Efficient Parameterizable Type Expansion for Typed Feature Formalisms

Hans-Ulrich Krieger, Ulrich Schäfer

RR-95-1
Research Repo

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report

RR-95-18

Efficient Parameterizable Type Expansion for Typed Feature Formalisms

Hans-Ulrich Krieger, Ulrich Schäfer

December 1995

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

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

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

Deutsches Forschungszentrum für Künstliche Intelligenz

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

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

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

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

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

Dr. Dr. D. Ruland
Director

Efficient Parameterizable Type Expansion for Typed Feature Formalisms

Hans-Ulrich Krieger, Ulrich Schäfer

DFKI-RR-95-18

A version of this paper has been published in: Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI-95, August 20–25, 1995, Montreal, Canada.

This work has been supported by a grant from The Federal Ministry of Education, Science, Research and Technology (FKZ ITWM-Verbmobil 01 IV 101 K/1).

ISSN 0946-008X

Efficient Parameterizable Type Expansion for Typed Feature Formalisms

Hans-Ulrich Krieger **Ulrich Schäfer**
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
phone: +49 681 302-5299 fax: +49 681 302-5341
{krieger,schaefer}@dfki.uni-sb.de

Abstract

Over the last few years, constraint-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics. From the viewpoint of computer science, typed feature structures can be seen as a record-like data structure that allow the representation of linguistic knowledge in a uniform fashion.

Type expansion is an operation that makes the idiosyncratic and inherited constraints defined on a typed feature structure explicit and thus determines its satisfiability. We describe an efficient expansion algorithm that takes care of recursive type definitions and permits the exploration of different expansion strategies through the use of control knowledge. This knowledge is specified on a separate layer, independent of grammatical information. The algorithm, as presented in the paper, has been fully implemented in COMMON LISP and is an integrated part of the typed feature formalism $\mathcal{TDL}$ that is employed in several large NL projects.

Acknowledgements. This paper has benefited from numerous people at various workshops where parts of it have been presented, in particular, the *EAGLES workshop on Implemented Formalisms* (Saarbrücken); the workshop on *Implementations of Attribute-Value Logics for Grammar Formalisms* at the European Summer School in Language, Logic, and Information (Lisbon); the workshop on *Neuere Entwicklungen der deklarativen KI-Programmierung* at KI-93 (Berlin); the *International Conference on Computational Linguistics, COLING-94* (Kyoto); and the *International Joint Conference on Artificial Intelligence, IJCAI-95* (Montreal).

Contents

1 Introduction

Over the last few years, constraint-based grammar formalisms [Shieber 1986] have become the predominant paradigm in natural language processing and computational linguistics. While the first approaches relied on annotated phrase structure rules (e.g., PATR-II [Shieber *et al.* 1983]), modern formalisms try to specify grammatical knowledge as well as lexicon entries entirely through feature structures. In order to achieve this goal, one must enrich the expressive power of the first unification-based formalisms with different forms of disjunctive descriptions. Later, other operations came into play, e.g., (classical) negation.

However the most important extension to formalisms consists of the incorporation of *types*, for instance in modern systems like TFS [Zajac 1992], CUF [Dörre and Dorna 1993], or $\mathcal{TDL}$ [Krieger and Schäfer 1994]. Types are ordered hierarchically as is known from object-oriented programming languages, a feature heavily employed in lexicalized grammar theories like Head-Driven Phrase Structure Grammar (HPSG) [Pollard and Sag 1987]. This leads to multiple inheritance in the description of linguistic entities. In general, not only is a type related to other types through the inheritance hierarchy, but is also provided with feature constraints that are idiosyncratic to this type. Hence, a type symbol can serve as an *abbreviation* for a complex expression and an untyped feature structure becomes a typed one. If a formalism is intended to be used as a stand-alone system, it must also implement *recursive types* if it does not provide phrase-structure recursion directly (within the formalism) or indirectly (via a parser/generator).[1] In addition, certain forms of relations (like *append*) or additional extensions of the formalism (like functional uncertainty) can be nicely modelled through recursive types.

