Parameterized Type Expansion in the Feature Structure Formalism $\mathcal{TDL}$

Diplomarbeit
an der Universität des Saarlandes
Fachbereich 14 Informatik

vorgelegt von
Ulrich Schäfer
Deutschherrnstraße 27
66117 Saarbrücken

November 1995

Abstract

Over the last few years, unification-based grammar formalisms have become the predominant paradigm in natural language processing systems because of their monotonicity, declarativeness, and reversibility. From the viewpoint of computer science, typed feature structures can be seen as data structures that allow representation of linguistic knowledge in a uniform fashion.

Type expansion is an operation that makes the constraints on a typed feature structure explicit and determines their satisfiability. We describe an efficient expansion algorithm that takes care of recursive type definitions and allows exploration of different expansion strategies through the use of control knowledge. This knowledge is specified in a separate layer, independently of grammatical information. Memoization of the type expansion function drastically reduces the number of unifications.

In the second part, nonmonotonic extensions to $\mathcal{TDL}$ and the implementation of well-typedness checks are presented. Both are closely related to the type expansion algorithm. The algorithms have been implemented in Common Lisp and are integrated parts of $\mathcal{TDL}$ and a large natural language dialog system.

Contents

Chapter 1

Introduction and Motivation

Typed feature structure formalisms have become the basis for modern natural language processing systems because of their monotonicity, declarativeness, and reversibility. They also facilitate reusability of existing NL grammars [Rupp & Johnson 94]. Several large national (e.g., Verbmobil) and Europe-wide research projects are built around typed HPSG-like grammars and constraint-based formalisms. The European Commission has recognized the need for developing a formalism for machine translation and natural language processing. Several CEC initiatives are working on related themes (e.g., EAGLES; see [Backofen et al. 93]).
Feature structures can be seen as data structures which allow the declarative specification of linguistic knowledge. They consist of (possibly nested) feature-value pairs which describe entities such as words, phrases, and sentences.
The values of features can be atoms, disjunctions, negated values or, again, feature structures.
Unification is the operation that combines the information encoded in two feature structures or establishes their incompatibility.
Types allow hierarchical ordering of feature structures in an inheritance network, and serve as abbreviations for complex feature structures via *type definitions*. Typically, a grammar for natural language consists of a large set of type definitions.
Feature structures augmented by types are called *typed feature structures*. *Type expansion* then is the operation that replaces type names in typed feature structures by their definitions using unification. The motivation for type expansion in manifold:

- *Economy*: It is not necessary always to work with fully expanded feature structures. Lexicon entries in highly lexicalized grammar theories such as HPSG [Pollard & Sag 87; Pollard & Sag 94], e.g., are very complex feature structures. A large lexicon can not be loaded fully expanded in memory because it would consume too much space. *Type expansion* expands lexicon entries and other typed feature structures at run time if they are required by the parser or generator and thus helps to save memory in linguistic processing. On the other hand, it makes sense to expand pre-lexical types statically (partial evaluation).

- *Efficiency*: Working with partially expanded feature structures reduces the costs of copying and unifying during parsing or generation. *Type expansion* is necessary to support this strategy.

- *Checking consistency*: Type definitions or partially expanded feature structures can be inconsistent. *Type expansion* is the operation that determines whether a typed feature structure (or a part of it) is consistent or inconsistent.

- *Making knowledge explicit*: Although working with partially expanded feature structures is useful at run time, there must be some point of time where all knowledge structured by types is made explicit by *type expansion* – otherwise, their specification would have been redundant. In this sense, type expansion is a structure building operation like unification.

- *Expressivity*: Admitting recursive type definitions increases the expressivity of typed feature structures. *Type expansion* is necessary to exploit this property and offers additional methods for grammar writers to formulate linguistic knowledge, e.g., relational *append* à la Prolog, finite state automata, functional uncertainty, and much more. Furthermore, recursive types are indispensable when working in the *parsing as deduction* paradigm without specialized parsers (or generators) for annotated context-free rules.

In this thesis, we describe a new approach to type expansion for the typed feature structure formalism $\mathcal{TDL}$ (Type Definition Language) developed at the Computational Linguistics Department of the German Research Center for Artificial Intelligence (DFKI). $\mathcal{TDL}$ is a comprehensive system designed for (but not restricted to) the development and run time support of HPSG-based natural language grammars [Krieger & Schäfer 93a; Krieger & Schäfer 94b; Krieger & Schäfer 94c; Krieger & Schäfer 94a; Krieger 95b; Uszkoreit et al. 94]. The main advantages of $\mathcal{TDL}$ in constrast to related systems are

- rich type system with atoms, atomic sorts, and complex feature types in either closed or open type world

- full boolean type logic

- type partitions and incompatible types can be declared

- specialized modules for feature unification ($\mathcal{UDiNe}$) and type system (hierarchy and simplification)

- coreferences, distributed disjunctions, classical negation, cyclic feature structures and feature unification are supported by the constraint solver $\mathcal{UDiNe}$

- templates (macros with parameters) support maintenance of large grammars

- 'instance' facility for lexicon entries and other feature structures that need not be defined as types

No other system offers the full range of these features that are demanded by grammar engineers.
The aim of the implementation part of this thesis is to devise a parameterizable expansion algorithm for $\mathcal{TDL}$ that meets the following requirements:

- type expansion as a proper module, in contrast to expansion mechanisms that are integrated into typed unification [Aït-Kaci et al. 94] or type definitions [Carpenter & Penn 94]
- support the sophisticated type system of $\mathcal{TDL}$, including disjunctive types, closed-world sorts and open or closed-world feature structure types
- support correct expansion of recursive types and partially expanded structures
- be parameterized for expansion strategy (depth-first, breadth-first), attribute selection, maximal path depth, preference information, etc.

Additionally, two further extensions related to type expansion are added to the $\mathcal{TDL}$ formalism:
Nonmonotonicity: Nonmonotonic type or instance definitions are useful for modelling defaults and exceptions which are linguistically motivated. Since monotonic unification is used during expansion, the *type expansion* algorithm is sensitive to the order in which feature structures are overwritten nonmonotonically.
Well-Typedness Check: In $\mathcal{TDL}$, the appropriateness specification of features can be derived from the type definitions. An optional check on feature structures uses this information to guarantee well-typedness as defined in [Carpenter 92]. The checking algorithm also uses results from *type expansion* to achieve this. Moreover, a feature structure is *totally well-typed* if it is *well-typed* and *fully expanded*.

The thesis is organized as follows. In the next chapter, we briefly describe the history of (typed) feature structures and the outcome of interdisciplinary research on that field in logic and computer science. In Chapter 3, we give an overview of the $\mathcal{TDL}$ formalism and architecture as far as is important for type expansion and introduce the main concepts and definitions used in the thesis. In Chapter 4, we describe extensions of the $\mathcal{TDL}$ representation of typed feature structures that are necessary for the implementation of the expansion algorithm. In the second part of this chapter, we describe the basic structure of the expansion algorithm. The algorithm will be refined during the subsequent chapters.
Chapter 5 introduces a technique we call indexed prototype memoization. It is used to reduce the number of unifications and to detect recursive types during expansion. In Chapter 6, the

treatment of recursive types and lazy expansion is described. We also discuss decidability results for feature logic with recursive type definitions. Chapter 7 explains how control information for the expansion algorithm is specified declaratively and how it is implemented in the algorithm.
Nonmonotonic extensions to the $\mathcal{TDL}$ language and their interaction with type expansion is the issue of Chapter 8. In Chapter 9, appropriateness and well-typedness is defined and the various implemented checks are presented. In Chapter 10, we discuss related systems such as ALE, TFS, and CUF and compare them to the implemented $\mathcal{TDL}$ system. Finally, a conclusion and a look at possible future work is given in Chapter 11.
Appendix A contains a BNF of $\mathcal{TDL}$'s type definition language and Appendix B a sample session.

Acknowledgements: First of all, I would like to thank my advisor, Hans-Ulrich Krieger, for his help, encouragement, and many discussions. Rolf Backofen and Christoph Weyers helped me to understand their feature constraint solver $\mathcal{UDiNe}$. Some $\mathcal{TDL}$ users and involuntary beta-testers at DFKI and CSLI discovered bugs and made helpful comments: Stephan Busemann, Elizabeth Hinkelman, Walter Kasper, Robert Malouf, Klaus Netter, Stephan Oepen, and Hannes Pirker (now at ÖFAI).
Finally, I am grateful to Elizabeth Hinkelman for reading a draft of this thesis, and to Hans-Ulrich Krieger and Hans Uszkoreit for helpful comments.

Chapter 2

Feature Structure Formalisms

2.1 Feature Structures

The concept of feature-value pairs is a natural one. It has arisen independently in linguistics and computer science. [Jakobson et al. 51] introduce so-called *distinctive features* in order to characterize phonemes in spoken language. A *bundle* of distinctive features with binary values describes each phoneme uniquely, e.g., the phoneme /p/ is

$$\begin{bmatrix} \text{PLOSIVE} & + \\ \text{VOICED} & - \\ \text{BILABIAL} & + \\ \text{NASAL} & - \end{bmatrix},$$

while /b/ has the same values except VOICED +. Later, binary values have been replaced by arbitrary values.[1]

The need for nested descriptions has evolved from research in syntax in the 60's and semantics in the 70's (cf. [Uszkoreit 88] for an overview). Martin Kay was the first one to propose encoding syntactic features and phrase structure uniformly in feature structures [Kay 84].

The following complex structure states syntactic agreement properties of a nominal phrase. $\boxed{1}$ expresses token identity between the values of the two features SUBJECT and AGREEMENT.

$$\begin{bmatrix} \text{CAT} & \texttt{np} \\ \text{AGREEMENT} & \boxed{1} \begin{bmatrix} \text{NUMBER} & \texttt{singular} \\ \text{PERSON} & \texttt{third} \end{bmatrix} \\ \text{SUBJECT} & \boxed{1} \end{bmatrix}$$

One can characterize such descriptions as rooted, directed, labelled graphs, where features are the labels, and values are the nodes of the graph.

[1] In the original work, the position of attributes (features) and values was value-attribute, but the attribute-value order gained acceptance because of its conveniance for nested descriptions. The feature notation in brackets is called avm notation (for attribute value matrix).

This idea has been extended to very complex descriptions for all kinds of linguistic entitites such as phonemes, morphemes, words, phrases, and sentences, which are uniformly encoded in feature structures while in earlier formalisms, feature structures have been used to annotate context free rules (LFG, [Bresnan 82], PATR II [Shieber et al. 83], GPSG [Gazdar et al. 85]).

2.2 Unification

The basic operation on feature structures is *feature unification* ($\sqcap$). Introduced by Martin Kay in Functional Grammar [Kay 79], feature unification is a monotonic, structure building operation which takes two descriptions, and results in the most general feature structure that satisfies both descriptions. If the two descriptions are not consistent, feature unification *fails*. For example:

$$\begin{bmatrix} \text{CAT} & \texttt{np} \\ \text{AGREEMENT} & \boxed{1}\begin{bmatrix} \text{NUMBER} & \texttt{singular} \\ \text{PERSON} & \texttt{third} \end{bmatrix} \\ \text{SUBJECT} & \boxed{1} \end{bmatrix} \sqcap \begin{bmatrix} \text{SUBJECT} & \begin{bmatrix} \text{GENDER} & \texttt{female} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} & & \begin{bmatrix} \text{GENDER} & \texttt{female} \\ \text{NUMBER} & \texttt{singular} \\ \text{PERSON} & \texttt{third} \end{bmatrix} \\ \text{SUBJECT} & \boxed{1} & \\ \text{AGREEMENT} & \boxed{1} & \\ \text{CAT} & \texttt{np} & \end{bmatrix}$$

but

$$\begin{bmatrix} \text{CAT} & \texttt{np} \\ \text{AGREEMENT} & \boxed{1}\begin{bmatrix} \text{NUMBER} & \texttt{singular} \\ \text{PERSON} & \texttt{third} \end{bmatrix} \\ \text{SUBJECT} & \boxed{1} \end{bmatrix} \sqcap \begin{bmatrix} \text{SUBJECT} & \begin{bmatrix} \text{NUMBER} & \texttt{plural} \end{bmatrix} \end{bmatrix}$$

is inconsistent.

Since the introduction of feature unification, feature structure formalisms have been enriched by negation, disjunction and set values [Karttunen 84]. For example,

$$\begin{bmatrix} \text{CAT} & \neg\ \texttt{vp} \end{bmatrix}$$

states that the value of feature CAT is not `vp`.

An example of a disjunction is

$$\begin{bmatrix} \text{SUBJECT} & \begin{bmatrix} \text{PERSON} & \left\{ \begin{matrix} \texttt{first} \\ \texttt{third} \end{matrix} \right\} \end{bmatrix} \end{bmatrix}$$

i.e., the value at path SUBJECT.PERSON can be either `first` or `third`.

An introduction to unification-based grammar formalisms can be found in [Shieber 86].

2.3 Types and Inheritance

At first glance, one could compare feature structures with record data types in imperative programming languages, object-oriented programming languages, or data base systems.

However, these structured data types are more general than feature structures in that they can consist of arbitrary data types such as arrays, hashtables, etc. On the other hand, they often lack negation and disjunction.

Moreover, the monotonic unification operation is not supported by these languages. Hence, they have not influenced computational linguistics very much, although they have often been used at low level to implement feature structures (e.g., Common Lisp's *structure* data type is employed in the $\mathcal{TDL}/\mathcal{UDiNe}$ system).

However, there are three fields in computer science that gave new impetus to the development of feature structure formalisms. First, the notion of *unification* originated from the logic programming and Prolog community, and functional and relational constraints have been adopted as extensions to unification-based formalisms (e.g., definite clauses in ALE, [Carpenter & Penn 94]).

Second, from knowledge representation has come the idea of *inheritance* that allows hierarchical ordering of feature structures and elimination of redundant definitions. Inheritance is especially useful in building and maintaining large lexica. Early formalisms provided templates for the purpose of abbreviating complex feature structures.

Shown below is a sample definition of the template `Intransitive` from PATR II [Shieber 86]:

```
Let Intransitive be MainVerb
                    <subcat first cat> = NP
                    <subcat rest> = end
                    <head trans arg1> = <subcat first head trans>.
```

Here, `Intransitive` inherits from template `MainVerb`, `<...>` denote feature paths, `NP` is a phrase symbol, and `end` is an atom.

Last but not least, feature structures have been augmented by hierarchically ordered *types* known from object-oriented programming languages [Cardelli & Wegner 85].

In *typed feature structures*, the nodes of the feature graph are enriched by type symbols. This holds for complex feature-value nodes as well as for atomic nodes. Throughout the thesis, type names are printed in *italics*, and features in UPPERCASE letters, $\langle\,\rangle$ denotes the empty list. Lists are usually encoded by first/rest feature structures (cf. the `CAR/CDR` representation in Lisp). The end of a list is encoded by a special type or atom such as *null* or *end*, depicted

by $\langle\,\rangle$ as below.

$$\begin{bmatrix} \textit{sign} \\ \text{SYNSEM} \begin{bmatrix} \textit{synsem} \\ \text{LOCAL} \begin{bmatrix} \textit{local} \\ \text{CATEGORY} \begin{bmatrix} \textit{cat} \\ \text{HEAD} \quad \textit{noun} \\ \text{SUBCAT} \ \langle\,\rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Unification is extended to typed unification which consists of feature unification plus the greatest lower bound (*glb*) operation on the corresponding types. Typed unification is successful if both operations are successful.
Interestingly, Hassan Aït-Kaci developed a very similar formalism for knowledge representation in his 1984 dissertation independently of linguistic motivation. His so-called ψ-terms [Aït-Kaci 86] are essentially the same as feature structures with types, coreferences, and unification.[2]
The main advantages of types in feature structures are:

- *efficiency*: Types can be used to (partially) replace complex feature structures and to impose additional constraints on typed feature structures, e.g., by partitioning the type hierarchy. Types have their own inference mechanism which coexists with untyped feature unification. Fast encoding techniques for type hierarchies like the bit vector encoding of [Aït-Kaci et al. 89] reduce the costs for the least upper bound, greatest lower bound and subsumption operations to nearly constant time (for a fixed type system).

- *modularity*: Like templates, types can be employed to abbreviate complex feature structures (via *type definitions* that establish ISA-links). Hierarchical ordering of types allows modular specification of complex grammars and lexica. Multiple inheritance is achieved by simply unifying the definitions of several supertypes.

- *safety*: As is the case in modern programming languages, type checking can contribute to reduction of the number of typographical or conceptual errors in large grammars. *Appropriateness specifications* describe what types are suitable attribute values in typed feature structures and are considered in type checking.

- *expressivity*: Types can be used to enrich the expressive power of feature structure formalisms. The feature that makes them Turing-equivalent is admission of recursive type definitions.

[2] There are other approaches in knowledge representation that bear a resemblance to typed feature structures, namely KL-ONE-like terminological languages [Brachman & Schmolze 85]. The main difference is that they generalize features (partial functions) to roles (relations), and often lack complements and coreferences.

It is worth emphasizing that these goals (except the second) cannot be achieved by templates which are comparable to purely syntactic macros in programming languages.
Typed (or sorted) feature structures became popular in connection with Head Driven Phrase Structure Grammar (HPSG, [Pollard & Sag 87; Pollard & Sag 94]). Equally, one might claim that HPSG became popular because it is the first grammar theory that uses typed feature structures effectively: they are applied uniformly to all kinds of linguistic knowledge.
Today, HPSG is the most important grammar theory in computational linguistics. It combines aspects of GB [Chomsky 81], GPSG, LFG, and Situation Semantics.
Because HPSG is a highly lexicalized grammar theory, that is, it tries to encode as much information as possible in the lexicon and the rest in very general rules and principles, (multiple) inheritance and types play an important rôle.
The success of HPSG stems from the fact that it unifies linguistic theories with ideas from knowledge representation and computer science, and addresses the structure of the lexicon very detailed.

2.4 Formalizations

Formalizations from different theoretical points of view have clarified the relationship between feature structures and (deterministic) finite automata, first-order logic and set-theoretical semantics of knowledge representation languages.
[Kasper & Rounds 86] lead off this work by characterizing feature structures via finite automata (where features are the transition labels and values correspond to states), developing the first attribute-value logic, and analyzing complexity.
[Johnson 88] shows that feature logic with classical negation is a decidable subset of first-order logic. [Smolka 88] presents a set-theoretical semantics for sorted feature structures similar to that for terminological languages like KL-ONE.
[Carpenter 92] gives a comprehensive introduction to the theory of typed feature structures, basic definitions, and various pointers to the literature.

2.5 Formalisms

Over the last years, many feature structure formalisms have been developed for applications in computational linguistics. Because there is no standard point of view about what criteria feature formalisms must fulfill, they differ not only in syntax, but also in expressivity and efficiency of implementation.
In addition, some of the formalisms are integrated parts of complex natural language systems with specialized modules such as a parser, generator, etc., whereas some others are stand-alone systems with interface facilities to provide connections with foreign modules.

One of the earliest and most famous formalisms is PATR-II [Shieber et al. 83]. Along with its derivatives and with Definite Clause Grammars (DCGs, [Pereira & Warren 80]) that are close to Prologs term representation and have fixed feature arity, they form the first generation of untyped feature structure formalisms. Because of their simplicity, many PATR- and DCG-based systems are still in use today in real applications.

The second generation of feature formalisms makes use of type hierarchies. The oldest representative, although not specifically designed for linguistic applications, is Aït-Kaci's LOGIN (LOGic and INheritance, [Aït-Kaci & Nasr 86]), a language based on Prolog with ψ-terms, types and inheritance.

Its successor LIFE provides functions in addition [Aït-Kaci & Lincoln 88; Aït-Kaci 93; Aït-Kaci et al. 94]. Both languages are mainly intended as programming and knowledge representation languages, but strongly influenced the implementation of NL formalisms.

TFS [Emele & Zajac 90] has been developed especially for the declarative specification of HPSG grammars. It provides relations and recursive types and has been strongly inspired by Aït-Kaci's work.

ALE [Carpenter & Penn 94] and CUF [Dörre & Dorna 93] share many properties with logic programming languages. Both have a restricted type system. ALE postulates fixed feature arity and does not provide features with disjunctive values.

$\mathcal{TDL}$ *ExtraLight* [Krieger & Schäfer 93b], the predecessor of $\mathcal{TDL}$, allows for multiple inheritance, and is, like $\mathcal{TDL}$, based on the constraint solver $\mathcal{UDiNe}$, a feature unification system with distributed disjunctions and full classical negation. While $\mathcal{TDL}$ *ExtraLight*'s type system relied on the type system of the CLOS and hence was very restricted in some respect, $\mathcal{TDL}$'s type hierarchy is rich and employs an extension of the bit-vector encoding from [Aït-Kaci et al. 89; Krieger 95a]. We will have a closer look at $\mathcal{TDL}$ in the following chapter.

Chapter 3

$\mathcal{TDL}$

In this chapter, we define basic concepts, and give an introduction to $\mathcal{TDL}$, its type definition language and architecture. We do not intend to describe $\mathcal{TDL}$ in full detail. Instead, we restrict the description to the parts of the formalism that are relevant to type expansion (e.g., templates or other syntactic sugar like $\mathcal{TDL}$'s rule syntax will be ignored). Furthermore, we abstract from the complex type system of $\mathcal{TDL}$ as much as possible in order to keep the expansion algorithm clear and to make it transferable to other formalisms. For a detailed description of $\mathcal{TDL}$, refer to [Krieger & Schäfer 94b], [Krieger & Schäfer 94c], and [Krieger 95b].

3.1 Basic Definitions

Definition 1 Type system
A $\mathcal{TDL}$ Type System (or Signature) Σ is a tuple $(\mathcal{F}, \mathcal{A}, \mathcal{T}, \top, \bot, \preceq, \Theta, \mathcal{V}, \Delta)$, where

- $\mathcal{F}$ is a set of *feature symbols* (attribute names)
- $\mathcal{A}$ is a set of *atoms* (e.g., symbols, numbers, strings) without ordering
- $\mathcal{T}$ is a set of *types*, itself partitioned in four disjoint sets:
  - $\mathcal{T}_a$, the set of complex *avm types*, which can bear features
  - $\mathcal{T}_s$, the set of *sorts*, which cannot have features (hierarchically ordered atoms)
  - $\mathcal{T}_b$, the set of *built-in sorts*, which correspond to admissible data types for atoms, e.g., *Integer*, *String*, *Symbol*, and *Number*
  - $\{\top\}$, the set that consists of the *top type* $\top$ of the hierarchy, i.e., the most general type that subsumes all other types of the hierarchy

  We assume that $\mathcal{A} \cap \mathcal{T} = \emptyset$
- $\bot \notin \mathcal{T}$ is the bottom symbol, which indicates inconsistency between types

- $\preceq \subseteq (\mathcal{T} \cup \{\bot\}) \times (\mathcal{T} \cup \{\bot\})$ is the *type subsumption order*, a reflexive partial order on types. $\preceq$ orders types according to their specificity and is induced through the type definition function Δ (see below). $\tau \preceq \sigma$ iff $\sigma \in \mathcal{T}$ is a *supertype* of $\tau \in \mathcal{T}$. We stipulate that $\forall \tau \in \mathcal{T} : \tau \preceq \top$
- Θ is a set of typed feature structures (definition follows)
- $\mathcal{V}$ is the set of variables. Variables indicate structure sharing (reentrancies) in feature structures
- $\Delta : \mathcal{T} \mapsto \Theta$ is the *type definition function*. It assigns to each type $\tau \in \mathcal{T}$ a typed feature structure $\theta \in \Theta$. θ is called the *skeleton* of the type definition.

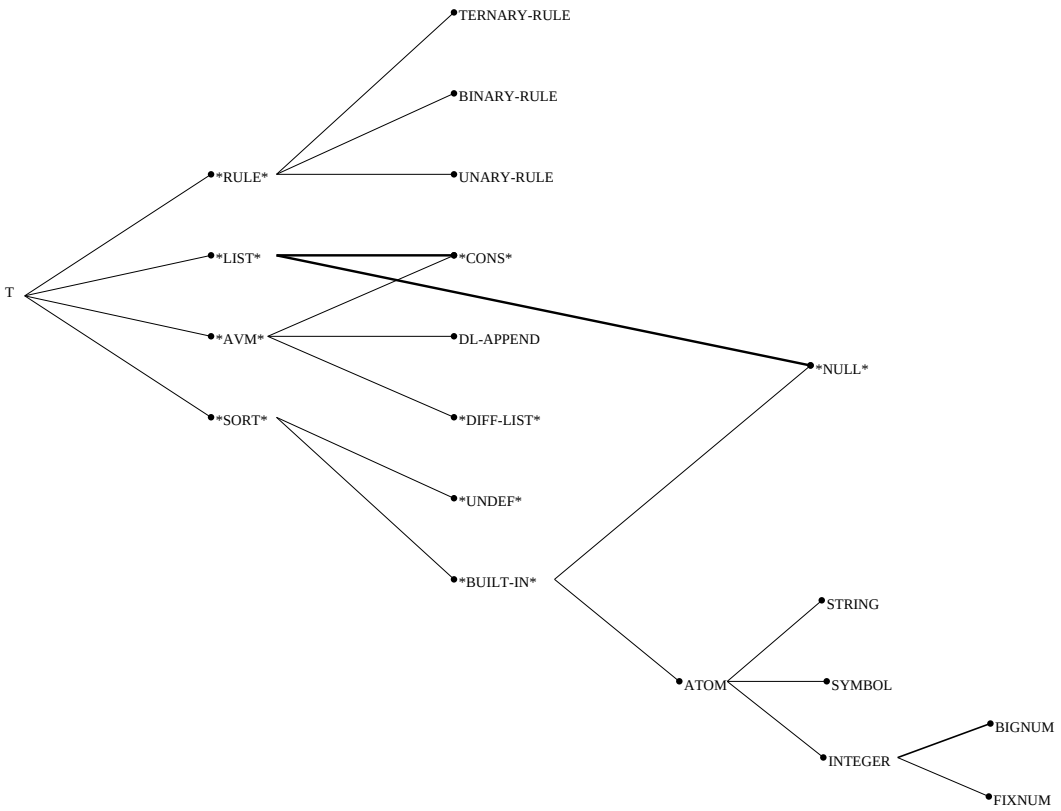


Figure 3.1: *An initial type hierarchy of TDL.*

It is important to realize that this definition of a type system does not impose any semantic constraints on the type hierarchy except that it be a partial order and that $\tau \preceq \top$ for all $\tau \in \mathcal{T}$. In contrast to [Carpenter 92], we do not require $\langle \mathcal{T}, \preceq \rangle$ to be a bounded complete partial order (BCPO).

The type subsumption order is determined by the type definition function Δ that establishes ISA-links between types in the hierarchy (cf. the following section). Further constraints on the hierarchy are imposed by type simplification rules.

Definition 2 Syntax of type expressions
Type Expressions are defined inductively:

- $\forall \tau \in \mathcal{T} : \tau$ is a type expression
- $\forall a \in \mathcal{A} : a$ is a type expression
- if σ is a type expression, then $\neg\sigma$ is a type expression (type negation)
- if σ and τ are type expressions, then $\sigma \wedge \tau$ is a type expression (type conjunction)
- if σ and τ are type expressions, then $\sigma \vee \tau$ is a type expression (type disjunction)

$(\mathcal{T} \cup \mathcal{A})^*$ is the set of all type expressions.

In the implementation, type simplification at definition and run time guarantees that type expressions are always in normal form. Either conjunctive or disjunctive normal form can be choosen through a global switch.
Expressions of the form $\rho_1 \wedge \cdots \wedge \rho_n$ can occur because $\mathcal{TDL}$ does not require that the greatest lower bound type of $\rho_1, \ldots, \rho_n$ always exists (the same holds for least upper bounds). This behavior ('open world *lub*/*glb* reasoning') can also be controlled by a global switch. In a closed *glb* world, the expression $\rho_1 \wedge \cdots \wedge \rho_n$ would be inconsistent if the greatest lower bound type did not exist. Most other typed feature structure formalisms only admit this 'closed world' view.

Definition 3 Syntax of typed feature structures (preliminary)
A *Typed Feature Structure* $\theta \in \Theta$ is either

- $\langle x, \tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n] \rangle$, a *conjunctive typed feature structure*, where
  $x \in \mathcal{V}$ is the Variable associated with the structure,
  $\tau \in (\mathcal{T} \cup \mathcal{A})^*$ is the (possibly complex) *type* (also *type entry* or *head*) of the feature structure,
  $f_1, \ldots, f_n \in \mathcal{F}$ are the *features* or *attributes*, and
  $\theta_1, \ldots, \theta_n \in \Theta$ the *values* of the features with $n \in \{0, 1, 2, \ldots\}$. If $n = 0$, then the structure bears no features. This is always the case for sorts and atoms.
- $\langle x, \{\theta_1, \ldots, \theta_m\} \rangle$ is a *disjunctive typed feature structure* or *disjunction*, where
  $x \in \mathcal{V}$ is the variable associated with the disjunction,
  $\theta_1, \ldots, \theta_m \in \Theta$ are the disjunction elements with $m \in \{1, 2, \ldots\}$.

This representation of typed feature structures is a generalization of Aït-Kaci's ϵ-terms which themselves are ψ-terms with a compact notation for disjunctions [Aït-Kaci 86]. $\mathcal{TDL}$ generalizes ϵ-terms in that the head of the term is not only a type symbol, but a complex type expression involving connectives $\wedge$, $\vee$, and $\neg$ in addition to simple type symbols.
The correlation of this representation with the avm notation for typed feature structures is obvious. Conjunctive feature structures correspond to the feature-value lists in brackets, disjunctive feature structures to alternative lists in braces. Variables which occur only once are omitted in the avm notation. Otherwise, they correspond to the boxed coreference tags which indicate structure sharing. For example, the avm

$$\left[\begin{array}{ll} \textit{npsg23} & \\ \text{AGREEMENT } \boxed{x} & \left[\begin{array}{l} \textit{sg23} \\ \text{NUMBER } \texttt{singular} \\ \text{PERSON } \left\{\begin{array}{l}\texttt{second}\\ \texttt{third}\end{array}\right\} \end{array}\right] \\ \text{CAT} & \texttt{np} \\ \text{SUBJECT} & \boxed{x} \end{array}\right]$$

can be translated into

$$\begin{array}{l} \langle \textit{npsg23}, [\text{AGREEMENT} \doteq \langle x, \textit{sg23}, [\text{NUMBER} \doteq \texttt{singular}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{PERSON} \doteq \langle \{\texttt{second}, \texttt{third}\}\rangle]\rangle, \\ \qquad \text{CAT} \doteq \texttt{np}, \\ \qquad \text{SUBJECT} \doteq x]\rangle \end{array}$$

Variables that occur only once are omitted, a tagged empty node such as $\langle x, \top, [\,]\rangle$ is abbreviated as x. Atomic and sort values are abbreviated to `atom` instead of $\langle \texttt{atom}, [\,]\rangle$.

Because the latter representation is hard to read if structures are large, we mainly use it to represent the general format of feature structures in algorithms. For sample structures, we prefer the avm notation.

Although the $\mathcal{UDiNe}$ feature constraint solver supports distributed (named) disjunctions, they add no expressive power and will therefore be ignored here.

