Tree Searching/Rewriting Formalism

Petr Němec

Institute of Formal and Applied Linguistics
Malostranské náměstí 25
118 00 Praha 1
Czech Republic
nemec@ufal.mff.cuni.cz

Abstract

We presents a formalism capable of searching and optionally replacing forests of subtrees within labelled trees. In particular, the formalism is developed to process linguistic treebanks. When used as a substitution tool, the interpreter processes rewrite rules consisting of left and right side. The left side specifies a forest of subtrees to be searched for within a tree by imposing a set of constraints encoded as a query formula. The right side contains the respective substitutions for these subtrees. In the search mode only the left side is present. The formalism is fully implemented. The performance of the implemented tool allows to process even large linguistic corpora in acceptable time. The main contribution of the presented work consists of the expressiveness of the query formula, in the elegant and intuitive way the rules are written (and their easy reversibility), and in the performance of the implemented tool.

1. Introduction

In this paper we aim to introduce a Tree Searching/Rewriting Formalism (TSRF) and its implementation. Originally, we developed the formalism for searching complicated linguistic phenomena in the tectogrammatic trees (TGTSs) of the Prague Dependency Treebank (Hajič et al., 2001) and then we extended it to a rewriting formalism for the purposes of machine translation (Hajič at el., 2002).

The formalism recognizes rules whose left side specifies a forest of subtrees to be found within a tree by imposing a set of constraints encoded as a query formula. The optional right side then contains respective substitutions for the found subtrees (or may be omitted in the searching only mode). The searched structures have to be rooted trees whose nodes may be labelled by a set of $(attribute - value)$ pairs.

There are several reasons for the development and implementation of such formalism. First, it can serve as a corpus searching tool allowing linguists to search for relevant linguistic phenomena. Second, its rewriting capabilities provide an elegant and intuitive way to perform rule-based tree transformations. Moreover, the rules can be easily reversed, i.e. given a rule that transforms tree A to tree B, it is easy to determine a rule that transforms B back to A. This has proven to be especially useful for the machine translation experiments where probabilities of respective rules could be estimated from the probabilities of the reversed rules.

An interpreter for the proposed formalism is fully implemented.

The paper is structured as follows: First, we introduce the general conception of the formalism in Section 2. The query formula and the supported predicates are discussed in Section 3. The substitution process is described in Section 4. Section 5 provides a few examples of the substitution rules. The implementation of the formalism and its performance is the subject of Section 6. Section 7 compares the interpreter to some other existing tools and Section 8 concludes the paper.

2. The Formalism Core

The proposed formalism is designed to allow searching for a specified forest of subtrees within queried trees. In the substitution mode, the found matches are then replaced by specified substitution trees. The nodes of the queried trees may be labelled by a set of *(attribute,value)* pairs[1] (such as e.g. TGTS trees).

In the following text we take a tree to be of the form (V, E), where V is the set of vertices[2] and E is the set of edges. Additionally, we will denote $\overline{V_v}$ and $\overline{E_v}$ the set of all vertices and edges, respectively, within the subtree whose root is $v \in V$.

In the following we will describe TSRF in the substitution mode. In the search mode, TSRF works in the same way, but it accepts only the left side of a rule (query formula) and yields the list of all the matches.

TSRF operates on a set of rules (a *program*) that are sequentially applied on a given tree. Schematically, a substitution rule looks as follows

$$F -> [S_1, S_2, ..., S_n],$$

where F is a query formula (see Section 3) containing (among other predicates) positive, i.e. non-negated, occurrences of structural predicates specifying subtrees (*templates*) T_1, T_2, ..., T_n to search for and S_1, S_2, ..., S_n are the respective substitution trees (*substitutions*) for the found subtrees. The template vertices are associated with actual tree vertices after a successful match. Substitutions represent new tree structures on the matched vertices.

More precisely, each vertex v of a template or substitution is assigned a *label* $l(v)$. l must be injective for the set of all template vertices (i.e. each vertex $v \in T_i, 1 \leq i \leq n$, is assigned a unique label). S_i, the corresponding substitution,

[1] Note that it is straightforward to convert e.g. edge-labelled trees to this representation as the label on an edge may be viewed as a value of a (special) attribute of the child node this edge leads to.