Now, because types allow us to refer to complex constraints through the use of symbol names, we need an operation that is responsible for deducing the constraints that are inherent to a type. This means, reconstructing the idiosyncratic constraints of a type, plus those that are inherited from the supertypes. We will call such a mechanism *type expansion* (TE) or type unfolding.[2] Thus TE is faced with two main tasks:

1. making some or all feature constraints explicit (TE is a structure-building operation)

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

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

[2] It is worth noting that our notion of TE shares similarities with Aït-Kaci's *sort unfolding* [Aït-Kaci *et al.* 1993] and Carpenter's *total well-typedness* [Carpenter 1992, Ch. 6]. However, the latter notion is not well-defined for true recursive typed feature structures in that such structures cannot be totally well-typed within finite time and space.

Types not only serve as a shorthand, like templates, but also provide other advantages which can only be accomplished if a mechanism for TE is available:

- STRUCTURING KNOWLEDGE
  Hierarchically-ordered types allow for a modular way of representing linguistic knowledge. Generalizations can be put at the appropriate levels of representation. *Type expansion*, then, is responsible for gathering the distributed information that is attached to the type symbols.

- SAVING MEMORY
  In practice, it is not possible to hold huge lexica in full detail in memory. However, only the idiosyncratic information of a lexicon entry needs to be represented. *Type expansion* is employed in making the constraints imposed by lexical types explicit.

- EFFICIENT PROCESSING
  Working with type symbols only or with partially expanded typed feature structures minimizes the costs of copying during processing and speeds up unification. This can only be accomplished if the system makes a mechanism for *type expansion* available.

- TYPE DISCIPLINE
  Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore disallowing one from writing inconsistent feature structures. Again, *type expansion* is necessary to determine the global consistency of a given description.

- RECURSIVE TYPES
  Recursive types give a grammar writer the opportunity to formulate certain functions or relations as recursive type specifications. Working in the type deduction paradigm forces a grammar writer to replace the context-free backbone through recursive types. Here, parameterized delayed *type expansion* is the key to controlled linguistic deduction [Uszkoreit 1991].

- ANYTIME BEHAVIOUR
  Complex architectures for NL processing require modules that can be interrupted at any time, returning an incomplete, nevertheless useful result [Wahlster 1993]. 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.

In the next section, we introduce the basic inventory to describe our own novel approach to TE. We then describe the basic structure of the algorithm, present several improvements, and show how it can be parameterized w.r.t. different dimension. Finally, we have a few words on theoretical results and compare our treatment with others. Further detailed material on this theme can be found in the PhD thesis of the first author [Krieger 1995a] and the master's thesis of the second [Schäfer 1995].

2 Preliminaries

In order to describe our algorithm, we need only a small inventory to abstract from the concrete implementation in $\mathcal{TDL}$ [Krieger and Schäfer 1994] and to make the approach comparable to others. First of all, we assume pairwise disjoint sets of *features* (attributes) $\mathcal{F}$, *atoms* (constants) $\mathcal{A}$, logical *variables* $\mathcal{V}$, and *types* $\mathcal{T}$. In the following, we refer to a *type hierarchy* $\mathcal{I}$ by a pair $\langle \mathcal{T}, \preceq \rangle$, such that $\preceq \subseteq \mathcal{T} \times \mathcal{T}$ is a decidable partial order, i.e., $\preceq$ is reflexive, antisymmetric, and transitive. A *typed feature structure* (TFS) θ is essentially either a ψ-term or an ϵ-term [Aït-Kaci 1986], i.e.,

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

such that $x \in \mathcal{V}$, $\tau \in \mathcal{T}$, $\Phi = \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}$, and $\Theta = \{\theta_1, \ldots, \theta_n\}$, where each $f_i \in \mathcal{F}$ and θ_i is again a TFS. We will call the equation $f \doteq \theta$ a *feature constraint* (or an attribute-value pair).[3] Φ is interpreted conjunctively, whereas Θ represents a disjunction. Variables are used to indicate structure sharing.

Let us give a small example to see the correspondences. The typed feature structure

$$\langle x, \textit{cyc–list}, \{\text{FIRST} \doteq 1, \text{REST} \doteq x\} \rangle$$

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

$$\boxed{x} \begin{bmatrix} \textit{cyc–list} \\ \text{FIRST} \;\; 1 \\ \text{REST} \;\; \boxed{x} \end{bmatrix}$$

It is worth noting that for the purpose of simplicity and clarity, we restrict TFS to the above two cases. Actually, our algorithm is more powerful in that it handles other cases, for instance conjunction, disjunction, and negation of types and feature constraints.