$\mathcal{TDL}$'s feature structure representation provides two ways to express disjunction, one at the type level and one at the feature structure level. Actually, the two feature structures $\langle x, \tau_1 \vee \cdots \vee \tau_n, [\,]\rangle$ and $\langle x, \{\langle x_1, \tau_1, [\,]\rangle, \ldots, \langle x_n, \tau_n, [\,]\rangle\}\rangle$ are equivalent modulo variable renaming (cf. their set-theoretical semantics in the next section). However, the second representation is the canonical one, since it conforms to $\mathcal{UDiNe}$'s distributed disjunction representation, that has a fixed disjunct order and admits coreferences between disjuncts only at the same position.[1] All

[1] For the sake of simplicity, we do not consider distributed disjunctions except in sample structures in this thesis. Disjunction names are treated 'invisible' to the expansion algorithms. This reflects exactly the implementation that ignores them as well. However, distributed disjunctions are handled by the implemented algorithms the same way as the trivial case of normal disjunctions.

disjunctions in $\mathcal{TDL}$'s feature structures are translated into this representation at definition time. The only occasion where the first representation can appear in $\mathcal{TDL}$, is the application of DeMorgan's law when negation takes place. As will be shown later, the type expansion process handles canonicalization of this case.
Disjunctive feature structures in $\mathcal{TDL}/\mathcal{UDiNe}$ are always untyped. The type of a disjunction node can be determined immediately by combining the types of its elements with the $\vee$ operator.

3.2 Set-Theoretical Semantics

$\mathcal{TDL}$ can be given a set-theoretical semantics along the lines of [Smolka 88]. Smolka's approach is closely related to the set-theoretical semantics of KL-ONE-like terminological logics.
We will only briefly sketch the semantics for $\mathcal{TDL}$ here; a more complete approach with fixed point construction is addressed in [Krieger 95b].

Definition 4 Interpretation of the type system Σ
The *interpretation* $\mathcal{I}$ assigns denotations to features, types, sorts, and atoms:

- $\top^{\mathcal{I}}$ is a set called the universe of $\mathcal{I}$
- $\bot^{\mathcal{I}}$ is the empty set
- if $\mathtt{a} \in \mathcal{A}$, then $\mathtt{a}^{\mathcal{I}}$ is a set consisting of exactly one element (singleton)
- $\forall \mathtt{a}, \mathtt{b} \in \mathcal{A}$: if $\mathtt{a} \neq \mathtt{b}$, then $\mathtt{a}^{\mathcal{I}} \neq \mathtt{b}^{\mathcal{I}}$, i.e., different atoms have different denotations
- if $\tau \in \mathcal{T}$, then $\tau^{\mathcal{I}} \subseteq \top^{\mathcal{I}}$, i.e., types denote subsets of the universe
- if $\sigma, \tau \in \mathcal{T}$ and $\rho \in \mathcal{T}$ is the greatest lower bound (*glb*) of σ and τ, then $\rho^{\mathcal{I}} = \sigma^{\mathcal{I}} \cap \tau^{\mathcal{I}}$, i.e., *glb* corresponds to set intersection
- if $\sigma, \tau \in \mathcal{T}$ and $\rho \in \mathcal{T}$ is the least upper bound (*lub*) of σ and τ, then $\rho^{\mathcal{I}} = \sigma^{\mathcal{I}} \cup \tau^{\mathcal{I}}$, i.e., *lub* corresponds to set union
- if $\sigma, \tau \in \mathcal{T}$ and $\sigma \preceq \tau$, then $\sigma^{\mathcal{I}} \subseteq \tau^{\mathcal{I}}$
- if $f \in \mathcal{F}$, then $f^{\mathcal{I}} : \mathcal{D}_f^{\mathcal{I}} \mapsto \top^{\mathcal{I}}$ is a function where $\mathcal{D}_f^{\mathcal{I}} \subseteq \top^{\mathcal{I}}$ is the domain of f in $\mathcal{I}$, i.e., features are interpreted as functions
- if $f \in \mathcal{F}$ and $\mathtt{a} \in \mathcal{A}$, then $\mathcal{D}_f^{\mathcal{I}} \cap \mathtt{a}^{\mathcal{I}} = \emptyset$, i.e., atoms cannot bear features
- if $f \in \mathcal{F}$ and $\tau \in \mathcal{T}_s \cup \mathcal{T}_b$, then $\mathcal{D}_f^{\mathcal{I}} \cap \tau^{\mathcal{I}} = \emptyset$, i.e., sorts cannot bear features

Definition 5 Denotation of variables
Let $\alpha : \mathcal{V} \mapsto \top^{\mathcal{I}}$ be the function that assigns variables to the universe. Then $[\![x]\!]_\alpha^{\mathcal{I}} := \{\alpha(x)\}$, i.e., the denotation of a variable x is a singleton set consisting of $\alpha(x)$.

Definition 6 Denotation of type expressions
The denotation of atoms and types is given by their interpretation $\mathcal{I}$ and Variable assignment α. Conjunction of type expressions denotes set intersection, disjunction (generalization) denotes set union, negation denotes set complement.

- $\forall \mathtt{a} \in \mathcal{A} : [\![\mathtt{a}]\!]^{\mathcal{I}}_{\alpha} := \mathtt{a}^{\mathcal{I}}$
- $\forall \tau \in \mathcal{T} : [\![\tau]\!]^{\mathcal{I}}_{\alpha} := \tau^{\mathcal{I}}$
- $\forall \sigma, \tau \in (\mathcal{T} \cup \mathcal{A})^* : [\![\sigma \wedge \tau]\!]^{\mathcal{I}}_{\alpha} := [\![\sigma]\!]^{\mathcal{I}}_{\alpha} \cap [\![\tau]\!]^{\mathcal{I}}_{\alpha}$
- $\forall \sigma, \tau \in (\mathcal{T} \cup \mathcal{A})^* : [\![\sigma \vee \tau]\!]^{\mathcal{I}}_{\alpha} := [\![\sigma]\!]^{\mathcal{I}}_{\alpha} \cup [\![\tau]\!]^{\mathcal{I}}_{\alpha}$
- $\forall \sigma \in (\mathcal{T} \cup \mathcal{A})^* : [\![\neg\sigma]\!]^{\mathcal{I}}_{\alpha} := \top^{\mathcal{I}} - [\![\sigma]\!]^{\mathcal{I}}_{\alpha}$

Definition 7 Denotation of feature-value constraints
The denotation of a feature-value constraint $f \doteq \theta$ ($f \in \mathcal{F}$, $\theta \in \Theta$) under interpretation $\mathcal{I}$ and variable assignment α is defined as follows

$$[\![f \doteq \theta]\!]^{\mathcal{I}}_{\alpha} := \{\alpha(x) | f^{\mathcal{I}}(\alpha(x)) \in [\![\theta]\!]^{\mathcal{I}}_{\alpha}\}.$$

The denotation of a set of feature-value constraints is interpreted as the intersection of their denotations.

Definition 8 Denotation of typed feature structures
The denotation of a typed feature structure $\theta \in \Theta$ under interpretation $\mathcal{I}$ and variable assignment α is defined as follows ($x \in \mathcal{V}$, $\tau \in (\mathcal{T} \cup \mathcal{A})^*$, $f_i \in \mathcal{F}$, $\theta_i \in \Theta$).

- If θ is a conjunctive typed feature structure, i.e., $\theta = \langle x, \tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n]\rangle$, then

$$\begin{aligned} [\![\theta]\!]^{\mathcal{I}}_{\alpha} &= [\![\langle x, \tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n]\rangle]\!]^{\mathcal{I}}_{\alpha} \\ &:= [\![x]\!]^{\mathcal{I}}_{\alpha} \cap [\![\tau]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{n} [\![f_i \doteq \theta_i]\!]^{\mathcal{I}}_{\alpha} \\ &= \{\alpha(x)\} \cap [\![\tau]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{n} \{\alpha(y) | f_i^{\mathcal{I}}(\alpha(y)) \in [\![\theta_i]\!]^{\mathcal{I}}_{\alpha}\} \end{aligned}$$

- If θ is a disjunctive typed feature structure, i.e., $\theta = \langle x, \{\theta_1, \ldots, \theta_n\}\rangle$, then

$$\begin{aligned} [\![\theta]\!]^{\mathcal{I}}_{\alpha} &= [\![\langle x, \{\theta_1, \ldots, \theta_n\}\rangle]\!]^{\mathcal{I}}_{\alpha} \\ &:= [\![x]\!]^{\mathcal{I}}_{\alpha} \cap \bigcup_{i=1}^{n} [\![\theta_i]\!]^{\mathcal{I}}_{\alpha} \\ &= \{\alpha(x)\} \cap \bigcup_{i=1}^{n} [\![\theta_i]\!]^{\mathcal{I}}_{\alpha} \end{aligned}$$

Finally, we are ready to define the denotation of unification.

Definition 9 Denotation of feature structure unification
The denotation of the unification ($\sqcap$) of two typed feature structures $\theta_1, \theta_2 \in \Theta$ under interpretation $\mathcal{I}$ and variable assignment α is

$$[\![\theta_1 \sqcap \theta_2]\!]^{\mathcal{I}}_{\alpha} := [\![\theta_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![\theta_2]\!]^{\mathcal{I}}_{\alpha}$$

That is, the denotation of the result of feature structure unification is defined as the intersection of the denotation of the input feature structures.

3.3 Type Definitions

In $\mathcal{TDL}$, type definitions are the basic means of building up a type hierarchy. A set of type definitions is often referred to as *grammar*. When a grammar is being processed, $\mathcal{TDL}$ starts with an empty hierarchy which contains only the most general type $\top$ (plus possibly some types predefined by the system such as **list**, **cons**, **null** for list representation, **sort**, **avm**, etc.).
A type definition adds a new type to the hierarchy, establishes ISA-links between the new type and its supertypes, and may introduce new features and refine values of inherited attributes. The right hand side of a type definition $\Delta(\tau)$ consists of a type expression that specifies the supertype(s) plus a possibly empty set of feature constraints (conjunctive definition). Alternatively, a type definition can be disjunctive, i.e., specify its subtypes. Thus, the rhs of the definition can be expressed through a *typed feature structure*. The defined type can be seen as an abbreviation for the rhs feature structure skeleton (cf. templates).
If $\Delta(\tau) = \theta$ is a conjunctive feature structure, then τ becomes the direct subtype of the head (the type) of θ, i.e., it inherits from the head type and may refine the feature constraints and introduce new features. If θ bears no features and τ is a conjunction of type symbols, say $\rho_1 \wedge \cdots \wedge \rho_n, n > 1$, then τ is marked as the least upper bound of $\rho_1, \ldots, \rho_n$. If θ is disjunctive, then τ is introduced as the least upper bound of the elements of the disjunction. All variables occurring in θ must be local, i.e., they must not be shared with variables in feature structures outside θ.

Although typed feature structures are used by $\mathcal{TDL}$ to represent type definitions, $\mathcal{TDL}$'s input syntax is more liberal. It admits any complex expression consisting of type names, atoms, feature-value lists and disjunctions, combined by the operators `&` (conjunction), `|` (disjunction), `^` (exclusive-or) and `~` (negation); cf. the BNF in Appendix A.
When such a type definition is processed, the following steps are performed (steps that are not always executed are marked by *):

1. parse input expression ($\mathcal{TDL}$ syntax)

2. build conjunctive or disjunctive normal form
3. translate into feature structure representation
4. store the feature structure skeleton
5. define intermediate types*
6. establish ISA-links in the hierarchy
7. mark new type as *glb/lub*
8. define features to be *appropriate* for a type* (cf. Chapter 9)

3.3.1 Examples

We now describe informally how type definitions are processed in TDL. Some simple examples should make clear what occurs when a type is being defined. The general syntax for a type definition is

newtype `:=` complex description (skeleton).[2]

Consider the following type definitions:

```
a := *avm* & [a ].
b := *avm* & [b ].
```

where **avm** (the most general avm type, predefined by TDL) is the common supertype of a and b. Both definitions are already in normal form, and their feature structure representation is

$$\Delta(a) = \langle *avm*, [\textsc{a} \doteq \top] \rangle, \text{ and}$$
$$\Delta(b) = \langle *avm*, [\textsc{b} \doteq \top] \rangle$$

Types a and b introduce the features A and B as appropriate because **avm** has no features. Type definition

```
c := a & b.
```

[2] The assignment syntax (`:=`) has been chosen to indicate that the left hand side is an *abbreviation* for the right hand side. This is contradictory from the semantic point of view (in general, if appropriateness is not stipulated) where a type definition has to be read as a *consequence* from left to right,

newtype $\Rightarrow$ complex description (skeleton).

i.e., if a feature structure is of type *newtype*, then it has to fulfill *at least* the constraints given by the right hand side of *newtype*'s definition.

introduces a new type c as subtype of a and c. c is marked as least upper bound and its feature structure representation is

$$\Delta(c) = \langle a \wedge b, [\,] \rangle$$

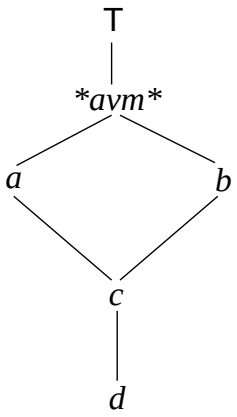
c does not introduce new features.

```
d := c & [a 1, b 2].
```

specifies a subtype of c which refines the values of the inherited attributes A and B. The feature structure representation is

$$\Delta(d) = \langle c, [\textsc{a} \doteq \langle 1, [\,] \rangle, \textsc{b} \doteq \langle 2, [\,] \rangle] \rangle, \text{or abbreviated, } \Delta(d) = \langle c, [\textsc{a} \doteq 1, \textsc{b} \doteq 2] \rangle$$

These four definitions build up the hierarchy



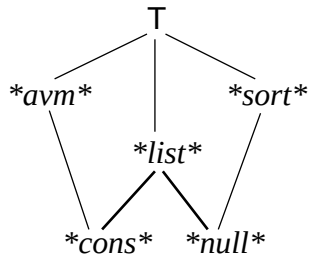
Alternatively, the grammar writer could have omitted the definition for c and instead have written

```
d := a & b & [a 1, b 2].
```

In this case, there are two possibilities. The first one is that a type with definition `a & b`, say c, already exists. Then, the definition of d will automatically be rewritten to `c & [a 1, b 2]`. If no such *glb* type exists, TDL introduces a so-called *intermediate type* with the same definition as c and with name $|a\&b|$. Then, d's definition is rewritten to `|a&b| & [a 1, b 2]`.
By default, intermediate types are only introduced for the supertype of a type definition. This is necessary to determine the correct location of the new type in the hierarchy (classification; cf. [Krieger 95a]). Inside a complex feature structure definition, i.e., at feature paths of length greater than zero, intermediate types are only explicitly represented if a global switch (that can be changed by the grammar writer) enforces this.
An example of a disjunctive type definition is TDL's **list** type, which is defined as

```
*list* := *null* | *cons*.
```

list is marked to be a least upper bound and $\Delta(\text{*list*}) = \langle\{\langle \text{*null*}, []\rangle, \langle \text{*cons*}, []\rangle\}\rangle$.
The resulting hierarchy is



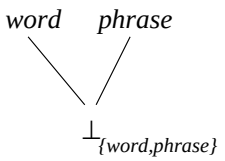
Declarations: There are other syntactic constructs where no features are involved, e.g.,

```
*NULL* :< *SORT*.
```

declares **null** to be a subsort of the most general sort **sort** (the same can also be done for avm types).

```
NIL = word & phrase.
```

declares *word* and *phrase* to be incompatible types (including their respective subtypes) by defining a special subtype of both which denotes inconsistency ('bottom type').



3.4 Typed Unification

The most important operation on typed feature structures is unification. Unification monotonically combines the information encoded in two feature structures. Either the most general structure that satisfies both arguments is returned or inconsistency is detected.
As defined in Section 3.2, the set-theoretical denotation of unification corresponds to set intersection of the denotation of the arguments:

$$[\![\theta_1 \sqcap \theta_2]\!]^{\mathcal{I}}_{\alpha} := [\![\theta_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![\theta_2]\!]^{\mathcal{I}}_{\alpha}$$

If we assume that arguments θ_1 and θ_2 are conjunctive feature structures (the extension to disjunctive arguments is straightforward), we can insert the definition of their semantics and exploit commutativity of conjunction and set intersection to rewrite this definition as follows:

$$
\begin{aligned}
[\![\theta_1 \sqcap \theta_2]\!]^{\mathcal{I}}_{\alpha} &= [\![\theta_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![\theta_2]\!]^{\mathcal{I}}_{\alpha} \\
&= [\![\langle \sigma, [f_{1_1} \doteq \theta_{1_1}, \ldots, f_{1_m} \doteq \theta_{1_m}]\rangle]\!]^{\mathcal{I}}_{\alpha} \cap [\![\langle \tau, [f_{2_1} \doteq \theta_{2_1}, \ldots, f_{2_n} \doteq \theta_{2_n}]\rangle]\!]^{\mathcal{I}}_{\alpha} \\
&= [\![\sigma]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{m} [\![f_{1_i} \doteq \theta_{1_i}]\!]^{\mathcal{I}}_{\alpha} \cap [\![\tau]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{n} [\![f_{2_i} \doteq \theta_{2_i}]\!]^{\mathcal{I}}_{\alpha} \\
&= [\![\sigma]\!]^{\mathcal{I}}_{\alpha} \cap [\![\tau]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{m} [\![f_{1_i} \doteq \theta_{1_i}]\!]^{\mathcal{I}}_{\alpha} \cap \bigcap_{i=1}^{n} [\![f_{2_i} \doteq \theta_{2_i}]\!]^{\mathcal{I}}_{\alpha} \\
&= [\![\sigma \wedge \tau]\!]^{\mathcal{I}}_{\alpha} \cap [\![[f_{1_1} \doteq \theta_{1_1}, \ldots, f_{1_m} \doteq \theta_{1_m}] \sqcap_f [f_{2_1} \doteq \theta_{2_1}, \ldots, f_{2_m} \doteq \theta_{2_m}]]\!]^{\mathcal{I}}_{\alpha}
\end{aligned}
$$

where $\sqcap_f$ is (untyped) feature unification. From the last line of these equations, we derive that operationally, typed unification has to

- determine the greatest lower bound of the types of the arguments
- unify the feature constraints (untyped feature structure unification)

if neither fails, the whole unification is consistent, and a single feature structure that contains the merged result is returned. Otherwise, $\bot$ is returned to indicate failure.
In the $\mathcal{TDL}$ system, the first part is accomplished by *type simplification* of the complex type expression $\sigma \wedge \tau$ ($\sigma, \tau \in (\mathcal{T} \cup \mathcal{A})$). The second part (untyped feature structure unification) is handled by the the constraint solver $\mathcal{UDiNe}$, which we treat as black box in this thesis.
The architecture of typed unification is depicted in Figure 3.2. Two feature structures with types σ and τ, are to be unified. Before feature unification takes place, type simplification computes the new type entry of the result, or fails. Finally, $\mathcal{UDiNe}$ unifies the feature structures and returns the result with the new type entry (or fails).

3.5 Type Simplification

Type simplification translates a $\mathcal{TDL}$ type expression into a normal form and determines whether it is consistent or not. Type simplification is invoked

- at type definition time, to normalize the type definitions and exploit information that can be inferred from previously defined types
- at unification time, to determine the type entry of the unified feature structure, or derive type inconsistencies
- as a separate function, invoked by external modules such as an NL parser or generator, e.g., to check type subsumption.

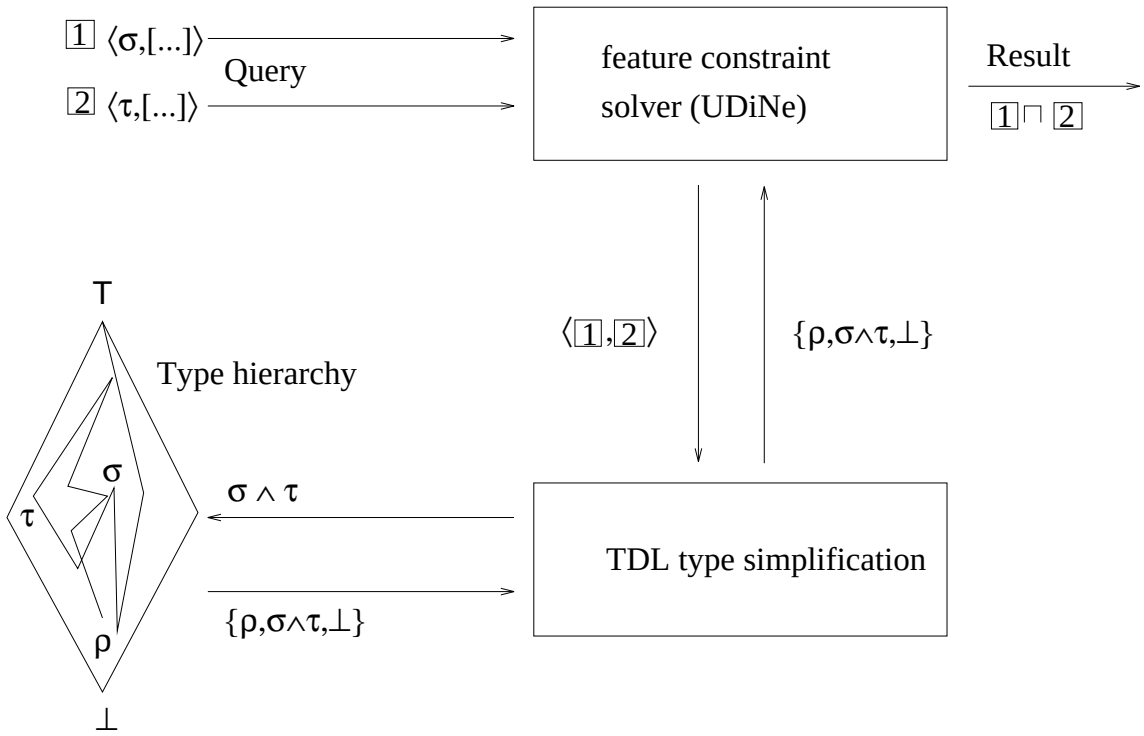


Figure 3.2: *Architecture of Typed Unification.*

TDL's function

$$\textit{simplify-type} : (\mathcal{T} \cup \mathcal{A} \cup \{\bot\})^* \mapsto (\mathcal{T} \cup \mathcal{A} \cup \{\bot\})^*$$

simplifies type expressions, where $(\mathcal{T} \cup \mathcal{A} \cup \{\bot\})^*$ is the set of complex type expressions over the connectives $\wedge$, $\vee$, and $\neg$. The function performs term rewriting on type expressions. A set of about 30 rewrite rules is applied iteratively to a type expression until a normal form (conjunctive or disjunctive, depending on a global switch) is reached or inconsistency is detected. Convergence is guaranteed by a total lexicographic order that is imposed on complex type expressions.

Efficient term rewriting is achieved by memoization of the type simplification function and a variant of bit-vector encoding of the type hierarchy [Aït-Kaci et al. 89] (e.g., for fast *lub*, *glb*, and subsumption computation).

The full set of the simplification rules and the basic algorithm can be found in [Krieger & Schäfer 94c]. We only show examples here to illustrate the different kinds of rules.

3.5.1 Purely Syntactic Schemata

This group comprises standard rules from Boolean algebra such as

$$\frac{\neg\,(\sigma \wedge \tau)}{\neg\,\sigma \vee \neg\,\tau} \quad \text{and} \quad \frac{\neg\,(\sigma \vee \tau)}{\neg\,\sigma \wedge \neg\,\tau} \quad \text{(DeMorgan's law)}$$

$$\frac{\sigma \wedge (\tau \vee \rho)}{(\sigma \wedge \tau) \vee (\sigma \wedge \rho)} \quad \text{and} \quad \frac{\sigma \vee (\tau \wedge \rho)}{(\sigma \vee \tau) \wedge (\sigma \vee \rho)} \quad \text{(Distributive law)}$$

$$\frac{\tau \wedge \tau}{\tau} \quad \text{and} \quad \frac{\tau \vee \tau}{\tau} \quad \text{(Idempotence)}$$

$$\frac{\tau \wedge \neg\,\tau}{\bot} \quad \text{and} \quad \frac{\tau \vee \neg\,\tau}{\top} \quad \text{(Inverse Element)}$$

$$\frac{\tau \wedge \top}{\tau} \quad \text{and} \quad \frac{\tau \vee \bot}{\tau} \quad \text{(Neutral Element)}$$

$$\frac{\neg\,\top}{\bot} \quad \text{and} \quad \frac{\neg\,\bot}{\top} \quad \text{(Negation)}$$

with $\rho, \sigma, \tau \in (\mathcal{T} \cup \mathcal{A} \cup \{\bot\})^*$, and others like absorption, double negation, commutativity, etc.

3.5.2 'Semantic' Schemata for Homogeneous Type Expressions

'Semantic' schemata are schemata that depend on the type hierarchy, i.e., exploit knowledge about subtype relations, least and upper bound types, etc. Homogeneous type expressions contain types from the same partition within the type hierarchy (e.g., sorts, built-in sorts, avm types, atoms) plus $\{\top\}$. Examples (again, the list of rules is not exhaustive):

$$\frac{\sigma \wedge \tau}{\sigma} \quad \text{and} \quad \frac{\sigma \vee \tau}{\tau} \quad \text{if } \sigma \preceq \tau \text{ and } \sigma, \tau \in \mathcal{T}$$

$$\frac{\sigma \wedge \neg\,\tau}{\bot} \quad \text{and} \quad \frac{\neg\,\sigma \vee \tau}{\top} \quad \text{if } \sigma \preceq \tau \text{ and } \sigma, \tau \in \mathcal{T}$$

$$\frac{\neg\,\sigma \wedge \neg\,\tau}{\neg\,\tau} \quad \text{and} \quad \frac{\neg\,\sigma \vee \neg\,\tau}{\neg\,\sigma} \quad \text{if } \sigma \preceq \tau \text{ and } \sigma, \tau \in \mathcal{T}$$

$$\frac{\sigma \wedge \neg\,\tau}{\sigma} \quad \text{if } glb(s,t) = \bot$$

The following rule explicitly states the 'closed world' for sorts. The closed world for complex avm types (which can be selected optionally in $\mathcal{TDL}$) depends on a global variable that enables or disables a similar rule.

$$\frac{\sigma \wedge \tau}{\bot} \quad \text{if } \sigma, \tau \in \mathcal{T}_s \text{ and } glb(\sigma, \tau) \notin \mathcal{T}$$

3.5.3 'Semantic' Schemata for Heterogeneous Type Expressions

Heterogeneous type expressions contain types of different partitions in the type hierarchy (e.g., sorts, built-in sorts, avm types, and atoms are mixed).
The set of rules in this group comprises (among others):

$$\frac{\sigma \wedge \tau}{\bot}$$ if $\sigma \in \mathcal{T}_s \cup \mathcal{T}_b$ and $\tau \in \mathcal{T}_a$ (Incompatibility of sorts and avm types)

$$\frac{\sigma \wedge \tau}{\bot}$$ if $\sigma \in \mathcal{T}_s$ and $\tau \in \mathcal{T}_b$ (Incompatibility of sorts and built-ins)

$$\frac{a \wedge b}{\bot}$$ and $$\frac{a \wedge \neg b}{a}$$ if $a, b \in \mathcal{A}$ and $a \neq b$ (Uniqueness of atoms)

$$\frac{a \wedge \sigma}{a}$$ and $$\frac{a \vee \sigma}{\sigma}$$ if $a \in \mathcal{A}$, $\sigma \in \mathcal{T}_b$ and the data type of $a \preceq \sigma$

$$\frac{a \wedge \sigma}{\bot}$$ if $a \in \mathcal{A}$, $\sigma \in \mathcal{T}$ and the data type of a is not σ and not a subtype of σ.

3.6 Architecture of the $\mathcal{TDL}$ System

As we have shown in the preceeding sections, $\mathcal{TDL}$ is both a type description language and the runtime support for this language, including a complex software system that administrates type definitions as input, generates feature structures suitable for the constraint solver $\mathcal{UDiNe}$, and contains the type hierarchy as well as type simplifier, and, last but not least, the type expansion module. Figure 3.3 shows the overall architecture of the $\mathcal{TDL}$ system.

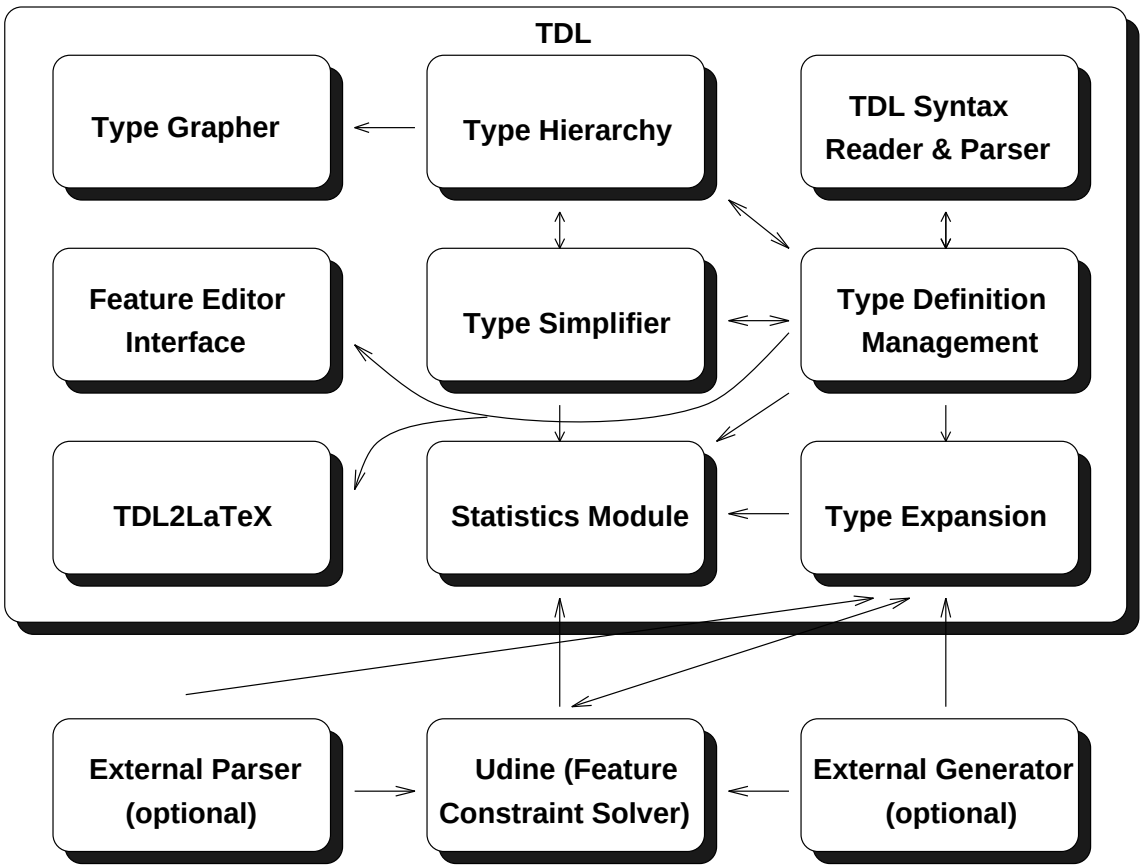


Figure 3.3: *Architecture of the TDL system.*

Chapter 4

Type Expansion

We now turn to the main topic of this thesis: type expansion. First, we give a definition of the notion and motivate it. Then, we modify the definition of typed feature structures in such a way that partially expanded feature structures can be represented. At the end of this chapter, we introduce the basic expansion algorithm which we will refine in the subsequent chapters.

