A Bag of Useful Techniques for Unification-Based Finite-State Transducers

Hans-Ulrich Krieger, Witold Drożdżyński, Jakub Piskorski, Ulrich Schäfer, Feiyu Xu
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
{krieger, witold, piskorsk, uschaefer, feiyu}@dfki.de

Abstract
We present several extensions to the shallow text processor SProUT, viz., (1) a fast imperfect unifiability test, (2) a special form of sets together with a polymorphic lazy and destructive unification operation, (3) a cheap form of negation, (4) a weak unidirectional form of coreferences, (5) optional context-free stages in the shallow cascade, (6) a compile time type check, (7) compile time transition sorting under subsumption, (8) several output merging techniques, and (9) a compaction technique for lexical resources. The extensions have been found relevant in several projects and might be of importance to other systems, even to deep processing.

1 Introduction
In the last decade, a strong tendency of deploying lightweight linguistic analysis to the conversion of raw textual information into structured and valuable knowledge can be observed. Recent advances in the areas of information extraction, text mining, and textual question answering demonstrate the benefit of applying shallow text processing techniques. Systems employing such shallow techniques are assumed to be considerably less time-consuming and more robust than deep processing systems, but are still sufficient to cover a broad range of linguistic phenomena (Hobbs et al., 1997).

This paper centers around several extensions to the shallow core engine of SProUT (Shallow Processing with Unification and Typed feature structures), a platform for the development of multilingual text processing systems (Becker et al., 2002; Drożdżyński et al., 2004). The extensions have been designed to either retain or even to speed up the run time performance of SProUT, and have been found useful in several other projects which employ SProUT to perform information extraction, hyperlinking, opinion mining, and text summarization. The extensions are worthwhile to be considered, not only by other shallow text processors, but even by deep processing engines.

1.1 SProUT
The motivation for developing SProUT came from the need to have a system that (i) allows a flexible integration of different processing modules and (ii) to find a good trade-off between processing efficiency and expressiveness of the formalism. On the one hand, very efficient finite-state devices have been successfully employed in real-world applications. On the other hand, unification-based grammars are designed to capture fine-grained syntactic and semantic constraints, resulting in better descriptions of natural language phenomena. In contrast to finite-state devices, unification-based grammars are also assumed to be more transparent and more easily modifiable. Our idea now was to take the best of these two worlds, basically having a finite-state machine that operates on typed feature structures (TFSs). Thus transduction rules in SProUT do not rely on simple atomic symbols, but instead on TFSs, where the left-hand side (LHS) of a rule is a regular expression over TFSs representing the recognition pattern, and the right-hand side (RHS) is a TFS specifying the output structure. Consequently, equality of atomic symbols is replaced by unifiability of TFSs and the output is constructed using TFS unification w.r.t. a type hierarchy.

1.2 Structure of Paper
The paper is structured as follows. The next section introduces XTDL, the formalism used in SProUT. Sections 3–11 then describe the extensions. Each of these sections explains the reasons for extending SProUT and estimates potential costs or even savings in terms of space and time, resp. We also try to motivate why these techniques might be of interest to other systems and paradigms.

2 XTDL—The Formalism in SProUT
XTDL combines two well-known frameworks: typed feature structures and regular expressions. We assume a basic familiarity with these concepts here.

2.1 The Basis: TDL
XTDL is defined on top of TDL, a definition language for TFSs (Krieger and Schäfer, 1994) that is used as a descriptive device in several grammar systems, such as LKB (Copestake, 1999), PAGE (Uszkoreit et al., 1994), or PET (Callmeier, 2000). We use the following fragment of TDL, including coreferences.

\[
\begin{align*}
\text{type-def} & \rightarrow \text{" : " atom } \\
\text{type} & \rightarrow \text{identifier}
\end{align*}
\]
Apart from the integration into the rule definitions, we also employ this fragment in \textit{SProUT} for the establishment of a type hierarchy of linguistic entities. In the example definition below, the \texttt{morph} type inherits from \texttt{sign} and introduces four morphosyntactically motivated attributes, together with their corresponding values.

\[
\text{morph} : = \text{sign} \land [\text{POS} \ \text{atom}, \ 
\text{STEM} \ \text{atom}, \ 
\text{INFL} \ \text{infl}, \ 
\text{SEGMENTATION} \ \ast \text{list}\ast].
\]

The next figure depicts a fragment of the type hierarchy used in the example.