A *type system* Ω is a pair $\langle \Theta, \mathcal{I} \rangle$, where Θ is a finite set of typed feature structures and $\mathcal{I}$ an inheritance hierarchy. Given Ω, we call $\theta \in \Theta$ a *type definition*.

[3] It should be noted that we define TFS to have a nested structure and not to be flat (in contrast to feature clauses in a more logic-oriented approach, e.g., [Aït-Kaci *et al.* 1993]) in order to make the connection to the implementation clear and to come close to the structured attribute-value matrix notation.

Our algorithm is independent of the underlying deduction system—we are not interested in the normalization of feature constraints (i.e., how unification of feature structures is actually done) nor are we interested in the logic of types, e.g., whether the existence of a greatest lower bound is obligatory (TFS [Zajac 1992]; ALE [Carpenter and Penn 1994]) or optional as in $\mathcal{TDL}$ [Krieger and Schäfer 1994]. We assume here that *typed unification* is simply a black box and can be accessed through an interface function (say *unify–tfs*). From this perspective, our expansion mechanism can be either used as a stand-alone system or as an integrated part of the typed unification machinery.

We only have to say a few words on the semantic foundations of our approach at the end of this paper. This is because we could either choose extensions of *feature logic* [Smolka 1989] or directly interpret our structures within the paradigm of (constraint) logic programming [Lloyd 1987; Jaffar and Lassez 1987].

3 Algorithm

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

- support a *complete* expansion strategy
- allow *lazy expansion* of recursive types
- *minimize* the number of unifications
- make expansion parameterizable for *delay* and *preference* information
- make expansion *incremental* to serve as an anytime module

Before we describe the algorithm, we modify the syntax of TFS to get rid of unimportant details. First, we simplify TFS in that we omit variables. This can be done without loss of generality if variables are directly implemented through structure-sharing (which is the case for our system). Hence conjunctive TFS have the form $\langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$, whereas disjunctive are of the form $\langle \tau, \{\theta_1, \ldots, \theta_n\}\rangle$.

Given a TFS θ, *type–of*(θ) returns the type of θ, whereas *typedef*(τ) then obtains the type definition without inherited constraints as given by the type system $\Omega = \langle \Theta, \mathcal{I} \rangle$. We call this TFS a *skeleton*. It is either $\langle \sigma, \{\theta_1, \ldots, \theta_n\}\rangle$ or $\langle \sigma, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle$, where σ are the direct supertype(s) of τ.

Because the algorithm should support partially expanded (delayed) types, we enrich each TFS θ by two flags:

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

2. *expanded*(θ)=true, iff Δ-*expanded*(θ)=true **and** *expanded*(θ_i)=true for all elements θ_i of TFS θ; false otherwise.

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

3.1 Basic Structure

The following functions briefly sketch the basic algorithm. It is a destructive depth-first algorithm with a special treatment of recursive types that will be explained in Section 3.3.

expand–tfs is the main function that initializes TE. The while loop is executed until the TFS θ is expanded or so-called "resolved" (see keyword :resolved-predicate in Section 3.5). Several passes may be necessary for recursive TFS.

---

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

---

depth–first–expand and *types–first–expand* recursively traverse a TFS. Which of both functions is employed, can be specified by the user. The visited check is done by comparing variables (actually, structure-sharing in the implementation makes variables obsolete). *types–first–expand* is defined analogously by first expanding the root type of a TFS, and then processing the feature constraints.

---

depth–first–expand(θ) :=
  if θ *has been already visited in this pass*
    then return
    else
      if $\theta = \langle \tau, \{\theta_1, \ldots, \theta_n\} \rangle$
        then
          for every $\theta \in \{\theta_1, \ldots, \theta_n\}$:
            depth–first–expand(θ)
        else do /* $\theta = \langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\} \rangle$ */
          for every $\theta \in \{\theta_1, \ldots, \theta_n\}$:
            depth–first–expand(θ);

if not Δ-*expanded*(θ)
  then *unify–type–and–node*(τ, θ)
od.

unify–type–and–node destructively unifies θ with the expanded TFS of τ. The index ι specifies which "prototype" of τ is chosen (see Section 3.2).