[2] For brevity reasons, we will also assume that each $v \in V$ contains its labelling as well and that exactly one $r \in V$ is designated as the root of the tree.

then defines a new tree structure by means of these labels. Additionally, for the substitution vertices l may introduce a new label (node adding), a single template label may be repeated (node copying), or a template label may be omitted whatsoever (node drop).

3. Query Formula

In this section we will describe the query formula, the searching part of the formalism, in detail.

As mentioned in the previous section, the purpose of the query formula is define a set of templates T_1, T_2, ..., T_n to be searched for. Each such template is defined by one non-negated occurrence of the structural predicate (described in the next subsection). The query result is the set of all matches of these templates that satisfy the query formula (see Section 5 for examples of complete query formulae).

The query formula is a propositional formula consisting of standard logical operators (and, or, not) that can be arbitrarily nested (i.e. unrestricted use of parentheses is supported). Its atoms are the predicates described in the following subsections.

The arguments of some of the predicates (e.g. attribute value or absolute position test) can be variables that are then subject to the standard unification procedure over the entire query formula.

From these facts a very important point regarding the overall expressive power of the query formula follows: because an occurrence of the structural predicate can viewed as an implicit existential quantifier on the vertex variables this predicate contains and because this occurrence can be arbitrarily nested and/or negated in the query formula, we in fact get the full power of first order logic over vertex variables, not just its existential fragment. (This follows from the fact that for any formula of first order logic there exists an equivalent formula without universal quantifiers.)

3.1. Structural predicate

The structural predicate specifying a template is of the form

$$(VertexVar, VertexConditions, SubTrees),$$

where $VertexVar$ is an (arbitrary) vertex variable identifying the template vertex, $VertexConditions$ is a propositional formula imposing conditions on the vertex, and $SubTrees$ is a list of other structural predicates specifying the children vertices of this vertex, i.e., the structural predicate is fully recursive on the members of $SubTrees$.

The propositional formula ($VertexConditions$) consists of arbitrarily nested standard logical operators (and, or, not) and the following predicates-atoms:

- Attribute value test

  (Attribute Operator Value)

  where $Operator \in \{=, =\sim, \neq, \neq\sim, <, >\}$ and $Value$ is either a number, a string, or a variable. The last option allows for the unification of attribute value variables throughout the entire query formula. $=\sim$ ($\neq\sim$) allows to test match (mismatch) against the given regular expression.

- Children count test

  $$(\#\ Operator\ Number),$$

  where $Operator \in \{=, <, >\}$

For example, the structural predicate specifying a template consisting of a conjunction (a) that coordinates two nouns (its children, b and c) that share the grammatical number would look as follows:

(a,func = 'CONJ', [(b, tag =∼ 'N' and number = X, []),*
(c, tag =∼ 'N' and number = X, [])])*

3.2. Path predicate

The $path$ predicate allows to impose constraints on the path between two nodes - $Start\_VertexVar$ and $End\_VertexVar$ - specified in the structural predicates. It is of the following form

$$path(Start\_VertexVar, End\_VertexVar,$$

$$Segments, VertexConditions),$$

$Segments$ are is the list of respective path segments. Each segment specifies the "movement" within the tree respective to the previously visited node (which is $Start\_VertexVar$ if this is the first segment, otherwise it is the last node of the previous segment). Form of a segment is as follows:

$$(Distances, Direction),$$

where $Distances$ is a list of intervals specifying the possible number of steps-nodes in the given direction. $Direction$ is either up (towards the parent), $down$ (towards a child), $left$ (towards the nearest node on the same tree level to the left), $right$ (towards the nearest node on the same tree level to the right). For example,

$$[((1 - inf), right), ((0, 2), up))]$$

denotes a two segment path, i.e., one node lies between $Start\_VertexVar$ and $End\_VertexVar$: it is located on the same level and to the right of $Start\_VertexVar$ and $End\_VertexVar$ is either this node itself or its grandfather.

