An Efficient Typed Feature Structure Index: Theory and Implementation

Bernd Kiefer and Hans-Ulrich Krieger
German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
{kiefer,krieger}@dfki.de

Abstract

Many applications (not necessarily only from computational linguistics), involving record- or graph-like structures, would benefit from a framework which would allow to efficiently test a single structure ϕ under various operations $\odot$ against a compact representation $\Im$ of a set of similar structures: $\phi \odot \Im$. Besides a Boolean answer, we would also like to see those structures stored in $\Im$ which are entailed by operation $\odot$. In our case, we are especially interested in $\odot$s that implement feature structure subsumption and unifiability. The urgent need for such a kind of framework is related to our work on the approximation of (P)CFGs from unification-based grammars. We not only define the mathematical apparatus for this in terms of finite-state automata, but also come up with an efficient implementation mostly along the theoretical basis, together with measurements in which we compare our implementation of $\Im$ against a discrimination tree index.

1 Introduction and Motivation

There exist several applications in which a single feature structure (FS) is tested against a large set of structures with operations like *forward and backward subsumption*, *unifiability*, or *containedness*.

One possibility to optimize this task is to separate the set into subsets with disjoint properties that help to quickly cut down the number of structures under consideration. This is usually called *indexing* of feature structures and several proposals have been published on how to achieve this efficiently, namely, (Goetz et al., 2001), (Kiefer and Krieger, 2002), (Ninomiya and Makino, 2002), and (Munteanu, 2003). All approaches use more or less the same strategy: *select a set of paths from the FSs, extract the values/types at the end of these paths, and then build indexing information based on these types*. The approaches differ mostly in the last step, viz., on how the information of the set of types is encoded or exploited, resp.

In cases where the structures originate from applications like constraint-based natural language processing, they are often structurally similar to one another. In graph-based implementations, all the above mentioned operations require a traversal of the graph structure and the execution of appropriate tests at the edges or nodes. It seems natural to exploit this similarity by avoiding multiple traversals, packing the information of the elements such that the tests can be performed for a whole set of structures at once.

With this idea in mind, we propose a new method to store a large set of FSs in a very compact form, which is more memory efficient than storing the set of single structures. The resulting data structure $\Im$ reminds us of *distributed disjunctions* (Maxwell III and Kaplan, 1991). Mathematically, we will characterize $\Im$ and the required operations in terms of finite automata and operations over these automata, will provide implementation details of the packed index data structure, and will compare $\Im$ with the index used in (Kiefer and Krieger, 2004).

The motivation to study this task arose from experiments to approximate a large-coverage constraint grammar, the LinGO ERG (Flickinger, 2011), in terms of CFGs. Besides this, feature structure indexing can and has also been used for

- feature structure operations for extracting context-free grammars (**this paper**);
- lexicon and chart lookup during parsing or generation, or for accessing FS tree banks;
- establishing a lookup structure for a semantic RDF triple repository.

The structure of this paper is as follows. In section 2, the application that motivates this work is described. After that, a mathematical characterization of the data structure and algorithms in terms of finite state automata and operations thereon is presented in section 3. The implementation and its relation to the mathematical apparatus is covered in section 4. Section 5 contains measurements which compare the im-

plementation of the new index to the discrimination tree index from (Kiefer and Krieger, 2004).

2 Our Application: CF Approximations

The need for this data structure arose from an attempt to compute a context-free approximation of the current LinGO ERG HPSG grammar (Flickinger, 2011). This grammar is one of the most complex constraint-based grammars ever written, containing 91934 types, 217 rules, and 36112 lexical types, resulting in a very high coverage of the English language.
The first tests were run using a reimplementation of (Kiefer and Krieger, 2004) in a 32-bit Java environment, all ending in memory overflow. After a reduction of the initial (lexicon) structures, the next test was manually cancelled after two weeks which made the need for a more efficient way of handling the vast amount of FSs and operations obvious. The algorithm outlined below is described in more detail in (Kiefer and Krieger, 2004), including various optimizations.

The Approximation Algorithm. Approximation starts by applying the available rules of a grammar to the already existing lexical entries, thereby generating new FSs which, again, are used as rule arguments in new rule instantiations. This process is iterated, until no further structures are computed, i.e., until a fixpoint is reached. For this process to terminate, it must be ensured that the FSs will not continue to grow forever. This is achieved by two means.

Applying restrictors. Restrictors are functions that delete or transform parts of a feature structure, and are applied to get rid of features that encode the constituent tree, or parts that will only increase the number of structures without imposing constraints during parsing, e.g., terminal string labels in lexicon entries. More than one restrictor may be used, e.g., to take proper care of lexical vs. phrasal structures.

Reducing the set of FSs under subsumption. The set of FSs T may not contain elements which are comparable under subsumption, i.e., there are no FS $\phi, \phi' \in T$ s.t. $\phi \sqsubseteq \phi'$. This implies that a newly generated FS ψ must be checked against T, and in case ψ is subsumed by some element from T, it is discarded; conversely, all elements that are subsumed by ψ must be found and removed from T.