\[
\begin{align*}
\text{atom} & \to \ast \text{top}\ast \\
\text{tense} & \to \ast \text{avm}\ast \land \ast \text{rule}\ast \\
\text{sign} & \to \ast \text{index-avm}\ast \\
\text{infl} & \to \ast \text{present}\ast \land \ast \text{token}\ast \\
\text{morph} & \to \ast \text{morph}\ast \land \ast \text{lang}\ast \land \ast \text{token-type}\ast \\
\text{de} & \to \ast \text{en}\ast \land \ast \text{separator}\ast \land \ast \text{url}\ast.
\end{align*}
\]

### 2.2 The Regular Extension: \textit{XTDL}

A rule in \textit{XTDL} is straightforwardly defined as a recognition pattern on the LHS, written as a regular expression, and an output description on the RHS. A label serves as a handle to the rule. Regular expressions over feature structures describe sequential entities. In the example definition below, the establishment of a type hierarchy of linguistic entities makes compositional processing modules simple, since input and output are all of the same abstract data type.

\[
\begin{align*}
\text{rule} & \to \text{rule-name} \ \ast : \ast \ \text{regex} \ \ast \rightarrow \ast \ \text{avm} \ \ast \ \text{fun-op} \ \ast \ \ast \\
\text{rule-name} & \to \text{identifier} \\
\text{regex} & \rightarrow \text{regex} \ | \ \text{regex} \ \text{regex} \ | \ \text{regex} \ | \ \text{regex} \ | \ \text{regex} \\
\text{fun-op} & \rightarrow \ \ast \ \text{where} \ \text{coref} \ \ast \ \ast \ \text{fun-app} \\
\text{fun-app} & \rightarrow \ \text{identifier} \ \ast \ \text{term} \ \ast \ \ast \ \text{term} \ \ast \ \ast
\end{align*}
\]

The choice of \textit{TDL} as a basis for \textit{XTDL} has a couple of advantages. TFSs as such provide a rich descriptive language over linguistic structures (as opposed to atomic symbols) and allow for a fine-grained inspection of input items. They represent a generalization over pure atomic symbols. Unifiability as a test criterion (viz., whether a transition is viable) can be seen as a generalization over symbol equality. Coreferences in feature structures describe structural identity. Their properties are exploited in two ways. They provide a stronger expressiveness, since they create dynamic value assignments while following the transitions in the finite-state automaton, thus exceeding the strict locality of constraints in an atomic symbol approach. Furthermore, coreferences serve as the means for information transport into the output description on the RHS of a rule. Finally, the choice of feature structures as primary citizens of the information domain makes composition of processing modules simple, since input and output are all of the same abstract data type.

### 2.3 Example

The \textit{XTDL} grammar rule below may illustrate the concrete syntax of the formalism. It describes a sequence of morphologically analyzed tokens of type \texttt{morph}. The first TFS matches one or zero items (?) with part-of-speech \texttt{det}. Then, zero or more \texttt{adj} items are accepted (*). Finally, one or two \texttt{noun} items (\{1,2\}) are consumed. The use of a variable (e.g., \#c) in different places establishes a coreference (i.e., a pointer) between features. This example enforces, e.g., agreement in case, number, and gender for the matched items. I.e., all adjectives must have compatible values for these features. If the recognition pattern on the LHS successfully matches the input, the description on the RHS creates a feature structure of type \texttt{phrase}. The category is coreferent with the category \texttt{noun} of the right-most token(s) and the agreement features result from the unification of the agreement features of the \texttt{morph} tokens.

\[
\begin{align*}
\text{np} : > \ & \text{morph} \land [\text{POS} \ \text{det}, \ 
\text{INFL} \ \text{infl}, \ 
\text{SEGMENTATION} \ \ast \text{list}\ast], \ 
\text{morph} \land [\text{POS} \ \text{adj}, \ 
\text{INFL} \ \text{infl}, \ 
\text{SEGMENTATION} \ \ast \text{list}\ast], \ 
\text{morph} \land [\text{POS} \ \text{noun} \ \& \ \text{cat}, \ 
\text{INFL} \ \text{infl}, \ 
\text{SEGMENTATION} \ \ast \text{list}\ast] \ (1,2) \\
\rightarrow \ & \text{phrase} \land [\text{CAT} \ \text{cat}, \ 
\text{AGR} \ \text{agr}, \ 
\text{INFL} \ \text{infl}, \ 
\text{SEGMENTATION} \ \ast \text{list}\ast].
\end{align*}
\]

### 3 Imperfect Unifiability Test

The challenge for the \textit{SProUT} interpreter is to combine regular expression matching with unification of TFSs. Since regular operators such as Kleene star cannot be expressed as a TFS (no functional uncertainty!), the interpreter is faced with the problem of mapping a regular expression to a corresponding sequence of input tokens, so that the coreference information among the elements in a rule is preserved. The solution is to separate the matching of regular patterns using unifiability (LHS of rules) from the construction of the output structure through unification (RHS). The positive side effect is that the
unifiability test filters the potential candidates for the space-consuming final unification.