$VertexConditions$ specify the attribute values constraints imposed on the nodes of the segment and are identical to attribute tests in the structural predicate (see the previous subsection).

3.3. Other predicates

There are several other predicates present:

- $testAttribute(VertexVar, Attribute, Operator, Value)$
- $testChildrenCount(VertexVar, Operator, Value)$
- $testDistance(VertexVar_1, VertexVar_2, Attribute, Operator, Value)$

The first two predicates correspond to their counterparts in the structural predicate. This way, however, they can appear independently in the query formula, thus increasing the expressive power of the formalism. For instance,

$$(a, , []) \; and \; (b, , []) \; and$$

$$(testAttribute(a, [attr =' \; value']) \; or \; testChildrenCount(b, [0 - 1]))$$

allows to search for nodes a and b, where either $a \rightarrow attr$ equals to $value$ or b has more than 1 child. This would be impossible to express with only the structural predicates at hand.

$testDistance$ allows to compare the difference in numerical attribute $Attribute$ of the specified nodes $VertexVar_1$ and $VertexVar_2$. This predicate compensates to some extent[3] the impossibility to use arithmetic expressions within predicate calls. The predicate holds if the difference is in the $Operator$ relation to the $Value$. For example,

$$testDistance(a, b,' \; sentord', =, 1)$$

tests whether the difference in $sentord$ attribute between nodes a and b is 1 (for a TGTS, this tests whether the words represented by the two nodes lie next to each other in the surface sentence).

4. Substitution Process

In this section we will describe the substitution process, the optional substitution part of the formalism, in detail.

Let $Q = (V^Q, E^Q)$ be a tree. Let R be a rule that is to be applied on Q, let T_1, T_2, ..., T_n be the respective templates and S_1, S_2, ..., S_n the corresponding substitutions. We will call any set of nodes that form a matching subtree for T_i within Q a *match* for T_i. Additionally, we will call the set of assignments of the vertices of template T to the vertices of a match M for $T = (V^T, E^T)$ the *map* m_{TM} between T and M. Let $val_{m_{TM}}(v), v \in V^T$, be a function that returns the matching node for v according to m_{TM}.

Throughout this section we will provide examples based on the situation depicted in Figure 1 for easier understanding.

R is applicable on Q if and only if the following conditions hold [4]:

1. For each $v \in V^Q$, v appears at most in one match from the set of all matches for the templates that are to be altered (i.e. the templates whose corresponding substitution is not identical to the templates themselves).

2. Let S_i be a substitution containing a vertex v such that $l(v) = l(w)$, $w \in T_j, i \neq j, 1 \leq i, j \leq n$ (i.e. v is defined in a *foreign* template T_j). Then T_j has only a single match in Q.

[3] In our experience, all the linguistically relevant queries over PDT that require a numerical operation are limited to the comparison of attribute value difference.

[4] If R is not applicable on Q, it is ignored.

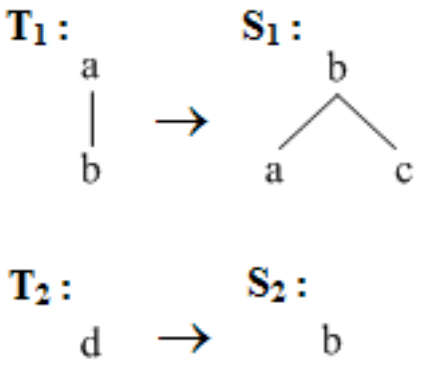

Figure 1: Example of a substitution rule featuring two templates T_1 and T_2 and the corresponding substitutions S_1 and S_2

These conditions[5] on applicability ensure that the substitution process may be carried out as intended.

The first one prevents a part of the tree to be subject to "multiple substitution" by specifying that the respective matches do not overlap (it is harmless for a template whose substitution is identical to it). Figure 2 illustrates such a malformed situation: M_1 and M_2 are two matches for the template T_1 from example depicted in Figure 1. Node 2 is contained in both matches and, as the substitution for S_1 is not identical to the template T_1, its application is not well defined at all.

