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 to represent 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 allows to explore 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 a large NL system.

Keywords: Natural language, typed feature formalisms, recursive feature types.
Knowledge representation, languages and systems for representing knowledge.

This paper has not already been accepted by and is not currently under review for a journal or another conference. Nor will it be submitted for such during IJCAI’s review period.
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 TDC [Krieger and Schäfer, 1994]. Types are ordered hierarchically as it is known from object-oriented programming languages, a feature heavily employed in lexicalized grammar theories like Head-Driven Phrase Structure Grammar (HPG) [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 to deduce the constraints that are inherent to a type. This means, to reconstruct 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 certain or all feature constraints explicit (type expansion is a

\(^1\)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 relations (equivalences), as is the case for CUF, such that parsing/generation roughly corresponds to SLD resolution.

\(^2\)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 recursive types, i.e., recursive types cannot be well-typed.
structure-building operation)

2. determining the global consistency of a type or more general, 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 to gather 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 need 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 to write inconsistent feature structures. Again, *type expansion* is necessary to determine the global consistency of a given description.

- **RECURSIVE TYPES**
  Recursive types give a grammar writer the opportunity to formulate certain functions or relations as recursive type specifications. Working in the type deduction paradigm enforces 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].

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 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.,

\[
\theta ::= \langle x, \tau, \Phi \rangle | \langle x, \tau, \Theta \rangle
\]

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 \( \theta_i \) is again a TFS. We will call the equation \( f \doteq \theta \) a feature constraint (or an attribute-value pair).\(^3\) \( \Phi \) 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 than the following two-dimensional attribute-value matrix (AVM) notation:

\[
\begin{bmatrix}
\text{cyc-list} \\
\text{FIRST} & 1 \\
\text{REST} & X
\end{bmatrix}
\]

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.

Our algorithm is independent of the underlying deduction system—we are not interested in the normalization of feature constraints (i.e., how unification

\(^3\)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.
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 TDC

[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 require-

ments:

- 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 conjunc-

tive 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 \( \theta \), 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 \( \langle \sigma, \{ \theta_1, \ldots, \theta_n \} \rangle \) or

\( \langle \sigma, \{ f_1 \doteq \theta_1, \ldots, f_n \doteq \theta_n \} \rangle \), 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. \( \Delta\text{-expanded}(\theta) = \text{true} \) iff \( \text{typedef(type-of(\( \theta \)))} \) and the definitions of all

   its supertypes have been unified with \( \theta \), and false otherwise.

2. \( \text{expanded}(\theta) = \text{true} \) iff \( \Delta\text{-expanded}(\theta) = \text{true} \) and \( \text{expanded}(\theta_i) = \text{true} \) 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 lead 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.

$\text{expand-tfs}$ is the main function that initializes TE. The while loop is executed until the TFS $\theta$ is expanded or resolved (see below). Several passes may be necessary for recursive TFS.

\[
\text{expand-tfs}(\theta) := \\
\quad \text{while not } (\text{expanded}(\theta) \text{ or } \text{resolved}(\theta) \text{ or } \\
\quad \quad \text{no unification occurred in the last pass}) \\
\quad \text{depth-first-expand}(\theta). \text{ /* or types-first-expand}(\theta), \text{ resp. */}
\]

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

\[
\text{depth-first-expand}(\theta) := \\
\quad \text{if } \theta \text{ has been already visited in this pass} \\
\quad \quad \text{then return} \\
\quad \quad \text{else if } \theta = \langle \tau, \{\theta_1, \ldots, \theta_n\} \rangle \\
\quad \quad \quad \text{then for every } \theta \in \{\theta_1, \ldots, \theta_n\} : \text{depth-first-expand}(\theta) \\
\quad \quad \quad \text{else } /* \theta = \langle \tau, \{f_1 = \theta_1, \ldots, f_n = \theta_n\} \rangle */ \\
\quad \quad \quad \text{for every } \theta \in \{\theta_1, \ldots, \theta_n\} : \text{depth-first-expand}(\theta) \\
\quad \quad \quad \text{if not } \Delta\text{-expanded}(\theta) \text{ then } \text{unify-type-and-node}(\tau, \theta). \\
\]

$\text{unify-type-and-node}$ destructively unifies $\theta$ with the expanded TFS of $\tau$.

\[
\text{unify-type-and-node}(\tau, \theta) := \\
\quad \text{if } \tau = \neg \sigma \\
\quad \quad \text{then } \text{unify-tfs} (\text{negate-fs} (\text{expand-type}(\sigma, \text{index})), \theta) \\
\quad \quad \text{else } \text{unify-tfs} (\text{expand-type}(\tau, \text{index}), \theta); \\
\quad \Delta\text{-expanded}(\theta) := \text{true}. \\
\]