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

We adapt Smolka's treatment of negation for our TFS [Smolka 1989]. Note that we only depict the conjunctive case here.

$$
\begin{array}{l}
\textit{negate–tfs}(\theta = \langle \tau, \{f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n\}\rangle) := \\
\quad \textbf{return} \\
\quad\; \langle \top, \{\langle \neg\tau, \{\}\rangle, \\
\qquad\quad \langle \top, \{f_1 \uparrow\}\rangle, \langle \top, \{f_1 \doteq \textit{negate–tfs}(\theta_1)\}\rangle, \ldots, \\
\qquad\quad \langle \top, \{f_n \uparrow\}\rangle, \langle \top, \{f_n \doteq \textit{negate–tfs}(\theta_n)\}\rangle\}\rangle.
\end{array}
$$

3.2 Indexed Prototype Memoization

The basic idea of *memoization* [Michie 1968] is to tabulate results of function applications in order to prevent wasted calculations. We adapt this technique to the type expansion function. The argument of our memoized expansion function is a pair consisting of a type name (or a name of a lexicon entry or a rule) and an arbitrary index that allows access to different TFS of the same type which may be expanded in different ways (fully expanded or partially w.r.t. to a certain specification). Such feature structures are called *prototypes*.

Once a prototype has been expanded according to the attached control information, its expanded version is recorded and all future calls return a copy of it, instead of repeating the same unifications once again:

expand–type$(\tau, index)$:=
  if *protomemo*$(\tau, index)$ **undefined**
    then $\theta \leftarrow$ *expand–tfs*(*typedef*(τ));
        protomemo$(\tau, index) \leftarrow \theta$;

return *copy–tfs*(θ)
else return *copy–tfs*(*protomemo*(τ, *index*)).

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

Some empirical results indicate the usefulness of indexed prototype memoization. Figure 1 contains statistical information about the expansion of a mid-size HPSG grammar with approx. 900 type definitions. About 250 additional lexicon entries and rules have been expanded from scratch, i.e., all types are unexpanded (are skeletons) at the beginning. The type and instance skeletons together consist of about 9000 nodes, whereas the resulting structures have a total size of approx. 50000 nodes (nodes undergoing garbage collections are not counted).

The measurements show that memoization speeds up expansion by a factor of 5 here (or 10 if all types except the lexicon entries are pre-expanded which seems to be the optimal setting at run time). These factors are directly proportional to the number of unifications. The time difference between the memoized and non-memoized algorithm may be even bigger if disjunctions are involved (in the ideal case exponential). The sample grammar contains only a few disjunctions.

algorithm	*depth–1st–expand*		*types–1st–expand*		*depth–1st–expand*		*types–1st–expand*	
memoization	yes		yes		no		no	
time (secs)	45	23*	46	23*	216		218	
unifications	27221	14495*	27207	14481*	155888		155876	
number of	853	*cons*	260	*cons*	8330	*avm*	8454	*avm*
calls to	316	cat-type	147	*diff-list*	2392	sem-expr	2503	sem-expr
expand-type	269	*diff-list*	143	morph-type	1379	term-type	1420	term-type
	243	morph-type	94	nmorph-head	1161	*cons*	1196	*cons*
*: with types	208	atomic-wff	83	sort-expr	1003	wff-type	1073	wff-type
pre-expanded	202	rp-type	71	atomic-wff	933	agr-feat	951	agr-feat
	146	conj-wff-type	62	rp-type	880	semantics	747	semantics
	120	var-type	53	subwff-inst	823	indexed-wff	730	indexed-wff
		...		...		...		...

Figure 1: *Efficiency of depth-first vs. types-first expansion with/without indexed prototype memoization.*

3.3 Detecting Recursion

The memoization technique is also employed in detecting recursive types. This is important in order to prevent infinite computations. We use the so-called "expand stack" of *expand–type* to check whether a type is recursive or not (see

Section 3.4). Each call of *expand–type*$(\tau, index)$ will push τ onto the expand stack. This stack then is passed to *expand–tfs*.