The second condition ensures that a substitution featuring foreign template variables is defined unambiguously. For the example depicted in Figure 1 this requires that either T_1 has only a single match or T_2 has no match within the queried tree as the node b appears also in S_2 (T_1 is a foreign template with respect to S_2).

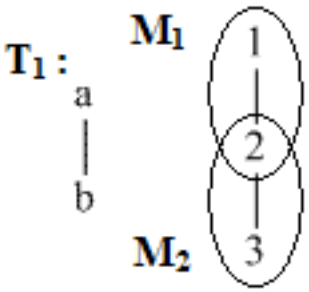

Figure 2: Example of an unapplicable rule.

If R is applicable the following substitution step is performed for each match M of each $T = (V^T, E^T) \in \{T_i, 1 \leq i \leq n\}$:

Let $S = (V^S, E^S)$ be the substitution associated with T. Let us define the partitioning of V^S into the set of nodes occurring in the corresponding template (V^S_O), the set of nodes occurring in foreign templates (V^S_F) and the set of

[5] Note that both of them represent "run-time" conditions - their fulfillment depends on not only on the rules themselves but also on the queried tree. It is the responsibility of the user to create meaningful rules that are applicable on all the processed trees.

new nodes (V_N^S):

$$V_O^S = \{v \in V^S : \exists w \; ; \; w \in V^T \; \wedge \; l(v) = l(w)\}$$

$$V_F^S = \{v \in V^S : \exists T', w \; ; \; T \neq T' \; \wedge \; w \in V^{T'} \; \wedge \; l(v) = l(w)\}$$

$$V_N^S = V^S - V_O^S - V_F^S$$

For the example depicted in Figure 1, the node $c \in S_1$ is a new node and node $b \in S_2$ is a node from the foreign template T_1, all other substitution nodes are occurring in the corresponding template.

Let $c_v(u), v \in V^S$, be a vertex *copy* function that returns a vertex with the same valuation as u, v is a substitution vertex that induces the copy[6]. We denote $\{c_v(u) : u \in V\}$ as $c_v(V)$, where V is a set of vertices. Similarly, $c_v(E)$ is $\{(c_v(u), c_v(w)) : (u, w) \in E\}$, where E is a set of edges. Finally, let us introduce two more abbreviations $val(v)$ and $c(v)$:

$$val(v) = val_{m_{TM}}(v) \Leftrightarrow v \in V_O^S$$
$$val(v) = val_{m_{T'M}}(v) \Leftrightarrow v \in V_F^S$$

$$c(v) = c_v(val(v)) \Leftrightarrow v \in V_O^S \cup V_F^S$$
$$c(v) = c_v(v) \Leftrightarrow v \in V_N^S$$

where T' is the one foreign template in which v occurs. Then S induces the following subtree $G = (V^G, E^G)$:

$$V^G = \bigcup_{v \in V_O^S \cup V_F^S} c_v\left(\overline{V^Q_{val(v)}} - \bigcup_{u \in (V_O^S \cup V_F^S) - \{v\}} \overline{V^Q_{val(u)}}\right) \cup \bigcup_{v \in V_N^S} c(v)$$

$$E^G = \{ \; (c(v), c(u)) : (v, w) \in E^S \; \} \; \cup \bigcup_{v \in V_O^S \cup V_F^S} \Big(c_v(\overline{E^Q_{val(v)}}) \; - \; \{ \; (u, c_w(w)) : u \in c_v(\overline{V^Q_{val(v)}}) \wedge w \in V^S \}\Big)$$

Additionally, the valuation of each vertex $val(v), v \in V^S$, is changed according to the labelling of v so that the values of the specified attributes are rewritten (the values of other attributes remain unchanged).

Informally, G consists of the copies of match nodes connected according to S (with the values of the specified attributes rewritten), copies of all the descendants of these nodes in Q (except for those that are already in some match for T) and the added nodes (according to S).

The substitution process then consists in removal of M from Q and attachment of G (if the root of M is identical to the root of Q then the updated tree is G itself).