Subsequently, a rule TFS with an instantiated LHS pattern is constructed. A TFS representation of a rule contains the two features IN and OUT. In contrast to the IN value in the matched input TFS representation, the IN value of the rule contains coreference information. The value of OUT is the TFS definition of the RHS of the rule. Given the input TFS and the uninstantiated rule TFS, the unification of the two structures yields the final output result.

As is the case for deep parsing, usually more than 90% of all LHS applications fail, and since we use unification \( \land \) for testing unifiability, a lot of space and time is wasted. However, the things are not that bleak as they seem, since our unifier eliminates redundant copying of TFSs through lazy incremental copying to achieve a great deal of structure sharing (see next section). Modified structures can be reset using invalidate() which simply increments a global generation counter, such that a modification in the copy slot of a TFS is no longer considered. And so unifiability testing of two conjunctive TFSs in the early SProUT reduced to \( \bot \) denotes incompatible information)

\[
\text{unifiable}(\text{TFS} \phi, \text{TFS} \psi) : \iff \\
\begin{align*}
\text{success} &\iff \phi \land \psi = \bot \\
\text{success} &\iff \text{false}; \\
\text{else} &\iff \text{true}; \\
\text{invalidate}(); \\
\text{return} \text{success};
\end{align*}
\]

Nevertheless, about 90% of the run time in the interpreter was due to the unification operation, may it be used for unifiability testing or to build up RHS structures. One might now argue that unifiability testing should be as fast and cheap as checking subsumption or equivalence, but this is not the case: a correct unifiability test must record the effects of type unification, i.e., must allocate memory. The deeper reason why this is so comes from the use of coreferences in unification-based grammars.

However, it is possible to implement an imperfect, but extremely fast unifiability test that does not require the space of standard unification. The test is imperfect in that there are very rare combinations of feature structures which are assumed to be unifiable, but which are not. Such combinations are detected later in the construction of the RHS of a rule when performing standard unification. The important thing, however, as explained above, is that almost all unifiability tests during grammar interpretation fail and for these negative cases, the fast test delivers a correct answer. During thorough experiments with the new unifiability test, even the positive answers were always right, i.e., subsequent RHS unifications had not failed.

The structure of the new algorithm is similar to the subsumption/equivalence test in SProUT, except that type subsumption/equality is substituted by type unification which reduces to a table lookup (note that the pseudo code is slightly simplified and does not work for cyclic structures):

\[
\text{unifiable}(\text{TFS} \phi, \text{TFS} \psi) : \iff \\
\begin{align*}
\text{if} \phi = \psi &\iff \text{return true}; \\
\text{if unifyTypes}(\phi, \psi) = \bot &\iff \text{return false}; \\
\forall (\text{feat . } \phi') \in \phi \text{ and } (\text{feat . } \psi') \in \psi &\iff \text{return false}; \\
\text{return true};
\end{align*}
\]

Compared to the old test, we achieved a speedup by a factor of 5.5-8, depending on different shallow grammars. It is worth noting that this imperfect test is related to another imperfect technique used in deep parsing, viz., quick-check filtering (Kiefer et al., 1999). Our method does not require offline training and additional data structures (which quick-check filtering does) and is comparable in performance when using mainly flat TFSs (which is the case for shallow grammars).

4 Polymorphic Lazy Set Unification

The original destructive lazy-copier unifier in SProUT was an optimized and corrected variant of (Emele, 1991) that was further extended by an efficient type unification operation, viz., bit vector bitwise AND on the type codes, together with result caching. The average-case complexity of computing the greatest lower bound (= result of type unification) is thus determined by a constant-time function.

Compared to the implementation of the original algorithm in (Emele, 1991), our improved version yields a speedup of 2-4.5 (depending on the shallow grammars) by computing the shared and unique feature-value pairs \( \text{SharedArcs}_1, \text{SharedArcs}_2, \text{UniqueArcs}_1, \text{and UniqueArcs}_2 \) of the two input structures in parallel (plus further minor improvements). This new unifier together with the imperfect unifiability test drastically speed up system performance, turning the original ratio of unification/interpreter time from 90/10 to 25/75. Overall, the two modifications lead to a speedup factor of 20 on the average.

