MonaSearch – Querying Linguistic Treebanks with Monadic Second-Order Logic

Hendrik Maryns and Stephan Kepser*

Collaborative Research Centre 441
University of Tübingen, Germany
{hendrik,kepser}@sfs.uni-tuebingen.de

Abstract

MonaSearch is a new powerful query tool for linguistic treebanks. The query language of MonaSearch is monadic second-order logic, an extension of first-order logic capable of expressing probably all linguistically interesting queries. In order to process queries efficiently, they are compiled into tree automata. A treebank is queried by checking whether the automaton representing the query accepts the tree, for each tree. Experiments show that even complex queries can be executed very efficiently. The tree automaton toolkit MONA is used for the computation of the automata.

1 Introduction

A frequent critique to most available query tools is their poor expressive power: most of them only incorporate a subset of first-order logic as a query language. Furthermore, often the semantics of the query language is not clearly defined; it is only defined by the functionality of the program. The importance of the expressive power of the query language is a consequence of the sizes of the available treebanks nowadays: tree banks of several tens of

*The work presented here is part of the project A2 in the Collaborative Research Centre 441 "Linguistic Data Structures" at the University of Tübingen, funded by a grant of the German Research Council (DFG SFB 441). We would like to thank Michael Jakl, Uwe Mönnich and Fang Wei for interesting and fruitful discussions. We are particularly grateful to Michael for pointing us to the function `mgCheck` in the MONA gta library. We also thank the anonymous reviewers for their remarks.

thousands of trees are no exception. It is clearly impossible to browse these treebanks manually searching for linguistic phenomena. But a query tool that does not permit the user to specify the sought linguistic phenomenon quite precisely is not too helpful, either. If the user can only approximate the phenomenon he seeks, answer sets will be very big, often containing several hundreds to thousands of trees. Weeding through answer sets of this size is still cumbersome and not really fruitful. If the task is to gain small answer sets, then query languages have to be powerful.

As a query language offering the necessary expressive power, we present Monadic Second-Order logic (henceforth MSO). [Kepser, 2004] describes its power and how it can be exploited to query tree banks in linear time. This paper describes how these ideas were successfully implemented to obtain the most powerful query engine for treebanks to date.

MSO is an extension of first-order predicate logic by set variables. These variables can be quantified over, representing (finite) sets of nodes in a tree. MSO offers a firm expressive power, while at the same time the automaton model discussed below assures the evaluation time of an MSO query on a tree is linear in the size of the tree.

An example of its strength is the ability to express the transitive closure of any binary relation which is definable in the language. Since MSO is an extension of first-order logic, any first-order relation can be expressed in it, and thus also the transitive closure of any first-order relation.

Transitive closures appear quite often and very naturally in linguistics. Most often, the need for them has been met by introducing atomic formulas which directly express the transitive relation. For the ancestor relation, for example, an atomic formula $x \lhd^+ y$ is introduced, next to the more basic parent relation $x \lhd y$. The limitation of this approach is that those relations are hard-coded in the logic. There is no way to define new ones apart from extending the language. In the case of a corpus query tool, this would mean the extra relation has to be implemented by the programmer himself. In MSO, this can be done dynamically by the user. As it is hard to foresee the need for such relations, this feature can come in handy.

An example illustrating this, which might be regarded as a 'natural' query for MonaSearch, is the following: Consider sentences ending in a sequence of prepositional phrases like

> The dog buried the bone [PP behind the tree [PP in the garden [PP in front of the house [PP at the end of the street]]]].

where the PPs are all embedded by one another as indicated. This contrasts with a structure like in

The dog buried the bone [PP with his paws] [PP under a stone] [PP behind the tree] [PP in the afternoon].

where each PP is an independent adjunct of the verb. Both examples can stepwise be extended to sequences of PPs of arbitrary length. The query for an arbitrary number of embedded PPs is not definable in first-order logic, but can be defined in MSO. We can define the transitive closure of a PP dominating an NP as a new relation and state that the highest PP node and the most deeply embedded NP node stand in this new relation.