Figure 3 presents an example of a tree transformation via the rule depicted in Figure 1. M_1 and M_2 are the only matches of T_1 and T_2 respectively. Node 3′ denotes a copy of node 3 and new is a new node (corresponding to template vartex c).

[6]We index c function(s) by this vertex in order to be able to identify the copy unambiguously.

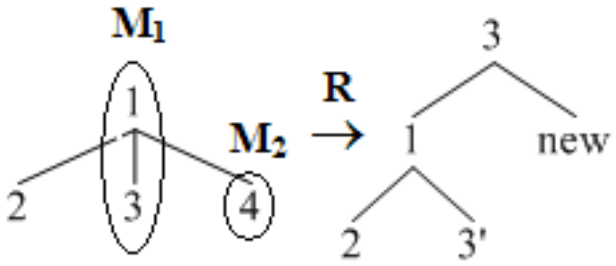

Figure 3: Example of a tree transformation via rule R depicted in Figure 1.

5. Test Examples

This section provides a few examples of the replacement rules. These examples were selected because their query parts (left side of the rules) represent relevant types of linguistic queries and because they well demonstrate the possibilities of TSRF. We have also used these types of substitution rules in the experiments with machine translation. Additionally, we will test the performance of the implemented software tool on these examples (see Section 6.3).

1. Copy the value of $form$ to $lemma$ for each preposition or conjunction (tag begins with *R* and *J* respectively).

   (a, tag =∼ '[RJ]' and form = X, []) →*
   (a, lemma = X, [])

2. Search for a template consisting of an auxiliary word ($afun$ begins with *Aux*) and its daughter - a locative adverbial ($func = LOC$). Swap the two nodes and mark the auxiliary as hidden:

   (a, afun =∼ 'Aux', [(b, func = 'LOC',[])]) →*
   (b,,[(a,TR = 'hidden',[])])

3. Delete all subtrees whose root is a noun and which does not contain an adjective that 1) matches the gender or the number of the noun and 2) precedes the noun at most in two positions in the linear surface order (value of $sentord$ attribute). Two templates are needed[7] as the adjective is not necessarily a direct dependant of the noun.

   (a, tag =∼ 'N' and gender = X and*
   number = Y, []) and not
   ((b, tag =∼ 'A' and (gender = X or number = Y),*
   []) and testDistance(a,b,'sentord',1-2) and
   path(b,a,[vu,1-INF])) → ()

4. Repair a possibly incorrect parsing result: a noun phrase, being a dependant of the main predicate ($func = Pred$), contains a temporal determination ($func = TWHEN$) which is more likely to be a modifier of the main predicate.

[7]Note that only the first template is subject to substitution as the second is negated.

(a, func = 'Pred', []) and (b, tag =∼ 'N', []) and path(b,a,[vu,1-INF]) and (c, func = 'TWHEN', []) and path(c,b,[vu,1-INF]) → (a,,[(c,,[])]),(b,,[]),()*

6. Software Tool

In this section we present the software tool that implements the described formalism. First, we will describe some of its basic characteristics, then we will discuss the complexity issues, and finally we will present performance results of the implemented tool.

6.1. Implementation

The algorithm used to process TSRF queries is quite simple. The query formula is being evaluated (with backtrack to get all the possible matches and variable instantiations) and the truth value of each predicate atom is tested. The free and ground variables from the already evaluated predicates are passed through and upon each successive predicate test these might be subject to unification.

When a structural predicate occurrence is being tested, i.e., there is an attempt to find a match for the corresponding template within the queried tree, the tree is traversed with backtrack starting with the root of a template.

All the possible matches (and variable instantiations) that fulfill the query formula are then collected and returned. If the tool is used in the substitution mode, the respective substitutions are then performed on the returned matches (providing the rules are applicable).

The nodes of the tree to be queried are first indexed so as to minimize the access time when evaluating the respective predicates (mainly the structural predicate) in the subsequent search run. There is large number of other optimizations (also for the optimal performance of the tests of respective designed predicates) but essentially the search run is optimized for a single tree (no optimizations are made for the entire treebank in advance).