4.1 Introduction and Motivation

A feature structure type is defined through local feature structures plus constraints that are inherited from its supertype. *Type expansion* is an operation on typed feature structures that combines the local information of a feature structure with the information given by type definitions through unification. Because the definitions of types themselves consist of typed feature structures, expansion is a recursive process that walks up the type hierarchy until the top type is reached.
Type expansion has two main functions:

- *structure building*: make constraints imposed by type definitions explicit locally
- *consistency checking*: test compatibility of local and inherited constraints

The major goal of this thesis is to devise an *efficient* algorithm for type expansion.
Further issues are *economy* and *expressivity*: *partial evaluation* can be employed in connection with type expansion to reduce the size of feature structures and hence save memory and copying time. Admission of *recursive type* definitions makes typed feature structures as powerful as Turing machines. This expressivity can only be exploited through type expansion.

There are other names for type expansion in the literature, e.g., *type checking* [Emele & Zajac 90], *sort unfolding* [Aït-Kaci 93] and *total well-typedness check* [Carpenter 92] (for well-typed feature structures). Some refer to it as type inference, but this is incorrect (although closely

```
LIEFER := trans-verb-lex &
          [ CAT @TRANS-VERB ( $PRED = 'LIEFER,
                              $SORT = accomplishment,
                              $STEM = < 'LIEFER > )  ].
```

Figure 4.1: *A TDL definition for the lexicon entry of the German transitive verb* liefern *from DISCO's HPSG-based grammar for German.*

related). Type inference starts with with an untyped (or partially typed) feature structure and tries to infer the correct type of the structure (or an approximation) according to the type definitions. Type expansion works the other way around: it starts with a typed skeleton, and inserts any features inherited through type definitions.
All feature structure formalism implementations with types include a variation of type expansion. In most cases, an implicit expansion mechanism is used. Actually, one can classify feature formalisms according to how expansion is executed.

Thesis: Expand as soon as possible

Systems like LOGIN, ALE, or *TDL ExtraLight* expand typed feature structures at definition time. The advantages are that

- no type expansion (i.e., additional unifications) has to be done at run time
- no facilities for partially expanded structure are needed

But there are several crucial disadvantages of that strategy:

- wasted memory: all lexicon entries must be held in memory. This is unacceptable for large lexica in real NL applications, cf. Figure 4.1 and Figure 4.2
- inefficient unification: the larger typed feature structures are, the more expensive is unification (in fact, nearly proportional to the number of unified nodes). It is known that often unification of partial descriptions suffices to rule out inappropriate readings
- restricted expressivity: if type definitions are expanded at definition time, recursive types cannot be admitted. Otherwise, expansion would either loop or be incomplete.

Antithesis: Expand as late as possible

The most radical version of this strategy is *lazy attribute inheritance* [Aït-Kaci 93]. The basic idea is that (1) expansion is an integrated part of unification and (2) not the whole skeleton of a type to be expanded is inherited, but only those attributes and its values of the definition that appear locally in the feature structure that has been returned as the result of feature unification. The advantage of this algorithm is that

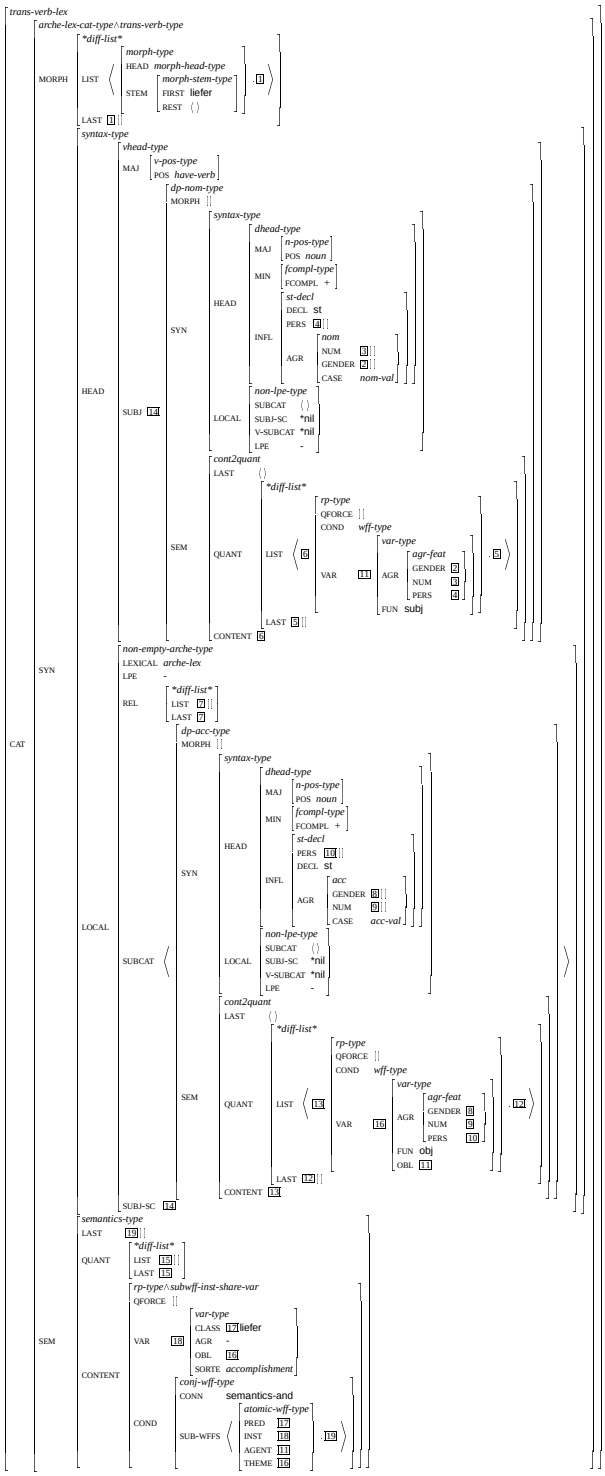


Figure 4.2: *The (almost) fully expanded lexicon entry of the German transitive verb* liefern. *The* NONLOCAL *features are omitted.*

- it is optimal with respect to the memory requirements if one is only interested in satisfiability
- lazy expansion of recursive types is treated simply and elegantly.

The disadvantages are that

- in general, full expansion is not performed although it may be necessary (e.g., to access the complete semantic information), i.e., only one half of the duties of type expansion are fulfilled, namely consistency checking
- the method slows down expansion at run time because *all* expansions are done at run time and because it contradicts memoization (cf. Chapter 5)
- in an implementation that represents coreferences by structure sharing (as $\mathcal{TDL}$ does), it is difficult (i.e., time-consuming) to preserve coreferences that occur in super types. Cf. Section 4.3.2 for a detailed discussion.

As it turns out, this is a classical conflict of time vs. space optimization. Neither of the two extremes presented above seems to be satisfactory for practical NL applications.

Synthesis: Expand if needed

The solution we present in this thesis is to treat type expansion as an explicit mechanism that can be employed at any time in linguistic processing: at type definition time as well as integrated in typed unification, but also in between, e.g., controlled explicitly by a NL parser or generator.
The expansion algorithm operates either destructively or non-destructively. It takes a typed feature structure and unifies each feature structure node that references to a type name with its definition. Because type definitions themselves are represented by typed feature structures, type expansion is a recursive process that walks through feature structures and up the type hierarchy until all types in the structure are expanded or a global inconsistency is detected. The expansion algorithm can be parameterized globally and locally for delay and preference information as well as search strategy. Memoization is used to minimize the number of unifications, and recursive types are treated properly.

4.2 Augmented Typed Feature Structures

Before we introduce the algorithm, the data structures that represent typed feature structures must be considered. Basically, we use the notation from Definition 3 on page 17. But some modifications of the type entries of conjunctive feature structures are necessary to be able to represent partially expanded (*postponed*) feature structures, e.g., for recursive and postponed types.

Definition 10 Type entries of conjunctive typed feature structures (final version)
The type expression τ in Definition 3 is replaced by a triple $\langle \tau, \Delta\text{-}set, expanded \rangle$, the *type entry* or *type info* where

- τ is the complex type expression as before
- $\Delta\text{-}set \in 2^{\mathcal{T} \cup \mathcal{A}}$ is a set that indicates which definitions of the components of τ (including their supertypes) have already been unified with the current node, i.e., $\Delta\text{-}set$ indicates whether a node is expanded *locally*. If all types (including their supertypes) of the node are expanded locally, then the node is said to be $\Delta\text{-}expanded$($\Delta\text{-}expanded \in$ {true,false}, see below)
- $expanded \in$ {true,false} is a flag that indicates whether *all* sub-feature structures including the current node are $\Delta\text{-}expanded$ (true) or not (false), i.e., *expanded* indicates whether a node is expanded *globally*

The *expanded* flag helps to drastically reduce the search space for type expansion. The expansion algorithm never visits those substructures of a feature structure node with *expanded* value true. If a whole feature structure is fully expanded, a single look at the root node's *expanded* flag suffices to decide whether further expansion is necessary or not.
The motivation of the $\Delta\text{-}set$ is to make it possible to postpone type expansion locally and to avoid unnecessary unification (expansion) of types that have already been expanded locally. Before a type definition is unified with a node in the expansion algorithm, the node's $\Delta\text{-}set$ is compared with the type to be expanded, say τ. If τ or one of its subtypes is already in the $\Delta\text{-}set$, then no unification takes place.
The only reason why $\Delta\text{-}set$ is a set (and not a flag) is that $\mathcal{TDL}$ admits complex conjunctive type expressions of the form $\tau_1 \wedge \cdots \wedge \tau_n$ in the type slot. Each τ_i can be selected separately through the $\Delta\text{-}set$. In a closed type world (which can be chosen in $\mathcal{TDL}$), $\Delta\text{-}set$ reduces to a single flag because there is always only one type per node.
$\Delta\text{-}set$ and *expanded* are only relevant to avm types that are defined as abbreviations for complex feature structures. Sorts, atoms and $\top$ are always $\Delta\text{-}expanded$ and *expanded*.
$\Delta\text{-}expanded$ can be computed easily from the $\Delta\text{-}set$ and the type expression τ of a type entry. Therefore, it is not stored explicitly in the type info.
The following functions access the values of the three components of the augmented type entries and the $\Delta\text{-}expanded$ flag.

Definition 11 *type-of*
type-of : $\Theta \mapsto (\mathcal{T} \cup \mathcal{A})^*$ is a function. *type-of*$(\theta), \theta \in \Theta$, returns

- the value of the first component of θ's type entry if θ is a conjunctive typed feature structure,

- $\bigvee_{i=1}^{n}$ *type-of*(θ_i) if θ is a disjunctive typed feature structure $\theta = \langle x, \{\theta_1, \ldots, \theta_n\}\rangle$, where $\bigvee$ is the type disjunction operator.

Definition 12 Δ-*set*
Δ-*set* : $\Theta \mapsto 2^{\mathcal{T}}$ is a function. Δ-*set*$(\theta), \theta \in \Theta$ returns a set of type symbols,

- the value of the second component of θ's type entry if θ is a conjunctive typed feature structure,
- the empty set if θ is a disjunction.

Definition 13 *expanded*
expanded : $\Theta \mapsto$ {true,false} is a function. *expanded*$(\theta), \theta \in \Theta$ returns

- the value of the third component of θ's type entry if θ is a conjunctive typed feature structure,
- $\bigwedge_{i=1}^{n}$ *expanded*(θ_i) if θ is a disjunctive typed feature structure $\theta = \langle x, \{\theta_1, \ldots, \theta_n\}\rangle$, where $\bigwedge$ is the Boolean *and* operator.

Definition 14 Δ-*expanded*
Δ-*expanded* : $\Theta \mapsto$ {true,false} is a function. Δ-*expanded*$(\theta), \theta \in \Theta$ returns

- if θ is a conjunctive typed feature structure:

$$\begin{cases} \text{true}, & \text{if } \textit{type-of}(\theta) \in \mathcal{A} \cup \mathcal{T}_s \cup \{\top\} \text{ or } \textit{type-of}(\theta) \in \Delta\text{-}\textit{set}(\theta) \text{ or} \\ & \textit{type-of}(\theta) = \tau_1 \wedge \cdots \wedge \tau_m, \text{ and } \forall \tau_i\text{: if } \tau_i \in \mathcal{T} \text{ then } \tau_i \in \Delta\text{-}\textit{set}(\theta) \\ \text{false}, & \text{otherwise,} \end{cases}$$

- $\bigwedge_{i=1}^{n}$ Δ-*expanded*(θ_i) if θ is a disjunctive typed feature structure $\theta = \langle x, \{\theta_1, \ldots, \theta_n\}\rangle$, where $\bigwedge$ is the Boolean *and* operator.

In the latter definition, we have assumed that type entries are in normal form. This is always guaranteed by the $\mathcal{TDL}$ type simplification mechanism.

4.2.1 Example

To illustrate the modified feature structure representation, we have a look at the *npsg23* type from the example above.
Suppose the type definition of *npsg23* is

```
npsg23  := np-type & [ AGREEMENT #x & sg23,
                       SUBJECT   #x ].
```

Then, the skeleton feature structure of *npsg23* is

$$\langle\langle \textit{np-type}, \{\}, \text{false}\rangle, [\textsc{agreement} \doteq \langle x, \langle \textit{sg23}, \{\}, \text{false}\rangle, [\,]\rangle, \\ \textsc{subject} \doteq x]\rangle$$

Again, we omit variables that occur only once and abbreviate atomic values to `atom` instead of $\langle \texttt{atom}, [\,]\rangle$.
Suppose now that *np-type* and *sg23* are defined by

```
np-type := *avm* & [ CAT 'np ].
sg23    := *avm* & [ NUMBER 'singular,
                     PERSON 'second | 'third ].
```

Then the fully expanded definition of *npsg23* is

$$\langle\langle \textit{npsg23}, \{\textit{npsg23}\}, \text{true}\rangle, [\textsc{agreement} \doteq \langle x, \langle \textit{sg23}, \{\textit{sg23}\}, \text{true}\rangle, \\ [\textsc{number} \doteq \texttt{singular}, \\ \textsc{person} \doteq \langle\{\texttt{second}, \texttt{third}\}\rangle]\rangle, \\ \textsc{cat} \doteq \texttt{np}, \\ \textsc{subject} \doteq x]\rangle$$

The type entry of the root node of a type definition skeleton contains the direct supertypes of the type if their definition is not Δ-*expanded* in the root node (*np-type* in the preceeding example). Otherwise, i.e., if all direct supertypes at the root node are Δ-*expanded*, the root type of the skeleton is the type the skeleton is associated with (*npsg23* in the example).
Since the feature structure representation is hard to read, we augment the AVM notation analogously.

4.2.2 Augmented AVM Notation

The Δ-*set* and *expanded* flags are adjoined to the AVM notation (Section 2.1) as 'virtual features', i.e., they are printed below the type of a conjunctive feature structure in the same way feature-value pairs are formatted (e.g., *:expanded false*).[1] If a node is *expanded*, then its *expanded* flag usually is omitted. If a node is Δ-*expanded*, then its Δ-*set* entry usually is omitted. Examples:

[1] Of course, they are not implemented as features (which are subject to unification), but are associated with the type entry of a node.

npsg23's skeleton:

$$\begin{bmatrix} \textit{np-type} \\ \textit{:expanded false} \\ \textit{:delta } \{\} \\ \text{AGREEMENT } \boxed{1} \begin{bmatrix} \textit{sg23} \\ \textit{:expanded false} \\ \textit{:delta } \{\} \end{bmatrix} \\ \text{CAT} \quad \texttt{np} \\ \text{SUBJECT} \quad \boxed{1} \end{bmatrix}$$

The expanded skeleton (prototype) of *npsg23*:

$$\begin{bmatrix} \textit{npsg23} \\ \textit{:expanded true} \\ \textit{:delta } \{ \textit{ npsg23 } \} \\ \text{AGREEMENT } \boxed{1} \begin{bmatrix} \textit{sg23} \\ \textit{:expanded true} \\ \textit{:delta } \{ \textit{ sg23 } \} \\ \text{NUMBER } \texttt{singular} \\ \text{PERSON } \left\{ \begin{matrix} \texttt{second} \\ \texttt{third} \end{matrix} \right\} \end{bmatrix} \\ \text{CAT} \quad \texttt{np} \\ \text{SUBJECT} \quad \boxed{1} \end{bmatrix}$$

4.3 Typed Unification

4.3.1 Integration into Feature Unification

In Section 3.4, typed unification has informally been described as type simplification plus untyped feature unification. Now that type entries are fully specified, we can go into the details of the type part of typed unification.
Function *unify-types* takes two typed feature structures and computes a new type entry for the resulting unified feature structure:

$$\textit{unify-types} : \Theta \times \Theta \mapsto (\mathcal{T} \cup \mathcal{A})^* \times 2^{\mathcal{T}} \times \{\text{true,false}\}$$

unify-types is called before feature unification. The main reason is that type conjunction can result in $\perp$ (failure), which will make feature unification superfluous. Because type conjunction (simplification) is cheaper than complex feature unification in general, this order of computation is reasonable.

In addition to type consistency checking, *unify-types* computes the Δ-*set* and *expanded* flag for the new type entry.
Both arguments of *unify-types* are conjunctive typed feature structures (including the trivial case of atoms and sorts). Disjunctive feature structures are simply unified component-wise at a higher level in the unification algorithm.
The return value $\langle \bot, \{\}, \text{true} \rangle$ triggers a unification failure.

function *unify-types* (θ_1, θ_2)**:**

$\tau :=$ *simplify-type*(*type-of*$(\theta_1) \wedge$ *type-of*(θ_2))**;**
if $\tau = \bot$ **or** $\tau \in \mathcal{A} \cup \mathcal{T}_s \cup \mathcal{T}_b$
then **return** $\langle \tau, \{\}, \text{true} \rangle$
else Δ-*set* $:=$ *combine-delta*(Δ-*set*$(\theta_1), \Delta$-*set*(θ_2))**;**
Δ-*expanded* $:= \Delta$-*set* $\supseteq$ *set-of*(τ)**;**
expanded $:= \Delta$-*expanded* **and** $\forall i :$ *expanded*(θ_{1_i}) **and** $\forall j :$ *expanded*(θ_{2_j})**;**
return $\langle \tau, \Delta$-*set*, *expanded*$\rangle$**.**

simplify-type returns a type expression in simplified normal form (cf. Section 3.5). Because disjunctions are translated to the feature structure level and negation is pushed to atoms at type definition time, τ is either $\bot$, a single type symbol or a complex expression of the form $\sigma_1 \wedge \cdots \wedge \sigma_n$ where $\sigma_i \in \mathcal{T} \cup \mathcal{A}$ (or negated). Therefore, and because of commutativity of type conjunction, we can treat type expressions as sets. Function *set-of*(τ) 'translates' a type expression in simplified normal form into a set.

$$\textit{set-of}(\tau) := \begin{cases} \{\tau\}, & \text{if } \tau \in \mathcal{T}_a \\ \{\tau_1, \ldots, \tau_n\}, & \text{if } \tau = \tau \wedge \cdots \wedge \tau_n, \tau_i \in \mathcal{T}_a, 1 \leq i \leq n \\ \{\}, & \text{otherwise (i.e., } \tau \in \mathcal{T}_s \cup \mathcal{T}_b \cup \mathcal{A} \cup \{\top\}) \end{cases}$$

An expression like $\tau_1 \wedge \tau_2$ with $\tau_1 \in \mathcal{T}_a$ and $\tau_2 \in \mathcal{T}_s$ cannot occur because type simplification rules it out ($\bot$). *combine-delta*(Δ-*set*$(\theta_1), \Delta$-*set*(θ_2)) can be characterized informally as set union over Δ-*set*$(\theta_i), i = 1, 2$ modulo type simplification. The function is implemented as follows. Suppose Δ-*set*$(\theta_1) = \{\sigma_1, \ldots, \sigma_m\}$ and Δ-*set*$(\theta_2) = \{\tau_1, \ldots, \tau_n\}$. Instead of set union (which does not take into account subtype relations and glb), we treat the input as a type conjunction $\sigma_1 \wedge \cdots \wedge \sigma_m \wedge \tau_1 \wedge \cdots \wedge \tau_n$, apply the function *simplify-type* to it, and retranslate the result into set notation through *set-of*.[2]
The result is the new Δ-*set*. The new *expanded* flag is obtained by a boolean *and* operation on the new Δ-*expanded* flag (if all types in the new type expression also occur in the new

[2]In the implementation, no translation is necessary because both sets and complex type expressions are represented by Common Lisp lists. Moreover, structure sharing between the type expression and the Δ-*set* (if the node is Δ-*expanded*) allows for a succinct representation:

```
#S(TYPE-INFO :TYPE (:AND . #1=(NP-TYPE CAT-TYPE GEN-TYPE)) :DELTA #1# :EXPANDED NIL)
```

E.g., the previous type info encodes a $\langle$*type*, Δ-*set*, *expanded*$\rangle$ triple where all types are Δ-*expanded*.

Δ-*set*, then the node is Δ-*expanded*) and the *expanded* flags of the values of attribute lists of both argument feature structures (θ_{ij}).
Finally, the new triple $\langle \tau, \Delta\text{-}set, expanded \rangle$ is returned and will become the type entry of the unified feature structure unless feature unification fails.
Because the unification algorithm of *UDiNe* is depth-first, the *expanded* and Δ-*set* values in a feature structure are established correctly after unification.

4.3.2 Lazy Attribute Inheritance

Although we did not implement it because of the disadvantages discussed on page 31, we briefly discuss the integration of lazy attribute inheritance into typed unification. Lazy attribute inheritance has been suggested by [Aït-Kaci 93]. In contrast to the strategy of a separate function for type expansion that we pursue in this thesis, lazy attribute inheritance integrates type expansion into unification.
Only those values whose attributes appear in the result of unification are copied from the corresponding type and are unified with the result. Therefore, the feature unification algorithm must call a second function different from *unify-types after* successful feature unification. Although the implementation seems to be straightforward, a problem arises because of the structure-sharing implementation of coreferences in *TDL*/*UDiNe*.
Consider the following example. Let $\langle a, [\textsc{arg1} \doteq \top] \rangle$ be a partially expanded structure, the result of a unification. If a is defined by $\Delta(a) = \langle \top, [\textsc{arg1} \doteq \langle x, \top, [\,] \rangle, \textsc{arg2} \doteq x] \rangle$, where x is a coreference, then the result of lazy expansion after unification would be $\langle a, [\textsc{arg1} \doteq \top] \rangle$ because feature ARG1 is explicit but ARG2 is not.
If ARG2 is made explicit later, the resulting structure is $\langle a, [\textsc{arg1} \doteq \langle \top, [\,] \rangle, \textsc{arg2} \doteq \langle \top, [\,] \rangle] \rangle$ which is different from the definition of a and hence incorrect.
Therefore, the algorithm must know about the coreferences in the super types which can only be achieved by an additional traversal of their definitions and this may be expensive.
Another variation of lazy attribute inheritance would be to expand only those features that are explicit in both unification arguments. The coreference problem yet remains.

4.4 The Basic Algorithm

Finally, we turn to the description of the basic expansion algorithm. It will be enriched by various control extensions in the subsequent chapters.

4.4.1 Implementation

The algorithms will be depicted in pseudo-code similiar to Pascal. The implementation is done in Common Lisp [Steele 90], as is the rest of the *TDL* and *UDiNe* system. The code is

portable; this has been tested successfully as a part of the DISCO NL system with Allegro Common Lisp 4.2, CLISP (on Linux), Lucid Common Lisp, and Macintosh Common Lisp. It is worth noting that the pseudo-code fragments we present are drastically simplified; we omit many $\mathcal{TDL}$-specific details such as domains, feature structure copying, prototype access as well as $\mathcal{UDiNe}$'s functional constraints and control objects.

4.4.2 Interface Functions

Type expansion operates on typed feature structures. There are different functions (procedures) that access feature structures and type definitions, destructively and non-destructively. They can be called either by the user (within grammar files) or from other NL modules such as parser, generator, etc.

- function *expand-fs* non-destructively expands a typed feature structure and returns the new structure
- function *expand-node* destructively expands a typed feature structure and returns the modified structure
- function *expand-type* expands the definition of an avm type (if necessary) and returns the expanded feature structure. This function can be called automatically at type definition time (the global switch `*EXPAND-TYPE-P*` controls this)
- procedure *expand-instance* expands the feature structure definition of an instance. Instances are feature structures that are not associated with a type in the hierarchy (but can inherit from types), e.g., lexicon entries or rules that are managed by a NL parser or generator. *expand-instance* can be called automatically at instance definition time (the global switch `*EXPAND-TYPE-P*` controls this).

All functions call *expand-tfs*, the main procedure, which expands typed feature structures destructively (see below). Roughly speaking, type expansion traverses the feature structure graph and destructively unifies type definitions with the given feature structure.

4.4.3 Search Strategies

We now discuss several search strategies for type expansion algorithms. It is clear that which strategy is best (fastest) depends on the purpose for which type expansion is used, e.g., consistency checking or structure building. Moreover, it can depend on the 'style' grammars are written in, as we will argue later.

Fan-out

The first formally specified expansion algorithm for typed feature structures is the one of [Aït-Kaci 86] (for ψ-terms in KBL and LOGIN [Aït-Kaci & Nasr 86]). It does not rely on unification, but rather uses DAG rewriting to merge type definitions into a feature structure. Hence, the order of rewriting is crucial for coreferring and overlapping structures. Fan-out rewriting rewrites type symbols closer to the root node first.

If one uses structure sharing to represent coreferences, then a preprocessing traversal is necessary for identifying coreferring structures.

Fan-out order is inflexible and, moreover, obsolete for feature structure formalisms like $\mathcal{TDL}$ that have a unification operation to combine the local feature structure with the definition of types. Therefore, we did not implement this strategy.

Breadth-First

Breadth-first expansion starts from the root node of the feature structure and then expands all types at feature path depth 0, then at depth 1, 2, 3,... (by unification). This strategy obviously leads to a complete algorithm for recursive types also, but has the general disadvantage of run time overhead for going back and forth in the feature structure to reach the nodes at the desired path depth. This is one reason why we did not implement this strategy within $\mathcal{TDL}/\mathcal{UDiNe}$. Another reason (which made it almost impossible to implement true breadth-first expansion efficiently) was $\mathcal{UDiNe}$'s technique for representing coreferences by marks at the feature structures nodes. If a node had been visited more than once (e.g., to jump to a longer path at a substructure), it would have been incorrectly treated as a coreference.

Depth-First

Depth-first expansion simply walks down the feature graph first, and expands the types on the way back (by unification). This is a simple strategy that also corresponds to the depth-first unification algorithm of $\mathcal{UDiNe}$. Therefore, *expanded* flags are guaranteed to be set correctly by function *unify-types* (during unification) without an extra traversal of the feature structure. The disadvantage is that parts of the feature structure must be visited several times if they contain recursive types in order to ensure fair and complete expansion. But the *expanded* flags help to restrict the search space to just those branches that contain the unexpanded (recursive) types.

Because of the simplicity and correspondence to the depth-first unification algorithm, depth-first has been chosen as the default strategy for $\mathcal{TDL}$'s type expansion.

Types-First

The types-first algorithm is similar to depth-first, but expands types *before* visiting the substructures (hence the name 'types-first'). This strategy helps to reduce the number of redundant type expansions (especially for well-typed grammars), although redundancy does not significantly slow down the depth-first algorithm in practice (cf. Section 5.4). Disadvantages are that each recursive type in a feature structure can only be expanded once in the same substructure (for each walk through the whole structure) even if it occurs more than once. This has to be done to prevent (incorrect) infinite expansion. Actually, this makes the algorithm inelegant and slower for recursive types than depth-first.
Since typed unification (and hence application of the *unify-types* function) is strictly depth-first, an additional walk through the feature structure may be necessary to set the *expanded* flags correctly. Therefore, slight advantages for unification run time are eaten up by the additional (partial) visit of the structure.
Having considered the above argumentation, we decided to provide types-first as an optional strategy for $\mathcal{TDL}$'s type expansion.

Other Strategies

As we noted above, only depth-first and types-first have been chosen for $\mathcal{TDL}$ type expansion. Depth-first is the generic strategy that is also advantageous for recursive types. Types-first may be slightly faster if only a few recursive types occur and if many types in the definitions that are unified with the structure have been postponed, or if the grammar is strictly well-typed. In the latter case, a depth-first strategy with memoization leads to redundant expansions/unifications, but although we did not have the opportunity to compare a well-typed and a non-well-typed grammar, we do not expect significant differences in run time. Expansion of a non-strictly well-typed sample grammar does not show great advantages for either strategy (see Section 5.4).
$\mathcal{TDL}$'s type expansion algorithm can easily be extended to support other search strategies than the preceeding ones. One simply has to define an appropriate procedure that takes the same arguments as *depth-first-expand* and *types-first-expand* and specify its name as a parameter in the expansion control (Chapter 7). Independently from the search strategy, heuristic information about the order in which attribute values are visited can speed up expansion (Section 7.10).

Example

To illustrate the different strategies, we give a short, in no way linguistically motivated, example. We define the following avm types:

```
p := [ f, g, h ].
```

```
q := [ l ].
r := [ m ].
s := p & [ f q,
           g [ k r ],
           h q & [ l r ] ].
```

and expand the definition of s with the command

```
expand-type 's.
```

Now, we can compare the different expansion orders induced by the different strategies (we omit fan-out order because it is obsolete for $\mathcal{TDL}$ as we argued above). The resulting expanded feature structures for s are shown below (which are of type s, of course, but type p at the root node here indicates the name of the supertype definition that has been unified with). The number in parentheses at each type name indicates the position in the sequence of calls to *expand-type* that are necessary to expand s. Because no recursive type occurs within s, full expansion is done within one walk through the feature structure.

breadth-first:

$$\begin{bmatrix} p(1) \\ \text{F} \begin{bmatrix} q(2) \\ \text{L} [\,] \end{bmatrix} \\ \text{G} \begin{bmatrix} \text{K} \begin{bmatrix} r(4) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \\ \text{H} \begin{bmatrix} q(3) \\ \text{L} \begin{bmatrix} r(5) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

depth-first:

$$\begin{bmatrix} p(5) \\ \text{F} \begin{bmatrix} q(1) \\ \text{L} [\,] \end{bmatrix} \\ \text{G} \begin{bmatrix} \text{K} \begin{bmatrix} r(2) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \\ \text{H} \begin{bmatrix} q(4) \\ \text{L} \begin{bmatrix} r(3) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

types-first:

$$\begin{bmatrix} p(1) \\ \text{F} \begin{bmatrix} q(2) \\ \text{L} [\,] \end{bmatrix} \\ \text{G} \begin{bmatrix} \text{K} \begin{bmatrix} r(3) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \\ \text{H} \begin{bmatrix} q(4) \\ \text{L} \begin{bmatrix} r(5) \\ \text{M} [\,] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Empirical results that compare depth-first vs. types-first expansion are discussed in Section 5.4 in context with memoization.

