

# 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 data structures that allow the representation of linguistic knowledge in a uniform fashion. Type expansion is an operation that makes constraints of a typed feature structure explicit and 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 *TDC* that is employed in several large NL projects.

## 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 come into play, e.g., (classical) negation.

---

This work was funded by the German Federal Ministry of Education, Science, Research, and Technology as part of the *Verbmobil* project. We are grateful to three anonymous IJCAI reviewers for helpful comments.

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 *TDC* [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).<sup>1</sup> 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.<sup>2</sup> Thus TE is faced with two main tasks:

1. making some or all feature constraints explicit (type

---

<sup>1</sup>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.

<sup>2</sup>It is worth noting that our notion of TE shares similarities with Ait-Kaci's *sort unfolding* [Ait-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.

2. determining the global consistency of a type or more generally, of a typed feature structure

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 names only or with partially expanded types minimizes the costs of copying structures during processing and speeds up unification. This can only be accomplished if the system makes a mechanism for *type expansion* available.

- TYPE CHECKING

Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore disallowing 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.

## 2 Preliminaries

In order to describe our algorithm, we need only a small inventory to abstract from the concrete implementation in *TDC* [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)  $\theta$  is essentially either a  $\psi$ -term or an  $\epsilon$ -term [Aït-Kaci, 1986], i.e.,

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

such that  $x \in \mathcal{V}$ ,  $\tau \in \mathcal{T}$ ,  $\Phi = \{f_1 \doteq \theta_1, \dots, f_n \doteq \theta_n\}$ , and  $\Theta = \{\theta_1, \dots, \theta_n\}$ , where each  $f_i \in \mathcal{F}$  and  $\theta_i$  is again a TFS.

We will call the equation  $f \doteq \theta$  a *feature constraint* (or an attribute-value pair).<sup>3</sup>  $\Phi$  is interpreted conjunctively, whereas  $\Theta$  represents a disjunction. Variables are used to indicate structure sharing.

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

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

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

$$\boxed{x} \begin{bmatrix} \text{cyc-list} \\ \text{FIRST } 1 \\ \text{REST } \boxed{x} \end{bmatrix}$$

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

<sup>3</sup>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 *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

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  $\langle \tau, \{f_1 \doteq \theta_1, \dots, f_n \doteq \theta_n\} \rangle$ , whereas disjunctive are of the form  $\langle \tau, \{\theta_1, \dots, \theta_n\} \rangle$ .

Given a TFS  $\theta$ , *type-of*( $\theta$ ) returns the type of  $\theta$ , whereas *typedef*( $\tau$ ) 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  $\langle \sigma, \{\theta_1, \dots, \theta_n\} \rangle$  or  $\langle \sigma, \{f_1 \doteq \theta_1, \dots, f_n \doteq \theta_n\} \rangle$ , where  $\sigma$  are the direct supertype(s) of  $\tau$ .

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

1.  $\Delta$ -*expanded*( $\theta$ )=true, iff *typedef*(*type-of*( $\theta$ )) and the definitions of all its supertypes have been unified with  $\theta$ , and false otherwise.
2. *expanded*( $\theta$ )=true, iff  $\Delta$ -*expanded*( $\theta$ )=true **and** *expanded*( $\theta_i$ )=true for all elements  $\theta_i$  of TFS  $\theta$ .

Hence  $\Delta$ -*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  $\Delta$ -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  $\theta$  is expanded or so-called “resolved” (see keyword *:resolved-predicate* in Section 3.5). Several passes may be necessary for recursive TFS.

```

expand-tfs( $\theta$ ) :=
  while not (expanded-p( $\theta$ ) or resolved-p( $\theta$ ) or
            no unification occurred in last pass)
    depth-first-expand( $\theta$ ).
  /* or types-first-expand( $\theta$ ), 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( $\theta$ ) :=
  if  $\theta$  has been already visited in this pass
  then return
  else
  if  $\theta = \langle \tau, \{\theta_1, \dots, \theta_n\} \rangle$ 
  then
    for every  $\theta \in \{\theta_1, \dots, \theta_n\}$ :
      depth-first-expand( $\theta$ )
  else do /*  $\theta = \langle \tau, \{f_1 \doteq \theta_1, \dots, f_n \doteq \theta_n\} \rangle$  */
    for every  $\theta \in \{\theta_1, \dots, \theta_n\}$ :
      depth-first-expand( $\theta$ );
    if not  $\Delta$ -expanded( $\theta$ )
    then unify-type-and-node( $\tau, \theta$ )
  od.

```

*unify-type-and-node* destructively unifies  $\theta$  with the expanded TFS of  $\tau$ . The index  $\iota$  specifies which “prototype” of  $\tau$  is chosen (see Section 3.2).

```

unify-type-and-node( $\tau, \theta$ ) :=
  if  $\tau = \neg\sigma$ 
  then unify-tfs (negate-fs (expand-type( $\sigma, \iota$ )),  $\theta$ )
  else unify-tfs (expand-type( $\tau, \iota$ ),  $\theta$ );
   $\Delta$ -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.