The tool was developed in Mercury (Somogyi et al., 1995) programming language. Mercury is a declarative language similar to Prolog but it goes above first order logic and provides a strict type system. Moreover, the Mercury compiler first translates the code into the programming language C and then compiles it as standard C code. This generates fast running code. Many optimizations, especially those connected to the backtracking backbone, are thus performed by Mercury.

The distribution package is available for Linux, Solaris and Windows [8] environments. Plain text files and fs, the native TGTS tree format, files are supported as input files.

6.2. Complexity Issues

Let us discuss the computational complexity of the respective parts of the algorithm presented in the previous section. The initial node indexing is performed once for each node, its time complexity is thus linear. The query formula is being evaluated in exponential time with respect to the number of occurrences of structural predicates contained in it.

The structural predicate test runs generally also in exponential time (searching for a subtree within a tree is known to be NP-hard), but not independently of the query formula evaluation - each successive match found by the structural predicate test run leads to the backtracking step of the query formula evaluation.

The path predicate test runs in $O(k * n)$, where k is the number of segments specified by the predicate and n is the number of nodes of the queried tree.

All other predicates listed in Section 3 run in constant time.

As each node of the queried tree is subject to at most one substitution (applicability condition), all the substitutions are performed in linear time with respect to the queried tree size.

In summary, the entire algorithm runs in exponential time (which cannot be avoided as there is in general exponentially many subtrees within a tree).

6.3. Performance Results

We have chosen the queries from Section 5 to measure the computation time over the PDT corpus. The primary purpose of doing so is to ensure that even such a large treebank as PDT (containing cca 50000 TGTS) can be queried in an acceptable time. The tests were performed on AMD Athlon 64 3800+, 1 GB RAM, running Windows XP. We have divided the tectogrammatic trees of PDT into three groups according to the number of their nodes. The resulting average computation times per tree from the given size range are listed in Table 1.

nodes	1	2	3	4
0 - 9	0.0004	0.0004	0.0006	0.0004
10 - 20	0.0011	0.0010	0.0021	0.0010
> 20	0.0021	0.0021	0.0049	0.0020
any size	0.0012	0.0011	0.0025	0.0011

Table 1: Average computation time per tree in seconds.

These results show that the tool is able to perform the relevant queries in acceptable times even for large corpora, in our case tens of seconds for the entire PDT. Although there is always space for improvement, based on these results we do not consider the performance issue to be critical here. (Moreover, the results are being retrieved sequentially so the user can in fact view them in realtime.)

7. Related Work

To our best knowledge, there is no similar tool that could be straightforwardly used both as a searching and rewriting tool. However, there is a large number of treebank searching tools, e.g. CorpusSearch (Randall, 2003), ICECUP III (Wallis et al., 2000), TGrep2 (Rohde, 2001), TIGERSearch (Konig, 2000), VIQTORYA (Kallmeyer, 2003), and Finite Structure Query(Kepser, 2003). Specifically for PDT querying purposes, NetGraph (Mírovský, 2002) tool was previously developed. All these tools implement some form of predicates for basic tree relations between nodes. As far as their overall expressive power is concerned, they rank from the most restricted ones featuring only limited possibilities to combine respective constraints by logical operators (CorpusSearch, ICECUP III,

[8] Under CygWin software tool.

NetGraph) via more general ones (TGrep2, TIGERSearch, VIQTORYA) up to those that use the full power of first order logic (Finite Structure Query). Additionaly, some of these tool can be used to query more general structures than strict trees (VIQTORYA, Finite Structure Query).

We will compare the presented tool to Finite Structure Query, the tool featuring the most powerful query expressiveness so far, and NetGraph, the tool directly designed to query PDT treebank, in greater detail.

TSRF is comparable to the query language used by Finite Structure Query. Although TSRF does not use overt quantification (and is therefore not strictly first order logic based), for any first order logic formula (over vertex variables) there exists an equivalent formula that can be expressed by TSRF. However, TSRF is much less powerful than Finite Structure Query in terms of generality of the queried structures - TSRF operates only on strict trees whereas Finite Structure Query can operate on an arbitrary finite structure. The expressive power of a formalism depends of course also on the set of supported predicates. We believe that in this aspect the set of predicates present in TSRF at least matches the set offered by Finite Structure Query, we were at least able to express all the examples presented at (Kepser, 2003) in the TSRF query formula[9].