4.4.4 Basic Functions and Procedures

Let us now turn to the skeleton of the expansion algorithm. *expand-tfs* is the main procedure that is called from the interface procedures like *expand-node*, *expand-type*, *expand-instance*, etc.

It takes the root node of the feature structure to be expanded as argument and applies the search strategy (*depth-first-expand*, *types-first-expand*, or user-defined; which one is applied can be chosen by control parameters) on it until the structure is either fully expanded or resolved (a predicate that can be defined by the user to decide whether structures containing recursive types are "complete"), or until no unification occurred in the last pass which ensures termination of expansion on feature structures that contain postponed types.

procedure *expand-tfs*(θ)**:**

while not (*expanded*(θ) **or** *resolved*(θ) **or** no unification occurred in last pass)
depth-first-expand(θ). /* or *types-first-expand*(θ) */

Procedures *depth-first-expand* and *types-first-expand* have been explained informally in the previous section. They only differ in the order in which the feature values and the local type info is visited. *depth-first-expand* first visits the substructures (if they exist) and then expand the type at the current node. *types-first-expand* first expands the type and then visits the substructures.
The visited check in the second line is necessary to ensure termination of expansion of coreferring and cyclic feature structures. The check can be done by comparing variables that we have omitted in the feature structure representation here. In the implementation with $\mathcal{UDiNe}$ where structure sharing is used to express coreferences, marks in the structures are checked instead.
If local unification fails, a global fail is triggered. We have omitted this in the code below for better readability. At the end of both procedures, the *expanded* flag must be updated. Here, m is the number of θ's features after unification, i.e., $m \geq n$.

procedure *depth-first-expand*(θ)**:**

if θ not already visited in this pass
then if $\theta = \langle\{\theta_1, \ldots, \theta_n\}\rangle$ /* θ is disjunctive */
then for i **from** 1 **to** n : *depth-first-expand*(θ_i);
else /* $\theta = \langle\tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n]\rangle$ (θ is conjunctive) */
for i **from** 1 **to** n : *depth-first-expand*(θ_i);
if not Δ-*expanded*(θ) **then** *unify-type-and-node*(*type-of*$(\theta), \theta)$;
expanded(θ) := Δ-*expanded*(θ) **and** $\bigwedge_{i=1}^{m}$ *expanded*(θ_i).

procedure *types-first-expand*(θ)**:**

if θ not already visited in this pass
then if $\theta = \langle\{\theta_1, \ldots, \theta_n\}\rangle$ /* θ is disjunctive */
then for i **from** 1 **to** n : *types-first-expand*(θ_i);
else /* $\theta = \langle\tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n]\rangle$ (θ is conjunctive) */
if not Δ-*expanded*(θ) **then** *unify-type-and-node*(*type-of*$(\theta), \theta)$;
for i **from** 1 **to** n : *types-first-expand*(θ_i);
expanded(θ) := Δ-*expanded*(θ) **and** $\bigwedge_{i=1}^{m}$ *expanded*(θ_i).

Procedure *unify-type-and-node* takes a feature structure θ and a type expression τ. It destructively unifies θ with all definitions of the types in the expression τ that are not yet member

of the Δ-*set* of θ. Since the type expression τ can be arbitrarily complex (but is guaranteed to be in normal form – CNF or DNF), *unify-type-and-node* recursively follows the structure of the type expression and unifies its types (viz., their definitions) with θ.
In a closed type world, complex conjunctive type expressions cannot occur. Type expressions of the form $\tau_1 \vee \cdots \vee \tau_n$ (τ_i possibly negated) can only occur if a conjunctive feature structure with type $\tau_1 \wedge \cdots \wedge \tau_n$ has been negated and τ_i are unexpanded (postponed). In all other cases, type disjunctions are translated to $\mathcal{UDiNe}$'s disjunction representation at definition time (cf. Chapter 3).
unify is the generic unification function for feature structures. We treat it as black box in this thesis. Function *expand-type* will be defined in the next section. It returns the (fully or partially) expanded feature structure definition of an avm type (i.e., of $\Delta(\tau)$).

procedure *unify-type-and-node*(τ, θ) **:** (preliminary)

case • $\tau \in \mathcal{T}_a$ **and** $\tau \notin \Delta$-$set(\theta)$ **then** *unify*(*expand-type*$(\tau), \theta)$**;**
/* this is the trivial case of a single avm type symbol */
• $\tau = \neg\sigma$ **and** $\sigma \in \mathcal{T}_a$ **then** *unify*(*negate-fs*(*expand-type*$(\sigma)), \theta)$**;**
/* σ is the name of an unexpanded avm type because τ is in normal form */
• $\tau = \tau_1 \wedge \cdots \wedge \tau_n$ **then for** i **from** 1 **to** n : *unify-type-and-node*(τ_i, θ)**;**
/* if τ is a complex type conjunction, then all τ_i (negated or positive) must be unified if not in Δ-$set(\theta)$ */
• $\tau = \tau_1 \vee \cdots \vee \tau_n$ **then** *unify*$(\theta, \langle \bigcup_{i=1}^{n}$ *unify-type-and-node*$(\tau_i, \langle \top, [\,] \rangle) \rangle)$**;**
/* if τ is a complex type disjunction, then a disjunctive node consisting of all τ_i is unified with θ */
• **otherwise return.**
/* i.e., τ is already in Δ-$set(\theta)$ or $\tau \in \mathcal{A} \cup \mathcal{T}_s \cup \mathcal{T}_b \cup \{\top\}$ */

Function *negate-fs* negates a typed feature structure. The only case where *negate-fs* is called during type expansion is if a complex avm type is negated within an unexpanded type expression. In this case (see *unify-type-and-node*), the definition of the type to be negated will be passed to *negate-fs*.
The negation schemata for feature structures are applied recursively. If the argument feature structure θ is disjunctive, then DeMorgan's law is applied, where conjunction ($\wedge$) of feature structures correponds to unification and negation ($\neg$) corresponds to application of the function *negate-fs*:

$$\neg\langle\{\theta_1, \ldots, \theta_n\}\rangle = \neg(\theta_1 \vee \cdots \vee \theta_n) = \neg\theta_1 \wedge \cdots \wedge \neg\theta_n$$

The negation schema for conjunctive feature structures looks somewhat complicated, but strictly follows the semantics of typed feature structures (cf. [Smolka 89]), which is defined

as conjunction of the semantics of the type and the semantics of the attribute-values pairs. If the feature structure has n features, then it can be seen as a conjunction of $n+1$ items: n feature-value pairs plus one conjunct for the type. Again, the negated complex conjunction is subject to DeMorgan's law, which propagates negation to the feature-value pairs.
A negated feature-value pair can either denote that the feature's value must be undefined, i.e., be different from all other possible values (we write $\uparrow$; the implementation of $\mathcal{UDiNe}$ uses a special atom `*undef*`), or that the feature's value is negated:

$$\begin{aligned}\neg(\tau \wedge f_1 \doteq \theta_1 \wedge \cdots \wedge f_n \doteq \theta_n) &= \neg\tau \vee \neg(f_1 \doteq \theta_1) \vee \cdots \vee \neg(f_n \doteq \theta_n) \\ &= \neg\tau \vee (f_1 \uparrow) \vee (f_1 \doteq \neg\theta_1) \vee \cdots \vee (f_n \uparrow) \vee (f_n \doteq \neg\theta_n)\end{aligned}$$

The resulting disjunction contains $2n+1$ elements. The schema is applied recursively because negation is propagated to the feature-value pairs. Therefore, negation can blow up a feature structure exponentially. This is why grammar writers should be careful with negation of large avm types.

function *negate-fs*(θ) **:**

if $\theta = \langle\{\theta_1, \ldots, \theta_n\}\rangle$ **/*** θ is disjunctive ***/**
then return $unify^n(negate\text{-}fs(\theta_1), \ldots, negate\text{-}fs(\theta_n))$
else return /* θ is conjunctive and has type τ and feature values $\theta_1, \ldots, \theta_n$ ***/**
$\langle\{\langle\neg\tau, [\,]\rangle\} \cup \bigcup_{i=1}^{n}\langle\top, [f_i \uparrow]\rangle \cup \bigcup_{i=1}^{n}\langle\top, [f_i \doteq negate\text{-}fs(\theta_i)]\rangle\rangle$.

Function $unify^n$ is the n-ary extension of the two-place function *unify*.

Now, we have presented the complete expansion algorithm for non-recursive types. Of course, the algorithm is still preliminary. It simply expands all types in a feature structure by unifying their definitions (or whatever *expand-type* returns) and is neither controllable by the grammarian nor can it handle recursive types. It would run into (mostly incorrect) infinite expansion of feature structures that contain recursive types. In the following two chapters, we will flesh out the skeleton.

Chapter 5

Indexed Prototype Memoization

5.1 Motivation

In this chapter, we describe the memoized *expand-type* function that returns the (partially or fully) expanded feature structure definition of a type, its so-called *prototype*. *expand-type* takes a type name τ and an index i (a symbol, integer, or string) that serves to identify a given member of a set of (possibly differently expanded) prototypes of τ. *expand-type* returns the prototype by expanding the skeleton $\Delta(\tau)$ according to the control information associated with index i.[1] The index can either be given as parameter to an explicit *expand-type* call or within control information specified for the feature structure containing τ.
Memoization is applied to *expand-type* in order to achieve the following goals.

- *Reducing the number of unifications*
  Once a prototype has been generated by expanding the skeleton of a type definition, it is stored in a table, and if the prototype is requested later again, a copy will be returned instead of repeating the unifications. Since copying feature structures is much faster than unifying, memoization will speed up the prototype access and hence the whole (recursive) expansion process.

- *Indexing prototypes*
  Instead of storing a single prototype for each avm type, the argument of the memoized expansion function is extended to a two-dimensional one: the type name plus a user-definable index that makes it possible to store differently expanded prototypes of a type (e.g., for partial evaluation at compile time).

- *Detecting recursive types*
  A recursive type is an avm type whose definition refers to itself directly in the skeleton or indirectly through inheritance. As a spin-off, memoization can be used to compute which

[1] If $i =$: `skeleton`, then the unexpanded skeleton is returned.

avm types are recursive. This knowledge is crucial to be able to postpone expansion that otherwise would result in infinite computation.

In this chapter, we will concentrate on the first and the second goal. Recursive types and how memoization is employed to detect them will be addressed in Chapter 6.

5.2 Memoization

The *memoization* or *tabulation* technique is as old as computer science. It can be seen as the simplest case of 'machine learning'. [Samuel 59] first proposes memoization for efficient implementation of the checkers game under the term 'rote learning'. [Michie 68] coins the term 'memoing', and develops a technique for translating arbitrary functions into memoized functions in functional programming languages. [Norvig 91] presents applications in natural language processing, e.g., for parsing.
The basic idea of memoization is to tabulate results of function application in order to eliminate redundant calculations. The more expensive the computation of a value is, the bigger the efficiency gain will be. A function to be memoized must meet the following requirements. First, it must be a true function with no side effects, because memoization of functions with side effects might cause erroneous results. Second, the function should be called more than once with the same argument, the more often, the better.[2] Recursive functions like *expand-type* (*expand-type* calls *expand-tfs* which in turn may call *expand-type* and so on) meet these two requirements and hence serve as good examples of the effectiveness of memoization.

The memoized expansion function is defined as follows.

function *expand-type*(τ, i) **:**

if *protomemo*$(\tau, i) =$ **undefined**
then $\theta :=$ *expand-tfs*(*copy-tfs*($\Delta(\tau)$)) **;**
if θ is conjunctive **and** Δ-*expanded*(θ) **then** *type-of*$(\theta) := \tau$**;**
protomemo$(\tau, i) := \theta$**;**
return *copy-tfs*(θ)
else return *copy-tfs*(*protomemo*(τ, i))**.**

expand-type checks whether a prototype of τ with index i already exists or not. If the prototype with index i already exists, a copy of the stored feature structure is returned.[3]

[2] A simple but impressive example is the memoized `fib` function [Abelson & Sussman 85; Norvig 91; Norvig 92] (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.

[3] A remark on copying is needed here. Since $\mathcal{UDiNe}$ is a destructive unifier, copies of feature structures are returned by *expand-type*. In a future version, $\mathcal{UDiNe}$ may use a more sophisticated copying mechanism that minimizes copying as proposed in [Emele 91]. In this case, the calls to *copy-tfs* simply can be omitted, and unification is responsible for it.

If the prototype with index i does not exist, the skeleton of τ is expanded according to control information that can be specified separately for each index. Finally, the prototype is stored in the prototype table (which can be accessed by function *protomemo*) and a copy is returned as the value of *expand-type*.

This is a drastic simplification of the real code. The implemented code also collects 'inherited' functional constraints that can be attached to type definitions, optionally translates the feature structure into disjunctive normal form, removes failed disjunction alternatives (through *UDiNe*'s *make-dnf* and *simplify-fs!* functions), and performs nonmonotonic overwriting (cf. Chapter 8).

5.3 Indexed Prototypes

It is sometimes desirable to store several prototypes for one type, under distinct indices. The most important reason to have more than one prototype per type is to be able to expand the definitions differently. E.g., it makes sense to postpone the expansion of semantic information in HPSG lexicon entries for parsing because semantics does not filter very much (cf. [Diagne et al. 95]). In other situations, a fully expanded lexicon entry may be needed as fast as possible.

Another application for indexed prototypes is *partial evaluation*. Often needed types (or all types, if lexicon entries are stored with *TDL*'s instance facility) can be expanded at compile time. This helps to minimize the number of unifications at run time. The prototypes can serve as *basic blocks* for building a partially expanded grammar.

An even more radical approach can also be pursued by indexed prototypes: *copy pools*, bundles of copies of prototypes, can be generated at compile time or in idle run time when the NL system waits for user input, each with another index. The destructive unifier then 'consumes' the fresh copies at run time without having to copy the feature structures. Heuristics can be applied to estimate the number of copies that are required. This strategy may speed up processing because time for copying is transferred from run time to compile time.[4] Heuristics about how many prototypes are needed can be obtained through training sessions.

Figure 5.1 shows how skeletons and prototypes are stored in memory. For each avm type, there is a skeleton, the definition of the type, and at least one default prototype (with index `nil`). Initially, the prototype feature structure is identical with the skeleton. A call to *expand-type* generates an expanded copy of the skeleton. Control information can be associated with each prototype index (e.g., for postponing types, etc.; see Chapter 7) that is consulted in *expand-tfs*.

[4] or, as Hassan Aït-Kaci formulated more concisely in a talk on a different topic he gave at DFKI, "Space is cheaper than time".

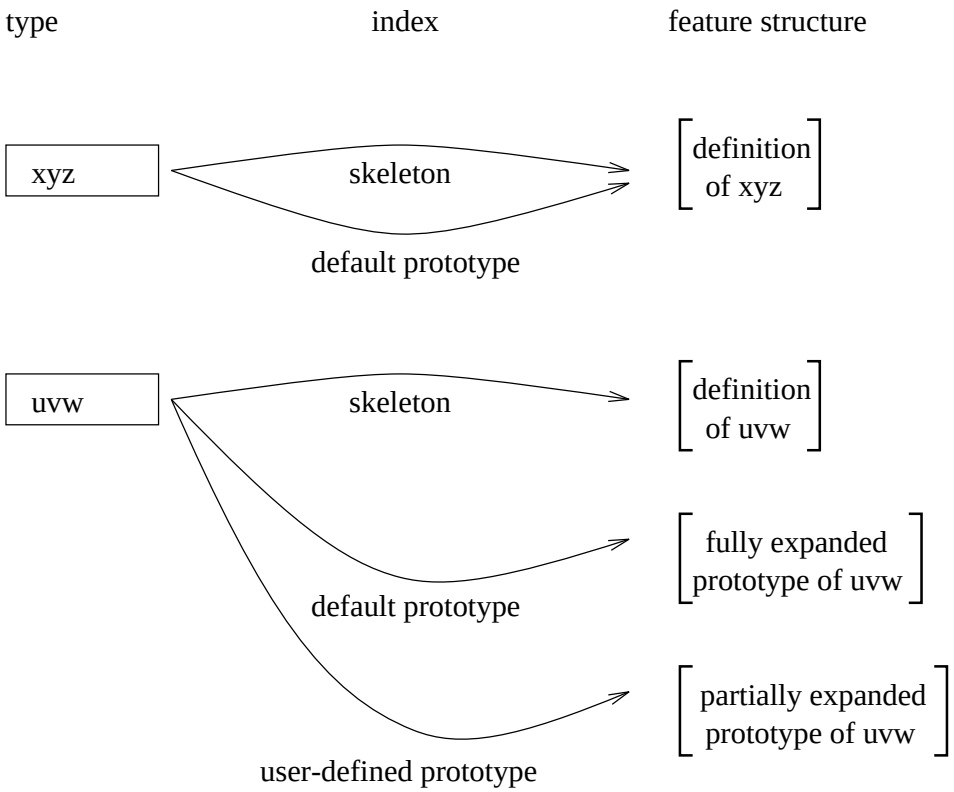


Figure 5.1: *Skeletons, Prototypes, and Indices: Type xyz's prototype is either unexpanded or contains no avm types. Thus, its prototypical feature structure is identical with its definition (skeleton). Type uvw has a (fully) expanded prototype and a user-defined prototype which are both (possibly partially) expanded copies of uvw's skeleton feature structure.*

5.4 Reducing the Number of Unifications – An Example

To show the performance gain caused by prototype memoization, we compare run time for full expansion of an HPSG lexicon with and without memoization. Moreover, we also compare depth-first vs. types-first expansion algorithms (described in Section 4.4.3).

Figure 5.2 contains statistical information about the expansion of a grammar with approx. 900 avm type definitions. It is an HPSG grammar for German, roughly as described in [Netter 93]. About 250 lexicon entries and rules have been expanded from scratch. They have not been defined as avm types, but as *instances*, feature structures that are not part of the type hierarchy, but inherit from lexical or rule types (they can be seen as the leaves of the hierarchy). Like avm types, instances have an associated name and definition, and different prototypes with control information. Their main purpose is to keep the type hierarchy small even if large lexica are defined.

All instances and types are unexpanded at the beginning (two further columns in the table

algorithm	*depth-first-expand*		*types-first-expand*		*depth-first-expand*		*types-first-expand*	
memoization	yes		yes		no		no	
run time	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 pre-	208	atomic-wff	83	sort-expr	1003	wff-type	1073	wff-type
expanded	202	rp-type	71	atomic-wff	933	agr-feat	951	agr-feat
lexical	146	conj-wff-type	62	rp-type	880	semantics	747	semantics
types	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

Figure 5.2: *Comparison of different expansion algorithms with and without prototype memoization. Run time is stated in seconds for a SPARC Station 10 with Allegro Common Lisp 4.2. the table shows (1) that there is no significant difference between depth and types first expansion (the grammar contains no recursive types) and (2) that prototype memoization speeds up full expansion of lexicon entries by a factor of 10 which is roughly proportional to the number of unifications.*

marked with an asterisk show run time with all types pre-expanded, but unexpanded instances). The type and instance skeletons together consist of about 9000 nodes. No control information for preference or postponement is given. The algorithm without memoization inserts only the unexpanded skeletons of a type definition, while the memoized version expands each complex type once and afterwards only returns copies of it.
The resulting structures consist of about 50000 nodes (27000 in type prototypes, 23000 in instance prototypes). Each instance is expanded once (and so are all avm types in the memoized version) using $\mathcal{TDL}$'s *expand-all-instances* command. A sample instance of a lexicon entry is depicted on page 32 and its $\mathcal{TDL}$ definition on page 31.
The measurements show that memoization speeds up expansion by a factor of 5 for this grammar (10 if all types are pre-expanded, which we consider a good strategy for real applications). The time difference between memoized and non-memoized algorithm may be even bigger if disjunctions are involved. The sample grammar contains only a few disjunctions (about 300 disjunction nodes in the definitions).

5.5 Accessing Prototypes

The following prototype access procedures are part of the $\mathcal{TDL}$ language. If a prototype with the specified index does not exist, it is created using a copy of the skeleton. *expand-control* is a complex structure which specifies parameters that control expansion (cf. Chapter 7). While *index* for avm types can be a string, number, or symbol for type prototypes, only *number*s are allowed for instance indices. The reason is that instances are stored differently from type prototypes (namely, in a list).

- `expand-type` *type* [`:index` *index*] [`:expand-control` *expand-control*] [`:domain` *domain*].
  generates a new prototype by expanding the definition skeleton of *type* or further expands an existing one.

- `expand-instance` *instance* [`:index` *number*] [`:expand-control` *expand-control*] [`:domain` *domain*].
  generates a new instance with index *number* by expanding the definition skeleton of *instance* or further expands an existing one.

- `expand-all-types` [`:index` *index*] [`:expand-control` *expand-control*] [`:except` *exception-list*] [`:domain` *domain*].
  expands the definition skeletons of all types with index *index* except those in the *exception-list*.

- `expand-all-instances` [`:index` *number*] [`:expand-control` *expand-control*] [`:except` *exception-list*] [`:domain` *domain*].

expands the definition skeletons of all instances with index *number* except those in the *exception-list*.

- `reset-proto` [*type* [`:index` *index*] [`:domain` *domain*]].
  resets the prototype with index *index* of an avm type to its skeleton.

- `reset-instance` [*instance* [`:index` *number*] [`:domain` *domain*]].
  resets the prototype with index *number* of an instance to its skeleton.

- `reset-all-protos` [`:domain` *domain*].
  resets all prototypes of all avm types.

- `reset-all-instances` [`:domain` *domain*].
  resets the prototypes of all instances.

- `defcontrol` { *type* | *instance* | `:global` } *expand-control* [`:index` *index*]
  [`:domain` *domain*].
  specifies control information for a type or instance with the given index.

The global variable `*DEFAULT-INDEX*` contains the name of the default index name that is assumed if no index argument is specified. Its default value is `nil`.

Chapter 6

Recursive Types

6.1 Introduction

Recursive types are avm type whose defining feature structure refers to the type itself directly (within the skeleton) or indirectly (through other types).
Although recursion is inherent in natural language representation, e.g., in context-free rules for syntax, not all feature structure formalisms support recursive types. Systems like ALE or *TDL ExtraLight* provide recursion only through definite clauses or through a (chart) parser, but not through types.
However, recursive types increase the expressivity of feature structure formalisms and enable the grammar writer to encode even more linguistic knowledge uniformly and elegantly within feature structures. This is why *TDL* has been designed as a successor of *TDL ExtraLight* to cope with recursive types. LIFE does so as well, but as we argued in Chapter 4, its implicit expansion mechanism as part of the unification process leads to some disadvantages that *TDL*'s architecture is designed to avoid.
In this chapter, we will first give an overview on decidability results, then discuss different kinds of recursion, their significance, and some (linguistic) applications. Then, we describe the expansion algorithm extensions necessary to treat recursive types correctly and to avoid infinite expansion. At the end of the chapter, we give some examples that illustrate the algorithm.

6.2 Motivation

Many NL systems avoid recursive types and instead provide various extensions beyond typed feature structures that support recursion such as definite clauses or context-free rules.
Nevertheless, recursive types can be used to elegantly formulate the following applications (among others):

- *Context-free backbone*: It is obvious that constituent structure can be expressed through

recursive types (as is the case for HPSG).

- *List types*: Lists can be defined recursively or non-recursively using feature structures. In both cases, an atom or sort **null** denotes end of list, and **list** is defined as a disjunctive type:

  ```
  *list* := *null* | *cons*.
  ```

  In the non-recursive version, **cons** is defined as

  ```
  *cons* := [ FIRST, REST ].
  ```

  whereas the recursive definition is

  ```
  *cons* := [ FIRST, REST *list* ].
  ```

  The recursive definition is stronger in that it stipulates that a finite list ends with **null**, while the non-recursive definition admits 'dotted pair' lists. Both definitions can be appropriate, depending on the application.

- *Finite state automata*: Morphology and phonology can be encoded by these devices. [Krieger et al. 93] show how to define finite state automata through recursive typed feature structures and present applications from allomorphy. We will see an expansion trace of a sample automaton at the end of the chapter.

- *Append*: List concatenation is frequently employed in NL processing. Recursive types even allow encoding of the relational version of append that works bidirectionally à la Prolog. A comprehensive example is given below.

- *Functional Uncertainty*: Recursive types can be used to model functional uncertainty constraints, which is an alternative device for describing long-distance dependencies and constituent coordination [Kaplan & Zaenen 88].

Of course, this list of possible applications is incomplete. Since recursive types make feature structure formalisms with coreferences Turing-equivalent, other applications are possible. Let us mention two paradigms in natural language processing that have been suggested for such powerful formalisms.
Parsing as deduction [Pereira 83] (and generation) can be supported by type expansion. One only needs a sufficiently specified grammar (using recursive types as described above). For parsing the sentence "Fido likes cookies", one specifies phonology only and starts expansion of the following structure.

$$\begin{bmatrix} \textit{phrase} \\ \textit{:expanded false} \\ \textit{:delta } \{\} \\ \text{PHON } \langle \texttt{"Fido"}, \texttt{"likes"}, \texttt{"cookies"} \rangle \end{bmatrix}$$

Type expansion then deduces the missing information. The result will be an expanded feature structure representing the whole analyzed sentence, including syntax and semantics.

For generation, one only specifies the semantic representation. Expansion will lead to a feature structure containing the missing syntax and phonology, i.e., a well-formed sentence.
This leads to purely declarative and very elegant parsing and generation without any additional parser or generator. Of course, ambiguity, termination, and efficiency problems emerge. But parameterized expansion (cf. Chapter 7) can help to moderate these difficulties.

The second paradigm has been proposed by [Mellish & Reiter 93]: using the formalism as a *programming language* that can also encode *extralinguistic* or *meta knowledge* in NL systems. While they used a KL-ONE derivative, namely I1, and classification instead of expansion to demonstrate the feasibility of their approach, we expect that feature structure expansion is also able to manage because of its comparable expressivity.
Mellish and Reiter present a schema for translation of Prolog programs into 'classification programs'. An example from their paper is the ubiquitous *append* function (see example at the end of this chapter). Actually, as Mellish and Reiter do for I1, one can see $\mathcal{TDL}$ as a declarative programming language that makes no distinction between data and procedures.

6.3 Decidability

Checking satisfiability of typed feature structures with variables (coreferences) is undecidable if we admit recursive type definitions. There are at least three different proofs in the literature we will shortly mention here.[1]
The first proof is by [Rounds & Manaster-Ramer 87]. They show that Kasper-Rounds logic enriched with recursive types can be used to construct a two-stack push-down automaton that is equivalent to a Turing machine. Thus, deciding satisfiability would imply that the Halting problem for Turing machines is decidable.
[Smolka 89] shows that coreference constraints are the source of undecidability in combination with recursive types. His proof is by encoding the word problem of Thue systems within feature structures.
The most recent (and most detailed) proof is by [Aït-Kaci et al. 93] and uses the compactness theorem of first-order logic. Moreover, they present an order-sorted feature theory unification algorithm which is encoded within 10 rewriting rules. If the algorithm is restricted to 9 rules, it is always terminating (but not complete). If one adds the tenth rewriting rule, one gains completeness but loses general termination. Finally, they clarified what is important for practical applications: non-satisfiability is semi-decidable, i.e., if the feature structure is inconsistent, it can be determined in finitely many steps.
Regrettably, their algorithm does not support disjunctions and cannot be translated to an efficient version for the $\mathcal{TDL}/\mathcal{UDiNe}$ system where feature structures are represented as dags,

[1] As Bob Carpenter (p.c.) pointed out, Hassan Aït-Kaci was the first to show that type expansion is Turing equivalent in his 1984 thesis. He did this implicitly by showing how to code up arbitrary Prolog programs.

since it is based on OSF clauses (decomposed ψ-terms).
What are the consequences of undecidability for $\mathcal{TDL}$'s expansion algorithm? First, it is clear that recursive types are the crucial point. We cannot forbid coreferences because they are indispensable for natural language representation. Second, the expansion algorithm should be complete in general if it is called by the user (or a parser etc.), and not restricted by user-specified control information (see next chapter). However, when called within memoization, type expansion must be guaranteed to be terminating (we will elaborate on this later). Third, expansion of recursive types should be postponed if infinite expansion can be foreseen (laziness). To be able to satisfy these requirements, we will have a closer look at how and where recursion occurs.

6.4 Recursion in Knowledge Representation Languages

[Smolka 88] mentions the relation between feature logic and knowledge representation languages like KL-ONE [Brachman & Schmolze 85] (we did so in Chapter 2). Leaving aside the difference between features (functions) and roles (relations), the main similarity is obviously that types in feature structure formalisms correspond to concepts in KL-ONE.
Thus, we can adopt some reflections on terminological cycles from [Nebel 90] and [Nebel 91]. While early papers on KL-ONE simply suggested forbidding terminological cycles, Nebel distinguishes two kinds of terminological cycles: component cycles and restriction cycles.

6.4.1 Component Cycles

A component cycle occurs if a concept to be defined inherits from itself. This obviously violates the condition for concepts as well as for feature types that they must be ordered hierarchically. Consequently, Nebel suggests forbidding such cycles. An example from [Nebel 90], translated into $\mathcal{TDL}$ syntax is:

```
man := human & male-human.
male-human := human & man.
```

We will follow his argumentation since, besides the philosophical consideration, $\mathcal{TDL}$'s encoding technique [Aït-Kaci et al. 89] does not admit such cycles.

6.4.2 Restriction Cycles

Although $\mathcal{TDL}$ and other feature structure languages do not provide restrictions in the sense KL-ONE does (namely, number and value restrictions), there is something corresponding: feature values. Actually, features in feature languages are just a special case of restriction in concept languages. A $\mathcal{TDL}$ type definition like

```
person := human & [ father person,
                    mother person ].
```

can be expressed as follows in KL-ONE:

$$T(\texttt{person}) = \texttt{human} \sqcap \exists^{\geq 1}\texttt{father} \sqcap \exists^{\leq 1}\texttt{father} \sqcap \forall\texttt{father} : \texttt{person} \\ \sqcap \exists^{\geq 1}\texttt{mother} \sqcap \exists^{\leq 1}\texttt{mother} \sqcap \forall\texttt{mother} : \texttt{person}$$

Again, we follow Nebel's approach: he suggested admitting restriction cycles, i.e., cycles where the concept (type) to be defined occurs as a restriction (feature value) in the definition (directly or indirectly through inheritance). The simple justification is that this kind of cycle makes sense and is useful for knowledge representation and natural language processing as we will show in the next section.

6.5 Algorithm

We now present the extensions to the expansion algorithm that are necessary to handle recursive types properly. The algorithm as described so far would expand recursive types repeatedly without returning at all. Therefore, we first examine sources of infinite expansion.

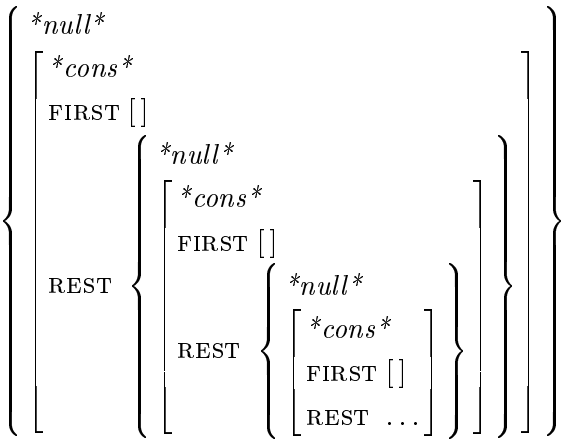
6.5.1 Sources of Infinite Expansion

Of course, Nebel's distinction of component vs. restriction cycles does not help to solve termination problems. It is only a distinction between meaningful and meaningless cycles in inheritance hierarchies. Therefore, closer inspection of restriction cycles and feature structures is necessary.
Generally, there are two sources of non-termination in feature structure expansion.