If a type τ on top of the expand stack also occurs below in the stack $(\tau, \sigma_n, \ldots, \sigma_1, \tau, \rho_m, \ldots, \rho_1)$, we immediate know that the types $\tau, \sigma_n, \ldots, \sigma_1$ are recursive. Furthermore, these types form a *strongly connected component* (SCC) of the type dependency (or occurrence) graph, i.e., each type in the SCC is reachable from every other type in the SCC. Examples for such SCCs are (*cons list*) and (*state1*) in the example below (Section 3.4).

Testing whether a type is recursive or not thus reduces to a simple *find* operation in a global list that contains all SCCs. The expansion algorithm uses this information in *expand–tfs* to delay recursive types if the expand stack contains more than one element. Otherwise, prototype memoization would loop.

If a recursive type occurs in a TFS and this type has already been expanded under a subpath, and furthermore no features or other types are specified at this node, then this type will be delayed, since it would expand forever (we call this *lazy expansion*). An instance of such a recursive type for which type expansion stops is the recursive version of *list*, as defined below.

3.4 Example

In the following, we define a finite automaton A as a family of recursive type definitions. A consists of two states and accepts the language $\mathcal{L}(A) = \mathtt{a}^*(\mathtt{a}+\mathtt{b})$.[4] The input is specified through a list under path INPUT ; cf. the definition of type *ab* below. The distributed (or named) disjunction [Eisele and Dörre 1990] headed by \$1 in type *state1* is used to map input symbols to state types (and vice versa). Defining FA this way provides a solid basis for the integration of automata-based allomorphy (e.g., 2-level morphology) and morphotactics within the same constraint-based formalism (cf. [Krieger *et al.* 1993]).

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

$$cons \Rightarrow \begin{bmatrix} \text{FIRST} & \top \\ \text{REST} & list \end{bmatrix} \quad \text{we abbreviate } cons \text{ via } \langle \ldots \rangle$$

$$non\text{-}final \Rightarrow \begin{bmatrix} \text{INPUT} & \langle \boxed{1} . \boxed{2} \rangle \\ \text{EDGE} & \boxed{1} \\ \text{NEXT} & \begin{bmatrix} \text{INPUT} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

[4] In [Krieger 1995b], it is shown that this special kind of encoding allows us to reconstruct all the nice properties of finite automata/regular expressions (viz., closedness under intersection, union, complement, concatenation, and Kleene closure) in terms of operations of the underlying feature calculus.

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

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

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

Fig. 2 shows a trace of the expansion of type *ab*. The algorithm is *depth–first–expand* without any delay or preference information. In this trace, we assume that it was not known before that the types *cons*, *list*, and *state1* are recursive, hence the SCCs will be computed on the fly.

step	*expand–type*	in type	under path	expand stack
1	*cons*	*ab*	INPUT.REST	(*ab*)
2	*list*	*cons*	REST	(*cons ab*)
3	*cons*	*list*	ϵ	(*list cons ab*) → (*cons list*) new SCC, delay *cons*
4	*cons*	*ab*	INPUT	(*ab*)
5	*state1*	*ab*	ϵ	(*ab*)
6	*state1*	*state1*	NEXT	(*state1 ab*) → (*state1*) new SCC, delay *state1*
7	*final*	*state1*	NEXT	(*state1 ab*)
8	*non-final*	*state1*	ϵ	(*state1 ab*)
9	*cons*	*non-final*	INPUT	(*non-final state1 ab*)
10	*state1*	*ab*	NEXT	(*ab*)

Figure 2: *Tracing the expansion of type ab. ab is consistent, hence the finite automata accepts input* $\langle\,\mathtt{a}, \mathtt{b}\,\rangle$.

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

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

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

$$abb \Rightarrow \begin{bmatrix} \textit{state1} & \\ \textsc{input} & \langle \mathtt{a}, \mathtt{b}, \mathtt{b} \rangle \end{bmatrix}$$

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

3.5 Declarative Specification of Control Information

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

- :expand-function {depth|types}-first-expand specifies the basic expansion algorithm.