Another property of MSO which comes in nicely is its decidability over trees[1]. This assures that, when a query gives no results, this effectively means no tree in the tree bank satisfies the formula, rather than that no tree was found.

In this paper, we present MonaSearch, a query tool which implements MSO as a query language using automata techniques. This way we show that the theoretical considerations are indeed also practically valid. We compare run times of a set of linguistically motivated queries to show that MonaSearch is not only powerful, but also faster than its main competitors.

MonaSearch is publicly available at `http://tcl.sfs.uni-tuebingen.de/MonaSearch`.

2 Overview of the Querying Process

What makes MSO usable as a query language is the existence of an automaton model for this logic. The type of automata that correspond to MSO is bottom-up tree automata. For a general introduction to tree automata, we refer the reader to Comon et al. [2007]. The connection between MSO and tree automata is very strong. A set of trees is definable by an MSO formula if and only if there exists a tree automaton accepting this set. This equivalence is constructive: there is an algorithm that constructs an automaton from a given MSO formula.

Hence the general evaluation strategy of MonaSearch can be summarized as follows: 1. convert the query from the user into a tree automaton, 2. run the automaton on each tree in the tree bank. In addition, some auxiliary steps are necessary to smooth computation.

This strategy has the advantage that checking whether an automaton accepts a tree is very fast, viz. linear in the size of the tree.

[1] More precisely, over graphs with bounded tree width.

3 MONA

The largest and most demanding subpart of this process is the development of a tree automata toolkit, a toolkit that compiles formulas into tree automata by performing standard operations on tree automata such as union, intersection, negation, and determinization. The most notable and successful is MONA [Klarlund and Møller, 2001]. Although developed for hardware verification, MONA can be used to query treebanks.

The language of MONA is pure monadic second-order logic of two successors. Note that there is no way to extend this language. This has important consequences: Firstly, we are restricted to using *binary* trees only; secondly, it cannot handle node labels.

The main part of MONA is a compiler that compiles formulas in an idiosyncratic logical language into a specific variant of tree automata. The input is a file containing the formulas. The default output is some information on whether the formula is satisfiable or not, it is however possible to let it output a representation of a compiled automaton into a file. Additionally, MONA offers a small library with functions that allow one to load those precompiled automata from file and to check if an automaton accepts a binary coded tree.

The strategy for employing MONA to query treebanks consists of two parts: Since we don't want to burden the user with learning the complex MONA language, a query from the user in MSO is translated into a MONA formula, and this formula is compiled into a tree automaton by the main program. The library functions are then used to check each tree in the treebank for acceptance. The trees from the treebank have to be transformed in a suitable way described below such that they can be used as input to MONA's particular type of automata.

4 MonaSearch

In this section, we'll go into some more detail about the steps involved in a query and the preparatory tasks, and explain how they are handled in MonaSearch.

4.1 Precompilation of the Treebank

Linguistic treebanks are usually provided in some type of data exchange format. This format is typically unsuitable for direct querying. Plain text files

are not suitable for efficient processing by their nature. Bouma and Kloosterman [2007] propose an efficient way of handling XML-encoded treebanks, but it requires the treebank to be in a specific format. E.g. the TüBa-D/Z [Telljohann et al., 2004] and TIGER [Brants et al., 2004] treebanks cannot be queried in that way. MonaSearch does not suffer from this restriction.

In the case of MonaSearch, there is a particular reason for a precompilation step. As mentioned before, MONA can only cope with *proper binary* trees. When translating trees from the treebank into MONA trees we consider proper trees only. Many treebanks contain more complex structures than proper trees. At the current stage of the development of MonaSearch we simplify these structures as follows: 1. secondary relations are ignored, 2. disconnected subparts are integrated into the tree by introducing a new virtual root and connecting the disconnected parts to this super root and 3. crossing edges are ignored by taking only the order of the children as seen by the parent into account. In principle, these restrictions can be released: There exists a more general method that encodes tree-like structures into proper binary trees, as is illustrated in [Kepser, 2004].