During the development of SProUT, it turned out that the description language XTTC misses constructs that assist unordered collections of information. Clearly, FIRST/REST lists in conjunctive TFS are the usual means to achieve this. However, by unifying two lists, we do not collect the sole information from both lists. Instead, the corresponding positions are unified, and lists of different length will never unify. Applications such as information extraction must work around this 'phenomena' and either apply fixed-arity named templates, implement difference list to achieve a kind of set union, apply
recursive type constraints, implement procedural attachment, or employ disjunctions as a way to express collective information. Explicit disjunctions in the TFS description language, however, have been consciously excluded, since they render almost linear (conjunctive) unification exponential.

In order to account for unordered collections, we decided to implement a special kind of non-standard sets, viz., multisets (or bags), which might contain equivalent, even equal objects. Elements of a set are either TFSs or again sets, even the empty set. Unifying two sets $S_1, S_2$ means to take multiset-union of $S_1$ and $S_2$:

$$S_1 \& S_2 := S_1 \cup S_2$$

This is an extremely cheap operation (even cheaper than normal set union) and is exactly what we need.

Sets should not be confused with disjunctions, since the unification of two disjunctions $D_1, D_2$ is defined in terms of the unification of their elements, a very expensive operation:

$$D_1 \& D_2 := \{ d_1 \& d_2 \mid d_1 \in D_1 \text{ and } d_2 \in D_2 \}$$

Up to now, we only considered the two cases that the unification method either takes two TFSs or two sets. But what is the result of unifying a TFS $\phi$ with a multiset $S$? We decided to open a third avenue here—this time we assume that the TFS argument acts as a filter on the elements of the multiset using unifiability. I.e., a unification failure leads to the deletion of the set element:

$$\phi \& S := \{ s \mid s \in S \text{ and } \phi \& s \neq \bot \}$$

This useful operation has, like ‘normal’ set unification, a counterpart in Lexical Functional Grammar (Bresnan, 1982). And again, it is relatively cheap, due to the use of the fast unifiability test.

Given these additions, unification now becomes a true polymorphic operation (and is realized this way in our JAVA implementation through method dispatching):

<table>
<thead>
<tr>
<th>&amp;</th>
<th>TFS</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFS</td>
<td>$\land$</td>
<td>$\in_c$</td>
</tr>
<tr>
<td>set</td>
<td>$\in_c$</td>
<td>$\cup$</td>
</tr>
</tbody>
</table>

Note the subtle differences when using singleton sets. Assuming that

$$\phi \& \psi = \bot$$

we have

$$\phi \& \{ \psi \} = \{ \phi \} \& \psi = 0$$

but

$$\{ \phi \} \& \{ \psi \} = \{ \phi, \psi \}$$

The important point is that the unifier in SProUT can be straightforwardly extended towards our special treatment of sets, without giving up any of the good properties of the original algorithm, viz., lazy non-redundant copying and almost linear run time complexity. And the good properties of the imperfect unifiability test can also be retained for multisets.

We are currently investigating to include a C(++)-like malloc allocation scheme for TFSs which should have a drastic effect on the turnwise run time performance of SProUT. The idea is to avoid the creation of new TFS objects and the application of Java’s garbage collector, if possible, by having our own TFS memory management. TFSs which are no longer relevant and which should be reused must be freed so that the allocator can take care of. If new TFSs are requested by unification, we first reuse the old objects before creating new ones.

## 5 Testing Negation

Negation in typed feature formalisms has often posed severe problems, either because of a complex or even wrong semantics, or because of a bad run-time performance. Classical negation of conjunctive TFS leads to the introduction of disjunctions (not that good as we have seen above), negated coreferences (easy!), and negated atoms/types (cheap!) when assuming negation normal form (Smolka, 1988). In case that negated information should be retained and accumulated in the TFS, additional memory must be allocated.

Several SProUT users have demanded that it would be nice to come up with some form of negation in order to compactly rule out certain input items. Clearly, when considering only types, negation can be laboriously spelled out through the introduction of additional types. For instance, the type not-1st (or 2nd-or-3rd) might be introduced under type person with subtypes 2nd and 3rd, in order to “reformulate” the negated TFS [pers $\sim$ 1st].