The informal algorithm from above makes use of an index structure $\Im$ to efficiently implement T, using the following operations thereon:

1. find potential unifiable members for an input feature structure during rule instantiations;
2. check if a new FS is subsumed by a structure in the index after a new structure has been created;
3. add a non-subsumed structure, and remove all current members which are subsumed by the new one.

It is worth noting that operation 1 is only an imperfect filter in all implementations, i.e., full unification still has to be performed on all structures that are returned, and there will still be unification failures for some structures. Contrary to other approaches, the subsumption operations in our implementation return all and only all correct answers.

3 A FSA Characterization of the Index

We start this section with some recap in order to motivate our decisions why we have deviated from some of the standard definitions. This includes a special definition of typed feature structures (TFS) as deterministic finite state automata (deterministic FSA or DFSA) which replaces the type decoration of nodes by additional type labels, attached to new edges.
Given the DFSA characterization of typed feature structures, we define TFS unification and subsumption in terms of operations on the corresponding automata and their recognized languages.
We then present the finite-state characterization of the index, focussing on the integration of new TFSs.
After that, we describe typed feature structure subsumption of an input query (a potentially underspecified TFS) against the index. This includes the definition of an effective procedure that constructs answer automata with their enclosing result structures.

3.1 Edge-Typed Feature Structures

In line with early work by Shieber on the PATR-II system (Shieber, 1984) and work by Kasper & Rounds (Kasper and Rounds, 1986; Rounds and Kasper, 1986) for the untyped case, Carpenter defines a *typed feature structure* (without disjunctions or sets) as a kind of *deterministic finite state automaton* (Hopcroft and Ullman, 1979) with the following signature (Carpenter, 1992, p. 36):

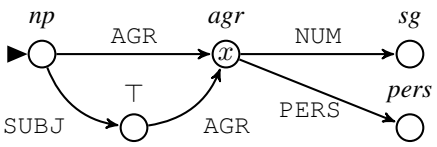
Definition 1 (TYPED FEATURE STRUCTURE)
A (conjunctive) typed feature structure (TFS) over a

finite set of types $\mathcal{T}$ and a *finite* set of features $\mathcal{F}$ is a quadruple $\langle Q, q_0, \theta, \delta \rangle$, such that

- Q is a *finite* set of nodes,
- $q_0 \in Q$ is *the* root node,
- $\theta : Q \to \mathcal{T}$ is a *total* node typing function, and
- $\delta : Q \times \mathcal{F} \to Q$ a *partial* feature value function.

The use of θ and δ thus allows us to attach labels (types and features) to nodes and edges of an automaton, representing a TFS. We note here that this definition does *not* employ a distinguished set of *final* states F. When extending the TFS model in Definition 3, we will assume that F always refers to the set of all *leaf nodes*, nodes that do not have outgoing edges. This will turn out important when we characterize TFS unification and subsumption as operations over the languages recognized by the FSA. The following TFS represents an underspecified singular NP.

Example 1 (FEATURE STRUCTURE)



We often use an alternative representation to depict TFSs, so-called *attribute-value matrices* (AVMs). *Reentrancies* in the above automaton are expressed through logical variables in the AVM (here: x).

Example 2 (ATTRIBUTE-VALUE MATRIX)

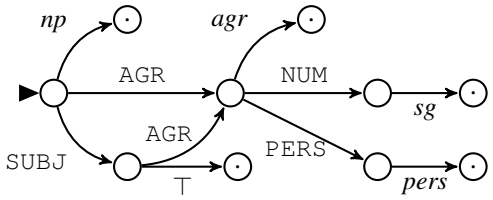
$$\begin{bmatrix} np & \\ \text{AGR} & \boxed{x} \begin{bmatrix} agr & \\ \text{NUM} & sg \\ \text{PERS} & pers \end{bmatrix} \\ \text{SUBJ} \mid \text{AGR} & \boxed{x} \end{bmatrix}$$

In the following, we no longer distinguish between TFSs and AVMs as they denote the same set of objects (there exists a trivial bijection between TFSs and AVMs). We also make use of the notion of a *type hierarchy*, underlying a typed feature structure and operations thereon, such as typed feature structure *unification* and *subsumption*.

Definition 2 (TYPE HIERARCHY)
Let $\mathcal{T}$ be a finite set of types and let $\leq$ be a binary relation over $\mathcal{T}$, called *type subsumption*. A decidable partial order $\langle \mathcal{T}, \leq \rangle$ then is called a type hierarchy (or an inheritance hierarchy). $\mathcal{T}$ contains two special symbols: $\top$ is called the *top type* (or the most general type) and $\bot$ is called the *bottom type* (or the most specific type), according to $\leq$.