We adapt Smolka’s treatment of negation for our TFS [Smolka, 1989]. Note that we only depict the conjunctive case here.
\[\text{negate-fs}(\theta = \langle \tau, \{ f_1 \mapsto \theta_1, \ldots, f_n \mapsto \theta_n \} \rangle) :=
\text{return } \langle T, \{ \{ \neg \tau, \emptyset \}, \{ f_1 \mapsto \neg \text{negate-fs}(\theta_1) \}, \ldots, \{ f_n \mapsto \neg \text{negate-fs}(\theta_n) \} \rangle \rangle.\]

### 3.2 Indexed Prototype Memoization

The basic idea of memoization 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 an lexicon entry or a rule) and an arbitrary index that allows to access different TFS of the same type which may be expanded in different ways (e.g., partially or fully). Such feature structures are called *prototypes*.

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

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

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

Some empirical results show the usefulness of indexed prototype memoization. The table on page 8 contains statistical information about the expansion of an HPSG grammar with approx. 900 type definitions. About 250 lexicon entries and rules have been expanded from scratch, i.e., all types are unexpanded (are skeletons) at the beginning.

The measurements show that memoization speeds up expansion by a factor of 5/10 for this grammar (this factor is directly related to the number of unifications). The time difference between the memoized and non-memoized algorithm may be even bigger if disjunctions are involved. The sample grammar contains only a few disjunctions.

### 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 “call stack” of \text{expand-type} to check whether a type is recursive or not.
<table>
<thead>
<tr>
<th>algorithm</th>
<th>depth–1st–expand</th>
<th>types–1st–expand</th>
<th>depth–1st–expand</th>
<th>types–1st–expand</th>
</tr>
</thead>
<tbody>
<tr>
<td>memoization</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>time (secs)</td>
<td>45</td>
<td>23*</td>
<td>46</td>
<td>23*</td>
</tr>
<tr>
<td>unifications</td>
<td>27221</td>
<td>14495*</td>
<td>27207</td>
<td>14481*</td>
</tr>
<tr>
<td>number of calls to expand-type</td>
<td>853</td>
<td><em>cons</em></td>
<td>260</td>
<td><em>cons</em></td>
</tr>
<tr>
<td>*: with types pre-expanded</td>
<td>316</td>
<td>cat-type</td>
<td>147</td>
<td><em>diff-list</em></td>
</tr>
<tr>
<td></td>
<td>269</td>
<td><em>diff-list</em></td>
<td>143</td>
<td>morph-type</td>
</tr>
<tr>
<td></td>
<td>243</td>
<td>morph-type</td>
<td>94</td>
<td><em>cons</em></td>
</tr>
<tr>
<td></td>
<td>208</td>
<td>atomic-wff</td>
<td>83</td>
<td>sort-expr</td>
</tr>
<tr>
<td></td>
<td>202</td>
<td>rp-type</td>
<td>71</td>
<td>atomic-wff</td>
</tr>
<tr>
<td></td>
<td>146</td>
<td>conj-wff-type</td>
<td>62</td>
<td>rp-type</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>var-type</td>
<td>53</td>
<td>subwff-inst</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>indexed-wff</td>
<td>53</td>
<td>cat-type</td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>nmorph-head</td>
<td>46</td>
<td>sign-type</td>
</tr>
</tbody>
</table>

(see Section 3.4). Each call of expand-type(τ, index) will push τ onto the call stack. This stack then is passed to expand-tfs.

If a type τ on top of the call stack also occurs below in the stack

\[(τ, σ_n, \ldots, σ_1, τ, ρ_m, \ldots, ρ_k)\]

we immediately know that the types τ, σ_n, ..., σ_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 call stack contains more than one element. Otherwise, prototype memoization would loop.