We have decided to allow negation only on the LHS of a SProUT rule and only on top of a description. These requirements have several theoretical and practical advantages. Firstly, restricting ourselves to the LHS means that we only test whether an input item meets the negated information or not. As we will see, this test is extremely cheap and does not require additional memory. Secondly, since negation inside a TFS is forbidden, no additional memory must be spent to represent that information. As a consequence of the LHS restriction, negated information is clearly no longer accessible after a successful LHS match. In a SProUT rule, positive and negative information can be arbitrarily mixed, and a top-level negation sign can be attached to types, TFSs, and even to coreferences.

Now, how does the negation test look like? When assuming a classical set-theoretical semantics for TFSs (as we do), the answer is really easy. Assume that the SProUT rule under inspection at a current input position contains the negated feature structure $\sim \phi$ and that the TFS for the input token is $\psi$. Testing for negation means that we want to decide whether $\sim \phi \land \psi = \bot$. The Venn diagram gives us
the answer (let \([\cdot]\) refers to the denotation of a TFS and let \(\mathcal{U}\) represents the universe of all denotations):

\[
\sim \phi \land \psi \text{ is not satisfiable, i.e., } [\sim \phi \land \psi] = \emptyset \iff [\psi] \subseteq [\phi]. \text{ This means that } \\
\sim \phi \land \psi = \bot \iff \psi \subseteq \phi
\]

I.e., only if \(\psi\) is more specific than or equivalent to \(\phi\) (\(\subseteq\)), rule interpretation has to be canceled. In every other case, there must exist elements in the denotation of \(\psi\) which are not in \(\phi\), i.e.,

\[
[\psi] \setminus [\phi] \neq \emptyset
\]

hence rule interpretation is allowed to continue. Testing for TFS subsumption (\(\subseteq\)) is again an extremely cheap operation.

6 Weak Unidirectional Coreferences

Several projects using SProUT have revealed a missing descriptive means in the original formalism. This no man’s land concerns the accumulation of information under Kleene star/plus or restricted repetition. Consider, for instance, the np rule from section 2.3 and assume that adjectives also have a relation attribute RELN. Our intention now is to collect all those LHS relations and to have them grouped in a set (section 4) on the RHS of the rule. In order to achieve this, we have implemented the concept of a weak, unidirectional coreference constraint, indicated by the percent sign in the concrete syntax:

\[
\text{np} :> \ldots \text{ (morph } \& \text{ [POS adj, } \ldots \text{, RELN } %r\ldots \text{ )} \ast \ldots \rightarrow \text{ phrase } \& \text{ [\ldots, RELN } %r\ldots \text{ ]}
\]

A usual coreference tag, say \(%r\) (instead of \(\%r\)) would enforce that the iterated values under RELN attribute are the same.

Collecting the LHS information in a set (instead of a list) perfectly matches our treatment of set unification: the result set on the RHS can be further processed and extended by succeeding stages in the shallow processing cascade. Recall that lists do not (easily) allow the accumulation of information (cf. section 4).

Implementing such weak coreferences is not that difficult and again extremely cheap. During Kleene expansion and restricted repetition, the rule interpreter introduces for each successful TFS instantiation and each occurrence of \(\%r\), a fresh variable which binds the value under the corresponding feature (in our case RELN). Consider, for instance, that the above np has matched two adjectives, before it has recognized the noun. The interpreter thus generates two bindings through the new variables \(\#r\_1\) and \(\#r\_2\) and constructs the set \([\#r\_1, \#r\_2]\) after a successful LHS match, as if the original RHS would have been

\[
\text{phrase } \& \text{ [\ldots, RELN } \{\#r\_1, \#r\_2\} \text{ ]}
\]

7 Context-Free Cascade Stages

SProUT permits to call additional rules during the course of a single rule interpretation (like a call to a subprocedure in a programming language) through the use of the seek operator. There are no objections for a rule to call itself, what clearly extends the expressiveness of the formalism, making it context-free (like the related recursive transition networks (Conway, 1963) are). The use of SDL (Krieger, 2003) together with context-free stages even allows SProUT to recognize context-sensitive languages.