We will now deviate from the TFS definition in (Carpenter, 1992) in that we move away from the *node-oriented* type representation to an *edge-based* implementation whose set of features is now given by $\mathcal{F} \uplus \mathcal{T}$. Thus, we have to make sure that $\mathcal{T}$ and $\mathcal{F}$ are disjoint (as can be easily realized via renaming). This extension results in *deterministic* automata with *directly interpretable* edge labels and *unlabeled* nodes. The following TFS depicts this modification when applied to the TFS from Example 1. Note that we have not renamed the types here, but have used a different style (*italics*) in order to distinguish them from the original features (`typewriter`).

Example 3 (FEATURE STRUCTURE, EXTENDED)



Given the above edge-based representation, we can now define an *edge-typed feature structure* (ETFS), mirroring exactly this modification.

Definition 3 (EDGE-TYPED FS)
An edge-typed feature structure (ETFS) over a *finite* set of types $\mathcal{T}$ and a *finite* set of features $\mathcal{F}$ is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, such that

- Q is a *finite* set of nodes,
- $\Sigma = \mathcal{F} \uplus \mathcal{T}$ is a *finite* input alphabet,
- $\delta : Q \times \Sigma \to Q$ is a *partial* feature value function,
- $q_0 \in Q$ is *the* root node, and
- $F = \{q \in Q \mid \delta(q, a) \uparrow, \text{ for all } a \in \Sigma\}$ is the *finite* set of final states.

We notice here that F corresponds exactly to the set of *leaf* nodes, nodes *without* outgoing edges, and that the total typing function θ is no longer needed.

We finally define a stronger class of ETFSs by enforcing *acyclicity*, meaning that these structures are only able to recognize *finite* languages. This assumption makes the definition of the index together with index unification and subsumption easy to understand. It is also worth noting that the applications in which we are interested (parsing with HPSG grammars (Pollard and Sag, 1994) and grammar approximation (Kiefer

and Krieger, 2004)) do forbid such cyclic TFS. In case cyclic structure result from a unification of two TFSs, the Tomabechi-style unification engines that we were using in our systems will signal a failure in the final copy phase.

Definition 4 (ACYCLIC ETFS)
An acyclic edge-typed feature structure is an ETFS which does not allow for infinite paths:

$$\nexists q \in Q, \nexists f \in \mathcal{F}^* \,.\, \hat{\delta}(q, f) = q$$

Recall, $\Sigma = \mathcal{F} \uplus \mathcal{T}$, so when we say *path* here, we usually refer to elements from $\mathcal{F}^*$ (but *not* from $\mathcal{F}^*\mathcal{T}$, for which we use the term *extended path*). Note that $\hat{\delta}$ above refers to the usual extension of δ, when moving from Σ to Σ^* (Hopcroft and Ullman, 1979, p. 17):

- $\hat{\delta}(q, \epsilon) := q$
- $\hat{\delta}(q, wa) := \delta(\hat{\delta}(q, w), a)$, for $q \in Q, w \in \Sigma^*$, and $a \in \Sigma$

3.2 TFS Unification and Subsumption

Given Definition 3, we are now able to define *ETFS subsumption* $\sqsubseteq$ and *ETFS unification* $\sqcap$ of two ETFSs ϕ and ψ in terms of the languages, recognized by ϕ and ψ. Since the types, represented by the typed edges, need to be interpreted against an inheritance hierarchy $\langle \mathcal{T}, \leq \rangle$, we first define the *recognized pre-language* of an ETFS. Due to space requirements, we restrict ourselves in the paper (but not in the oral presentation) to *coreference-free ETFSs*, as a proper handling of coreferences would require a further modification of the ETFS definition (edges in the DFSA need to be labelled with sets of elements from $\mathcal{F}^*$, expressing equivalence classes).

Definition 5 (ETFS PRE-LANGUAGE)
The pre-language $\mathcal{L}^-(\phi)$ recognized by an ETFS ϕ is defined as follows:

$$\mathcal{L}^-(\phi) := \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Now, the type hierarchy $\langle \mathcal{T}, \leq \rangle$ comes into play when defining the *recognized language* of an ETFS ϕ. Essentially, we "expand" type t at the end of each word from $\mathcal{L}^-$ by $\{s \mid s \leq t\}$.

Definition 6 (ETFS LANGUAGE)
The language $\mathcal{L}(\phi)$ recognized by an ETFS ϕ is defined as follows:

$$\mathcal{L}(\phi) := \{vs \in \Sigma^* \mid vt \in \mathcal{L}^-(\phi) \textit{ and } s \leq t\}$$

Note that the languages we are dealing with are not only regular, but even *finite*, due to the acyclicity assumption, imposed on ETFSs (see Definition 4).

In order to define ETFS subsumption and unification, we need a further definition that introduces an operator Π that basically *chops off* the types at the end of extended paths.

Definition 7 (STRIPPED-DOWN FS)
A stripped-down FS can be obtain from an ETFS ϕ by applying Π to $\mathcal{L}(\phi)$, where

$$\Pi(\mathcal{L}(\phi)) := \{w \in \mathcal{F}^* \mid ws \in \mathcal{L}(\phi) \textit{ and } s \in \mathcal{T}\}$$

We are now almost ready to define the usual ETFS operations. Before doing so, need to talk about the *unique paths* (depicted by $\mathcal{L}_=$) and *shared paths* (depicted by $\mathcal{L}_\neq$), relative to ϕ and ψ.