As stated above, the precompilation of trees into formulas has to perform two tasks: 1. the trees, which are arbitrarily branching, must be transformed into binary trees and 2. the linguistic labels have to be taken care of.

For the transformation into binary trees, we employ the First-Child-Next-Sibling encoding, a rather standard technique. An arbitrary node x in the original tree is transformed into x' in the binary tree as follows:

- If x has any children, call its leftmost child y, then y' will become the left child of x'.
- If x has any right siblings, call the leftmost one z, then z' will become the right child of x'.

For example, the left tree in fig. 1, stemming from the TüBa-D/Z treebank, is transformed into the tree on the right. For real trees, the original tree can easily be recreated from this binary form; in fact, that is what the formula translation described later is implicitly doing. The simplifications mentioned before, however, are not retrievable from the binary form and are therefore not available for querying. Since we refer to trees using an identifier, we do not need to do this back-translation explicitly, we simply use the original tree for further processing.

Note how the disconnected punctuation node at the lower right corner in the original tree becomes the right child of the SIMPX node in the transformed tree; it is treated as if it was its sibling in the original tree.

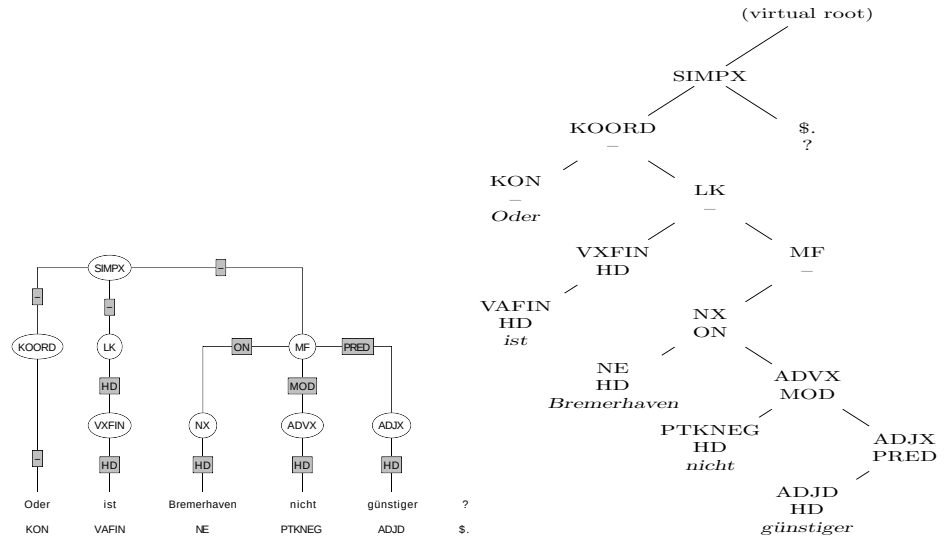

Figure 1: An example sentence from TüBa-D/Z and its binary coding.

As can be seen in the figure, edge labels are moved down to the node below it; thus, a node in the translated tree carries a set of labels. This is necessary either way, since even in the original tree, the leaves carry multiple labels. We'd like to mention that it is only a coincidence of the chosen annotation scheme and source tree that all and only the leaf nodes of the binary tree are labelled with words. This need not be the case in general.

Presently, labels for the category of a node, its function, the word at a leaf, its lemma or its morphological information occur, however, other labels can be treated the same way.

The preprocessing of the labels is still not enough to enable MONA to handle the binary trees. The trees it expects contain bit vectors as labels, which are closely related to the translation process. These bit vectors are, however, dependent on the specific query and are therefore generated while querying.