- :delay { ({*type* | (*type* [*pred*])} {*path*}$^+$) }* specifies types at *path* to be delayed. *path* may be a feature path or a complex path pattern with wildcard symbols *, +, ?, feature and segment variables. *pred* is a test predicate to compare types, e.g., = or $\preceq$ (checked in *unify–type–and–node*).

- {:expand|:expand-only} { ({*type* | (*type* [*index* [*pred*]])} {*path*}$^+$) }* There are two mutually exclusive modes, concerning the expansion of types. If the :expand-only list is specified, only types in this list will be expanded with the specified prototype *index*, all others will be delayed. If the :expand list is specified, all types will be expanded (checked in *unify–type–and–node*).

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

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

- :use-{conj|disj}-heuristics {t|nil} [Uszkoreit 1991] suggested exploiting numerical preferences to speed up unification. Both keywords control the use of this information in functions *depth–first–expand* and *types–first–expand*.

- :resolved-predicate {resolved-p|always-false|...} This slot specifies a user definable predicate that may be used to stop recursion (see function *expand–tfs*). Such a predicate might suffice in practice to guarantee a terminating expansion without violating correctness. The default predicate is always-false which leads to a complete expansion algorithm (if no other delay information is specified).

- :ask-disj-preference {t|nil} If this flag is set to t, the expansion algorithm interactively asks for the order in which disjunction alternatives should be expanded (checked in *depth–first–expand* and *types–first–expand*). This option is useful during the debug phase of a grammar.

- :ignore-global-control {t|nil} Specifies whether globally specified :expand-only, :expand, and :delay information should be ignored or not.

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

```
defcontrol verb
;; delay all subtypes of sign under spec. path pattern
;; ? matches INHERITED and TO-BIND
  ((:delay ((sign Subsumes) SYNSEM.NONLOCAL.?.SLASH))
;; attribute preference during expansion by this order
   (:attribute-preference SYNSEM DTRS SUBCAT HEAD)
   (:use-disj-heuristics T)
   (:ignore-global-control T)
;; expand type local with index initial
;; * matches all paths in type local
   (:expand ((local initial) *)))
;; these control specs are used for type verb with index 1
  :index 1.
```

3.6 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 TFS and decides whether a description is satisfiable or not (see also Section 5). However, there are several ways to prevent infinite expansion in our framework:

- The first method is part of the expansion algorithm (lazy expansion) as described before.

- The second way is brute force: use the :maxdepth slot to cut expansion at a suitable path depth.

- The third method is to define :delay patterns or to select the :expand-only mode with appropriate type and path patterns.

- The fourth method is to use the :attribute-preference list to define the "right" order for expansion.

- Finally, one can define an appropriate :resolved-predicate that is suitable for a class of recursive types.

4 Applications

In Section 3.4, we have already mentioned an NL application in which type expansion was employed, viz., in the formulation of the interface between allomorphy and morphotactics [Krieger *et al.* 1993]. Let us quickly present two other areas that profit from type expansion: parsing/generation as type expansion and distributed parsing with partially expanded information.

Parsing and generation can be seen in the light of type expansion as a uniform process, where ideally only the phonology (for parsing) or the semantics (for generation) must be given, for instance:

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

Type expansion together with a sufficiently specified grammar then is responsible in both cases for constructing a fully specified feature structure which is maximal informative and compatible with the input structure.

Distributed parsing is a strategy which reduces the representational overhead: given one grammar which cospecifies syntax and semantics, proper constraints (i.e., filters) are separated from purely representational constraints. The resulting subgrammars are then processed by two parsers in parallel [Diagne *et al.* 1995]. This presupposes that we can properly handle partially expanded typed feature structures.

5 Theoretical Results

It is worth noting that testing for the satisfiability of feature descriptions admitting recursive type equations/definitions is in general undecidable. [Rounds and Manaster-Ramer 1987] were the first to have shown that a Kasper-Rounds logic enriched with recursive types allows one to encode a Turing machine. Later, [Smolka 1989] argued that the undecidability result is due to the use of coreference constraints. He demonstrated his claim by encoding the word problem of Thue systems. Hence, our expansion mechanism is faced with the same result in that expansion might not terminate.