- *Strongly recursive feature structures* are the defining feature structures of recursive types. If the *skeleton* of a recursive type is expanded, no finite 'input' is given that stops the definition of the type from being unified repeatedly (at increasing path depth). Examples are the following **list** and **cons** type definitions.

```
declare sort : *null*.
*list* := *null* | *cons.
*cons* := *avm* & [ first [],
                    rest *list* ].
```

`expand-type '*list*` with naive expansion (using the algorithm as presented so far) would result in infinite computation:



We will develop a method of detecting and stopping such infinite expansion.

- *Weakly recursive feature structures* are feature structures that are not skeletons of a recursive type but in which at least one type entry in the structure itself or in inherited constraints refers to a recursive type. An example is the following feature structure.

  $$\begin{bmatrix} \top \\ \text{LIST} \;\; *list* \end{bmatrix}$$

  Naive expansion may or may not stop on weakly recursive feature structures. It stops if finite 'input' (i.e., non-empty feature list) that brakes recursion at finite path depth is specified together with the recursive type. It definitely does not stop if the recursive type stands alone. Examples of such structures are first-rest lists with finite length (**null** terminates the recursion) or context-free rules that are saturated by terminal symbols.

The idea for the expansion algorithm is to exploit strongly recursive feature structures in order to detect recursive types and to postpone expansion on strongly and weakly recursive feature structures as appropriate.
Another source of infinite expansion stems from memoization. The memoization algorithm as presented would try to expand recursive types endlessly. Here, recursive types must be postponed within memoization. But before we modify the algorithm to postpone expansion of recursive types, we show how to compute which types are recursive.

6.5.2 Computing Recursive Types

From the above consideration, it follows that it is crucial for the expansion algorithm to know which types are recursive in order to be able to postpone their expansion. There are two ways of detecting recursive types.

The first one is static and can be done at type definition time. When an avm type is being defined, all type names that occur in the skeleton are stored in a set associated with the type, the so-called occurrence type set (*ots*). A type is (strongly) recursive if it is contained in its own *ots* or in the transitive closure of the types in its *ots*.
The advantage of this method is that it can be done at type definition time. The disadvantage is that it is expensive and furthermore may compute a superset of the types that can actually cause infinite expansion. This is because the occurence type sets cannot be computed correctly without full expansion of the definitions (which in turn would need the knowledge about which types are recursive to prevent infinite computation); the *ots* contain a superset of the actual occurrence types in the expanded definitions since disjunction branches containing avm types may be cut by unification failure. Although the method of computing *ots* has been implemented within $\mathcal{TDL}$, we decided not to use it because it slows down type definitions and hence thwarts incremental grammar development.

The second method is dynamic, preserves efficient incrementality and, furthermore, computes the correct (minimal) set of recursive types. It is a by-product of prototype memoization and hence relies on memoization (which is not bad since memoization is advantageous as we have shown above). The idea is to record the type of the feature structure being expanded in a stack that is passed from one memoized *expand-type* call to another (via *expand-tfs*). Each call of *expand-type*$(\tau, index)$ pushes τ onto the stack and passes the the new stack to *expand-tfs*. If a 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 immediately 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 each other type in the scc by a sequence of *expand-type* and *expand-tfs* calls. An example of such scc is (**cons* *list**) for the recursive list type defined above. Each time a *new* scc is detected, it is stored in a global variable `*RECURSIVE-SCCS*`.
We modify functions *expand-type*, *expand-tfs*, *unify-type-and-node*, *depth-first-expand*, and *types-first-expand* such that they take another parameter *stack* which is simply passed from one function to another.
At the beginning of function *expand-type* (page 48) the following line is inserted:

check-recursive-avm$(\tau, stack)$; /* check if τ is recursive */
push$(\tau, stack)$; /* push τ onto type stack */

and the new stack is passed to *expand-tfs*. In procedure *check-recursive-avm*, a new scc is added to `*RECURSIVE-SCCS*` if τ is in *stack* but not already marked as a recursive type.
Testing whether a type is recursive or not then reduces to a simple *find* operation in the global `*RECURSIVE-SCCS*` list (which is typically rather small compared to the total number of types). *find-recursive* τ returns the scc that contains τ iff τ is recursive, and `nil` otherwise

and hence serves as a predicate, too. We present Common Lisp code here because it is more concise in this case.

```
(DEFUN find-recursive (type)
   (FIND-IF #'(LAMBDA (scc)
                (MEMBER type scc))
            *RECURSIVE-SCCS*))
```

6.5.3 Postponing Recursive Types: Lazy Expansion

There are two situations where recursive types must be postponed (leaving aside postponement enforced by control information which is addressed in the next chapter). As the postponement is done automatically to make life easier for grammarians, it resembles *laziness* in modern functional (or object-oriented) programming languages, so we adopt this term.

Situation 1: Within Implicit Expansion

Implicit expansion is feature structure expansion that has been induced by memoization (i.e., not by a user-call to *expand-type* or *expand-instance*). The strategy is simply to postpone *all* recursive types that occur in implicitly expanded structures. This kind of postponement has been hard-wired in the code for two reasons: (1) It prevents infinite loops in memoized expansion, i.e., expansion is guaranteed to terminate when called implicitly. (2) It avoids copying overhead. As borne out in practice (proven by run time measurements), it is not a good idea to pre-expand and memoize recursive types, since they are copied in vain if inconsistency occurs (which is always the case for finite structures).
The disadvantage is that inconsistent type definitions containing recursive types may not be detected immediately unless one calls explicit expansion. Nevertheless, the resulting structures are correct and, more importantly, it is not a good idea to define inconsistent types (consistency can optionally be checked at type definition time in $\mathcal{TDL}$).

Situation 2: Within Explicit Expansion

Explicit expansion is feature structure expansion explicitly called by the user or functions different from memoization. Here, we generally chose a complete expansion algorithm, because we assume that the caller knows what he is doing (cf. the problem termination vs. completeness; of course, complete expansion can be restricted by control parameters).
However, there are structures where recursive types can be anticipated to expand infinitely, but never contribute to inconsistency. Lazy expansion recognizes these cases and then postpones the recursive type (after having it expanded once).
The *default recognition criterion* is that the type to be postponed automatically must be recursive, and the node containing it must not bear features. Moreover, the type (or a

subtype) already must have been Δ-*expanded* at a proper subpath. A *proper* subpath is a proper prefix of a given path that contains identical disjunction alternatives.
Consider the feature structure from the *weakly recursive feature structure* paragraph above. Expansion stops after **list** has been expanded once because afterwards, **list** (viz., **cons**) has been expanded once at path LIST and **list** does not bear features at path LIST.REST. So the resulting structure is

$$\left[\begin{array}{l} \top \\ :expanded\ false \\ :delta\ \{\} \\ \text{LIST} \left\{\begin{array}{l} *null* \\ \left[\begin{array}{l} *cons* \\ :expanded\ false \\ :delta\ \{\ *cons*\ \} \\ \text{FIRST}\ [\,] \\ \text{REST} \left[\begin{array}{l} *list* \\ :expanded\ false \\ :delta\ \{\} \end{array}\right] \end{array}\right] \end{array}\right\} \end{array}\right]$$

It is worth emphasizing that $\mathcal{TDL}$'s lazy expansion is completely different from *lazy attribute inheritance* as proposed by [Aït-Kaci 93]. In lazy attribute inheritance, only attribute values that appear locally are inherited from the type definition, regardless of the question of whether it is recursive or not (cf. Section4.3.2). The disadvantage of this strategy is that it is inherently slow and only yields partially expanded structures. The advantage over our strategy is that structures are minimal which suffices if one is only interested in satisfiability.

Resolvedness

Resolvedness is a property of typed feature structures that is checked at the beginning of each loop in *expand-tfs*. Its main purpose is to serve as an additional means of stopping infinite expansion of recursive types. While the laziness criterion is always safe, i.e., it cannot lead to incompleteness unless the feature structures is infinite, different criteria are possible for resolvedness (because of undecidability), depending on the recursive types involved. Therefore, $\mathcal{TDL}$ permits different definitions of resolvedness. User-defined criteria, so-called *resolved-predicate*s, can be specified as control parameters (see next chapter). The *resolved-predicate* is checked in the top-level loop in procedure *expand-tfs*. It takes a complex control structure comprising various information, such as *recursive-paths*, a list of paths containing recursive types that is computed on the fly.
The default resolvedness predicate is *always-false*. It always returns false, and hence does not stop expansion in any case. An example of a recursion that stops using the default resolvedness criterion is the recursive version of the **list** type as defined above.

Depth-First vs. Types-First Search

A special case additionally occurs if the search strategy *types-first-expand* has been chosen. In contrast to *depth-first-expand*, a recursive type must be expanded only once at a path in each top-level loop in procedure *expand-tfs* to guarantee fair expansion. Otherwise, explicit expansion would loop at the first recursive type encountered. Again, a feature structure containing the **list** type is an example of a weakly recursive feature structure whose types-first expansion would expand forever without this hard-wired brake. If an unexpanded list of length 5 has to be expanded, five top-level loops in *expand-tfs* are necessary. In contrast, *depth-first-expand* only needs one loop through the structure. This is why we do not recommend the types-first strategy if many recursive types are involved.

Distinguishing Explicit vs. Implicit Expansion

As we argued above, expansion of feature structures is different depending on if we are at the top-level feature structure that is to be expanded (explicit expansion) or if memoization of a depending feature structure (implicit expansion) takes place. The method the algorithm uses to distinguish between these two expansion modes is somewhat tricky. It is done by looking at the memoization type stack *stack*.

$$\text{expansion is } \begin{cases} \text{explicit}, & \text{if } |stack| = 1, \\ \text{implicit}, & \text{otherwise, i.e., if } |stack| > 1. \end{cases}$$

Therefore, in a user call to *expand-type*, *stack* is initialized with the empty stack (*expand-type* pushes the type name onto the stack immediately before calling *expand-tfs*). User calls to *expand-node*, *expand-instance*, and *expand-tfs* initialize *stack* with $(\top)$, where $\top$, the top type of the hierarchy which can never be recursive because it bears no features, is employed as a dummy item that guarantees stack height 1 for expansion of the top level structure.

The *delay* Predicate

Whether type expansion is postponed or not, i.e., whether the type definition is unified with the node that refers to it or not, is checked in the first case in procedure *unify-type-and-node* (page 45) by calling the *delay* predicate:

procedure *unify-type-and-node*$(\tau, \theta, stack)$ **:** (revised)

case • $\tau \in \mathcal{T}_a$ **and** $\tau \notin \Delta\text{-}set(\theta)$ **and not** $delay(\tau, \theta, stack, path)$
then $unify(expand\text{-}type(\tau), \theta)$**;**
/* this is the trivial case of a single avm type symbol ***/**
• $\tau = \neg\sigma$ **and** $\sigma \in \mathcal{T}_a$ **and not** $delay(\sigma, \theta, stack, path)$ **then** ...
⋮

The implemented code is more complicated. We have omitted here the case of expanding a recursive type for the first time. In this situation, *delay* always returns false because it not known that the type is recursive *before expand-type* is called. Additional code ensures that unification of the recursive type definition with the local node will not take place by calling *delay* a second time (only in this case). If *delay* returns true, the feature structure returned by *expand-type* is thrown away. Because this occurs only once for each recursive type, memory is not wasted significantly. Alternative methods would have been to set the recursive sccs manually/statically (which contradicts incrementality as we argued above), or to expand the recursive type for the first time anyhow (which would add some kind of indeterminacy).

The *delay* predicate is a straightforward formalization of the postponement rules we gave informally.

function $delay(\tau, \theta, stack, path)$ **:**

return *find-recursive*(τ) **and**
$((|stack| > 1)$ **or** (θ has no features **and**
$\exists$ proper subpath of *path* where τ is Δ-*expanded*)).

To efficiently access the subpaths containing recursive types, they are collected on the fly. Only two memory cells are necessary per path: one for the pointer to the actual path, and one for the type name.

6.6 Examples

The algorithm now treats recursive types correctly. Loops in memoization are prevented and expansion of feature structures containing recursive types is lazy. Finally, we can demonstrate the algorithm using two of the applications for recursive types motivated in Section 6.2: *append* and finite state automata. Since we already gave examples of laziness (the list types), we will concentrate on complex structures with finite 'input' to recursive types whose expansion will lead to a fully expanded, finite feature structure (or inconsistency).

6.6.1 Append

The first example is the *append* relation (cf. [Aït-Kaci 86], [Mellish & Reiter 93]). It concatenates two lists specified at the FRONT and BACK values and returns the result at the WHOLE value. Since *append* works bidirectionally as in Prolog, it can also be used to synthesize possible input from given output (and, optionally, partial input). If there are several possibilities, they are represented as disjunctive alternatives.
Using the following definitions,

```
defdomain :append :load-built-ins-p NIL.
begin :domain :append.
  begin :declare.
    sort: *null*.
  end :declare.

begin :type.
  *cons* := [ FIRST, REST *list* ].
  *list* := *null* | *cons*.
  append0 := [ FRONT *null*,
               BACK  #1 & *list*,
               WHOLE #1 ].
  append1 := [ FRONT < #first . #rest1 >,
               BACK  #back & *list*,
               WHOLE < #first . #rest2 >,
               PATCH append & [ FRONT #rest1,
                                BACK  #back,
                                WHOLE #rest2 ] ].
append := append0 | append1.
```

we can expand the following definition

```
test := append & [ front < "Fido", "likes" >,
                   back  < "cookies" > ].

expand-type 'test.
```

to obtain the concatenation of the input lists at the WHOLE feature.
The following trace of `expand-type 'test` has been generated automatically by the type expansion algorithm. The search strategy is *depth-first-expand* (as we recommend for recursive types); no additional control information has been specified. We assume that it was not known before that **list**, **cons**, *append*, and *append1* are recursive types to illustrate how the recursive sccs are computed on the fly. Subsequent expansions using the *append* type will need less expansion steps.
Each call to *expand-type* is recorded in the trace as well as the following action (expand definition if it is unexpanded, or return memoized feature structure if already expanded). The expanded feature structures are printed at the end of each pass in *expand-tfs*. *unif-occ*=... prints the value of the flag that indicates whether a unification has taken place during the last pass in *expand-tfs*. *resolved*=... prints the value of the resolvedness predicate (always false in our example). List structures are printed in their first-rest encoding instead of using <...> to show where the types come from.

In order to save space, we abbreviate the *:delta* and *:expanded* slots by two boxes that are printed on the right of a type name within a feature structure. The left box is ☐, if the value of *:expanded* is false, and Ⓔ, if the value is true. Because no complex type expressions occur in the sample feature structures, we can treat *:delta* as a flag (i.e., Δ-*expanded*). If the type is not locally expanded (Δ-*expanded*=false), the right box is ☐. If the type is locally expanded (Δ-*expanded*=true), then is Δ is printed.

expand-type(**cons**, `nil`) in prototype of *test* under path FRONT.REST, *stack=(test)*:
Expanding definition of **cons**, index `nil`.
expand-type(**list**, `nil`) in prototype of **cons** under path REST, *stack=(*cons* test)*:
Expanding definition of **list**, index `nil`.
New recursive type **cons** detected. *recursive-sccs=((*cons* *list*))*.
End of pass 0 in **list**, index `nil`, *unif-occ*=false, *resolved*=false:

$$\left\{ \begin{array}{l} \left[\text{*cons*} \; \square\square \right] \\ \text{*null*} \end{array} \right\}$$

End of pass 0 in **cons**, index `nil`, *unif-occ*=false, *resolved*=false:

$$\left[\begin{array}{l} \text{*cons*} \; \square\Delta \\ \text{FIRST} \; [\,] \\ \text{REST} \; \left[\text{*list*} \; \square\square \right] \end{array} \right]$$

---

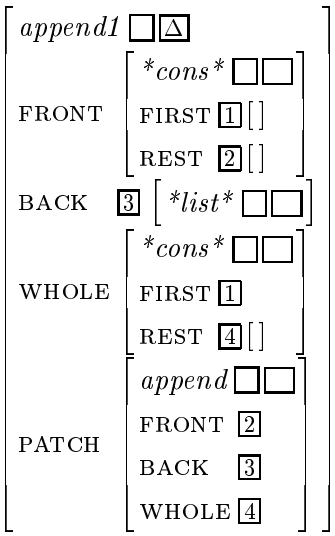
expand-type(**cons**, `nil`) in prototype of *test* under path FRONT, *stack=(test)*:
Returning *protomemo*(**cons**, `nil`).
expand-type(**cons**, `nil`) in prototype of *test* under path BACK, *stack=(test)*:
Returning *protomemo*(**cons**, `nil`).
expand-type(*append*, `nil`) in prototype of *test* under path ϵ, *stack=(test)*:
Expanding definition of *append*, index `nil`.
expand-type(*append0*, `nil`) in prototype of *append* under path ϵ, *stack=(append test)*:
Expanding definition of *append0*, index `nil`.
Delaying recursive type **list** in prototype of *append0* under path BACK.
End of pass 0 in *append0*, index `nil`, *unif-occ*=false, *resolved*=false:

$$\left[\begin{array}{l} \text{append0} \; \square\Delta \\ \text{FRONT} \; \text{*null*} \\ \text{BACK} \; \boxed{1} \left[\text{*list*} \; \square\square \right] \\ \text{WHOLE} \; \boxed{1} \end{array} \right]$$

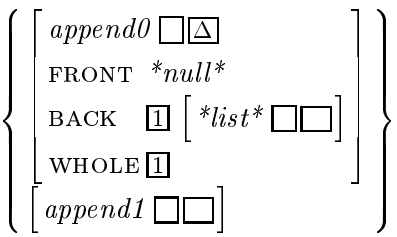
---

expand-type(*append1*, `nil`) in prototype of *append* under path ϵ, *stack=(append test)*:

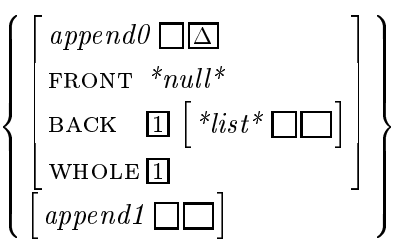
Expanding definition of *append1*, index `nil`.
Delaying recursive type **cons** in prototype of *append1* under path FRONT.
Delaying recursive type **list** in prototype of *append1* under path BACK.
Delaying recursive type **cons** in prototype of *append1* under path WHOLE.
New recursive type *append* detected. *recursive-sccs=((append append1) (*cons* *list*))*.
End of pass 0 in *append1*, index `nil`, *unif-occ*=false, *resolved*=false:



End of pass 0 in *append*, index `nil`, *unif-occ*=true, *resolved*=false:



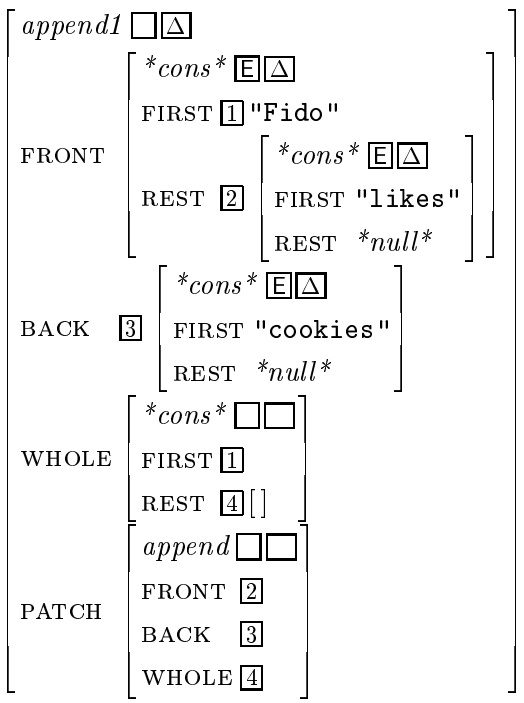
Delaying recursive type **list** in prototype of *append* under path WHOLE.
Delaying recursive type *append1* in prototype of *append* under path ϵ.
End of pass 1 in *append*, index `nil`, *unif-occ*=false, *resolved*=false:



---

expand-type(*append1*, `nil`) in prototype of *test* under path ϵ, *stack=(test)*:
Returning *protomemo*(*append1*, `nil`).

End of pass 0 in *test*, index `nil`, *unif-occ*=true, *resolved*=false:



---

expand-type(**cons**, `nil`) in prototype of *test* under path WHOLE, *stack=(test)*:

Returning *protomemo*(**cons**, `nil`).

expand-type(**list**, `nil`) in prototype of *test* under path PATCH.WHOLE, *stack=(test)*:

Returning *protomemo*(**list**, `nil`).

expand-type(**cons**, `nil`) in prototype of *test* under path PATCH.WHOLE, *stack=(test)*:

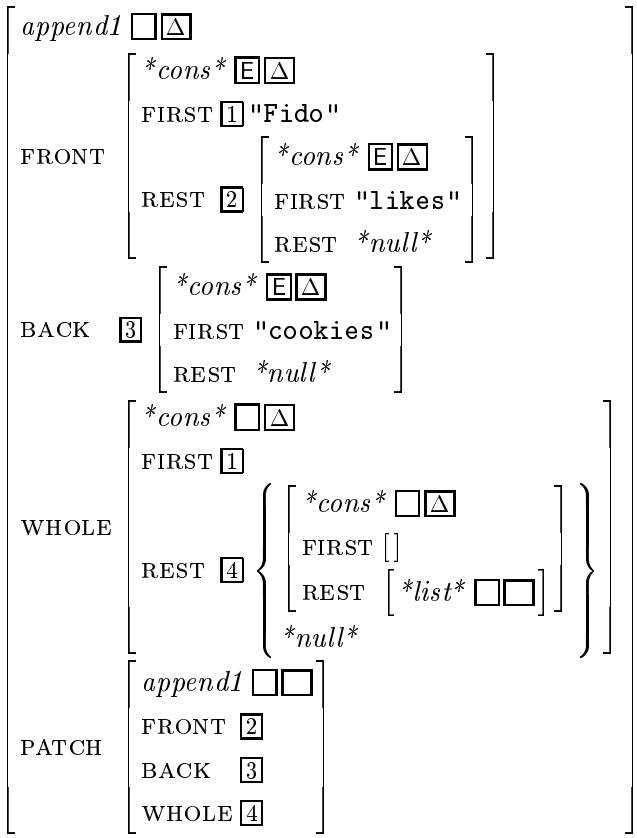
Returning *protomemo*(**cons**, `nil`).

expand-type(*append*, `nil`) in prototype of *test* under path PATCH, *stack=(test)*:

Returning *protomemo*(*append*, `nil`).

Delaying recursive type *append1* in prototype of *test* under path PATCH.

End of pass 1 in *test*, index `nil`, *unif-occ*=true, *resolved*=false:

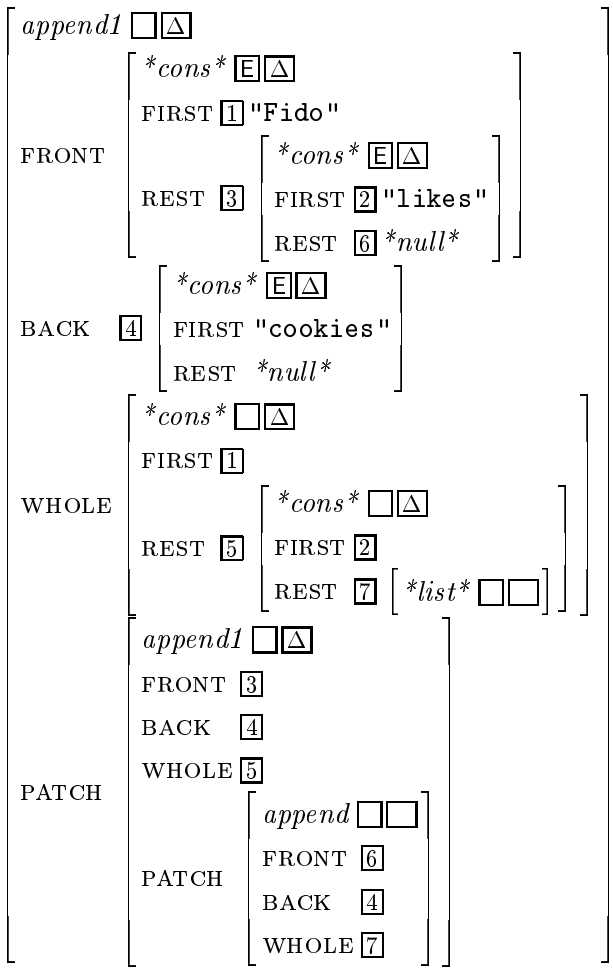


Delaying recursive type **list** in prototype of *test* under path WHOLE.REST.REST.

expand-type(*append1*, `nil`) in prototype of *test* under path PATCH, *stack=(test)*:

Returning *protomemo*(*append1*, `nil`).

End of pass 2 in *test*, index `nil`, *unif-occ*=true, *resolved*=false:



Delaying recursive type **list** in prototype of *test* under path WHOLE.REST.REST.

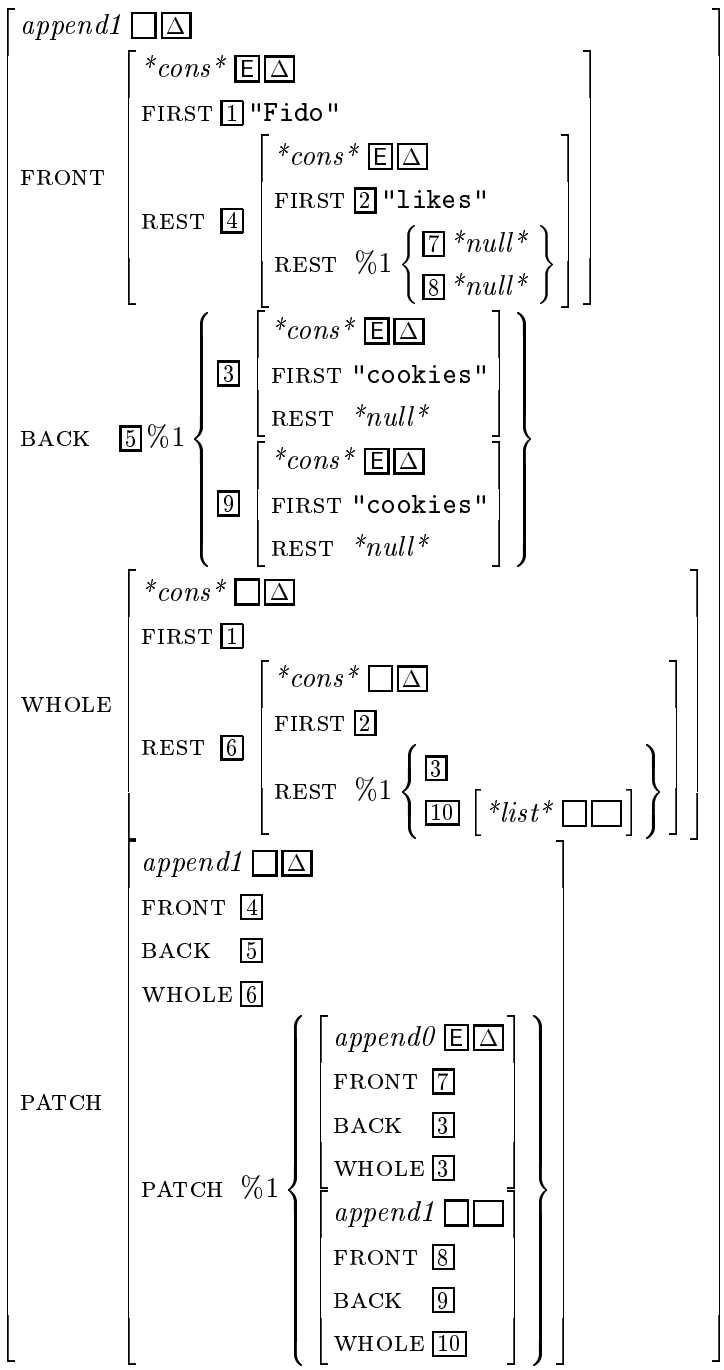
expand-type(*append*, `nil`) in prototype of *test* under path PATCH.PATCH, *stack=(test)*:

Returning *protomemo*(*append*, `nil`).

Delaying recursive type *append1* in prototype of *test* under path PATCH.PATCH.

Delaying recursive type **list** in prototype of *test* under path PATCH.WHOLE.REST.

End of pass 3 in *test*, index `nil`, *unif-occ*=true, *resolved*=false:

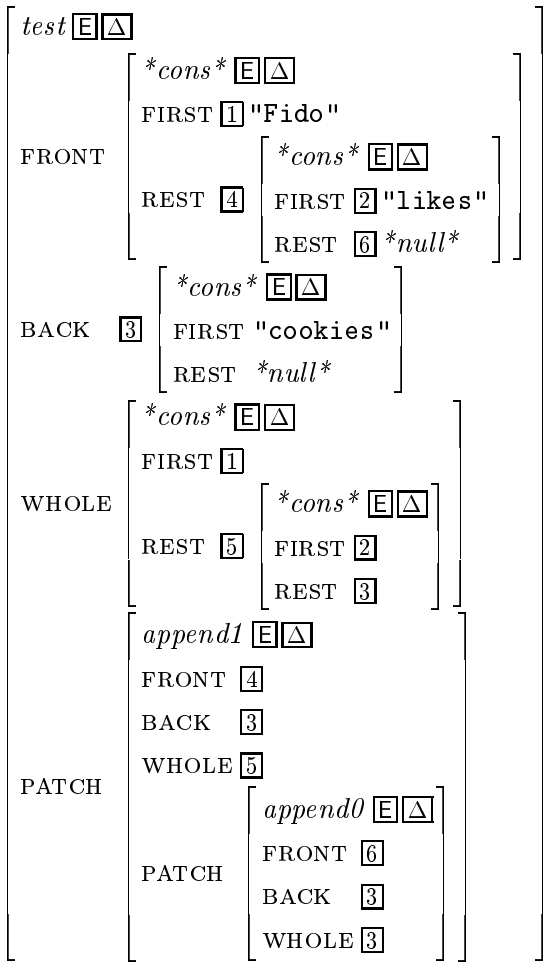


Delaying recursive type **list** in prototype of *test* under path WHOLE.REST.REST.

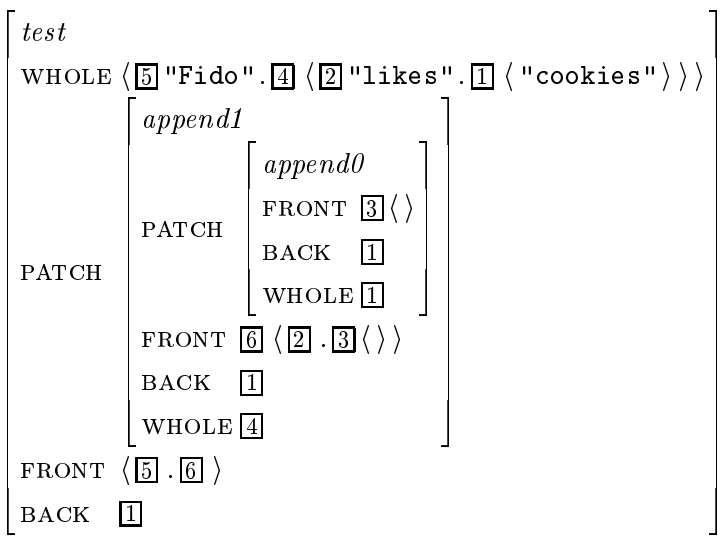
expand-type(*append1*, `nil`) in prototype of *test* under path PATCH.PATCH, *stack=(test)*:

Returning *protomemo*(*append1*, `nil`).