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

### 3.4 Example

In the following, we define a finite state machine [Krieger et al., 1993] 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).
\[
\begin{align*}
\text{list} & \Rightarrow \{\text{cons}, \emptyset\} \\
\text{cons} & \Rightarrow \left[ \begin{array}{c}
\text{FIRST} \\
\text{REST} \leftarrow \text{list}
\end{array} \right] \quad \text{we abbreviate cons via } \{\ldots\} \\
\text{non-final-config} & \Rightarrow \left[ \begin{array}{c}
\text{INPUT} \ \langle 1, 2 \rangle \\
\text{EDGE} \ \leftarrow 1 \\
\text{NEXT} \leftarrow \text{INPUT}[2]
\end{array} \right] \\
\text{final-config} & \Rightarrow \left[ \begin{array}{c}
\text{INPUT} \ \langle \emptyset \rangle \\
\text{EDGE} \ \leftarrow \text{undef} \\
\text{NEXT} \leftarrow \text{undef}
\end{array} \right] \\
\text{state1} & \Rightarrow \left[ \begin{array}{c}
\text{non-final-config} \\
\text{EDGE} \ \$1\{a, \{a, b\}\} \\
\text{NEXT} \leftarrow \$1\{\text{state1, final-config}\}
\end{array} \right] \\
\text{ab} & \Rightarrow \left[ \begin{array}{c}
\text{state1} \\
\text{INPUT} \ \langle a, b \rangle
\end{array} \right]
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>step</th>
<th>expand-type</th>
<th>in type</th>
<th>under path</th>
<th>call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textit{cons}</td>
<td>\textit{ab}</td>
<td>INPUT.REST</td>
<td>(ab)</td>
</tr>
<tr>
<td>2</td>
<td>\textit{list}</td>
<td>\textit{cons}</td>
<td>REST</td>
<td>(cons ab)</td>
</tr>
<tr>
<td>3</td>
<td>\textit{cons}</td>
<td>\textit{list}</td>
<td>e</td>
<td>(list cons ab)</td>
</tr>
<tr>
<td>4</td>
<td>\textit{cons}</td>
<td>\textit{ab}</td>
<td>INPUT</td>
<td>(ab)</td>
</tr>
<tr>
<td>5</td>
<td>\textit{state1}</td>
<td>\textit{ab}</td>
<td>e</td>
<td>(ab)</td>
</tr>
<tr>
<td>6</td>
<td>\textit{state1}</td>
<td>\textit{state1}</td>
<td>NEXT</td>
<td>(state1 ab)</td>
</tr>
<tr>
<td>7</td>
<td>\textit{final-config}</td>
<td>\textit{state1}</td>
<td>NEXT</td>
<td>(state1 ab)</td>
</tr>
<tr>
<td>8</td>
<td>\textit{non-final-config}</td>
<td>\textit{state1}</td>
<td>e</td>
<td>(state1 ab)</td>
</tr>
<tr>
<td>9</td>
<td>\textit{cons}</td>
<td>\textit{non-final-config}</td>
<td>INPUT</td>
<td>(non-final-config state1 ab)</td>
</tr>
<tr>
<td>10</td>
<td>\textit{state1}</td>
<td>\textit{ab}</td>
<td>NEXT</td>
<td>(ab)</td>
</tr>
</tbody>
</table>

The result of \textit{expand-type(ab)} is the following feature structure:
If we ran our automaton on the input abb,

\[
abb \Rightarrow \left[ \text{state1} \right]
\]

it would be rejected: \( \text{expand-type}(abb) \Rightarrow \text{fail} \).

### 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 \text{expand-tfs} call. The following control keywords have been implemented so far.

- \text{expand-function} \{depth|types\}-\text{first-expand} specifies the basic expansion algorithm.
- \text{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 \( \leq \) (checked in \text{unify-type-and-node}).
- \{\text{expand}|\text{expand-only}\} \{ \{type | (type [index [pred]])\} \{path\}^+ \}^*
  There are two mutually exclusive modes concerning expansion of types. If the \text{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 \text{expand} list is specified, all types will be expanded (checked in \text{unify-type-and-node}).
- \text{maxdepth integer} specifies that all types at paths longer than integer will be delayed anyway (checked in \text{unify-type-and-node}).
- \text{attribute-preference} \{attribute\}^* defines a partial order on attributes that will be considered in the functions \text{depth-first-expand} and \text{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 to exploit numerical preferences to speed up unification. Both keywords control the use of this information in functions \texttt{depth-first-expand} and \texttt{types-first-expand}.

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

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

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

```lisp
(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 definition is undecidable in general, i.e., there is no complete algorithm that halts on arbitrary input (TFS) and decides whether a description is satisfiable or not (see Section 4). However, there are several ways to stop infinite expansion in our framework:

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

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

- The third method is to define \texttt{:delay} patterns or to select the \texttt{: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 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 having 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, [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).

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

Semantically, we can formally account for such recursive feature descriptions (with respect to a type system) in different ways: either directly on the descriptions, or indirectly through a transformational approach into (first-order) logic. Both approaches rely on the construction of a fixpoint over a certain continuous function.\footnote{In both cases, there is, in general, more than one fixpoint, but it seems desirable to choose the greatest one.} 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 Moeshier, 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):
\[
\forall x.\text{cyc–list}(x) \leftrightarrow \exists y, z. \text{list}(x) \land \text{FIRST}(x, y) \land \text{REST}(x, z) \land y \neq 1 \land z \neq x
\]

5 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, 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.

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. the results of our grammar example on page 8). 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 to delay the evaluation of a relation until the specified parameters are instantiated.

6 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 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 unification, parsing, etc. The algorithm, as presented in the paper, has been fully implemented within TDL [Krieger and Schäfer, 1994] and is an integrated part of the DISCO system [Uszkoreit et al., 1994].

References