However, we conjecture that non-satisfiability and thus failure of type expansion is semi-decidable. The intuitive argument is as follows: given an arbitrary recursive TFS and assuming a fair type unfolding strategy, the only event under which TE terminates in finite time follows from a local unification failure which then leads to a global one. In every other case, the unfolding process goes on by substituting types through their definitions. Recently, [Aït-Kaci *et al.* 1993] have formally shown a similar result by using the compactness theorem of first-order logic. However, their proof assumes the existence of an infinite OSF clause (generated by unfolding a ψ-term). Furthermore, they have not addressed disjunction.

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

Semantically, we can formally account for such recursive feature descriptions (with respect to a type system) in different ways: either directly on the descriptions, or indirectly through a transformational approach into first-order logic (see [Krieger 1995c] for a transformational approach of typed feature structures into definite equivalences). Both approaches rely on the construction of a fixpoint over a certain (downward) continuous function.[5] The first approach is in general closer to an implementation (and thus to our algorithm) in that the function which is involved in the fixpoint construction corresponds more or less to the unification/substitution of TFS (see for instance [Aït-Kaci 1986] or [Pollard and Moshier 1990]). The latter approach is based on the assumption that TFS are only syntactic sugar for first-order formulae. If we transform these descriptions into an equivalent set of definite clauses, we can employ techniques that are fairly common in logic programming, viz., characterizing models of a definite program through fixpoints. Take, for instance, our *cyc–list* example from the beginning to see the outcome of such a transformation (assume that *cyc–list* is a subtype of *list*):

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

Under the least fixpoint interpretation, *cyc–list* will be assigned an empty denotation (assuming a rational tree domain), whereas the greatest fixpoint interpretation leads to a non-empty denotation, containing even infinite feature trees ([Krieger 1995c] is a detailed investigation of this and other related areas).

6 Comparison to other Approaches

To our knowledge, the problem of type expansion within a typed feature-based environment was first addressed by Hassan Aït-Kaci [Aït-Kaci 1986]. The language, he described, was called KBL and shared great similarities with LOGIN; see [Aït-Kaci and Nasr 1986]. However, his expansion mechanism was order dependent in that it substituted types by their definition instead of unifying the information. Moreover, it was non-lazy, thus it will fail to terminate for recursive types and performs TE only at definition time as is the case for ALE

[5] In both cases, there is, in general, more than one fixpoint, but it seems desirable to choose the *greatest* one, as it would not rule out, for instance, cyclic structures or types which are not "grounded" on atoms.

[Carpenter and Penn 1994]. However, ALE provides recursion through a built-in bottom-up chart parser and through definite clauses. Allowing TE only at definition time is in general space consuming, thus unification and copying is expensive at run time.

Another possibility one might follow is to integrate TE into the typed unification process so that TE can take place at run time. Systems that explore this strategy are TFS [Zajac 1992] and LIFE [Aït-Kaci 1993]. However, both implementations are not lazy, thus hard to control and moreover, might not terminate. In addition, if prototype memoization is not available, TE at run time is inefficient; cf. Fig. 1). A system that employs a lazy strategy on demand at run time is CUF [Dörre and Dorna 1993]. Laziness can be achieved here by specifying delay patterns as is familiar from Prolog. This means delaying the evaluation of a relation until the specified parameters are instantiated.

7 Summary

Type expansion is an operation that makes constraints of a typed feature structure explicit and determines its satisfiability. We have described an expansion algorithm that takes care of recursive types and allows us to explore different expansion strategies through the use of control knowledge. Efficiency is addressed through specialized techniques: (i) prototype memoization reduces the number of unifications, and (ii) preference information directs the search space. Because our notion of type expansion is conceived as a stand-alone module, one can freely choose the time of its invocation, e.g., during typed unification, parsing, etc.

The algorithm, as presented in the paper, has been fully implemented within the *TDL/UDiNe* system [Krieger and Schäfer 1994; Backofen and Weyers 1994] and is an integrated part of Disco [Uszkoreit *et al.* 1994].

We are convinced that our approach is also of interest to those who are working with (possibly recursive and hierarchically-ordered) record-like data structures in other areas of computer science.