End of pass 4 in *test*, index `nil`, *unif-occ*=false, *resolved*=false:



The resulting structure in the abbreviated, more readable list format is



6.6.2 A Finite State Automaton

The second sample trace is for a simple finite state automaton defined by using distributed (named) disjunctions as proposed in [Krieger et al. 93].

We define a finite automaton for the language $\mathcal{L} = \mathtt{a}^*(\mathtt{a} + \mathtt{b})$.

Type `final-config` models the final state, `non-final-config` is the supertype of all non-final states. The automaton accepts an input given as list value at the INPUT path, if and only if complete expansion is consistent. If a word is rejected, the corresponding feature structure is `*fail*` (inconsistent).

```
begin :declare.
  sort : *undef*.
end :declare.
begin :type.
  non-final-config := [ INPUT < #1 . #2 >,
                        EDGE #1,
                        NEXT [ INPUT #2 ] ].
  final-config := [ INPUT *null*,
                    EDGE *undef*,
                    NEXT *undef* ].
  state1 := non-final-config & [ EDGE %1( 'a,     'a | 'b ),
                                 NEXT %1( state1, final-config ) ].
  test-ab := state1 & [ INPUT < 'a, 'b > ].

  expand-type 'test-ab.
```

In contrast to the first example, we assume that all types have already been expanded (except `test-ab`) and hence the recursive sccs `(*cons* *list*)` and `(state1)` have already been computed before the call to `expand-type 'test-ab`.

expand-type(**cons**, `nil`) in prototype of *test-ab* under path INPUT.REST, *stack=(test-ab)*:

Returning *protomemo*(**cons**, `nil`).

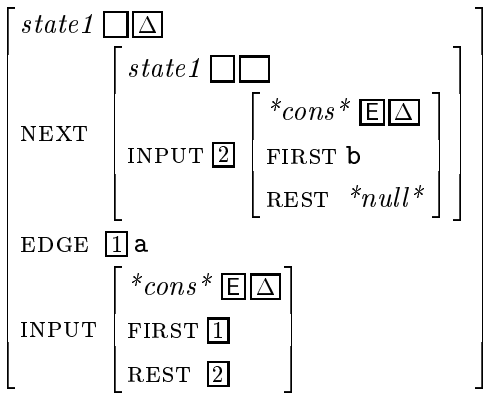
expand-type(**cons**, `nil`) in prototype of *test-ab* under path INPUT, *stack=(test-ab)*:

Returning *protomemo*(**cons**, `nil`).

expand-type(*state1*, `nil`) in prototype of *test-ab* under path ϵ, *stack=(test-ab)*:

Returning *protomemo*(*state1*, `nil`).

End of pass 0 in *test-ab*, index `nil`, *unif-occ*=true, *resolved*=false:

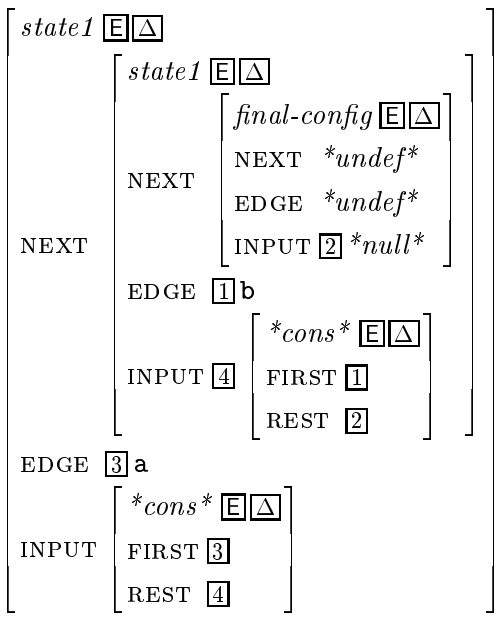


---

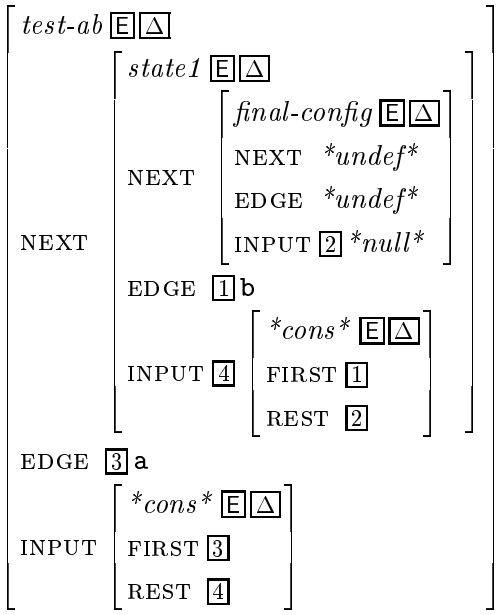
expand-type(*state1*, `nil`) in prototype of *test-ab* under path NEXT, *stack*=*(test-ab)*:

Returning *protomemo*(*state1*, `nil`).

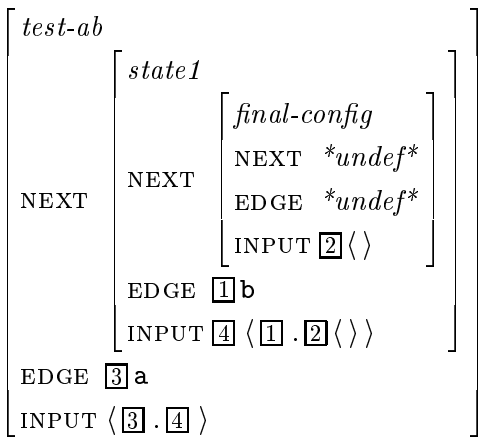
End of pass 1 in *test-ab*, index `nil`, *unif-occ*=true, *resolved*=false:



End of pass 2 in *test-ab*, index `nil`, *unif-occ*=false, *resolved*=false:



The resulting structure in the abbreviated, more readable list format is



If unacceptable input is given to the automaton, type expansion results in global unification failure, e.g.,

```
test-ai := state1 & [ INPUT < 'a, 'i > ].
expand-type 'test-ai.
```

Again, we assume that it was known that **list**, **cons**, and *state1* are recursive.

expand-type(**cons**, `nil`) in prototype of *test-ai* under path INPUT.REST, *stack=(test-ai)*:
Returning *protomemo*(**cons**, `nil`).

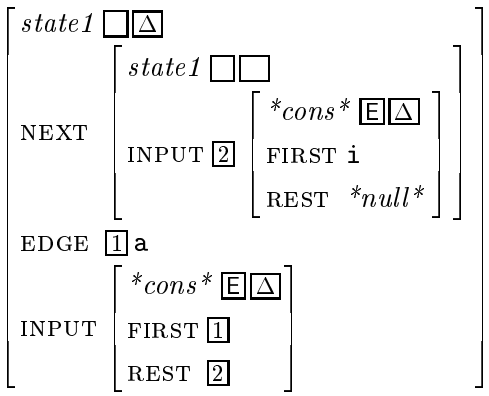
expand-type(**cons**, `nil`) in prototype of *test-ai* under path INPUT, *stack=(test-ai)*:

Returning *protomemo*(**cons**, `nil`).

expand-type(*state1*, `nil`) in prototype of *test-ai* under path ϵ, *stack=(test-ai)*:

Returning *protomemo*(*state1*, `nil`).

End of pass 0 in *test-ai*, index `nil`, *unif-occ*=true, *resolved*=false:



---

expand-type(*state1*, `nil`) in prototype of *test-ai* under path NEXT, *stack=(test-ai)*:

Returning *protomemo*(*state1*, `nil`).

Warning: Type *test-ai* **is inconsistent.**

Chapter 7

Controlling Type Expansion

7.1 Motivation

In the last three chapters, we have presented the basic expansion algorithm, the memoization technique that helps to reduce the number of unifications, and proper treatment of recursive types. In this chapter, we will enrich the basic algorithm by control that may speed up linguistic processing by (1) keeping feature structures small (partially expanded) and (2) reducing search space for expansion. Moreover, control information can serve as an additional means (besides lazy expansion) to stop expansion of recursive types. The instruments for controlled expansion are

- postpone expansion of types
- choose specific prototypes to be inserted by expansion
- set a maximal path depth for expansion
- select the expansion search strategy
- exploit heuristics for backtracking and expansion order
- specify preference information for the order in which attribute values are expanded
- define resolvedness criteria

In the following sections, we will describe syntax and semantics of control information and briefly sketch their introduction into the expansion algorithm implementation.

7.2 Declarative Specification of Expansion Control

The overall design goal for expansion control is to keep its specification completely declarative, as the grammars themselves are. Control can be specified globally or locally for a prototype

(or both), either in a separate file or mixed with the type and instance definitions of the grammar.
The `begin :control ... end :control` environment can be used to indicate control information. It has been introduced for symmetry with other $\mathcal{TDL}$ environments to structure grammar files, but it is completely optional and can be omitted. Within a control environment, only the `defcontrol` statement is admitted.
The $\mathcal{TDL}$ statement for defining control information is `defcontrol`. Its syntax is:

`defcontrol` { *type* | *instance* | `:global` } *expand-control*
[`:index` *index*] [`:environment` {`:type` | `:instance`}] .

The first parameter in the `defcontrol` statement is a symbol, either the name of a type or the name of an instance. Which of both it designates depends on the surrounding environment (`:type` or `:instance`). If we are in the `:control` environment, its surrounding environment (either `:type` or `:instance`) determines the kind of argument. Alternatively, the `:environment` keyword can be used to enforce interpretation of the first parameter as either type or instance name. If the first argument is `:global`, then global control information is specified, independently of the environment. This allows the user to choose a depth first search strategy generally, e.g., but to replace it locally by another strategy.
A `defcontrol` statement can be placed anywhere in a grammar file, before or after the corresponding type or instance definitions (but always within a control, type or instance environment). A newer `defcontrol` declaration with the same type/instance and *index* replaces an older one; the same holds for global control, i.e., `defcontrol` is *not* cumulative.
Global control can *only* be defined by the `defcontrol` statement, whereas there are two more possible locations for local expansion control.

- Control information for the default prototypes of avm types or instances can be attached directly to the type or instance definition using the following syntax:
  name `:=` *body* `, expand-control:` *expand-control* .
- *expand-control* can also be passed as an optional keyword argument to all expansion interface procedures such as `expand-node`, `expand-type`, etc. (cf. $\mathcal{TDL}$ syntax in Appendix A), e.g.,
  `expand-type` *name* [`:index` *index*]
  [`:expand-control` *expand-control*] .

In the latter case, other local control information (for type and instance prototypes) is ignored. The values of missing slots are always inferred from global control.

7.3 Syntax of Expansion Control

The syntax of *expand-control* is defined by the following BNF:

expand-control → ([(`:expand` { ({*type* | (*type* [*index* [*pred*]])} {*path*}$^+$) }*) |
(`:expand-only` { ({*type* | (*type* [*index* [*pred*]])} {*path*}$^+$) }*)] |
[(`:delay` { ({*type* | (*type* [*pred*])} {*path*}$^+$) }*)] |
[(`:ignore-global-control` {`t` | `nil`})] |
[(`:maxdepth` { *integer* | `nil` })] |
[(`:expand-function` {{`depth` | `types`}`-first-expand` | ... })] |
[(`:resolved-predicate` {`resolved-p` | `always-false` | ... })] |
[(`:use-conj-heuristics` {`t` | `nil`})] |
[(`:use-disj-heuristics` {`t` | `nil`})] |
[(`:attribute-preference` {*identifier*}*)] |
[(`:ask-disj-preference` {`t` | `nil`})])

where

path → {*identifier* | *pattern*} {.{*identifier* | *pattern*}}*
pattern → `?` | `*` | `+` | `?`[*identifier*][`?`|`*`|`+`]
pred → `eq` | `subsumes` | `extends` | ...
type → *identifier*
index → *integer* | *identifier* | *string*

The syntax will be described in detail in the rest of the chapter. We only give an overview in this section. The `:expand` and `:expand-only` slots are mutually exclusive. Default is `:expand` for all types with default prototypes. Both slots determine the basic mode for expansion and can be used to specify which prototype of a type is unified at which path(s) in the feature structure. The `:delay` slot can additionally be used to postpone types at certain paths independently from the basic expansion mode. Path patterns and type comparison predicates such as the subsumption test facilitate the use of these three path and type-related slots. If the value of `:ignore-global-control` is `nil`, then global and local path patterns are merged, otherwise, local patterns override global ones.
`:maxdepth` defines the maximal path depth where expansion takes place. Types at longer paths will be postponed. `:maxdepth nil` sets the maximal depth to ∞. The basic search strategy is specified by `:expand-function`, i.e., `types-first-expand`, `depth-first-expand`, or a user-defined strategy. The `:resolved-predicate` can be used to replace complete expansion ('`always-false`') of recursive types by other strategies. `:use-conj-heuristics` and `:use-disj-heuristics` enable or disable the use of heuristic information for expansion order of attributes (conjuncts) and backtracking (disjuncts). Finally, one can enable interactive choice of disjunct order (`:ask-disj-preference`), and define a canonical order in which attribute values are expanded (`:attribute-preference`).
An example of the use of the syntax is

```
defcontrol intrans-verb-lex
  ((:delay ((semantics Subsumes) ?.?.SYNSEM.*))
```

```
   (:attribute-preference HEAD FIRST)
   (:use-disj-heuristics T)
   (:ignore-global-control T)
   (:expand ((morphology initial) CAT.MORPH.LIST)))
  :index 1.
```

7.4 The Control Structure

The control information specified in *expand-control* is stored in a control structure that can be accessed by all functions and procedures of the expansion algorithm. Each prototype of a type or an instance has its own control structure that is re-used if the prototype is expanded again. For functions *expand-node* and *expand-fs*, fresh control structures are created for each call. The original definition of the control structure is

```
(DEFSTRUCT control
  ;;; slots definable through expand-control
  (expand                 ()                    :type LIST)
  (expand-only            ()                    :type LIST)
  (delay                  ()                    :type LIST)
  (maxdepth               NIL                   :type (OR NULL INTEGER))
  (attribute-preference   ()                    :type LIST)
  (ask-disj-preference    NIL                   :type (MEMBER NIL T))
  (use-disj-heuristics    ()                    :type (MEMBER NIL T))
  (use-conj-heuristics    ()                    :type (MEMBER NIL T))
  (resolved-predicate     'Resolved-P           :type (OR SYMBOL FUNCTION))
  (expand-function        'Depth-First-Expand   :type (OR SYMBOL FUNCTION))
  ;;; internal slots:
  (timestamp              '(-1 . -1)            :type LIST)
  (pass                   0                     :type INTEGER)
  (index                  NIL                   :type ATOM)
  (unification-occurred-p NIL                   :type (MEMBER NIL T))
  (functional-constraints NIL                   :type LIST)
  (recursive-paths        ()                    :type LIST)
 )
```

In addition to the slots that correspond to the *expand-control* input syntax, the control structure contains slots that are used internally by the expansion algorithm:

- `timestamp` is a pair consisting of a timestamp for local (prototype) expansion control, and a timestamp for global expansion control. The timestamps are set up in such

a way that the control structure need not be re-initialized if neither local nor global control specification has changed, i.e., they help to avoid unnecessary computations and copying.

- `pass` is the counter for passes in procedure *expand-tfs* (0 at the beginning).

- `index` contains the index name of the current prototype being expanded.

- `unification-occurred-p` is a flag that indicates whether expansion (unification) has occurred in the last expansion pass. It is set to `nil` at the beginning of each pass, and set to `t` if unification takes place locally in the structure being expanded. The flag helps to stop expansion of postponed (and recursive) types as it is checked at the top-level of procedure *expand-tfs* (page 44).

- `functional-constraints` is a list (set) containing functional constraints that is associated with the feature structure being expanded. For the sake of simplicity, we omit description of $\mathcal{UDiNe}$'s functional contraints in this thesis, but they have indeed been integrated into the expansion algorithm. 'Inheritance' of functional constraints (through type expansion) is done by non-destructive union of the functional constraint sets. The `functional-constraints` slot is used to collect the constraint set temporarily.

- `recursive-paths` is a list consisting of all recursive types that have been visited within the last toplevel loop in procedure *expand-tfs*. Only recursive types are recorded in the list. Each entry in the list has the following form

  (*type* Δ-*expanded node path*)

  type is the name of the recursive type, Δ-*expanded* indicates whether the definition of *type* is already expanded or not. *node* is a pointer to the node where *type* occurred, *path* is the path from the root node to *node*. The list is mainly used for lazy expansion and can also be accessed by user-defined resolvedness predicates (see below). It is reset to the empty list at the beginning of each pass in *expand-tfs*.

When code is presented below, we use the C notation for structure accessing functions, e.g., *control.recursive-paths* is the access function for the control slot `recursive-paths`.

Other parameters required by the expansion algorithm, such as *path* (the current path within the structure being expanded), *stack* (the type stack of calls to *expand-type*), *domain* (the current type domain), and $\mathcal{UDiNe}$'s private control object, are passed directly from one expansion procedure to another, and hence are not part of the control structure.

7.5 Expanding and Postponing Prototypes

Syntax and Description

Three slots control which types are expanded or postponed, and which prototypes are inserted under which paths. `:expand` and `:expand-only` are mutually exclusive:

({`:expand` | `:expand-only`} { ({*type* | (*type* [*index* [*pred*]])} {*path*}$^+$) }*)

In the `:expand` mode (default), all types are expanded (as in the algorithm presented so far). If not otherwise specified, the default prototypes with index `nil` are inserted. For each type, paths and indices can be defined to indicate where to insert which prototype. In the `:expand-only` mode, only those types are expanded that are explicitly mentioned (with specific prototypes and paths), all others are postponed.
Independently from the expansion mode, types are postponed anyway if listed in the `:delay` slot (again, with specific paths).

(`:delay` { ({*type* | (*type* [*pred*])} {*path*}$^+$) }*)

This delay information as well as postponement of recursive types (cf. Chapter 6) overrides both expansion modes.
The reasons why there are three somewhat redundant slots for expansion and postponement are the following. First, creating prototypes with partially expanded types should be as easy as possible. Here, the `:expand-only` mode is suitable to generate such feature structures before run time. The `:expand` mode is more intended to be used at run time, where as much information from types as possible should be gathered as fast as possible. Nevertheless, some type constraints can be postponed with the `:delay` syntax.
The second reason for the threefold syntax for postponement and expansion is to be able to express exceptions from generalizations that can be made by the path patterns and type comparison predicates (*pred*).

Path Patterns

Instead of exact feature paths, one can specify path patterns that are matched against the actual paths during expansion by pattern matching with wildcards, variables, and segment variables. This allows powerful generalizations over paths. The syntax for path patterns is

path → {*identifier* | *pattern*} {.{*identifier* | *pattern*}}*

pattern → `?` | `*` | `+` | `?`[*identifier*][`?`|`*`|`+`]

Figure 7.1 explains the meaning of path patterns.[1]

[1]Note that `*` is rather a wildcard than a Kleene star in the sense of functional uncertainty where regular expressions over features are used. However, functional uncertainty can be modeled through recursive types in $\mathcal{TDL}$. One can restrict expansion of possibly infinite paths by path patterns in the `:delay` slot.

pattern	meaning
feature	match feature exactly
?*identifier*	segment variable that matches one feature
?*identifier*?	segment variable that matches zero or one feature
?*identifier*+	segment variable that matches one or more features
?*identifier**	segment variable that matches zero, one, or more features
?	unique pattern variable that matches one feature
??	unique pattern variable that matches zero or one feature
+, ?+	unique pattern variable that matches one or more features
, ?	unique pattern variable that matches zero, one, or more features

Figure 7.1: *The meaning of path patterns.*

Segment variables are local to each path pattern.
Examples:

- `?x.rest.?x` matches FIRST.REST.FIRST, LAST.REST.LAST
- `?x*.rest.?x*` additionally matches REST, A.B.REST.A.B
- `+.last` matches REST.LAST, FIRST.REST.LAST
- `*.last` additionally matches LAST
- `?.?.?` matches all paths of length 3
- `?.?+` matches all paths of length > 2

Usually, disjunctions are not recorded in a path, i.e., the REST value at path FIRST.REST may be a direct value of FIRST or can be a disjunction alternative if the FIRST value is a disjunction. Occasionally, one may wish to choose only one disjunction alternative (especially for postponing expansion). In this case, one can include the position of the disjunct within the disjunction node (starting from 1) like a attribute names. This is possible because $\mathcal{UDiNe}$'s distributed disjunction representation guarantees fixed positions of disjuncts, provided `*SIMPLIFY-FS-P*` is set to `nil` to ensure that failed alternatives do not change the positions.
Example:

- `first.2` matches the second disjunction alternative under path FIRST
- `first.2.*.3` matches all third alternatives at arbitrary depth under the second alternative of the FIRST disjunction.

If no path pattern contains numbers, disjunctions are ignored (i.e., treated as the empty path).

Type Comparison predicates

The second way to generalize over types to be expanded or postponed is by type comparison predicates. The predicates are used to check whether the types in the control slot (`:expand`, `:expand-only`, or `:delay`) match the type to be expanded.

pred → `eq` | `subsumes` | `extends` | ...

The following predicates are predefined

- `eq`: type identity (this is the default if no predicate is specified)
- `subsumes`: the type in the control slot subsumes the type in the feature structure to be expanded
- `extends`: the type in the feature structure to be expanded subsumes the type in the control structure

Other, user-defined predicates can be specified as well. The predicate must take two arguments, the first argument will be *type* in the `:expand`, `:expand-only`, or `:delay` list, the second is the type to be compared in the structure that is subject to expansion. An example is the following user-defined predicate
disjunctive-subtype-p(x, y) = `t`(true) iff y is below x in the type hierarchy and y is defined disjunctively (e.g., **list**)

```
begin :lisp.
   (defun disjunctive-subtype-p (x y)
     (and (subsumes x y)
          (udine::disjunction-node-p (feature-structure-term
                                      (get-prototype y)))))
end :lisp.
```

Examples

The following global control definition states that for all types that are subtypes of the *semantics* type, a prototype with index `semindex` has to be inserted, but only at paths that have SYNSEM as first or second attribute. At all other paths, and for all other types, the prototypes with index `nil` are inserted, except for the types below *syntax*, where prototypes with index `synindex` are expanded under all paths. Moreover, expansion of the *lex-type* is postponed for all paths.

```
defcontrol :global ((:expand ((semantics semindex subsumes)
                              synsem.* ?.synsem.*)
                             ((syntax synindex subsumes) *))
                   (:delay  (lex-type *))).
```

The special index name `:skeleton` refers to the skeleton of a type definition. An alternative to setting the global variable `*USE-SKELETON-P*` (Section 7.15) would be to define

```
defcontrol :global ((:expand ((*top* :skeleton subsumes) *))).
```

Then, instead of using memoization, each occurrence of any type (`subsumes *top*`) under any path will have its skeleton inserted.
To define control information for a single type or instance, its name (instead of `:global`) and an index has to be specified, where `nil` designates the default index. In the following example, `:environment :instance` forces `kommen` to be interpreted as an instance name, even if `defcontrol` is enclosed by a `:type` environment.

```
defcontrol kommen ((:delay ((semantics subsumes) *)))
           :index initial-lex
           :environment :instance.
```

Then,

```
expand-instance 'kommen :index initial-lex.
```

generates an instance prototype with all `semantics` types (including subtypes) postponed. The same result could be achieved by specifying control information as an argument to the `expand-instance` function:

```
expand-instance 'kommen
                :index initial-lex
                :expand-control '((:delay ((semantics subsumes) *))).
```

Implementation

It is clear from the expansion algorithm code presented in the preceeding chapters that function *unify-type-and-node* (page 45 and page 63) is responsible for checking the `:expand`, `:expand-only`, and `:delay` lists before unifying type definitions with the local feature structure node, and for choosing the requested prototypes. Because the implementation is straightforward (but the code is lengthy), we only sketch it briefly.
The order of checking the different alternatives is as follows (we assume that the argument of *unify-type-and-node* is a single type symbol, otherwise, recursion over complex type expression is done as in the code already presented). First, check for recursive types. If they must be postponed, this has to be done independently from control information in order to avoid infinite loops. Second, check for types in the `:delay` list, because this information overrides `:expand` and `:expand-only`. Third, check for `:expand` or `:expand-only` (only one of both is allowed, the other must be empty).

If the contents of the `:expand`, `:expand-only` and `:delay` lists are checked, the type is compared first, because type subsumption (and, of course, equality) can be computed in $O(1)$ with respect to the size of the type hierarchy using [Aït-Kaci et al. 89]'s encoding technique. If the type is OK, then pattern matching is done to compare the current path with the path patterns in control information. The pattern matcher we use is from [Norvig 92, Section 6.2] with some minor modifications.
If type and path match one of the entries in the expand/delay list, the corresponding action (postpone or unify prototype with specified index) is performed. If there is more than one expression that matches the current path and type, the leftmost is taken (the rest is not considered). Local and global control are merged at compile time (in case the value of `:ignore-global-control` is `nil`) in such a way that local control is checked before global control.
Let us have a closer look at one of the path and type checking functions, say the one for `:expand-only`. At compile time, `defcontrol` translates the `:expand-only` input syntax using the ZEBU LALR(1) parser [Laubsch 93] into the following list representation.

$$
\begin{array}{l}
(\ (\ (type_1\ index_1\ predicate_1)\ path_{1_1}\ \ldots\ path_{1_n}\ ) \\
\quad\vdots \\
\quad (\ (type_m\ index_m\ predicate_m)\ path_{m_1}\ \ldots\ path_{m_p}\ )\ )
\end{array}
$$

Path patterns are translated into the internal format of Norvig's pattern matcher during parsing of the control syntax at definition time (e.g., `?x?` is translated to `(?? ?x)`). The generated list is stored in the control structure and can be accessed through the *control.expand-only* function.
The test function for `:expand-only` then is defined as follows (again, we present the original, slightly simplified Common Lisp code because is much more concise).

```
(DEFUN Type-is-in-Expand-Only-List (type control path)
  "type is current type, path is current path, control is control struct"
  (MEMBER type (control.expand-only control)
          :test #'(LAMBDA (name name-path)            ;;; check type first
                    (AND (FUNCALL (CADDAR name-path)  ;;; EQ, SUBSUMES,...
                                  (CAAR name-path)    ;;; type in list
                                  name)               ;;; current type
                         (SOME #'(LAMBDA (x)          ;;; then check path
                                   (PAT-MATCH x       ;;; patterns in list
                                              path))  ;;; current path
                               (CDR name-path))))))
```

The function returns `nil` if no type and path in `:expand-only` matches the current type and path. Otherwise, the rest of the `:expand-only` list starting with the matching entry is returned (non-`nil`).

7.6 Merging Global and Local Control

Syntax and Description

`(: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 in local control. If the flag is `nil`, the locally and globally specified type and path lists are merged where the local values preceed the global ones. The value of `:ignore-global-control` does not affect the values of the other control slots described in the subsequent sections. For these, local values always override global ones. The values of global control information are only considered if no local information is given.

Implementation

Time stamps associated with the control structures are checked each time a new control definition is processed to avoid unnecessary copying.

7.7 Maximal Path Depth for Expansion

Syntax and Description

`(:maxdepth {` *integer* | `nil` `} )`

This slot sets the maximal path depth where types are expanded. If not `nil`, all types at paths longer than *integer* will be postponed. This control may be used as a brute force method of stopping infinite expansion and ensuring termination. Moreover, it can be useful to easily generate partially expanded prototypes, e.g., lists of any length.

Implementation

The current path depth is passed from one expansion procedure to the next, and its length is increased by one in each call to *depth-first-expand* and *types-first-expand*. This avoids computation of the current path length from scratch in each call to these procedures. The current path depth is checked in *depth-first-expand* and *types-first-expand*. If it exceeds the value of `:maxdepth`, these functions will not be invoked again on the feature values.
Because of the structure-sharing representation of coreferences in $\mathcal{UDiNe}$, results are unpredictable for feature structures that contain coreferences between nodes where one node has path depth $>$ `:maxdepth`, and the other has path depth $<$ `:maxdepth`. If the node at the smaller path is visited first, then expansion is cut later than if the longer path is visited first. This is because coreferring nodes are only visited (expanded) once. Which path is visited first depends on the search strategy and on the order of attributes. We believe that this indeterminacy will do no harm in practice.

7.8 Search Strategy

Syntax and Description

`(:expand-function {{depth | types}-first-expand | ...} )`

The `:expand-function` slot specifies the basic expansion search strategy. Predefined strategies are `depth-first-expand` (the default) and `types-first-expand` which we recommend only for some special cases (e.g., if feature structures with many postponed types are to be fully expanded). The details, advantages and disadvantages have been discussed in Section 4.4.3. Other user-defined strategies can be specified as well by defining a Common Lisp function that takes the same parameters (namely 12). The parameters are, among others, the current node, path, path depth, type stack, type domain, $\mathcal{UDiNe}$'s control object and the expansion control structure.

Implementation

The implementation can be discussed briefly. The preliminary code of *expand-tfs* from page 44 has to be changed in such a way that the fixed call of `depth-first-expand` is replaced by a `FUNCALL` to the value of *control.expand-function* with the parameters mentioned above.

7.9 Resolvedness Predicates

Syntax and Description

`(:resolved-predicate {always-false | ...})`

This slot specifies a user definable predicate that may be used to prematurely stop the toplevel loop in *expand-tfs* (page 44). The predicate has mainly been introduced to enable the grammar writer to stop expansion of recursive types as appropriate.
The default predicate is `always-false` which will make the expansion algorithm complete (if no delay or maxdepth restriction is given, of course).

Implementation

The resolvedness predicate is checked at the toplevel loop in *expand-tfs* (page 44). Instead of a fixed predicate, the value of *control.resolved-predicate* has to be `FUNCALL`ed with the following arguments

```
(FUNCALL control.resolved-predicate control domain node ctrl-obj)
```

where `control` is the whole current control structure, `domain` is the current type domain, `node` is the root node of the feature structure to be expanded, `ctrl-obj` is $\mathcal{UDiNe}$'s internal control object that contains information about fail contexts etc.

These parameters make all important information accessible to a user-defined resolvedness predicate.
To write one's own resolvedness predicate, extensive knowledge about *UDiNe*'s feature structure representation and *TDL*'s internal structure is necessary. It is beyond the scope of this thesis to go into that much detail, and therefore, we informally describe an example, say, for the *append* type. The predicate uses the `:recursive-paths` slot of the control structure to check the values under the FRONT{.REST}*.FIRST, BACK{.REST}*.FIRST, and WHOLE{.REST}*.FIRST subpaths of each node that is associated with the *append1* type. If all these values are typed with $\top$, and bear no features, and all other nodes with recursive types are either expanded or postponed by laziness, then the resolvedness predicate assumes that there is no input and output for *append* and returns true; false otherwise. Because this check may be expensive if structures are large, we recommend it mainly for development and debugging. For real-world applications, the grammar writer should either ensure termination by sufficiently specified input to recursive types (which one can assume in NL processing), or by less expensive methods such as `:maxdepth`; also cf. Section 7.14.