NetGraph uses its own form of query formula. This formula in fact directly represents an underspecified tree template rather than being a logical formula containing predicates. As described in (Mírovský, 2002) , NetGraph is obviously less powerful than TSRF as its query formula (when stated in logical terms) features only a restricted disjunction and no negation at all (thus it forms only a positive existential fragment). However, it is known to us that NetGraph was recently significantly improved as it developed a powerful functionality within its underspecified template (e.g. a specific type of negation, a form of attribute value unification etc.). So at least for the purposes of querying PDT for relevant linguistic phenomena it may now be as suitable as TSRF.

8. Conclusion

The expressive power of presented formalism is able to capture complex linguistic structures in the tree structures and is comparable to the currently most advanced tools available. However, there are still structures which cannot be captured (such as two templates connected by a potentially infinite set of vertices with properties that cannot be expressed by the path predicate). Likewise, there are substitutions that cannot be performed (such as reversion of potentially infinite path within a tree). As far as the query formula is concerned, the expressiveness may be quite easily increased by extending the existing predicates or adding a new ones.

The implemented tool showed acceptable performance which makes it possible to use it to process even large data sources. However, many optimizations (such as indexing of the entire corpus in advance) can yet be performed. From the engineering point of view, the tool could support more formats, e.g. XML.

In summary, we see the contribution of the presented work mainly in the expressiveness of the query formula, in the elegant and intuitive way the rules are written (and their easy reversibility), and in the performance of the implemented tool.

[9]For example, we are not sure whether unification of attribute value variables supported by TSRF is supported by Finite Structure Query,too.

9. Acknowledgements

The development of the presented work has been supported by the following organizations and projects: the LC536 grant of the Ministry of Education of the Czech Republic, Information Society Project No. 1ET201120505 of the Grant Agency of the Academy of Sciences of the Czech Republic, and the Rodipas grant IIS-9982329 of the National Science Foundation of the USA.

10. References

J. Hajič, E. Hajičová, P. Pajas, J. Panevová, P. Sgall, B. Vidová-Hladká. 2001. *Prague Dependency Treebank 1.0. CDROM*. CAT:LDC2001T0. ISBN 1-58563-212-0.

J. Hajič, M. Čmejrek, B. Dorr, Y. Ding, J. Eisner, D. Gildea, T. Koo, K. Parton, D. Radev, and O. Rambow. 2002. *Natural language generation in the context of machine translation*. Technical report, Center for Language and Speech Processing, Johns Hopkins University, Baltimore. Summer Workshop Final Report.

Laura Kallmeyer, Ilona Steiner. 2003. *Querying treebanks of spontaneous speech with VIQTORYA*. Traitement Automatique des Langues. 43(2).

Stephan Kepser. 2002. *Finite Structure Query*. Proceedings of 10th Conference of The European Chapter of The Association for Computational Linguistics.

Esther König, Wolfgang Lezius. 2000. *A description language for syntactically annotated corpora*. Proceedings of the COLING Conference. 10561060.

Jiří Mírovský, Roman Ondruška, Daniel Pruša. 2002. *Searching through Prague Dependency Treebank*. Proceedings of Treebanks and Linguistic Theories. 114122.

Beth Randall. 2000. *CorpusSearch users manual*. Technical report, University of Pennsylvania. http://www.ling.upenn.edu/mideng/ ppcme2dir/.

Douglas Rohde. 2001. *Tgrep2*. Technical report, Carnegie Mellon University. http://tedlab. mit.edu/dr/Tgrep2/.

Zoltan Somogyi, Fergus Henderson, Thomas Conway. 1995. *Mercury: an efficient purely declarative logic programming language*. Proceedings of the Australian Computer Science Conference, Glenelg, Australia. 499-512.

Sean Wallis, Gerald Nelson. 2000. *Exploiting fuzzy tree fragment queries in the investigation of parsed corpora*. Literary and Linguistic Computing. 15(3):339361.