Definition 8 (UNIQUE AND SHARED PATHS)
Given two ETFSs ϕ and ψ, we deconstruct their corresponding languages as follows:

$$\mathcal{L}(\phi) := \mathcal{L}_=(\phi) \uplus \mathcal{L}_\neq(\phi)$$
$$\mathcal{L}(\psi) := \mathcal{L}_=(\psi) \uplus \mathcal{L}_\neq(\psi)$$

such that

$$\Pi(\mathcal{L}_=(\phi)) = \Pi(\mathcal{L}_=(\psi))$$

Thus

$$\mathcal{L}_\neq(\phi) = \mathcal{L}(\phi) \setminus \mathcal{L}_=(\phi)$$
$$\mathcal{L}_\neq(\psi) = \mathcal{L}(\psi) \setminus \mathcal{L}_=(\psi)$$

Definition 9 (ETFS SUBSUMPTION/UNIFICATION)

$$\phi \sqsubseteq \psi :\Longleftrightarrow \mathcal{L}_=(\phi) \subseteq \mathcal{L}_=(\psi) \textit{ and } \mathcal{L}_\neq(\psi) = \emptyset$$

$$\phi \sqcap \psi = \pi :\Longleftrightarrow \mathcal{L}(\pi) = \mathcal{L}_\neq(\phi) \cup \mathcal{L}_\neq(\psi) \cup \{wu \mid ws \in \mathcal{L}_=(\phi), wt \in \mathcal{L}_=(\phi), \textit{ and } u = s \wedge t\}$$

It is worth noting that the unification operation directly above assumes that the underlying type hierarchy is GLB-completed as it assumes a unique (and not many) u, resulting from taking the GLB $s \wedge t$. We further note that if $s \wedge t$ fails at any point (i.e., returns $\bot$), π is supposed to be inconsistent.

3.3 A TFS Index

In this subsection, we will establish a typed feature structure index $\Im$ that arises from a collection of ETFSs $\{\phi_1, \ldots, \phi_n\}$ by taking the *union* of the corresponding automata, an operation that can be *effectively* constructed, resulting in, what we call later, an *answer automaton*:

$$\Im = \bigsqcup_{i=1}^{n} \phi_i$$

With *effectively*, we mean here and in the following that there exists an algorithm that directly constructs

the resulting DFSA (the *answer* FSA) from the input DFSAs. When an ETFS ϕ has already been inserted in $\Im$, we write

$$\phi \in \Im$$

In order to address all this properly, we need to slightly extend the signature of an ETFS by a total indexing function ι defined on F, enumerating those ETFSs $\phi \in \iota(q)$ at a final node $q \in F$, such that $\hat{\delta}(q_0, w) = q$ *and* $w \in \mathcal{L}^-(\phi)$. Thus, ι records TFSs at q that share an *extended path* $w = ft$ from root node q_0 to q ($f \in \mathcal{F}^*, t \in \mathcal{T}$).

As we we will see in a moment, ι will be utilized to return *all* and *only all* TFSs recorded in the index $\Im$ that are more specific (or equal) than or which are unifiable with a given *query* TFS. Due to space requirements, we restrict ourself here to coreference-free TFSs as this simplifies the formal description of the index.

Definition 10 (ETFS, REVISED)
An edge-typed feature structure (ETFS) is a sextuple $\langle Q, \Sigma, \delta, q_0, F, \iota \rangle$, where Q, Σ, δ, q_0 and F is given in Definition 3 and ι defined as follows:

- $\iota : F \to 2^I$ is a *total* indexing function.

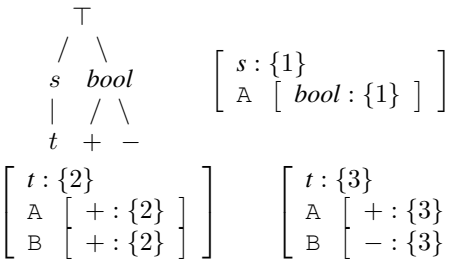
As we see from this definition, ι does not access the feature structures directly. Instead, it utilizes a set of IDs I that identify the corresponding ETFSs Φ from the index through the use of an additional *bijective* function, viz., $id : I \to \Phi$ and $id^{-1} : \Phi \to I$.

This strategy gives an implementation the freedom to relocate the TFSs to a separate table in RAM or even to store them in an external file system. Since it is possible to *reconstruct* feature structures from the index $\Im$, given a specific ID from I, we can even use a memory-bounded internal cache to limit the memory footprint of $\Im$, which is the method of choice in the implementation described in section 2.