7.10 Numerical Preferences

[Kogure 90] and [Uszkoreit 91] suggested that exploitation of preference information for features and disjunctions would speed up unification. For conjunctions of feature-value pairs, this can be achieved by unifying features with highest failure probability first; for disjunctions, the alternatives with highest success probability first. Another possibility is a backtracking strategy that picks only one disjunct, proceeds with unification as if this were the only value, and backtracks if this fails.
It is convenient to store the failure potential, numerically represented, directly at the feature structure nodes. The values can either be computed statically by training sessions, or dynamically at run time. In the latter case, several algorithms for rearranging the conjunct and disjunct order are possible.
Because expansion is essentially unification, numerical preference information can also be used to speed up the expansion process. Features with high failure probability are expanded first, disjuncts with high failure potential last.

Implementation

The implementation is simple from the point of view of type expansion. *UDiNe* is responsible for storing and gathering preference information because the unification algorithm itself has to be modified to cope with preferences. The interface to expansion only consists of a predicate. We assume that there is an ordering predicate that can be used to sort features and disjunctions according to the local preference values (the same predicate is used within

unification). Procedures *depth-first-expand* and *types-first-expand* are modified in such a way that they rearrange disjunct and conjunct order[2] before the elements are visited.

Syntax and Description

The control slots

`(:use-conj-heuristics` {`t` | `nil`}`)`

and

`(:use-disj-heuristics` {`t` | `nil`}`)`

trigger the use of preference information during expansion. *control.use-conj-heuristics* and *control.use-disj-heuristics* are checked in procedures *depth-first-expand* and *types-first-expand*. If the value of the flags is true, the feature or disjunct order is determined by a predicate that uses the preference information for $\mathcal{UDiNe}$.
Because the preference extension to $\mathcal{UDiNe}$ is not yet finished, we cannot present empirical results to prove the usefulness of this approach.

7.11 Attribute Order

Syntax and Description

`(:attribute-preference` {*identifier*}*`)`

This slot defines a partial order on attributes by *non-numerical preference* information. The sub-feature structures at the leftmost attributes will be expanded first. This 'hand-coded' preference information can be used if no heuristic information from the unifier is available (cf. the previous section).
The following global expansion control

```
defcontrol :global ((:attribute-preference first rest head-dtr
                                           comp-dtrs front back)).
```

forces the algorithm to expand FIRST before REST, HEAD-DTR before COMP-DTRS, FRONT before BACK.

Implementation

Procedures *depth-first-expand* and *types-first-expand* are responsible for the expansion order of feature values. Before the feature list is visited, it is sorted according to the preference list if *control.attribute-preference* is nonempty.

[2]Because of the distributed disjunction representation in $\mathcal{UDiNe}$, disjunct order cannot be altered destructively. Instead, the ordering predicate is consulted for each disjunct.

7.12 Interactive Disjunct Selection

Syntax and Description

`(:ask-disj-preference {t | nil})`

Because failed disjuncts can disappear during expansion, it is difficult to uniquely identify disjuncts in feature structures. Therefore, the only feasible way to select disjuncts for expansion is to do it interactively. Of course, this is only useful for debugging or speeding up grammars, and only if the number of disjunctions is not too large. Therefore, it is preferable to specify interactive disjunct choice in local control, but not in global control.
If the `:ask-disj-preference` flag is set to `t`, the expansion algorithm interactively asks for the order in which disjunction alternatives are to be expanded.
As one can see from the sample output, there are several possibilities to continue if a disjunction node is reached.

```
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 for continuation without asking again in G) ? _
```

Implementation

As is the case for `:attribute-preference`, the order in which disjuncts are visited is determined in procedures *depth-first-expand* and *types-first-expand*. A function is called to manage the dialog if *control.ask-disj-preference* is true.

7.13 Printing Control Information

The *TDL* statement

`print-control` { *type* | *instance* | `:global` }
[`:index` *index*] [`:environment` {`:type` | `:instance`}] .

takes the same optional arguments as `defcontrol`. It prints the control information defined by `defcontrol` in an internal format with path patterns replaced by a special syntax. It can

be used anywhere in a grammar file or interactively to show the current status of control specification.

7.14 How to Stop Recursion

Type expansion with recursive type definitions is undecidable in general, i.e., there is no complete algorithm that halts on arbitrary input (type definitions) and decides whether a description is satisfiable or not.
However, there are several ways to stop infinite expansion which we will discuss now briefly. All of them except the first require defining appropriate control information so that it is clear how to formulate them in $\mathcal{TDL}$.
The first method is part of the expansion algorithm. If a recursive type occurs in a typed feature structure that is to be expanded, and this type has already been expanded at a subpath, and no features or other types are specified locally at this node, then this type will be postponed, since it would expand forever (this is called lazy expansion, cf. Section 6.5.3). An example of a recursion that stops for this reason is the recursive version of the **list** type. A counterexample, i.e., a type that will not stop without a finite input (using the default resolvedness predicate `always-false` and no delay pattern), is Aït-Kaci's *append* type. Expanding `append` with finite input will stop, of course (cf. the examples at the end of Chapter 6.
The second way is brute force: One can set the `:maxdepth` slot to cut off expansion at a suitable path depth. The third method is to define `:delay` patterns or to select the `:expand-only` mode with appropriate types and paths.
The fourth method is known from Prolog and works for some kinds of recursive types. One uses the `:attribute-preference` list to direct expansion into a feature branch that ensures termination. Finally, one can define an appropriate `:resolved-predicate` that is suitable for the application.

7.15 Global Variables

The following global variables contain the default values for global control that are used if not specified otherwise in `defcontrol`.

- `*EXPAND-FUNCTION*` (default value: `'Depth-First-Expand`)
- `*RESOLVED-PREDICATE*` (default value: `'Always-False`)
- `*MAXDEPTH*` (default value: `NIL`)
- `*IGNORE-GLOBAL-CONTROL*` (default value: `NIL`)

- `*ASK-DISJ-PREFERENCE*` (default value: `NIL`)
- `*ATTRIBUTE-PREFERENCE*` (default value: `()`)
- `*USE-DISJ-HEURISTICS*` (default value: `NIL`)
- `*USE-CONJ-HEURISTICS*` (default value: `NIL`)

Other global variables that influence the expansion algorithm (but cannot be changed by defcontrol) are:

- `*SIMPLIFY-FS-P*` (default value: `t`)
  If set to `t`, *UDiNe*'s *simplify-fs!* function removes `*fail*` items (failed disjunction alternatives) from a feature structure. This has to be done explicitly after unification because distributed disjunctions have a fixed number of disjuncts that can not be altered until unification has terminated. *simplify-fs!* is called after expansion in *expand-fs*, *expand-instance*, and *expand-type*.
- `*MAKE-DNF-P*` (default value: `nil`)
  If set to `t`, *UDiNe*'s *make-dnf* function translates all distributed disjunctions in a typed feature structure into disjunctive normal form (DNF). *make-dnf* is called after expansion in *expand-fs*, *expand-instance*, and *expand-type*. The flag is *UDiNe*-specific and only makes sense in cases where *simplify-fs!* could not simplify a structure completely (an example of such a structure is the *less* type from [Krieger & Schäfer 94b, page 42]). Because DNF may blow up a feature structure exponentially in the worst case, we do not recommend setting it to `t` unless *simplify-fs!* did not succeed.
- `*VERBOSE-EXPANSION-P*` (default value: `nil`)
  If `t`, type expansion is verbose, i.e., each call to *expand-type* is documented, as well as the current path and detection of recursive types. Moreover, the feature structures obtained at the end of each expansion pass in *expand-tfs* are printed using *UDiNe*'s ASCII representation (via function *print-fs*).
- `*LATEX-STREAM*` (default value: `nil`)
  If `*VERBOSE-EXPANSION-P*` is `t` and `*LATEX-STREAM*` has a non-`nil` value which is assumed to be a Common Lisp stream, then LaTeX code that protocols the expansion computation is written to that stream. Examples are the expansion traces at the end of Chapter 6. There exists a function *latex-expand-type* that takes the same parameters as *expand-type*, automatically creates a LaTeX file and writes the protocol stream of type expansion (to which `*LATEX-STREAM*` is temporarily bound to) to it.
- `*USE-SKELETON-P*` (default-value: `nil`)
  If `t`, the function *expand-type* returns the unexpanded skeleton of a type instead of

a memoized expanded prototype. The same result can also achieved by setting the prototype name of `:expand` slots in global control to `:skeleton` for all paths and all types (see below). `*USE-SKELETON-P*` has only been introduced to be able to correctly compare memoized vs. unmemoized expansion (cf. Figure 5.2 on page 51). As it turned out, the run time difference is minimal.

7.16 Statistics

Statistics Module

The *TDL* system can be compiled with an optional statistics module (by `(PUSHNEW :STATISTICS *FEATURES*)`) to investigate and compare grammars or different control strategies (cf. the table of Figure 5.2).
The expansion module has been augmented with the following functions:

`print-expand-statistics` [`:domain` *domain*] [`:stream` *stream*] .

prints expansion statistics after types or instances have been expanded. If *domain* is not specified, the current type domain is assumed. *stream* is the output stream (default: `t` for standard output).
Example:

```
Expand Statistics in domain DISCO:
----------------------------------
 852 yes *CONS*
 316 yes CAT-TYPE
 269 yes *DIFF-LIST*
 243 yes MORPH-TYPE
 208 yes ATOMIC-WFF-TYPE
 202 yes RP-TYPE
 146 yes CONJ-WFF-TYPE
 120 yes VAR-TYPE
  63 yes INDEXED-WFF
  59 yes NMORPH-HEAD
  53 yes SUBWFF-INST-SHARE-VAR
  53 yes TERM-TYPE
  51 yes SEMANTICS-TYPE
   ⋮
```

The first column contains the number of calls to *expand-type*, i.e., the number of prototypes and skeletons that have been returned. The second column indicates whether the default prototype is fully expanded (yes) or not (no). The last column contains the name of the type.

`reset-expand-statistics` [`:domain` *domain*].

resets expansion statistics. All expansion-specific statistics are set to zero. If *domain* is not specified, the current type domain is assumed.

Size of Feature Structures

To determine the size of feature structures, or of all avm types or instances, the universal function `count-nodes` has been defined.

`count-nodes` {*type* | *instance* | `:all`}
[`:table` {`:avms` | `:instances`}]
[`:expand-p` {`t` | `nil`}]
[`:verbose` {`t` | `nil`}]
[`:domain` *domain*]
[`:index` *index*]
[`:stream` *stream*]
[`:filename` {`nil` | *filename*}].

counts the number of nodes in an avm type or instance with the specified index (default is `nil` for types and `0` for instances). Instead of a name, the `:all` keyword can be specified to count all nodes in all instances or types with *index*. In this case, `:verbose t` will output the number of nodes for each type or instance. Otherwise, only the total will be printed.

The `:filename` or `:stream` argument can be used to redirect the output to a file or a stream (default: standard output). `:expand-p t` will expand structures before counting if necessary (default is `nil`). When called from Lisp, the function returns 9 values (integers) in the order as below. Here is an example output:

```
Total number of nodes in all instances:
# of conj avm nodes: 13868
# of atomic nodes:   5564
# of sortal nodes:   3697
# of attributes:     24717
# of disj nodes:     644
# of disj elements:  1606
# of fail nodes:     0
# of undef nodes:    0
# of shared nodes:   2763
total # of nodes:    23773
```

Counting the Number of Unifications

TDL's global variable `*COUNT-UNIFICATIONS*` can be used to count the number of unifications, e.g.,

```
set-switch *COUNT-UNIFICATIONS* 0.
expand-all-instances.
print-switch *COUNT-UNIFICATIONS*.
```

Chapter 8

Nonmonotonicity and Single Link Overwriting

8.1 Introduction and Motivation

Unification as defined in Chapter 3 is a monotonic operation on feature structures. Information can be added (refined), but never be withdrawn or revised. However, there are good reasons to admit nonmonotonicity. The first is a philosophical or aesthetic one. When describing natural language, we often have to deal with exceptions, especially in the lexicon, e.g., morphology of irregular verbs. If we want to encode exceptions in a monotonic fashion, we must introduce a feature (or a type) that not only contains the exceptions in, say, one percent of all items, but also repeats the default value in the remaining 99 percent as well. Because many such features can cross non-orthogonally, feature structures become complex and ugly, and errors are likely to occur within encoding and maintenance of the lexicon.
The second reason is a more pragmatic one. Because of the increased complexity if we are restricted to a purely monotonic formalism, feature structures are large, their representation is space-consuming, the type hierarchy blows up, and unification slows down.
Hence, *nonmonotonic extensions to feature structure unification* have been proposed (for an overview cf. [Bouma 92]). *Nonmonotonic inheritance* allows encoding of defaults and exceptions in type or template definitions in a more succinct fashion and thus help to reduce the number of types.
Because this thesis is about type expansion, we concentrate on the second part, *nonmonotonic inheritance*, or, more specifically, *single link overwriting* (SLO). A future version of $\mathcal{TDL}$ will also implement *nonmonotonic unification* [Krieger 95b] based on SLO, but this is beyond the scope of this thesis.
Single link overwriting allows types having a single supertype to be defined inconsistently with that supertype. This restriction avoids inheritance conflicts that can be caused by multiple inheritance, and makes nonmonotonic unification easier (cf. [Krieger 95b]). But

nonmonotonicity can also be used without a special kind of unification. One only has to guarantee that feature structures with default values will not unify with feature structures that have been overwritten nonmonotonically. Good candidates are lexical types (defaults) and lexicon entries (with possibly overwritten values).

8.2 Syntax of Nonmonotonic Definitions

To define nonmonotonic inheritance for lexicon entries, a new syntax is introduced. `!=` instead of `:=` for monotonic instance definitions indicates nonmonotonic overwriting. The different syntax ensures that the grammarian is aware of the special semantics. Although mainly designed for lexicon entries which are usually represented by instances in *TDL*, SLO nonmonotonicity can also be applied to avm types. The syntax for both kinds of definitions is the same.

identifier `!=` *nonmonotonic* [`where (` *constraint* {`,` *constraint*}* `)`] {`,` *option*}* `.`

where

nonmonotonic → *type* `&` `[` *overwrite-path* {`,` *overwrite-path*}* `]`

and

overwrite-path → *identifier* { `.` *identifier* }* *disjunction*

As for monotonic definitions, the left hand side of the definition (*identifier*) designates the type or instance name to be defined. *constraint*s are optional functional constraints and can also be used to separate coreference constraints syntactically from the rest of the feature structure definition (as in monotonic definitions).
type is the super type to be overwritten. In contrast to monotonic type or instance definitions, multiple inheritance is not allowed. The `[...]` syntax bears some similarity with the feature structure syntax for monotonic type or instance definitions. However, *overwrite-path* must be a true path, and *everything* following the path (*disjunction* is the start symbol for a complex *TDL* expression, cf. Appendix A), is interpreted as the overwrite value. In other words, the comma-separated elements between the brackets denote *sets* of overwrite paths and overwrite values, in contrast to nested feature-value pairs in monotonic definitions.
An example is the following definition.

```
strikepp != strikelex & [ CAT.MORPH.STEM "struck", PHON "struck" ].
```

`strikepp` (either a type or an instance) inherits from type `strikelex` and nonmonotonically overwrites the values at the specified paths with the atom `"struck"`.
Because of the structure sharing representation of coreferences, it is also possible to destroy coreferences by overwrite values.

8.3 Value Restrictions

To restrict the possible values that can be overwritten at a path, we augment typed feature structures by another type slot (in addition to *:delta* and *:expanded*), namely *:restriction*.
Restrictions in $\mathcal{TDL}$ are comparable to type restrictions in programming languages (typing variables). The main motivation for restrictions is to have an additional means for checking correctness of inheritance definitions because nonmonotonicity is powerful and dangerous. Value restrictions prevent feature structures from being overwritten by arbitrary values.
The additional restriction slots are subject to monotonic unification (type conjunction) like normal type slots.[1] A *:restriction* slot contains a possibly complex expression consisting only of type symbols joined with the operators `&`, `|`, and `^`, and negation (`~`).
Restriction types can be specified within avm type definitions as an optional suffix for attribute names, i.e., instead of a single attribute *attribute*, one can write

attribute : *restriction*

where

restriction → *conj-restriction* { { `|` | `^` } *conj-restriction* }*

conj-restriction → *basic-restriction* { `&` *basic-restriction* }*

basic-restriction → *type* | `~` *basic-restriction* | `(` *restriction* `)`

The restriction is checked if nonmonotonic overwriting takes place. A (continuable) error is signalled if the overwrite value violates the restriction type.
The default restriction type (if no restriction is specified) is $\top$, the top type of the hierarchy, i.e., the value can be overwritten by any value.
Example:

```
begin :declare.
  built-in: Integer.
end :declare.

begin :type.
  a := [ person_x : Integer, person_y : Integer ].
  b := a & [ person_x 1 | 2, person_y 1 | 2 ].
end :type.

begin :instance.
  c != b & [ person_x 3 ].
```

[1] Note that the restriction facility is only available if the $\mathcal{TDL}$ system has been compiled with the `#+TDL-Restriction` compiler switch.

```
  d != b & [ person_x "three" ].
end :instance.
```

Expanding the prototype of instance `c` results in the following structure

$$\left[\begin{array}{l} c \\ \text{PERSON\_X } 3 \\ \text{PERSON\_Y } \left\{\begin{array}{l}1\\2\end{array}\right\} \end{array}\right]$$

If we expand instance `d`, an error is signalled:

```
Error: Restriction INTEGER is inconsistent with overwrite value
       (:ATOM "three") under path PERSON_X in D
Restart actions (select using :continue):
 0: Continue; overwrite restriction anyway.
```

8.4 Implementation

The implementation of the restriction slot is straightforward. $\mathcal{TDL}$'s type infos are augmented by the *:restriction* slot as mentioned above. Additional syntax rules for $\mathcal{TDL}$'s ZEBU grammar (cf. Appendix A) are introduced to pass the restriction types to $\mathcal{TDL}$'s type infos. When typed unification takes place, not only are the values 'unified' (type conjunction, cf. Chapter 3), but also the restriction types.

Overwriting is more tricky. In the current implementation, it is done after feature structure expansion in functions *expand-type* and procedure *expand-instance*. It is necessary that the prototype has been fully expanded before overwriting to avoid later interaction with unification of feature structures that intersect with the overwrite paths.

Then, the values at the overwrite paths are destructively eliminated, and the new values are unified with the 'amputated' structure. Because of the structure sharing representation of coreferences in $\mathcal{UDiNe}$, this solution has the nice advantage that although the coreferences in the paths that have been overwritten are destroyed, their counterparts outside are left untouched.

Problems can occur if nonmonotonically overwritten feature structures are not fully expanded, caused by postponement of types at subpaths of the overwrite paths. In this case, a warning is signalled. The only proper alternative solution would be to store the overwrite value within the feature structures themselves (which is not allowed in the current implementation of $\mathcal{UDiNe}$), and only overwrite if the (sub)structure is fully expanded by adding a kind of *after method* to feature structure expansion. Currently, overwrite paths and values are stored together with the prototype feature structure of a type or instance.

Furthermore, warnings are given if the overwrite path does not exist in the feature structure to be overwritten. In this case, the missing path is generated anyway and the overwrite value is unified monotonically. As mentioned above, a continuable error is signalled if the overwrite value is incompatible with the restriction type.

Chapter 9

Appropriateness and Well-Typedness

9.1 Introduction and Motivation

The last task of this thesis is to add appropriatess, well-typedness and total well-typedness to the $\mathcal{TDL}$ formalism. These three properties impose restrictions on the typed feature structures and type systems we have defined in Chapter 3. *Appropriateness* is a relation between types and features. It states that only those features can occur in a feature structure of a certain type that are defined for it, namely its *appropriate* features. Correspondingly, only some types are *appropriate* for a feature. Moreover, appropriateness also restricts the kind of values a feature may have. *Well-typedness* and *total well-typedness* are properties of typed feature structures that are based on appropriateness (formal definitions follow).
The notion of *appropriateness* was first introduced informally by [Pollard & Sag 87]. Their main motivation was to forbid typed feature structures to bear features that do not belong to their types, e.g., a structure of type *sign* may have a PHONOLOGY attribute, but a feature structure of type *category* must not.
Because from the viewpoint of satisfiability of feature structure unification, there is no difference whether *sign* has a feature PHONOLOGY with value $\top$ or no such feature at all, [1] it is necessary to impose additional appropriateness conditions on feature structures and type systems to preclude these cases.
There are good reasons to require grammars and feature structure formalisms to meet the appropriateness conditions.

- *Debugging and maintainability*
  Grammars that are to meet the appropriateness conditions can be checked by the compiler. More conceptual or typographical errors are likely to be detected because of the

[1] although their set-theoretical semantics differ, cf. Section 3.2

additional restriction.

- *Upper bounds for feature arity*
  If a feature structure formalism is restricted by appropriateness, the number of features that can occur is limited for every type. This property can be exploited to efficiently represent feature structures (e.g., by fixed-size arrays for compilation or by Prolog terms). However, this also requires that the type system is closed, in contrast to the (optionally) open type world of $\mathcal{TDL}$.

- *Type inference*
  Appropriateness is indispensable for a feature structure formalism with a type inference mechanism. Otherwise, another source of termination and efficiency problems arises because of the indeterminacy to which type a feature belongs.

- *Portability and intertranslatability of NL grammars*
  Many implemented feature structure formalisms based on Prolog require feature structures to meet the appropriateness conditions, or, even more restrictively, to be totally well-typed. In this case, a grammar written for a more open formalism (like $\mathcal{TDL}$ without appropriateness check) may not work on the appropriateness formalism without major changes. [Rupp & Johnson 94] discuss portability and related problems for NL grammars.

- *Object orientation*
  [Cardelli & Wegner 85] define three criteria that must be fulfilled by a programming language to be considered called 'object-oriented'. *Strong typing* requires every object to be typed. *Data abstraction* allows reference to complex objects by their name without knowing about their internal structure. Finally, *inheritance-based polymorphism* ensures that methods defined for a type are also applicable for its subtype. Appropriateness is the basis for these three criteria. If they are fulfilled, methods and techniques developed for the implementation of object-oriented programming languages can be used for feature structure formalisms as well.

9.2 Definitions

It is due to Bob Carpenter that precise definitions for appropriateness and well-typedness in feature structure formalisms exist [Carpenter 92, Chapter 6].
We translate and extend his definitions to fit into our formalization of $\mathcal{TDL}$ (Chapter 3).

Definition 15 Appropriateness
A $\mathcal{TDL}$ type system meets the appropriateness condition iff $\forall f \in \mathcal{F} : \exists \tau \in \mathcal{T}_a : \tau$ is the most general type such that $\Delta(\tau) = \langle \sigma, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n] \rangle$ and $f = f_i$ for some $i \in \{1, \ldots, n\}$.

Then the appropriateness specification is a partial function $Approp : \mathcal{F} \times \mathcal{T}_a \mapsto (\mathcal{T} \cup \mathcal{A})$ and $Approp(f, \tau) := type\text{-}of(\theta_i)$.
If $Approp(f, \tau)$ is defined and $\sigma \preceq \tau$, then $Approp(f, \sigma)$ is defined and $Approp(f, \sigma) \preceq Approp(f, \tau)$ (closure).

The *Approp* function is defined on every feature for every avm type that has a feature (by definition or by inheritance) at the toplevel path.
Carpenter stipulates that there is exactly one most general type that introduces a feature. The reason is that this eliminates a non-determinism in type inference algorithms.[2] $\mathcal{TDL}$ is not so restrictive, and allows more than one type to introduce a feature (feature polymorphism). However, optional warnings can be printed if there is no unique type that introduces a feature.

Definition 16 Well-typedness
A conjunctive feature structure $\theta = \langle \tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n] \rangle$ is *well-typed*, iff θ_i are well-typed $\forall i \in \{1, \ldots, n\}$ *and*

$$\begin{cases} \text{if } \tau \in \mathcal{T}_a: & Approp(f_i, \tau) \text{ is defined and } type\text{-}of(\theta_i) \preceq Approp(f_i, \tau) \\ & \forall i \in \{1, \ldots, n\} \\ \text{if } \tau = \tau_1 \wedge \cdots \wedge \tau_m: & Approp(f_i, \tau_j) \text{ is defined for some } \tau_j,\ 1 \leq j \leq m, \text{ and} \\ & type\text{-}of(\theta_i) \preceq Approp(f_i, \tau_j)\ \forall i \in \{1, \ldots, n\} \end{cases}$$

A disjunctive feature structure $\theta = \{\theta_1, \ldots, \theta_n\}$ is *well-typed*, iff θ_i are well-typed $\forall i \in \{1, \ldots, n\}$.

Informally, a feature structure is well-typed, if every feature occurring in it is licensed by the *Approp* function and if its value is equal to or more specific than its *Approp* value.

Definition 17 Total well-typedness
A conjunctive feature structure $\theta = \langle \tau, [f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n] \rangle$ is *totally well-typed*, iff θ_i are totally well-typed $\forall i \in \{1, \ldots, n\}$ *and*

$$\begin{cases} \text{if } \tau \in \mathcal{T}_a: & \text{if } Approp(f_i, \tau) \text{ is defined then } f = f_i \text{ for some } i \in \{1, \ldots, n\} \\ \text{if } \tau = \tau_1 \wedge \cdots \wedge \tau_m: & \forall \tau_j,\ 1 \leq j \leq m\text{: if } Approp(f_i, \tau_j) \text{ is defined then } f = f_i \\ & \text{for some } i \in \{1, \ldots, n\} \end{cases}$$

A disjunctive feature structure $\theta = \{\theta_1, \ldots, \theta_n\}$ is *totally well-typed*, iff θ_i are totally well-typed $\forall i \in \{1, \ldots, n\}$.

In other words, a feature structure is totally well-typed if it is well-typed and if every feature that is appropriate for each type occurring in the feature structure is explicit in the structure. Operationally, total well-typedness corresponds to *full type expansion* of well-typed feature structures. Note that Carpenter's total well-typedness is not well-defined for recursive typed feature structures, because the definition would 'loop' in recursive types.

[2] E.g., if both type *a* and type *b* introduce feature F, then the inference algorithm cannot immediately determine to which type of both a feature structure belongs.

9.3 Implementation of the Appropriateness Function

As we mentioned in Chapter 3, the *Approp* function is computed incrementally at type definition time. For every new feature that occurs at the toplevel of a avm type definition, an entry is added to the *Approp* table.
Instead of storing the complete two-place *Approp* function $\mathcal{F} \times \mathcal{T}_a \mapsto (\mathcal{T} \cup \mathcal{A})$ for each type and feature as defined by Carpenter, we only store a total one-place function $\mathcal{F} \mapsto (\mathcal{T}_a \times (\mathcal{T} \cup \mathcal{A}))^+$ that contains only one entry per feature, namely the most general avm type(s) plus its or their admissible feature values. The admissible values for all subtypes can be inferred quickly by a lookup in the prototypical feature structure of the requested type (its default prototype with index `nil`). To infer the actual value types for a feature, the default prototypes of all types must be expanded at least at path depth 0 (i.e., its supertypes). Expansion is done automatically (i.e., if necessary) by the procedure that computes or updates the appropriateness table.
The main advantage of storing only the most general intro-type for every feature is that this representation needs less space.

Procedure `compute-approp` computes the appropriateness table according to the avm type definitions. Normally, `compute-approp` need not to be called by the user. This is done by the well-typedness checking procedures if necessary (e.g., if new types have been added).
However, if one needs to know whether there is more than one type that introduces a feature, it is possible top call `compute-approp` by hand. The optional keyword `:warn-if-not-unique` controls whether a warning is printed (`t`) or not (`nil`), if a feature is not defined uniquely; default is `nil`. Syntax:

`compute-approp` [`:domain` *domain*] [`:warn-if-not-unique` {`t` | `nil`}].

`compute-approp` expands the prototypes of all avm types at path depth 0 (if necessary), and inserts for each feature the most general type(s) that introduce(s) the feature into the *Approp* table.

Procedure

`print-approp` [`:domain` *domain*].

prints the current appropriateness table of a $\mathcal{TDL}$ type domain. The first column contains the feature name, the second column contains a list of dotted pairs. Each dotted pair consists of the most general type that introduces the feature and the admissible value type.

```
Feature       ((Intro-Type . Value-Type)*)
-----------------------------------------
        HOUR  ((TIME-VALUE . *TOP*))
     NON-LOC  ((NON-LOCAL . NON-LOCAL-TYPE))
```

```
    SUBJ-SC  ((SUBJ-SUBCAT-TYPE . *TOP*))
  SEM-MOOD  ((QUESTION-SEMANTICS . SYMBOL))
    SUBCAT  ((SUBCAT-TYPE . *TOP*))
FILLER-DTR  ((FILLER-DTR-TYPE . MAX-SIGN-TYPE))
    SEMINF  ((SPAC-4-TYPE . (:AND CLOSED-SEM-LISTS EMPTY-QUANT-TYPE))
      LAST  ((*DIFF-LIST* . *TOP*)
             (CONT2QUANT . *NULL*)
             (CLOSED-DIFF-LIST . *NULL*))
      LIST  ((*DIFF-LIST* . *TOP*))
     LISZT  ((MRS-TYPE . *LIST*))
   HAENDEL  ((MRS-TYPE . MRS-HANDLE))

        ⋮
```

9.4 Checking Well-Typedness

Different kinds of well-typedness checks are provided, at type or instance definition time (for debugging purposes), at unification time (to rule out non-well-typed feature structures during unification), and explicitly on typed feature structures and prototypes.
In every case, the global variable `*VERBOSE-WELLTYPEDNESS-CHECK-P*` controls whether the source of non-well-typedness is printed (`t`) or not (`nil`). The warnings that are printed if the check is verbose are:

- "`Feature ... is not welltyped under path ... in` ⟨*instance or type name*⟩" if the feature value is more general than or incompatible with the appropriateness type.
- "`Feature ... under path ... in` ⟨*instance or type name*⟩ `has no appropriateness specification in domain ...`" if the feature is not licensed by an appropriateness definition, i.e., there is no type that introduces the feature at the toplevel.

Independently from the value of the global switch, a warning is printed if an undefined type occurs in a feature structure being checked.

9.4.1 Well-Typedness Checks at Definition Time

The global variable `*CHECK-WELLTYPEDNESS-P*` controls whether the check is done at type and instance definition time (`t`) or not (`nil`). If `t`, full expansion of the default prototype of the structure being defined takes place before the check. Therefore, the grammar writer has to be careful if recursive types are involved. The check is done by calling *check-welltypedness-node* (see below) on the expanded prototype feature structure.

9.4.2 Well-Typedness Checks at Unification Time

Well-typedness checks at unification time can be useful if feature structures are generated by mechanisms different from $\mathcal{TDL}$'s avm type or instance definition facility, e.g., by an external NL generator. In an open type world, if only well-typed instances and types are used in a grammar, well-typedness checks at unification time are superfluous.