The following example presents a simple grammar that matches \(n\) occurrences of “a” followed by \(n\) occurrences of “b” and counts \(n\) by representing it as a list of \(n\) bars |. Considering the recognition part of the rule, \([a^nb^n] \mid n > 0\) is, in fact, a context-free language.

\[
S :> a \text{ (\#seek(S) } \& \text{ [COUNT } #1\text{]}\} \text{? b} \rightarrow \text{ [COUNT } <"!" \text{ } . \#1\text{]}.
\]

Note that we represent “a” and “b” as types a and b, whose surface form is “a” and “b”, resp.

\[
a := \text{ token } \& \text{ [SURFACE } "a"\text{]}.
\]

\[
b := \text{ token } \& \text{ [SURFACE } "b"\text{]}
\]

In some special cases, a sequence of seek calls can generate a rule cycle. Of course, if no input is consumed within a cycle of seek calls, we end up in an infinite loop at runtime. To avoid such a situation, we implemented a special left-recursion check that is performed during the compilation of the grammar and which generates a compile time error, if the grammar contains infinite cycles of seek calls.

Concerning runtime performance, there are two aspects to be considered when using seek. Firstly, regular languages which are specified with the help of seek can be rewritten using only regular operators. In such cases, the efficiency of a system which provides seek although a grammar does not use it is on par with an implementation that does not have the possibility of calling seek. For each automaton state, our system is given a disjoint partition of outgoing edges, viz., a set of seek edges and a set of non-seek edges. These sets are computed at compile time and testing for an empty seek set is negligible. Secondly, applying seek at runtime forces the interpreter to produce new binding environments. To improve efficiency here, we introduced an optimizing mechanism for seek calls. The compiler tries to replace seek calls with the body of the called rule in case the RHS of the rule is empty (not possible in case of a circle). There exist several other configurations which are recognized by the compiler and
which obviates the generation of new environments. Such optimizations make the compiled finite-state automaton larger, but can lead to a speedup of up to 30%, depending on the grammar.

8 Compile Time Type Check
The basic building blocks of SPoUT rules are typed feature structures. A compile time type check has been added to the system, checking appropriateness of features, well-typedness of feature structures, and strong typing of rule definitions.

(Carpenter, 1992) introduces formal definitions for appropriateness and well-typedness. Informally, a feature in a TFS is said to be appropriate if the type bearing it or one of the supertypes introduces the feature. A SPoUT rule meets the appropriateness condition if every feature, relative to the TFS it occurs in, is appropriate. A SPoUT rule is well-typed if every feature that occurs is appropriate and has an appropriate value, i.e., a value that is subsumed by the value of the feature of the type that introduces that feature. Finally, we say that a SPoUT rule is strongly typed if every feature structure, occurring in it and bearing at least one feature, also has a type that is more specific than the most general type of the type hierarchy.

To sum up, strong typing, appropriateness, and well-typedness conditions impose additional constraints on typed feature structures occurring in rules. These restrictions are defined in and imposed by the TDL type hierarchy associated with the rule grammar. Practical advantages of these meta-constraints in grammar formalisms (not even in SPoUT) are threefold (there are others as well, such as type inference; cf. (Schäfer, 1995)).

Debugging, safety and maintainability. Conceptual or typographical errors in SPoUT rules (e.g., feature names or types) are likely to be detected at compile time, due to the above restrictions.

Portability of grammars. Many implemented TFS formalisms require feature structures to meet some of the above conditions. Grammars written for a less restricted formalism may not work on systems requiring appropriateness without major changes.

Efficiency. Generally, strong typing and restriction to a fixed set of features can be exploited for compact representations. We give a small example where type checking at compile time leads to efficiency gains at runtime by revealing impossible unifications. Given the following SPoUT rules

\[
S :> \ldots \rightarrow z \& \{ \ldots \}.
U :> @@seek(S) \& x \& \{ \ldots \} \rightarrow \ldots.
V :> @@seek(S) \& y \& \{ \ldots \} \rightarrow \ldots.
\]

and assume that \(z \land x \neq \bot\), but \(z \land y = \bot\). Compile time type checking then uncovers that the LHS of rule \(V\) is inconsistent under all possible interpretations. Related to this technique is rule filtering in deep parsing (Kiefer et al., 1999) and partial evaluation, known from logic programming.

For appropriateness checking, the system builds up a table of features that are admissible for a type when reading in type definitions. Well-typedness is checked on the basis of prototypical feature structures that are associated with each type defined in the type hierarchy. Checking well-typedness, appropriateness, and strong typing is achieved by recursively walking through a SPoUT rule (represented in an intermediate XML expression which in turn has been generated on the basis of a JavaCC parser for ATDL), checking types and features at every feature structure node. Error messages with detailed reasons and links to character positions in ATDL source files are generated.