References

[Aït-Kaci and Nasr 1986] 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.* 1993] Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. Technical Report 32, Digital Equipment Corporation, DEC Paris Research Laboratory, France, May 1993. Also in Proceedings of the International Symposium on Logic Programming, Oct. 1993, MIT Press.

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

[Aït-Kaci 1993] Hassan Aït-Kaci. An introduction to LIFE—programming with logic, inheritance, functions, and equations. In *Proceedings of the International Symposium on Logic Programming*, pages 52–68, 1993.

[Backofen and Weyers 1994] Rolf Backofen and Christoph Weyers. $\mathcal{UDiNe}$—a feature constraint solver with distributed disjunction and classical negation. Unpublished documentation note, 1994.

[Carpenter and Penn 1994] Bob Carpenter and Gerald Penn. ALE—the attribute logic engine user's guide. version 2.0. Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, August 1994.

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

[Diagne *et al.* 1995] Abdel Kader Diagne, Walter Kasper, and Hans-Ulrich Krieger. Distributed parsing with HPSG grammars. In *Proceedings of the 4th International Workshop on Parsing Technologies, IWPT-95*, pages 79–86, 1995. Also available as DFKI Research Report RR-95-19.

[Dörre and Dorna 1993] Jochen Dörre and Michael Dorna. CUF—a formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I.* DYANA, 1993.

[Eisele and Dörre 1990] Andreas Eisele and Jochen Dörre. Disjunctive unification. IWBS Report 124, IWBS, IBM Germany, Stuttgart, 1990.

[Jaffar and Lassez 1987] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Krieger and Schäfer 1994] Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, pages 893–899, 1994.

[Krieger *et al.* 1993] Hans-Ulrich Krieger, John Nerbonne, and Hannes Pirker. Feature-based allomorphy. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics, ACL-93*, pages 140–147, 1993. A version of this paper is available as DFKI Research Report RR-93-28.

[Krieger 1995a] 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, September 1995.

[Krieger 1995b] Hans-Ulrich Krieger. Typed feature formalisms as a common basis for linguistic specification. In *Machine Translation and the Lexicon.* Springer, Berlin, 1995. Lecture Notes in Artificial Intelligence 898. A version of this paper is available as DFKI Research Report RR-94-39.

[Krieger 1995c] Hans-Ulrich Krieger. Typed feature structures, definite equivalences, greatest model semantics, and nonmonotonicity. In *Proceedings of the 4th Meeting on Mathematics of Language, MOL4*, 1995. Also available as DFKI Research Report RR-95-20.

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

[Michie 1968] Donald Michie. "Memo" functions and machine learning. *Nature*, 218(1):19–22, 1968.

[Pollard and Moshier 1990] Carl J. Pollard and M. Drew Moshier. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language, and Cognition. Vol. 1 of Vancouver Studies in Cognitive Science*, pages 285–322. University of British Columbia Press, 1990.

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

[Rounds and Manaster-Ramer 1987] William C. Rounds and Alexis Manaster-Ramer. A logical version of functional grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 89–96, 1987.

[Schäfer 1995] Ulrich Schäfer. Parametrizable type expansion for TDL. Master's thesis, Universität des Saarlandes, Department of Computer Science, 1995.

[Shieber *et al.* 1983] 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, editors, *Research on Interactive Acquisition and Use of Knowledge*, pages 39–79. AI Center, SRI International, Menlo Park, Cal., November 1983.

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

[Smolka 1989] Gert Smolka. Feature constraint logic for unification grammars. IWBS Report 93, IWBS, IBM Germany, Stuttgart, November 1989. Also in Journal of Logic Programming, 12:51–87, 1992.

[Uszkoreit *et al.* 1994] Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of COLING-94*, pages 436–440, 1994. A version of this paper is available as DFKI Research Report RR-94-38.

[Uszkoreit 1991] Hans Uszkoreit. Strategies for adding control information to declarative grammars. In *Proceedings of the 29th Meeting of the Association for Computational Linguistics (ACL)*, pages 237–245, 1991.

[Wahlster 1993] Wolfgang Wahlster. VERBMOBIL—translation of face-to-face dialogs. Research Report RR-93-34, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1993. Also in Proc. MT Summit IV, 127–135, Kobe, Japan, July 1993.

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