When entering a TFS ϕ to the index, a brand-new identifier from I is obtained and added to every final node of ϕ. This assignment is important when we define subsumption and unification over the index in Section 3.4. In the end, the index $\Im$ is still a ETFS/DFSA, but its ι-function differs from a single TFS in that it does *not* return a singleton set, but usually a set with more than one ID.

Before moving on, let us take an example that explains the ideas, underlying the index. The example will display the index after three ETFSs have been added. To ease the pictures here, we will not use the DFSA model of ETFSs, but instead employ the corresponding equivalent AVM description from Example 2. As already explained, types occur as "ordinary" features, and the AVM for the index usually comes with more than one outgoing type features at every node in the DFSA. The set of IDs that refer to those TFSs "entailed" by a current extended path at a final node in the index are attached (via ':') to the type features in the AVMs below.

Example 4 (INDEX WITH THREE ETFSS)
We assume a type hierarchy with the following six types ("lower" depicted types are more specific) and will add the following three ETFSs to an empty index



The resulting index $\Im$, integrating 1, 2, and 3, is:

$$\Im \equiv \left[\begin{array}{l} s : \{1\} \\ t : \{2,3\} \\ \texttt{A} \left[\begin{array}{l} bool : \{1\} \\ + : \{2,3\} \end{array}\right] \\ \texttt{B} \left[\begin{array}{l} + : \{2\} \\ - : \{3\} \end{array}\right] \end{array}\right]$$

What this example shows is that the index does in fact result from taking the *union* of the three DFSAs/ETFSs. We also see that the associated ID sets at the final nodes of the index are extended by the IDs of the ETFSs that are going to be inserted.

In order to obtain the *complete* set of ETFSs Φ stored in the index, we need to walk over the set of final nodes and take the union of *all* ID sets (actually, the TFSs associated with the indices):

$$\Phi = \bigcup_{q \in F} \bigcup_{i \in \iota(q)} \{id(i)\}$$

3.4 Querying the Index

We will now define two further natural operations w.r.t. index $\Im$ and a query ETFS ψ, and will present a construction procedure for one of them, viz., $\Im \sqsubseteq \psi$:

1. *return all ϕ_i which are equal or more specific than query ψ*:
   $\Im \sqsubseteq \psi :\Longleftrightarrow \cup_i \phi_i$ *s.t.* $\phi_i \sqsubseteq \psi$ *and* $\phi_i \in \Im$

2. *return all ϕ_i which are unifiable with query ψ*:
$\psi \sqcap \Im :\Longleftrightarrow \cup_i \phi_i$ *s.t.* $\phi_i \sqcap \psi \neq \bot$ *and* $\phi_i \in \Im$

(1.) essentially reduces to the construction of a subautomaton, whereas (2.) results in the construction of an intersecting FSA, again two operations that can be *effectively* constructed. By this, as before, we mean that we can directly construct a new ETFS/DFSA from the *signatures* (see Definition 10) of $\Im$ and ψ. Note that we let the indexing function ι^ψ for query ψ always map to the empty set, as it is a query, and *not* a TFS that is part eof the index, i.e., $\iota^\psi : F^\psi \rightarrow \emptyset$.
Since the result of these two operations are again DFSAs, we call them *answer automata*, as the subsumed or unifiable structures can be obtained through the use of ι and *id*—the resulting FSA, *as such*, is *not* the answer.

Before presenting the construction procedure for $\Im \sqsubseteq \psi$, we need to report on two important observations. Firstly, the following equality relations always hold for the resulting structure $\cup_i \phi_i$ w.r.t query ψ:

$$\forall i \;.\; \Pi(\mathcal{L}^-_=(\psi)) = \Pi(\mathcal{L}^-_=(\phi_i))$$

In other words, the subsumed ETFSs ϕ_i from $\Im$ must at least contain the *same* paths from $\mathcal{F}^*$ than query ψ, and perhaps come up with more specific type labels (and, of course, additional *own* unique paths).

Secondly, if $\phi \in \Im$ *and* $w, w' \in \mathcal{L}^-(\phi)$ s.t. $w = ft$ and $w' = ff't'$ ($f \in \mathcal{F}^*; f' \in \mathcal{F}^+; t, t' \in \mathcal{T}$), then $id^{-1}(\phi) \in \iota(q)$ *and* $id^{-1}(\phi) \in \iota(q')$, given $\hat{\delta}(q_0, w) = q$ and $\hat{\delta}(q_0, w') = q'$ (q_0 being the root node of $\Im$). I.e., a recorded ETFS ϕ in $\Im$ with index $id^{-1}(\phi)$ under path f will also be recorded with the same index under the *longer* path ff'.

Given these two remarks, it thus suffices to construct an answer automaton $\Im \sqsubseteq \psi$ that is *structural equivalent* to ψ and whose ι-function is no longer empty, but instead constructed from ι of $\Im$ w.r.t. a type hierarchy.