4.2 Querying

The detailed steps of a query are as follows:

1. getting a query from the user;
2. translating the query into a MONA formula on binary trees;
3. compiling the MONA formula into a MONA tree automaton;

Relation	Formula	Translation
parenthood	$x \lhd y$	`ex1 z: x.0 = z & right_branch(z,y)`
precedence	$x \prec y$	`ex1 z: ( x = z | dom(z.0,x) ) & dom(z.1,y)`
dominance	$x \lhd^+ y$	`ex1 z: x.0 = z & dom(z,y)`

Table 1: MONA translation of the more involved base relations.

4. for each tree from the precompiled treebank:
   1 preparing the tree for the query,
   2 running the automaton on the translated tree, noting whether it is accepted or not;
5. presenting the results to the user.

Steps 1 and 5 are handled in a graphical user interface.

For step 2 one has to keep in mind that the user formulates his query on the original treebank, but the trees that are used for querying are the binary coded trees. Hence it is necessary to translate the original query ϕ in such a way into a MONA formula ϕ' that the original query ϕ is true for an original tree t if and only if the translated formula ϕ' is true on the binary recoding t' of t.

This step also has the advantage that it makes us independent of the idiosyncratic MONA language. We can offer the user an easily understandable, semantically clear language that is almost the pure MSO logic. It also makes extending the query language with typical linguistic relations such as c-command relatively easy.

For most connectives, the MONA counterpart can be taken over directly. This is the case for all boolean connectives, quantification and some of the atomic relations. The main problem lies with the relations that express something about the shape of the tree, dominance and precedence being the most relevant. Their translation uses two auxiliary predicates on the binary tree, which can be defined in the MONA language:

- `dom`(x, y) expresses that x dominates y in the binary tree.
- `right_branch`(x, y) expresses that y lies at the branch of right children starting at x.

Note that both express the transitive closure of a first-order relation (parenthood and right child, respectively) and as such cannot be expressed in first-order logic.

Table 1 shows the translation of the parent and precedence relations on the original trees into a formula on binary trees in the language of MONA; see the user manual for the notation. Note that the dominance relation could equally well be expressed as the transitive closure of the parent relation. We

include it since it turns out that a direct translation yields a simpler formula.

Finally, all types of linguistic labels are treated as free set variables. This yields a MONA formula ϕ'.

In step 3, the MONA formula is compiled into a tree automaton by a call to the MONA compiler. This implies that MONA has to be installed on the system in order for MonaSearch to work.

Step 4 comprises the actual querying of the treebank. The aim is to run the tree automaton on each binary coded tree.

Remember that MONA is not able to handle the preprocessed trees directly. The process of compiling a formula into an automaton translates variables into bit vectors. More precisely the label of each node in a tree processable by the automaton is a bit vector. The index of the vector states which variable is stored at that place. The value (0 or 1) indicates if the node is in the set of this variable. Since linguistic labels are treated as set variables, each binary tree must undergo this translation process. We impose some (arbitrary, but fixed) order on the labels. The length of the bit vector is the number of labels in the query. The index states which particular label is referred to. Now consider a node in the binary tree and its original set L of labels. The new label is a bit vector. If the index refers to a label in L, the bit is set to 1, otherwise it is set to 0. As a consequence, the bit vectors in the translated tree only refer to labels occurring in the query.

After this translation step the input tree is finally ready for processing. The MONA library contains functions that load the precompiled tree automaton from file and apply it to a translated tree returning true or false depending on whether the tree automaton accepts the tree or not.

It remains to solve one small technicality. Remember that in the binarization process, a virtual root node was introduced to connect disconnected parts of the 'tree'. Care has to be taken to ignore this node when evaluating the query, in order not to skew the results. Here, fortune has smiled upon us: the particular form of tree automaton MONA uses happens to do just that when evaluating a simple tree: it ignores the root node and starts looking at its left child. This left child will necessarily be the first root node of the original tree, no further tweaking is needed.

4.3 User interface

MonaSearch offers MSO as a query language. The usual logical connectives are provided, along with first- and second-order quantification. For the atomic formulas, those relevant for relations of nodes in a tree are chosen, along with relations that express relations between nodes and node sets.

