany, Stuttgart, November 1989. Also in Journal of Logic Programming, 12:51–87, 1992.

[Uszkoreit et al. 94] Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. *DISCO—An HPSG-based NLP System and its Application for Appointment Scheduling.* In: Proceedings of COLING-94, Kyoto, Japan, pp. 436–440, 1994.

[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.

[Wahlster 93] Wolfgang Wahlster. *VERBMOBIL—Translation of Face-to-Face Dialogs.* Research Report RR-93-34, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany, 1993. Also appeared in: MT Summit IV, Kobe, Japan, July 1993.

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