Algorithm 1 (INDEX SUBSUMPTION)
Given an index $\Im = \langle Q^\Im, \Sigma^\Im, \delta^\Im, q_0^\Im, F^\Im, \iota^\Im \rangle$, a query $\psi = \langle Q^\psi, \Sigma^\psi, \delta^\psi, q_0^\psi, F^\psi, \iota^\psi \rangle$, and a type hierarchy $\langle \mathcal{T}, \leq \rangle$, we define the *subsumed answer automaton*

$$\Im \sqsubseteq \psi := \langle Q, \Sigma, \delta, q_0, F, \iota \rangle$$

where $Q := Q^\psi, \Sigma := \Sigma^\psi, \delta := \delta^\psi, q_0 := q_0^\psi$, and $F := F^\psi$. Now let $\hat{\delta}(q_0, ft) = q \in F$, where $f \in \mathcal{F}^*$, $t \in \mathcal{T}$, and $ft \in \mathcal{L}^-(\psi)$. ι then is given by the following definition ($q \in F$):

$$\iota(q) := \bigcup_{s \leq t} \begin{cases} \emptyset, & \textit{if } \hat{\delta}^\Im(q_0^\Im, fs)\uparrow \\ \iota^\Im(\hat{\delta}^\Im(q_0^\Im, fs)), & \textit{otherwise} \end{cases}$$

The set $\Phi = \cup_i \phi_i$ of subsumed ETFSs ϕ_i w.r.t. ψ is finally given by

$$\Phi = \bigcap_{q \in F} \bigcup_{i \in \iota(q)} \{id(i)\}$$

It is worth noting that the transition function for fs is not necessarily defined in $\Im$, since query ψ might employ a feature from $\mathcal{F}$ and/or use a type labeling from $\mathcal{T}$ that is not literally present in $\Im$.

Let us now present a further example, showing how $\Im \sqsubseteq \psi$ looks for two queries w.r.t the index depicted in Example 4.

Example 5 (ANSWER FA FOR TWO QUERIES)
Given the index from Example 4 and queries

$$\psi_1 \equiv \begin{bmatrix} s \\ \texttt{A} \;\; + \\ \texttt{B} \;\; \textit{bool} \end{bmatrix} \text{ and } \psi_2 \equiv \begin{bmatrix} s \\ \texttt{A} \;\; \textit{bool} \\ \texttt{C} \;\; \textit{bool} \end{bmatrix}$$

the answer automata have the following structure:

$$\Im \sqsubseteq \psi_1 \equiv \begin{bmatrix} s : \{1\} \cup \{2,3\} = \underline{\{1,2,3\}} \\ \texttt{A} \;\; [\; + : \underline{\{2,3\}} \;] \\ \texttt{B} \;\; [\; \textit{bool} : \emptyset \cup \{2\} \cup \{3\} = \underline{\{2,3\}} \;] \end{bmatrix}$$

$$\Im \sqsubseteq \psi_2 \equiv \begin{bmatrix} s : \{1\} \cup \{2,3\} = \underline{\{1,2,3\}} \\ \texttt{A} \;\; [\; \textit{bool} : \{1\} \cup \{2,3\} \cup \emptyset = \underline{\{1,2,3\}} \;] \\ \texttt{C} \;\; [\; : \emptyset \cup \emptyset \cup \emptyset = \underline{\emptyset} \;] \end{bmatrix}$$

The indices *Id* for the ETFSs from $\Im$ hidden in these answer automata are given by the intersection of the ID sets associated with the final nodes (see Algor. 1):

$$Id(\Im \sqsubseteq \psi_1) = \{1,2,3\} \cap \{2,3\} \cap \{2,3\} = \underline{\{2,3\}}$$

$$Id(\Im \sqsubseteq \psi_2) = \{1,2,3\} \cap \{1,2,3\} \cap \emptyset = \underline{\emptyset}$$

I.e., ETFS 2 and 3 from Example 4 are subsumed by ψ_1, whereas *no* ETFS can be found in $\Im$ for ψ_2.

Due to space limitations, we are not allowed here to describe the answer automata for index construction $\phi \sqcup \Im$, for the inverse case of index subsumption $\psi \sqsubseteq \Im$, and for index unifiability $\psi \sqcap \Im$. This will be addressed in the oral presentation, where we will also indicate how coreferences in the theoretical description of the index are represented.

4 Implementing the Index

The proposed data structure exploits the similarity of the structures by putting all information into the tree structure of $\Im$, which contains all paths occurring in the stored TFSs. Now, ι is *implemented* by attaching *bit vectors* to the nodes, which is very compact since the member indices are themselves small integer numbers. In contrast to the mathematical description, they encode not only the presence of a type under a specific path, but also the presence or absence of other values, namely features and coreferences.

Our implementation can straightforwardly be used in a parallel execution environment, which allows to scale up the target application more easily with modern machines, as can be seen in the next section. Subsumption and generalization in our implementation always return correct results, while other indexing techniques require the full test to be applied on the results, thus thread-safe versions of the FS operations have to be provided by them. Since almost all unification engines draw their efficiency from invalidating intermediate results by increasing a global counter, concurrent evaluation is only possible at higher costs.[1] Our implementation provides multiple-read/single-write, beneficial for the application described in Section 2, as the amount of queries by far surmounts that of adding or removing structures.