```

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

```

### 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 (e.g., partially or fully). Such feature structures are called *prototypes*.

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

```

expand-type( $\tau$ , index) :=
  if protomemo( $\tau$ , index) undefined
  then  $\theta \leftarrow \textit{expand-tfs}(\textit{typedef}(\tau))$ ;
        protomemo( $\tau$ , index)  $\leftarrow \theta$ ;
        return copy-tfs( $\theta$ )
  else return copy-tfs(protomemo( $\tau$ , index)).

```

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

Some empirical results indicate the usefulness of indexed prototype memoization. Figure 1 contains statistical information about the expansion of an 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.

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). 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. The sample grammar contains only a few disjunctions.

### 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  $\tau$  onto the expand stack. This stack then is passed to *expand-tfs*.

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

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 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 that stops is the recursive version of *list*, as defined below.

### 3.4 Example

In the following, we define a finite automaton with two states that accepts the language  $a^*(a + b)$ . 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  $\Rightarrow \{ \textit{cons}, \langle \rangle \}$ 

cons  $\Rightarrow \left[ \begin{array}{cc} \text{FIRST} & \top \\ \text{REST} & \textit{list} \end{array} \right]$  we abbr. cons via  $\langle \dots \rangle$ 

non-final  $\Rightarrow \left[ \begin{array}{cc} \text{INPUT} & \langle \boxed{1} \cdot \boxed{2} \rangle \\ \text{EDGE} & \boxed{1} \\ \text{NEXT} & [\text{INPUT } \boxed{2}] \end{array} \right]$ 

final  $\Rightarrow \left[ \begin{array}{cc} \text{INPUT} & \langle \rangle \\ \text{EDGE} & \textit{undef} \\ \text{NEXT} & \textit{undef} \end{array} \right]$ 

state1  $\Rightarrow \left[ \begin{array}{cc} \textit{non-final} \\ \text{EDGE} & \$1 \{ \mathbf{a}, \mathbf{b} \} \\ \text{NEXT} & \$1 \{ \textit{state1}, \textit{final} \} \end{array} \right]$ 

ab  $\Rightarrow \left[ \begin{array}{cc} \textit{state1} \\ \text{INPUT} & \langle \mathbf{a}, \mathbf{b} \rangle \end{array} \right]$ 

```

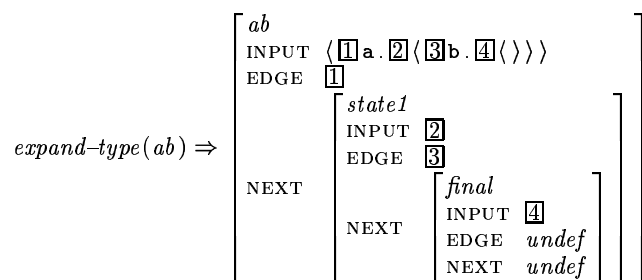
Fig. 2 shows a trace of the expansion of type *ab*. The algorithm is *depth-first-expand* without any delay or preference information. In this trace, we assume that

algorithm	<i>depth-1st-expand</i>		<i>types-1st-expand</i>		<i>depth-1st-expand</i>		<i>types-1st-expand</i>	
memoization	yes		yes		no		no	
time (secs)	45	23*	46	23*	216		218	
unifications	27221	14495*	27207	14481*	155888		155876	
number of calls to <i>expand-type</i>	853	*cons*	260	*cons*	8330	*avm*	8454	*avm*
316	cat-type	147	*diff-list*	2392	sem-expr	2503	sem-expr	
269	*diff-list*	143	morph-type	1379	term-type	1420	term-type	
243	morph-type	94	nmorph-head	1161	*cons*	1196	*cons*	
*: with types pre-expanded	208	atomic-wff	83	sort-expr	1003	wff-type	1073	wff-type
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.

it was not known before that the types *cons* (abbreviated as (...)), *list*, and *state1* are recursive, hence the sccs will be computed on the fly.