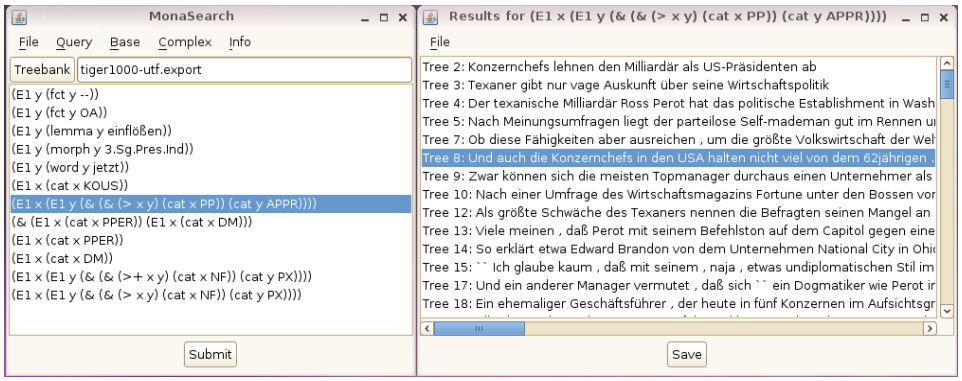

Figure 2: The MonaSearch GUI: compose and result window

Furthermore, (linguistic) labels of nodes can be queried. First-order variables are supposed to range over nodes, whereas second-order variables, the particular property of MSO, range over sets of nodes.

The user can compose these formulas stepwise in a bottom-up fashion in the graphical user interface, which acts like a scrap book of formulas that can be edited, connected using the logical connectives, and submitted for querying. Base formulas can be entered from menus.

We illustrate the query language by an example of a query. In most main sentences in German – as in English – the subject precedes the inflected verb. In yes/no-questions, this order is reversed. To find trees where the verb precedes the subject in the TüBa-D/Z treebank, the following query may be used:

$$\exists x, y, z(cat(x) = \text{SIMPX} \wedge cat(y) = \text{VXFIN} \wedge fct(z) = \text{ON} \wedge x \lhd^+ y \wedge x \lhd^+ z \wedge y \prec z)$$

"There exist a node with category SIMPX, a node with category VXFIN and a node with function ON; the first dominates the other two, and the second comes before the third." (*ON* codes the grammatical function *subject* in the TüBa-D/Z.) The tree in fig. 1 is a sample match for this query.

5 Performance

In order to give an impression of the performance of MonaSearch we compare query times for a series of queries for three query tools: TIGERSearch [Lezius, 2000], fsq [Kepser, 2003], and MonaSearch. These tools were chosen

Query	TIGERSearch		fsq		MonaSearch	
1	5.5	5.5	23	13.5	15	10
2	9	5.5	23	13.5	15	10
3	15	16	23	13.5	15	10
4	–	–	23	13.5	15	10
5	–	–	–	–	15	10

Query times in seconds. For each query tool, the left column contains the query time on the TIGER treebank, the right column the one on the TüBa-D/Z. A dash (–) indicates that the query language of the tool is not powerful enough to express the query.

System: Pentium D, 3.2GHz, 1GB Ram, OS: Linux openSUSE 10.3

Table 2: Query times of MonaSearch in comparison to other tools

because they can be used to query any treebank and already have rather high expressiveness. We tested their forces on the TIGER treebank and on TüBa-D/Z. The queries are ordered by complexity starting with simple ones.

1. An NP dominating an S dominating a PP.
2. An NP dominating an S dominating a PP and an NP, which do not dominate each other.
3. Sentences where the verb precedes the subject.
4. An NP *not* dominating an S which dominates a PP.
5. A PP dominating an NP which is part of a chain of embedded PPs (see introduction).