To perform the operations, the index structure is traversed in a canonical order which in turn implies an order over all occurring paths and establishes a one-to-one correspondence between index nodes and nodes in the query feature structure. Because of this, when we subsequently talk about the *current node*, we always mean the pair of corresponding index and query structure node.

Boolean bit vector operations are employed to exclude invalid structures. As a starting point for traversal, a bit vector a is used, where all the bits, representing the current index feature structures, are set. When checking unifiability, for example, we collect at every current node all those defined types t which are incompatible with type s in the query structure ψ at the same current node q. We then use the stored bit vectors b_t^q that encode the index feature structures bearing t to remove them from the set of valid candidates a by executing $a \wedge (\bigwedge_t \neg b_t^q)$.

We describe the implementation of $\mathit{Id}(\Im \sqsubseteq \psi)$ and $\mathit{Id}(\psi \sqsubseteq \Im)$ in detail below, that is, determining TFSs from the index that are subsumed by (resp. subsuming) query structure ψ. The currently implemented unifiability method only deals with type constraints, and ignores coreferences.

Every member from $\Im$, whose type t is *not subsumed (not subsuming, resp.)* by type s in the current node of the query ψ, is removed. For the generalization case, all contexts with outgoing features missing in the query are removed. After that, the coreference constraints are evaluated. Coreference constraints (sets of paths leading to the same node in an FS) establish equivalence classes of index nodes (exploiting the one-to-one correspondence between query FS nodes and index nodes). We write $\mathcal{E}_\psi^q$ for the equivalence class at node q of the query structure ψ. These sets can be collected during traversal. A valid member $\phi \in \Im$ for a subsumption query ψ in terms of coreferences is one where $\mathcal{E}_\psi^q \subseteq \mathcal{E}_\phi^q$, for every q. This condition has to be checked only at nodes where a new coreference is introduced, whereas the information that nodes are coreferent has to be stored also at paths that reach beyond such an introduction node.

In the index, the sets $\mathcal{E}_\phi^q$ are only encoded implicitly. Every non-trivial equivalence class (non-singleton set) is replaced by its *representative* which is the lowest node in the canonical order that belongs to this class. Analogous to the bit vectors for types, there are bit vectors b_r^q for all members, using r as the representative at node q.

To test the subset condition above, we have to make sure that all members of $\mathcal{E}_\psi^q$ point to the same representative for some index member. Thus for the valid contexts, we have to compute $\bigvee_{r \in \mathcal{E}_\phi^q} (\bigwedge_{q' \in \mathcal{E}_\psi^q} b_r^{q'})$.

For the generalization case, computing that the equivalence class of the index is a subset of that in the query, we have to check if a representative node r in $\mathcal{E}_\phi^q$ is not an element of $\mathcal{E}_\psi^q$, and to remove all members where this holds by computing $a \wedge (\bigwedge_{r \in \mathcal{E}_\phi^q \wedge r \notin \mathcal{E}_\psi^q} \neg b_r^q)$.

5 Measurements

To be able to compare the performance of the two index structures, we have designed the following syn-

[1] We plan to implement a thread-safe unifier in the near future, which will enable us to assess a parallel version of the discrimination, tree. For the approximation, however, the packed index outperforms the discrimination tree even in sequential mode.

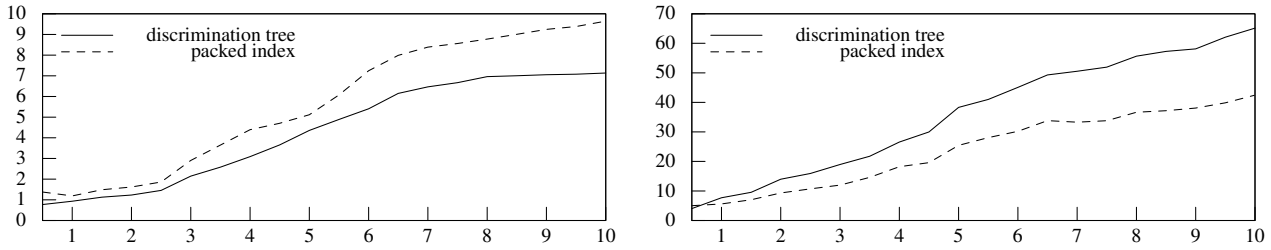


Figure 1: Time to get all subsumed (left) and all unifiable members (right) from the index. The graphs show the time needed for 10,000 operations (in 100 ms steps on the vertical axis) in relation to the index size ($\times 1,000$ members). Experiments were executed on a 2.4 GHz Quad-Core Opteron with 64GB main memory which only ran the test process. We would like to draw the reader's attention to the difference in scale on the vertical axes.

thetic experiment: 20 million feature structures generated during the approximation were dumped, from which three random sets were selected, such that indices of up to 10,000 TFSs are created, and three random sets of 10,000 query structures. The numbers from the resulting nine experiments were averaged to remove effects that are due to characteristics of the data set. Every index and query set was used to perform two of the operations executed in the CF approximation: (1) determining the set of all subsumed elements in the index and (2) returning all unifiable elements (see Figure 1). Where the index only acts as a filter, the time to compute the correct result is included, i.e., the full unification or subsumption. The indexing method we are comparing against is described in detail in (Kiefer and Krieger, 2004) and is an instance of discrimination tree indexing.