In addition to the type check, a unification of incompatible types (conjunction of types with the \& operator) in type expressions is signaled, and an information is issued when two types in a conjunctive type expression have a common subtype in the type hierarchy. Furthermore, the seek operator undergoes a special handling: For appropriateness and well-typedness checking, the output type and feature structure of the called SPoUT rule is computed, and checked together with the feature structure (if present) of the calling part.

9 Transition Sorting
Since ATDL grammars are compiled into finite-state devices whose edges are labeled with TFSs, the standard finite-state optimization techniques can not be exploited directly. The application of conventional determinization and minimization neither reduces the size, nor the degree of nondeterminism of the finite-state network significantly.

Instead of applying these optimizations to non-atomic TFS-labeled edges, we have introduced a technique for ordering the outgoing edges of an automaton state, which resembles topological sorting of acyclic graphs. To be more precise, we sort all outgoing transitions of a given state via the computation of a transition hierarchy under TFS subsumption. Obviously, such an ordering can be computed offline, since edge labels do not change at run time. In the process of traversing an extended finite-state grammar, these transition hierarchies are utilized for inspecting outgoing transitions from a given state, starting with the least specific transition(s) first (remember, TFS subsumption induces only a partial order), and moving downwards in the hierarchy, if necessary. The important point now is that if a less specific TFS does not match, the more specific ones will not match as well, hence the corresponding edges need not be inspected (the fast unifiability test is employed here as well).

In this manner, the number of time-consuming unifications can be potentially reduced. Our initial experiments reveal that the utilization of this data structure, in particular, comes in handy in case of the initial state and its close neighbors (the initial state in one of our test grammar had about 700 out-
going transitions).

However, since most of the states have only two outgoing edges on the average, the application of transition sorting to all states is not a good idea in terms of efficiency. Therefore, a threshold on the number of outgoing arcs is used in order to classify states for which the transition sorting is applied. Due to the fact that transition hierarchies, created solely from the arcs in the grammar, exhibit a somewhat flat character (average depth: 2–3), we provide a further option for deepening and fine-tuning them through a semi-automatic introduction of artificial edges.

Firstly, for each major type which occurs in the grammar, e.g., morph, token, and gazetteer, nodes in the transition hierarchy are introduced. Secondly, for all appropriate features \( f \) of a given major type \( t \) and all feature-value pairs \([f \ v_1], \ldots, [f \ v_k]\) found in the grammar, we introduce additional nodes in the transition hierarchy, representing \( t \& \sim [f \ v_1, \ldots, v_k] \), i.e., TFSs whose \( f \) value is different from \( v_1, \ldots, v_k \). Finally, for all TFSs \( t \& [f \ v_i] \), we compute a separate table of links to the corresponding least specific nodes.

These rough extensions allow for an efficient traversal of the hierarchy while processing input TFSs like for instance \( t' \& [\ldots, f \ v] \) \( (t' \sqsubseteq t) \). Transition sorting, as briefly introduced here, proves to speed up the grammar interpreter by a factor of 3–4.

The proximate paper (Krieger and Piskorski, 2004) gives a deeper insight into the application of this and related techniques.

10 Output Merging Techniques

Shallow analysis in SProUT (and other IE systems) often yields multiple results for a single recognized chunk, originating from different ambiguity sources.

Local ambiguities. The lexicon contains morphosyntactic (e.g., gender, number, person) and semantic (senses) variations which might blow up the search space of the grammar interpreter, resulting in multiple readings. There exist, however, several techniques which help to lower the ambiguity rate by compacting and unifying lexicon entries (see next section). Typed feature structures are a necessary requirement for applying such techniques.

Spurious ambiguities. Depending on the form of the grammar, multiple recursive rule calls might lead to attachment ambiguities which however produce equivalent RHS structures. In case we are only interested in the output (which usually is the case), we are allowed to remove such duplicate TFSs.

Rule ambiguities. We have often encountered rule sets, which, for specific input items, produce output structures that are related according to their degree of informativeness. I.e., we have found structures within these results which are more general or more specific than others.

In each of the above cases, we locally reduce the number of output TFSs for a fixed input span with-out giving up any information. This is achieved by the virtue of TFS equivalence, TFS subsumption, and TFS unifiability/unification. In SProUT, a user can specify (i) whether the output TFSs are left untouched, (ii) whether duplicate structure should be removed, (iii) whether the most general-specific structures only prevail, or (iv) whether we maximally integrate the output structures through unification. Very often, a single TFS remains at the end. Due to the fact that the SProUT interpreter employs a longest-match strategy, further ambiguities are avoided at this stage.