Table 2 shows that MonaSearch performs quite well. TIGERSearch may be faster on simple queries, but its deficit in expressive power makes it unsuitable for querying more advanced linguistic phenomena. In fsq, such phenomena can be queried, but MonaSearch is faster and even more powerful. Furthermore, fsq requires a certain skill in composing a query. Processing a query with quantifier depth 4 takes 23 seconds when the query is composed optimally. But it can also take 46 minutes, when the query is composed in a less skilled fashion. MonaSearch does not suffer from this problem. The processing time of logically equivalent queries is largely independent of the particular phrasing of the queries. This makes MonaSearch significantly easier to use.

Since all trees are evaluated separately, the evaluation time of a query is linear in the number of trees in the treebank. Effectively, a huge treebank with one million trees can be queried in about five minutes, i.e., quite fast.

6 Related Work

Alternative query tools and their performance have just been discussed in the previous section. The aim of this section is to explain how the current approach differs from the one in [Kepser, 2005]. Kepser [2005] explores a different strategy for using MONA to query linguistic treebanks. This different strategy proved unsuccessful. The reason for this being that trees were encoded as MONA automata featuring all linguistic labels of the treebank. As a result, the automata which encoded the trees got so big they could not be processed any more. The current approach does *not* try to code trees as automata.

The current approach avoids large label sets on trees by equipping each tree that is to be checked by an automaton with only those labels which are relevant to the current query, as described in step 4.1 in section 4.2. By avoiding these pitfalls the method presented in this paper is successful.

7 Conclusion and Future Work

In this paper, we presented MonaSearch, a powerful tool for querying linguistic treebanks. The two main features of MonaSearch are the very high expressive power of the query language and the high performance of the query engine. The query language is monadic second-order logic. This is – to our knowledge – the most expressive query language offered in any query tool useful for linguists. MonaSearch is on the other hand also the fastest query system available for advanced queries. Thanks to MonaSearch's speed it is for the first time within reach to query automatically annotated treebanks with several million trees with non-trivial queries.

Future work has two directions. Firstly, usability will be enhanced by simplifying the syntax of the query language, extending the range of treebank input formats and enhancing the GUI. An important component would be a graphical treebank browser. Secondly, it would be worth investigating whether the restriction to proper trees can be overcome. As stated, there exists a theoretical solution for this problem. But the recoding of trees and formulas that is involved in this solution is very complicated. Formulas may get very large. It is therefore an open question whether these formulas can still be compiled into automata. But the potential advantage of coping with arbitrary tree-like structures should make it worthwhile to investigate the practicability of this solution.

References

Gosse Bouma and Geert Kloosterman. Mining syntactically annotated corpora with xquery. In *Proceedings of the Linguistic Annotation Workshop*, pages 17–24, Prague, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W07/W07-1503`.

Sabine Brants, Stefanie Dipper, Peter Eisenberg, Silvia Hansen, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit. TIGER: Linguistic interpretation of a German corpus. *Research on Language and Computation*, 2(4):597–620, 2004.

Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2007. URL `http://tata.gforge.inria.fr/`.

Stephan Kepser. Finite Structure Query: A tool for querying syntactically annotated corpora. In Ann Copestake and Jan Hajič, editors, *Proceedings EACL 2003*, pages 179–186, 2003.

Stephan Kepser. Querying linguistic treebanks with monadic second-order logic in linear time. *Journal of Logic, Language, and Information*, 13: 457–470, 2004.

Stephan Kepser. Using MONA for querying linguistic treebanks. In Chris Brew, Lee-Feng Chien, and Katrin Kirchhoff, editors, *Proceedings HLT/EMNLP 2005*, pages 555–563, 2005.

Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. URL `http://www.brics.dk/mona/`. Notes Series NS-01-1. Revision of BRICS NS-98-3.

Wolfgang Lezius. Tigersearch – ein suchwerkzeug für baumbanken. In Stephan Busemann, editor, *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002)*, Saarbrücken, 2000.

Heike Telljohann, Erhard W. Hinrichs, and Sandra Kübler. The TüBa-D/Z treebank – annotating German with a context-free backbone. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 2229–2232, 2004.