The global variable `*CHECK-UNIFICATION-WELLTYPEDNESS-P*` controls whether the argument nodes of typed unification are checked to be well-typed (`t`) or not (`nil`) in function *unify-types* (cf. page 38).

If `*CHECK-UNIFICATION-WELLTYPEDNESS-P*` has value `t`, the global variable `*RETURN-FAIL-IF-NOT-WELLTYPED-P*` determines whether a unification failure is triggered if one of the unified nodes is not well-typed (`t`). If the value is `nil`, a warning is printed instead.

9.4.3 Explicit Well-Typedness Checks for Feature Structures

Finally, function *check-well-typedness-node* checks well-typedness of a feature structure node.

`check-welltypedness-node` *node* [`:domain` *domain*]
[`:ctrl-obj` *ctrl-obj*]
[`:verbose` {`t` | `nil`}] .

where *node* is a feature structure node, *domain* is the name of a $\mathcal{TDL}$ type domain, *ctrl-obj* is a $\mathcal{UDiNe}$ control object, and the `:verbose` flag sets verbosity (default value: value of `*VERBOSE-WELLTYPEDNESS-CHECK-P*`).

The function recursively traverses the feature structure in depth-first order and compares the types and feature at every node with the values in the *Approp* table.

`check-welltypedness-node` returns `t` if the feature structure is well-typed and `nil` otherwise. As a side effect, warnings are printed to indicate non-well-typed features and their paths according to the value of the `:verbose` keyword.

The function

`check-welltypedness` [*type* | *instance* | `:all` [`:instances` | `:avms`
[`:domain` *domain*] [`:index` *index*] [`:verbose` {`t` | `nil`}]]].

provides a well-typedness check for a single avm type or instance as well as for all types or instances with the specified *index* (default is `:all :instances`) The function calls function *check-welltypedness-node* with the specified type or instance prototype and returns `t` if it is well-typed and `nil` otherwise. If `:all` is specified as the first argument, the return value is undefined.

9.5 Total Well-Typedness Checks

There is no special function for checking total well-typedness because total well-typedness directly follows from full expansion of well-typed feature structures:

total well-typedness = well-typedness + type expansion

A simple check of the *expanded* flag at the root node of a well-typed feature structure suffices to decide whether it is totally well-typed or not.

Chapter 10

Comparison to Related Systems

In this chapter, we compare the *TDL* system to other implemented grammar formalisms. Because there is a steadily growing number of feature structure formalisms, we limit the comparison to widespread formalisms with type hierarchies that implement at least two of the topics described in this thesis.[1]

Moreover, we concentrate on formalisms that have been designed specifically for the development of unification-based grammars such as HPSG (and have proven that HPSG grammars work). Although LOGIN [Aït-Kaci & Nasr 86] and LIFE [Aït-Kaci et al. 94] have influenced the design and implementation of grammar formalisms for HPSG, they are not used for HPSG grammars, mainly because disjunction and negation and other important features are missing. Like Oz [Henz et al. 93], LIFE is rather a general purpose programming language. However, the next version of Oz will implement open features and hence might be better suited for natural language representation.

A new system developed at the University of Tübingen, named TROLL, is not considered in our comparison because it makes very strong assumptions on the type and feature system that differ from the common interpretation of HPSG (cf. [Gerdemann & King 94]).

We compare the following systems: *TDL* (literature: cf. introduction), *TDL ExtraLight* [Krieger & Schäfer 93b], ALE [Carpenter & Penn 94], CUF [Dörre & Dorna 93], and TFS [Zajac 92]. The focus of the comparison lies, of course, in type expansion and related topics, as well as expressivity of the formalisms, well-typedness and nonmonotonicity.

All systems based on types have some kind of type expansion. ALE and *TDL ExtraLight* expand types fully at definition time. Therefore, they cannot handle recursive types. However, ALE provides recursion through a built-in bottom-up chart parser and through definite clauses, as most other systems without recursive types do, including CUF. Although the expressivity is the same, the disadvantage of this solution is that it differs from the framework proposed for HPSG. E.g., the Head Feature Principle cannot be formulated as a type defini-

[1] A more detailed overview of 16 systems can be found in [Backofen et al. 93]. [Manandhar 93] gives a thorough comparison of ALE, CUF, and TFS.

tion if recursive types are not supported. Allowing type expansion only at definition time is in general space consuming, thus unification and copying is expensive at run time.

TFS also expands types at definition time (type checking), but delays recursive types that can be expanded at run time.

Another strategy one might follow is to integrate type expansion into the typed unification process so that type expansion can take place at run time. This approach has been suggested for LIFE; it is also possible in *TDL*. Moreover, in *TDL* one can freely choose when type expansion takes place: at type definition time or at run time.

Feature structure memoization in *TDL* helps to reduce the number of unifications which is important for run time expansion, cf. Figure 5.2 on page 51. Partial expansion keeps the feature structures small and makes expansion of lexicon entries about 10 times faster. A system that employs postponement on demand at run time is CUF [Dörre & Dorna 93]. Laziness can be achieved here by specifying delay patterns as is familiar from Prolog. This means postponing the evaluation of a relation until the specified parameters are instantiated.

	TDL	*TDL ExtraLight*	ALE	TFS	CUF
Type expansion					
• at def./compile time	✓[1]	✓	✓	✓	✓
• at run time	✓[1]	-	-	[2]	-
• within unification	✓[3 1]	-	-	-	-
• recursive types	✓	-	-	✓	-
• partial expansion	✓	-	-	-	✓
• control, preferences	✓	-	-	-	[4]
• fs memoization	✓	-	-	-	-
Well-typedness	✓[1]	-	✓	-	-
Nonmonotonicity	✓	-	-	-	-
Open type world	✓[1]	✓	-	-	✓
Complex negated types	✓	-	-	-	✓
Disjunctive Type definitions	✓	-	-	✓	✓
Open feature arity	✓	✓	-	✓	✓
Feature polymorphism	✓	✓	-	✓	✓
Complex disj. values	✓	✓	-	-	-
Complex fs negation	✓	-	-	-	✓
Coreference variables	✓	✓	-	✓	-

Figure 10.1: *Comparison of typed feature structure formalisms.*

Currently, *TDL/UDiNe* is the only system that supports preferences to direct unification and speed up processing. [Brew 93] presents an approach for integrating preferences in CUF. Welltypedness checks are only performed in ALE and (optionally) in *TDL*. Nonmonotonic definitions, that are especially useful for lexicon specification, are unique in *TDL*. *TDL*, *TDL ExtraLight*, and CUF are the only systems that allow for an open type world (i.e., conjunction of types that have no common subtype is consistent). In *TDL*, one can also select a closed type world to ensure compatibility with grammars written for other formalisms.
Criteria like complex disjunctive feature values, complex feature negation, and coreferences can be used to estimate the expressivity of the feature constraint solver.
To sum up, *TDL* is more general than the other systems in that many features are optional or parameterized such as closed type world, well-typedness check, expansion time, and control. Therefore, it may also be used to develop or check grammars written for other systems, and can support intertranslatability and portability of natural language grammars (cf. [Rupp & Johnson 94]).

[1] optional

[2] recursive types only [Emele & Zajac 90]

[3] currently not implemented, but can be done by modifying the *TDL-UDiNe* unification interface

[4] not implemented; has been proposed by [Brew 93]

Chapter 11

Conclusion and Future Work

In this thesis, we have presented a new approach to type expansion. By considering type expansion a proper system module, instead of an implicit mechanism, the time for type expansion can be chosen freely during linguistic processing. It can also be integrated into unification.

Partial expansion at definition time helps to reduce space requirements, e.g., for lexicon entries. Incrementality is achieved through *expanded* flags in the feature structures that allow partially expanded structures and avoid redundant unifications. Moreover, the flags drastically reduce the search space for subsequent expansions. Memoization of expanded structures minimizes the number of unifications.

Lazy expansion of recursive types allows exploitation of the increased expressivity that recursive types admit. This allows grammars to be specified close to the definitions given in the HPSG books.

The expansion algorithm is parameterized to be either complete or always terminating. Declarative specification of control information can increase processing speed, e.g., by preference information (controlled linguistic deduction, cf. [Uszkoreit 91]).

Furthermore, nonmonotonicity has been included in the type expansion algorithm for type and instance definitions, e.g., for a more succinct lexicon specification with defaults and exceptions. Appropriateness conditions can be checked optionally to guarantee well-typedness of the feature structures (at definition or at run time). Full expansion of well-typed feature structures leads to total well-typedness (modulo recursive types).

Because many of the added features can be parameterized or are optional, $\mathcal{TDL}$ is more general than other formalisms, and can also be used to process grammars that have been developed for other formalisms, leaving aside different syntax that can be translated automatically in most cases (full Boolean logic for types and feature constraints).

In this thesis, we have laid the foundations for more sophisticated linguistic processing with typed feature structures. Various strategies can be tested. While the usefulness of memoization has already been proven for expansion of a grammar with (currently) 1800 types and 450

lexicon entries (speed up factor is up to 10 compared to a naive algorithm that unifies only the skeletons), other issues are challenging:

- *Expansion of lexicon entries at run time*
  The most promising application of controlled and delayed expansion is related to lexicon entries. The current NL applications of $\mathcal{TDL}$ still use fully expanded lexicon entries. For larger HPSG lexica, it is indispensable to postpone expansion of lexicon entries until they are required by the parser in order to save memory. A lexicon entry of the DISCO grammar has an average size of 100 complex nodes.

- *Postponement of non-filtering semantic parts of lexicon entries during parsing*
  Controlled expansion of lexical semantics can drastically improve parsing speed because the structures become smaller (i.e., copying and unification is faster). Expansion of semantics is only necessary if the syntactic part did not lead to failure. The sophisticated delay specification developed in this thesis can be used to only make explicit the filtering parts of semantic information; cf. [Diagne et al. 95] for first results.

- *Exploitation of increased expressivity by recursive definitions*
  The *append* relation (list concatenation), finite state automata (morphology), and phrase structure are examples for applications of recursive types that lead to more elegant grammar specifications.

- *Copy pools*
  The prototype and memoization techniques presented in this thesis can be pushed to the extreme, where (almost) all copying is done at compile time and within idle run time. This helps to speed up unification, e.g., for parsing. Heuristics can be obtained from training sessions to estimate the required number of copies per prototype.

- *Psycholinguistically motivated preferences*
  can be supported by training sessions and can be used to first process the most probable readings or rule out some others depending on parameters.

While these applications can be performed within the $\mathcal{TDL}$ formalism as implemented and presented in the thesis, some future extensions of the formalism are interesting:

- *Type expansion as an anytime module*
  Complex architectures for NL processing require modules that can be interrupted at any time, returning an incomplete but nevertheless useful result [Wahlster 93]. Such modules are able to continue processing with only a negligible overhead, instead of having been restarted from scratch. Type expansion can serve as an anytime module for linguistic processing. Since the representation of partially expanded feature structures trough expanded flags already supports incrementality, the integration of anytime behaviour into the expansion algorithm is straightforward.

- *Type inference*
  tries to infer the correct type of an untyped or partially typed feature structure. It can be seen as the inverse of type expansion that makes feature explicit from type definitions. Type inference is especially useful for natural language generation. Because the TDL language is very powerful, its expressivity must be limited if type inference is to be decidable, or approximations are computed to ensure termination. The memoization technique can also be used for an efficient inference algorithm.

- *Lazy attribute inheritance and integration of expansion into unification*
  Although we do not believe that this expansion technique is generally useful for natural language processing (because much more unifications take place compared to prototype memoization), it is elegant for lazy expansion of recursive types, and it would be nice to have it as an alternative expansion strategy.

- *Abstract interpretation, data flow analysis, call patterns, mode declarations*
  These techniques have been developed for more efficient processing of Turing-equivalent computations in logic programming languages (among others, cf. [Warren 92]). Although TDL becomes Turing-equivalent through the admission of recursive types, their processing is not always (space-) efficient. 'Junk slots' like the PATCH feature in the *append* type are necessary to pass and store arguments during computation and therefore blow up the size of the feature structures being processed. The techniques mentioned above help to reduce this overhead by compiling declarative grammars into more efficient ones.

The TDL system including the expansion mechanism presented in this thesis has been installed and successfully employed at several sites, e.g., DISCO and PARADICE projects at DFKI Saarbrücken, CSLI Stanford (Dan Flickinger, Ivan Sag), IBM Germany, Heidelberg (Tibor Kiss), ÖFAI Vienna (Harald Trost), PRACMA project at the Computer Science Department of Saarbrücken, IMS Stuttgart (Martin Emele), Simon Fraser University (Fred Popowich), GMD-IPSI, Darmstadt (Renate Henschel), grammar engineering course at the Computational Linguistics Department of Saarbrücken (Hans Uszkoreit, Stephan Oepen), and Brandeis University (James Pustejovsky).

Appendix A

Syntax of $\mathcal{TDL}$

Introduction

The $\mathcal{TDL}$ syntax is given in extended BNF (Backus-Naur Form). Terminal symbols (characters to be typed in) are printed in `typewriter` style. Nonterminal symbols are printed in *italic* style. The grammar starts with the *start* production. The following table explains the meanings of the metasymbols used in extended BNF.

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

$\mathcal{TDL}$ Main Constructors

start → {*block* | *statement*}*

block → `begin :control.` { *type-def* | *instance-def* | *start* }* `end :control.` |
`begin :declare.` { *declare* | *start* }* `end :declare.` |
`begin :domain` *domain*`.` {*start*}* `end :domain` *domain*`.` |
`begin :instance.` { *instance-def* | *start* }* `end :instance.` |
`begin :lisp.` {*Common-Lisp-Expression*}* `end :lisp.` |
`begin :template.` { *template-def* | *start* }* `end :template.` |
`begin :type.` { *type-def* | *start* }* `end :type.`

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* | nil })] |
[(:ask-disj-preference {t | nil})] |
[(:attribute-preference {*identifier*}*)] |
[(:use-conj-heuristics {t | nil})] |
[(:use-disj-heuristics {t | nil})] |
[(:expand-function {{depth | 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 | ...

Instance Definitions

instance-def → *instance avm-def* .

instance → *identifier*

Template Definitions

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

Declarations

declaration → *partition* | *incompatible* | *sort-def* | *built-in-def*

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

incompatible → nil = *type* {& *type*}$^+$.

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

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

Statements

(as far as of importance for type expansion and welltypedness)

statement → `check-welltypedness` [{*type* | *instance* | `:all`} [{`:instances` | `:avms`}
[`:domain` *domain*] [`:index` *index*] [`:verbose` {`t` | `nil`}]]]. |
`compute-approp` [`:domain` *domain*] [`:warn-if-not-unique` {`t` | `nil`}]. |
`defcontrol` {`:global` | *type* | *instance*} *expand-control* [`:index` *index*]. |
`expand-all-instances` {*expand-option*}* . |
`expand-all-types` {*expand-option*}* . |
`expand-instance` [*instance* [`:index` *integer*] {*expand-option*}*] . |
`expand-type` [*type* [`:index` *index*] {*expand-option*}*] . |
`print-approp` [`:domain` *domain*]. |
`print-control` { *type* | *instance* | `:global` } . |
`print-expand-statistics` [`:domain` *domain*] [`:stream` *stream*] . |
`print-recursive-sccs` [`:domain` *domain*] . |
`reset-all-instances` [*domain*] . |
`reset-all-protos` [*domain*] . |
`reset-expand-statistics` [`:domain` *domain*] . |
`reset-proto` [*type* [`:domain` *domain*] [`:index` *index*]] .

expand-option → `:domain` *domain* |
`:expand-control` *expand-control*

domain → `'`*identifier* | `:`*identifier* | `"`*identifier*`"`

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*}*`"`

Appendix B

Sample File

```
;;; -*- Mode: TDL -*-
;;; ---------------------------------------------------------------------------
;;; Parameterized Type Expansion in TDL. Some Examples.
;;; ---------------------------------------------------------------------------

defdomain "DEMO".  ;;; built-in types will be loaded automatically
begin :domain "DEMO".
  set-switch *WARN-IF-REDEFINE-TYPE* NIL.       ;;; switch off warnings
  set-switch *WARN-IF-TYPE-DOES-NOT-EXIST* NIL. ;;; dto
  set-switch *PRINT-SORTS-AS-ATOMS* T.          ;;; for fegramed/pgp
  set-switch *VERBOSE-EXPANSION-P* T.           ;;; verbose expansion
  set-switch *PRINT-SLOT-LIST* (CONS :DELTA *PRINT-SLOT-LIST*). ;;; show :delta
  set-switch *LABEL-SORT-LIST* '(FIRST REST LAST INPUT EDGE NEXT  ;;; for output
                                 WHOLE FRONT BACK A B C D X Y Z). ;;; only
  fegramed.     ;;; start Feature Editor
  set-switch FEGRAMED:*DEF-FILENAME* "/tmp/".
  grapher.      ;;; start Type Grapher
  begin :type.

;;; ---------------------------------------------------------------------------
;;; Parameterized Expansion: expand-only mode for type d
;;; ---------------------------------------------------------------------------

    a1 := [a 1].
    b2 := [b 2].
    c  := [c  ].
    zz := [z  ].
    d  := zz & [x a1,
                y b2,
                z c & [c 3]].
```

```
    defcontrol d ((:expand-only ((c 1 EQ) z.*) ((a1) x))
                  (:attribute-preference z x y)).
    expand-type 'd.

;;; ---------------------------------------------------------------------------
;;; Parameterized Expansion with delay and prototype index 1
;;; ---------------------------------------------------------------------------

    defcontrol d ((:delay (c *))
                  (:attribute-preference z x y)
                  (:expand-function types-first-expand))
              :index 1.
    expand-type 'd :index 1.

;;; ---------------------------------------------------------------------------
;;; Interactively ask for disjunct order
;;; ---------------------------------------------------------------------------

    inter := [disj  a1 | b2 | c,
              disj2 b2 | d  | 42].

    defcontrol inter ((:ask-disj-preference t)).
    expand-type 'inter.

;;; ---------------------------------------------------------------------------
;;; Negation `a la [Smolka 89]
;;; ---------------------------------------------------------------------------

    xn := [a 1, b 2, c 3].
    nx := [n ~xn].

    expand-type 'nx.

;;; ---------------------------------------------------------------------------
;;; Nonmonotonicity (single link overwriting)
;;; ---------------------------------------------------------------------------

    a := [ person_x:INTEGER,
           person_y:INTEGER ].

    b := a & [ person_x  1 | 2 ].

    px3 != b & [ person_x 3 ].
```

```
    expand-type 'px3.

    pxs != b & [ person_y "string" ].

    expand-type 'pxs.

;;; ---------------------------------------------------------------------------
;;; Welltypedness Check for an instance at definition time:
;;; ---------------------------------------------------------------------------

    set-switch *VERBOSE-EXPANSION-P* NIL.
    set-switch *CHECK-WELLTYPEDNESS-P* T.
    ;;; now expand-instance will be done automatically!
    begin :instance.
       zi := zz & [z c & [c 3],
                   x a1        ].
    end :instance.
    set-switch *CHECK-WELLTYPEDNESS-P* NIL.

;;; ---------------------------------------------------------------------------
;;; Automata - Basic Configurations
;;; ---------------------------------------------------------------------------

  begin :declare.
    sort: *undef*.
  end :declare.

  proto-config := *avm* &
    [EDGE, NEXT, INPUT].

  non-final-config := proto-config &
    [EDGE  #first,
     NEXT.INPUT  #rest,
     INPUT  <#first . #rest>].

  final-config := proto-config &
    [INPUT *null*,
     EDGE  *undef*,
     NEXT  *undef*].

  config := non-final-config | final-config.

;;; ---------------------------------------------------------------------------
;;; consider the two regular expressions U=(a+b)^*c and X=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].

  Y := non-final-config &
         [EDGE  'b,
          NEXT  Y | Z].

  Z := config &
         [EDGE  %covary( 'c, *undef*),
          NEXT  %covary(  Z, *undef*)].

;;; -----------------------------------------------------------------------
;;; now we intersect the two automata U and X  --> a(b^+)c
;;; -----------------------------------------------------------------------

  UX := U & X.

  test1 := UX & [INPUT <'a,'b,'c>].      ;;; accepted
  test2 := UX & [INPUT <'a,'b,'b,'c>].   ;;; accepted
  test3 := UX & [INPUT <'b,'c>].         ;;; is inconsistent
  test4 := UX & [INPUT <'a,'b,'c,'d>].   ;;; is inconsistent

  set-switch *VERBOSE-EXPANSION-P* NIL.      ;;; silent expansion
  set-switch *PRINT-SLOT-LIST* (REMOVE :DELTA *print-slot-list*).
                                             ;;; don't print delta list
  expand-type 'test1.
  expand-type 'test2.
  expand-type 'test3.
  expand-type 'test4.

;;; -----------------------------------------------------------------------
;;; Ait-Kaci's version of APPEND
;;; -----------------------------------------------------------------------

  set-switch *VERBOSE-EXPANSION-P* T.
```

```
  *cons* := *avm* & [FIRST,REST *list*].  ;;; redefine *LIST* recursively

  append0 := *avm* & [FRONT *null*,
                      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.

  r:=append & [FRONT <'a,'b>,
               BACK  <'c,'d>].    ;;; result will be in WHOLE

  expand-type 'r.

  set-switch *VERBOSE-EXPANSION-P* NIL.

  q:=append & [WHOLE <'a,'b,'c>].   ;;; compute possible inputs (4)
  expand-type 'q.

;;; ---------------------------------------------------------------------------
;;; Print Recursive Types (SCCs)
;;; ---------------------------------------------------------------------------

  message "~%~%List of recursive sccs:".
  print-recursive-sccs.

;;; ---------------------------------------------------------------------------
;;; Print Appropriateness table
;;; ---------------------------------------------------------------------------

  message "~%Computing appropriateness table~%".
  compute-approp :warn-if-not-unique T.
  print-approp.
```

Bibliography

[Abelson & Sussman 85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Aït-Kaci & Lincoln 88] Hassan Aït-Kaci and Patrick Lincoln. *LIFE—A Natural Language for Natural Language.* Technical Report ACA-ST-074-88, MCC, Austin, TX, 1988.

[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. 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.* Research Report 32, Digital Equipment Corporation, DEC Paris Research Laboratory, France, May 1993. Also in Proceedings of the International Symposium on Logic Programming, Vancouver, BC, Canada, October 1993, edited by Dale Miller, published by MIT Press, Cambridge, MA.

[Aït-Kaci et al. 94] Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. *The Wild LIFE Handbook (prepublication edition).* DEC Paris Research Laboratory, France, March 1994.

[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 1993 International Symposium on Logic Programming, 1993.

[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, Germany, March 1993. Deutsches Forschungszentrum für Künstliche Intelligenz. DFKI Research Report D-93-27.

[Bouma 92] Gosse Bouma. *Feature Structures and Nonmonotonicity.* Computational Linguistics, 18(2):183–203, 1992.

[Brachman & Schmolze 85] Ronald J. Brachman and James G. Schmolze. *An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science, 9:171–216, 1985.

[Bresnan 82] Joan Bresnan (ed.). *The Mental Representation of Grammatical Relations.* Cambridge, Mass.: MIT Press, 1982.

[Brew 93] Chris Brew. *Adding Preferences to CUF.* In: [Dörre 93], pp. 54–69. DYANA-2 Deliverable R1.2.A.

[Cardelli & Wegner 85] Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 17(4):471–522, 1985.

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

[Carpenter 92] Bob Carpenter. *The Logic of Typed Feature Structures*, volume 32: Cambridge Tracts in Theoretical Computer Science. Cambridge, UK: Cambridge University Press, 1992.

[Chomsky 81] Noam A. Chomsky. *Lectures on Government and Binding.* Dordrecht: Foris, 1981.

[COLING 94] *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, Kyoto, Japan, 1994.

[Diagne et al. 95] Abdel Kader Diagne, Walter Kasper, and Hans-Ulrich Krieger. *Distributed Parsing With HPSG Grammars.* Technical report, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1995.

[Dörre & Dorna 93] Jochen Dörre and Michael Dorna. *CUF—A Formalism for Linguistic Knowledge Representation.* In: [Dörre 93], pp. 1–22. DYANA-2 Deliverable R1.2.A.

[Dörre 93] Jochen Dörre (ed.). *Computational Aspects of Constraint-Based Linguistic Description I.* ILLC/Department of Philosophy, University of Amsterdam, 1993. DYANA-2 Deliverable R1.2.A.

[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, Helsinki, Finland, 1990.

[Emele 91] Martin Emele. *Unification with Lazy Non-Redundant Copying.* In: Proceedings of the 29^{th} Annual Meeting of the Association for Computational Linguistics (ACL), pp. 323–330, Berkeley, CA, 1991.

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

[Gerdemann & King 94] Dale Gerdemann and Paul John King. *The Correct and Efficient Implementation of Appropriateness Specifications for Typed Feature Structures.* In: [COLING 94].

[Henz et al. 93] Martin Henz, Gert Smolka, and Jörg Würtz. *Oz—A Programming Language for Multi-Agent Systems.* In: Proceedings of the 13th International Joint Conference on Artificial Intelligence IJCAI-93, Vol. 1, pp. 404–409, 1993.

[Jakobson et al. 51] R. Jakobson, G. Fant, and M. Halle. *Preliminaries to speech analysis: The distinctive features and their correlates.* Cambridge, MA, 1951.

[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.

[Kaplan & Zaenen 88] Ronald M. Kaplan and Annie Zaenen. *Long-distance Dependencies, Constituent Structure, and Functional Uncertainty.* In: M. Baltin and A. Kroch (eds.), Alternative Conceptions of Phrase Structure. Chicago: University of Chicago Press, 1988.

[Karttunen 84] Lauri Karttunen. *Features and Values.* In: Proceedings of the 10^{th} 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 24^{th} Annual Meeting of the Association for Computational Linguistics (ACL), pp. 257–266, New York, NY, 1986. Columbia University.

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

[Kay 84] M. Kay. *Functional Unification Grammar: a formalism for machine translation.* In: Proceedings of the 10^{th} International Conference on Computational Linguistics and the 22^{nd} Annual Meeting of the Association for Computational Linguistics, pp. 75–78, Stanford, Ca., July 2–6 1984.

[Kogure 90] Kiyoshi Kogure. *Strategic Lazy Incremental Copy Graph Unification.* In: Proceedings of the 13^{th} International Conference on Computational Linguistics, COLING-90, pp. 223–228, Helsinki, Finland, 1990.

[Krieger & Schäfer 93a] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL – A Type Description Language for Unification-Based Grammars.* In: Proceedings Neuere Entwicklungen der deklarativen KI-Programmierung, KI-93 Workshop, pp. 67–82, Berlin, September 1993. Humboldt-Universität. DFKI Research Report RR-93-35.

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

[Krieger & Schäfer 94a] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for Constraint-Based Grammars.* In: [COLING 94], pp. 893–899.

[Krieger & Schäfer 94b] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for HPSG. Part 1: Overview.* DFKI Research Report RR-94-37, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1994.

[Krieger & Schäfer 94c] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for HPSG. Part 2: User Manual.* DFKI Document D-94-14, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1994.

[Krieger & Schäfer 95] Hans-Ulrich Krieger and Ulrich Schäfer. *Efficient Parameterizable Type Expansion for Typed Feature Formalisms.* In: Proceedings of the 14^{th} International Joint Conference on Artificial Intelligence (IJCAI), pp. 1428–1434, Montréal, Canada, 1995.

[Krieger et al. 93] Hans-Ulrich Krieger, John Nerbonne, and Hannes Pirker. *Feature-Based Allomorphy.* In: Proceedings of the 31^{st} Annual Meeting of the Association for Computational Linguistics (ACL), Columbus, Ohio, 6 1993. Ohio State University. Also available as DFKI Research Report RR-93-28, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany.

[Krieger 95a] Hans-Ulrich Krieger. *Classification and Representation of Types in TDL.* In: Proceedings of the International KRUSE Symposium, Knowledge Retrieval, Use, and Storage for Efficiency, University of California, Santa Cruz, 1995.

[Krieger 95b] 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, Saarbrücken, Germany, 1995.

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

[Manandhar 93] Suresh Manandhar. *CUF in context.* In: [Dörre 93], pp. 43–53. DYANA-2 Deliverable R1.2.A.

[Mellish & Reiter 93] Chris Mellish and Ehud Reiter. *Using Classification as a Programming Language.* In: Proceedings of the 13^{th} International Joint Conference on Artificial Intelligence (IJCAI), pp. 696–701, Chambéry, France, 1993.

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

[Nebel 90] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422: Lecture Notes in Artificial Intelligence. Berlin: Springer, 1990.

[Nebel 91] Bernhard Nebel. *Terminological Cycles: Semantics and Computational Properties.* In: John F. Sowa (ed.), Principles of Semantic Networks: Explorations in the Representation of Knowledge. San Mateo, CA: Morgan Kaufmann, 1991.

[Netter 93] Klaus Netter. *Architecture and Coverage of the DISCO Grammar.* In: Stephan Busemann and Karin Harbusch (eds.), Proceedings of the DFKI Workshop on Natural Language Systems: Reusability and Modularity, pp. 1–10, Saarbrücken, Germany, October 1993. DFKI Document D-93-03.

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

[Norvig 92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1992.

[Pereira & Warren 80] Fernando C.N. Pereira and David H.D. Warren. *Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks.* Artificial Intelligence, 13:231–278, 1980.

[Pereira 83] Fernando C.N. Pereira. *Parsing as Deduction.* In: Proceedings of the 21^{st} Annual Meeting of the Association for Computational Linguistics (ACL), pp. 137–144, Cambridge, MA, 1983. Massachusetts Institute of Technology.

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

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

[Rounds & Manaster-Ramer 87] William C. Rounds and Alexis Manaster-Ramer. *A logical version of functional grammar.* In: Proceedings of the 25^{th} Annual Meeting of the Association for Computational Linguistics (ACL), pp. 89–96, Buffalo, NY, 1987. State University of New York at Buffalo.

[Rupp & Johnson 94] C. J. Rupp and Rod Johnson. *On the Portability of Complex Constraint-based Grammars.* In: [COLING 94], pp. 900–905.

[Samuel 59] A. L. Samuel. *Some Studies in Machine Learning, Using the Game of Checkers.* IBM Journal of research and development, 3(3):210–229, April 1959.

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

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

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

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

[Steele 90] Guy L. Steele. *Common Lisp: The Language.* Bedford, MA: Digital Press, 2nd edition, 1990.

[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: [COLING 94].

[Uszkoreit 88] Hans Uszkoreit. *From Feature Bundles to Abstract Data Types: New Directions in Representation and Programming of Linguistic Knowledge.* In: Albrecht Blaser (ed.), Natural Language at the Computer—Contributions to Syntax and Semantics for Text Processing and Man-Machine Translation, volume 320: LNCS, pp. 31–64. Berlin: Springer, 1988.

[Uszkoreit 91] Hans Uszkoreit. *Adding Control Information to Declarative Grammars.* In: Proceedings of the 29^{th} Annual Meeting of the Association for Computational Linguistics (ACL), pp. 237–245, Berkeley, CA, 1991.

[Wahlster 93] Wolfgang Wahlster. *Verbmobil Translation of face-to-face Dialogs.* DFKI Research Report RR-93-34, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993.

[Warren 92] David S. Warren. *Memoing for Logic Programs.* Communications of the ACM, 35(3):93–111, 1992.

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