Merging is only applied at the very end of a single stage in the shallow cascade, and thus not very expensive overall. Worst-case running time is a quadratic function in the number of output structures. The TFS operations involved in this merging are cheap, as explained in the previous sections.

11 Compacting Lexical Resources

Morphological resources in SProUT are usually built on top of the full form lexical databses of MMorph. However, many lexical entries in MMorph possess spurious ambiguities. When integrating MMorph lexicons as they are, a runtime system might have a serious space problem, and in particular, performs redundant unifications.

We have developed a method which compacts MMorph resources by replacing several readings through a compact reading, by deleting redundant readings, and by substituting specialized readings through more general ones, using type generalization and subsumption checking. These techniques go hand in hand with a moderate enlargement of the original type hierarchy (no additional costs—recall that average-case complexity of type unification can be estimated by a constant-time function) and increase the efficiency of systems using MMorph, since they shrink the size of the lexicon and come up with fewer readings for a morphological form. Clearly, such an approach not only is interesting to MMorph, but also to other lexicons, which build on an feature-value representation of lexical information.

Entries in MMorph relate word forms to their base forms and their morphosyntactic descriptions (MSDs), which are sets of flat feature-value pairs. Here are two of the 11 MMorph readings of the German word “evaluieren” (to evaluate, evaluated):

```
Verb[mode=indicative vform=fin tense=imperfect number=plural person=13 ...]
Adjective[gender=masc|fem|neutrum number=singular case=nom|gen|dat|acc degree=pos ...]
```

MMorph represents atomic disjunctions by using the vertical bar, e.g., 113 (see example). We represent such disjunctions through exhaustive disjunctive types, e.g., 1st_or_3rd, together with proper type definitions, e.g.,

\[
1st := 1st_or_2nd \& 1st_or_3rd
\]
These types are automatically generated by our method when reading in an MMorph data base.

Given a full form database, we store information for the same word form (example: evaluierten) in an index structure of the following form (POS abbreviates part of speech):

\[
\begin{align*}
\text{word form} & \rightarrow \text{POS}_1 \rightarrow \text{stem}_{11} \rightarrow \text{set of MSDs} \\
& \hspace{1cm} \vdots \hspace{1cm} \vdots \hspace{1cm} \vdots \\
& \rightarrow \text{POS}_n \rightarrow \text{stem}_{n1} \rightarrow \text{set of MSDs} \\
& \hspace{1cm} \vdots \hspace{1cm} \vdots \hspace{1cm} \vdots \\
& \rightarrow \text{set of appropriate values} \\
& \rightarrow \text{set of appropriate values}
\end{align*}
\]

An MSD is encoded as a table of the following form:

\[
\begin{align*}
\text{feature}_1 & \rightarrow \text{set of appropriate values} \\
& \hspace{1cm} \vdots \\
\text{feature}_l & \rightarrow \text{set of appropriate values}
\end{align*}
\]

Given the set of all MSDs \( M \) for a specific word form, the compaction method applies the following operations to \( m_1, m_2 \in M \), until \( M \) remains constant (i.e., until a fixpoint is reached):

- **Equality test**: If \( m_1 = m_2 \), remove \( m_1 \) from \( M \).
- **Subsumption test**: If \( m_1 \) is a subset of values of features in \( m_2 \), remove \( m_2 \) from \( M \) (\( m_2 \) is more general than \( m_1 \)).
- **Set union**: If \( m_1 \) differs from \( m_2 \) at only one feature \( f \), then merge the two values, remove \( m_1 \) from \( M \), and replace the value of \( f \) in \( m_2 \) by \( v \), where \( v \coloneqq m_1 \cup m_2 \). Only 195 type definitions are produced by the above method for the German lexicon. Overall, the average speedup measured for the German named entity grammars in SProUT was about a factor of 3. A thorough investigation of the approach is presented in (Krieger and Xu, 2003).

We are currently investigating the impact of packing morphosyntactical information across several features-value pairs. Our automated compaction method can be easily extended to handle such superfeatures/-values.

**Acknowledgments**

This paper has benefited from discussions with our colleagues Stephan Busenmann, Bernd Kiefer, and Hans Uszkoreit. Thanks much! We also like to thank our reviewers for their awesome assessments. This work has been partially funded by the German BMDF under grant nos. 01 IN A01 (Collate) \& 01 IW C02 (Quetal), and by the EU under grant nos. IST-12179 (Airforce), IST-2000-25045 (Memphis), and IST-2001-37836 (DeepThought).

**References**