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



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

$$abb \Rightarrow \left[ \begin{array}{l} \text{state1} \\ \text{INPUT} \langle \text{a, b, b} \rangle \end{array} \right]$$

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*}<sup>+</sup> ) }\* 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  $\leq$  (checked in *unify-type-and-node*).
- {*:expand|:expand-only*} { ( {*type* | (*type* [*index* [*pred*]]) } {*path*}<sup>+</sup> ) }\* There are two mutually exclusive modes concerning expansion of types. If the *:expand-only* list is specified, only types in this list

will be expanded with the specified prototype *index*, all others will be delayed. If the *:expand* list is specified, all types will be expanded (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*). 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*)
- *: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 *TDC*.

step	expand-type	in type	under path	expand stack
1	<i>cons</i>	<i>ab</i>	INPUT.REST	( <i>ab</i> )
2	<i>list</i>	<i>cons</i>	REST	( <i>cons ab</i> )
3	<i>cons</i>	<i>list</i>	$\epsilon$	( <i>list cons ab</i> ) → ( <i>cons list</i> ) is new scc, delay <i>cons</i> here
4	<i>cons</i>	<i>ab</i>	INPUT	( <i>ab</i> )
5	<i>state1</i>	<i>ab</i>	$\epsilon$	( <i>ab</i> )
6	<i>state1</i>	<i>state1</i>	NEXT	( <i>state1 ab</i> ) → ( <i>state1</i> ) is new scc, delay <i>state1</i> here
7	<i>final</i>	<i>state1</i>	NEXT	( <i>state1 ab</i> )
8	<i>non-final</i>	<i>state1</i>	$\epsilon$	( <i>state1 ab</i> )
9	<i>cons</i>	<i>non-final</i>	INPUT	( <i>non-final state1 ab</i> )
10	<i>state1</i>	<i>ab</i>	NEXT	( <i>ab</i> )

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

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

### 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 input (TFS) and decides whether a description is satisfiable or not (see 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 only the phonology (for parsing) or the semantics (for generation) must be given, for instance:

**Parsing:**  $\left[ \begin{array}{l} \textit{phrase} \\ \text{PHON} \langle \textit{“John” “likes” “bagels”} \rangle \end{array} \right]$

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 co-specifies 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. 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, in general, semi-decidable. The intuitive argument is as follows: given an arbitrary recursive TFS and assuming a fair type unfolding strategy, the only event under which TE terminates in finite time follows from a local unification failure which then leads to a global one. In every other case, the unfolding process goes on by substituting types through their definitions. Recently, [Ait-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  $\psi$ -term).

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

Semantically, we can formally account for such recursive feature descriptions (with respect to a type system) in different ways: either directly on the descriptions, or indirectly through a transformational approach into (first-order) logic. Both approaches rely on the construction of a fixpoint over a certain continuous function.<sup>4</sup> 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 [Ait-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 the models of a definite program through a fixpoint. Take, for instance, our *cyc-list* example from the beginning to see the outcome of such a transformation (assume that *cyc-list* is a subtype of *list*):

$$\forall x. \text{cyc-list}(x) \leftrightarrow \exists y, z. \text{list}(x) \wedge \\ \text{FIRST}(x, y) \wedge \text{REST}(x, z) \wedge \\ y \doteq 1 \wedge z \doteq x$$

## 6 Comparison to other Approaches

To our knowledge, the problem of type expansion within a typed feature-based environment was first addressed by Hassan Ait-Kaci [Ait-Kaci, 1986]. The language he described was called KBL and shared great similarities with LOGIN; see [Ait-Kaci and Nasr, 1986]. However, the expansion mechanism he outlined 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 [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.

<sup>4</sup>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.

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 [Ait-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 here, 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 *TDC/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

- [Ait-Kaci and Nasr, 1986] Hassan Ait-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
- [Ait-Kaci *et al.*, 1993] Hassan Ait-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.
- [Ait-Kaci, 1986] Hassan Ait-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.
- [Ait-Kaci, 1993] Hassan Ait-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. *UDiNe*—A Feature Constraint Solver with Distributed Disjunction and Classical Negation. Unpublished documentation note.
- [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.
- [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 *Proc. of 14th POPL*, pages 111–119, 1987.
- [Krieger and Schäfer, 1994] Hans-Ulrich Krieger and Ulrich Schäfer. *TDC*—a type description language for constraint-based grammars. In *Proc. of 15th COLING*, pages 893–899, 1994.
- [Krieger *et al.*, 1993] Hans-Ulrich Krieger, John Nerbonne, and Hannes Pirker. Feature-based allomorphy. In *Proc. of 31st ACL*, pages 140–147, 1993.
- [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 *Proc. of 25th ACL*, pages 89–96, 1987.
- [Shieber *et al.*, 1983] S. Shieber, H. Uszkoreit, F. Pereira, J. Robinson, and M. 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., 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] H. Uszkoreit, R. Backofen, S. Busemann, A.K. Diagne, E.A. Hinkelman, W. Kasper, B. Kiefer, H.-U. Krieger, K. Netter, G. Neumann, S. Oepen, and S.P. Spackman. DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proc. of 15th COLING*, pages 436–440, 1994.
- [Uszkoreit, 1991] Hans Uszkoreit. Strategies for adding control information to declarative grammars. In *Proc. of 29th ACL*, pages 237–245, 1991.
- [Wahlster, 1993] Wolfgang Wahlster. VERBMOBIL—translation of face-to-face dialogs. Proc. of 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.