The performance of the subsumed members operation is slightly worse, while the unifiability test is superior as it avoids many of the costly full unification operations. At first sight, the graphs in figure 1 don't seem to indicate a large improvement. However, we ran these synthetic experiments to demonstrate the raw performance; When applying it to grammar approximation, the picture is quite different. The unifiability test is the first and therefore most important step, and together with the potential that it can be used almost effortlessly with low locking overhead in concurrent environments, the packed index by far outperforms the discrimination tree. To show this, we ran the approximation in three different setups, with the discrimination tree, and the packed index without and with parallelization. The numbers below show the *real time* needed to complete the third iteration of the fixpoint computation of the CF approximation:

1. discrimination tree: 139,279 secs
2. packed sequential: 49,723 secs ($2.8\times$ faster)
3. packed parallel: **15,309** secs ($9.1\times$ faster)

To measure the *space requirements*, we used the 59,646 feature structures at the end of this third iteration and, running the system in a profiling environment, stored it in both implementations. This gave us, subtracting the 144 MB that the profiler showed after loading the grammar, 103 MB for the packed and 993 MB for the discrimination tree index, a factor of **9.64**. As is true for most techniques that optimize feature structure operations, the effectiveness strongly depends on the way they are used in the application, e.g., the number of executions of the different operations, the implementation of basic functions like type unification or subsumption, etc. This means that the presented method, while very effective for our application, may have to be adapted by others to produce a significant gain. And there is still a lot of room for improvement, e.g., by combining the tree and packed index, or tuning the implementation of the bit set operations, which will often operate on sparse sets.

6 Summary and Outlook

In this paper, we have described a new indexing method for typed feature structures that deviates from the index techniques mentioned in the introduction. Our measurements have shown that the new method outperforms the discrimination tree index, at least when applied to our approximation experiments.

We note here that the new methods might also be important to other areas in computational linguistics, such as lexicon lookup for a large lexical data base or tree bank, unification-based parsing under packing, or even chart-based generation. Other areas involving record-like structures would also benefit from our approach and we envisage semantic repositories for storing RDF graphs, as similar operations to unifiability and subsumption are of importance to OWL.

Acknowledgements

The research described here was financed by the German Federal Ministry of Education and Research (BMBF) through the project *∂eepen∂ance* (contract no. 01IW11003) and by the EU FP7-ICT Programme projects ALIZ-E (grant agreement no. 248116) and TrendMiner (grant agreement no. 287863). The authors would like to thank the reviewers for their comments.

References

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

Dan Flickinger. 2011. Accuracy vs. robustness in grammar engineering. In E.M. Bender and J.E. Arnold, editors, *Language from a Cognitive Perspective: Grammar, Usage, and Processing*, pages 31–50. CSLI Publications, Stanford.

Thilo Goetz, Robin Lougee-Heimer, and Nicolas Nicolov. 2001. Efficient indexing for typed feature structures. In *Proceedings of Recent Advances in Natural Language Processing, Tzigov Chark*.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Robert T. Kasper and William C. Rounds. 1986. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, ACL-86*, pages 257–266.

Bernd Kiefer and Hans-Ulrich Krieger. 2002. A context-free approximation of Head-Driven Phrase Structure Grammar. In S. Oepen, D. Flickinger, J. Tsuji, and H. Uszkoreit, editors, *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*, pages 49–76. CSLI Publications.

Bernd Kiefer and Hans-Ulrich Krieger. 2004. A context-free superset approximation of unification-based grammars. In H. Bunt, J. Carroll, and G. Satta, editors, *New Developments in Parsing Technology*, pages 229–250. Kluwer Academic Publishers.

John T. Maxwell III and Ronald M. Kaplan. 1991. A method for disjunctive constraint satisfaction. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 173–190. Kluwer. Also available as report SSL-90-06, XEROX, System Sciences Laboratory, Palo Alto, 1990.

Cosmin Munteanu. 2003. Indexing methods for efficient parsing with typed feature structure grammars. In *Proceedings of the 6th Pacific Association for Computational Linguistics Conference*.

Takashi Ninomiya and Takaki Makino. 2002. An indexing scheme for typed feature structures. In *In Proceedings of the 19th International Conference on Computational Linguistics (COLING-02)*, pages 1248–1252.

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

William C. Rounds and Robert T. Kasper. 1986. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual Symposium of the IEEE on Logic in Computer Science*.

Stuart M. Shieber. 1984. The design of a computer language for linguistic information. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 362–366.