Approximating Principles and Parameters Grammars with MSO Tree Logics

Frank Morawietz and **Tom Cornell**
Seminar für Sprachwissenschaft
Universität Tübingen
Wilhelmstr. 113
72074 Tübingen, Germany
{frank,cornell}@sfs.nphil.uni-tuebingen.de

Introduction. In recent years there has been a continuing interest in computational linguistics in both model theoretic syntax and finite state techniques. The (weak) monadic second-order logic (MSO) of two successor functions (WS2S) is the extension of first-order logic with quantification over (finite) sets, i.e., monadic predicates. The decidability proof for WS2S furnishes a technique for deriving (bottom-up) tree automata for recognizable relations from the MSO constraints effectively (Thatcher and Wright 1968, Doner 1970).[1] The tree automata represent the relations in the sense that they recognize their satisfying assignments. While these results were known for some time, the suitability of this formalism as a constraint language for Principles-and-Parameters (P&P) based linguistic theories has only recently been shown by Rogers (1994).

The abstract proceeds by first giving an example of an MSO constraint and the corresponding tree automaton, followed by a discussion of possible uses of the compilation technique for natural language processing. After the outline of the inherent problems of a naïve approach, we discuss the parameterization of the grammar for a particular input as an approximation to the specification of a P&P grammar. We conclude by applying a constraint logic programming (CLP) extension—resulting from the embedding of the compiler as its constraint solver—to remaining engineering problems. We assume familiarity with both the decidability proof for WS2S (a good presentation is in Gurevich 1985) and tree automata (an introduction can be found in Gécseg and Steinby 1984).

As an example for MSO logic in practice consider the formula denoting directed asymmetric c-command.[2] We use the more natural tree logic signature of Rogers which is interdefinable with S2S. Uppercase letters denote second order, lowercase ones first order variables, $\lhd^*$ reflexive and $\lhd^+$ proper domination and $\prec$ precedence:

$$
\begin{array}{ll}
 & \mathsf{AC\text{-}Com}(x,y) \stackrel{def}{\Longleftrightarrow} \\
 & \quad \%\ x \text{ c-commands } y\text{:} \\
 & (\forall z)[z \lhd^+ x \Rightarrow z \lhd^+ y] \wedge \neg(x \lhd^* y) \wedge \\
(1) & \quad \%\ y \text{ does not c-command } x\text{:} \\
 & \neg((\forall z)[z \lhd^+ y \Rightarrow z \lhd^+ x] \wedge \neg(y \lhd^* x)) \wedge \\
 & \quad \%\ x \text{ precedes } y\text{:} \\
 & x \prec y
\end{array}
$$

The corresponding tree automaton is shown in equation (2).

$$
(2) \qquad
\begin{aligned}
\mathcal{A} &= \langle A, \Sigma, a_0, F, \alpha \rangle \\
A &= \{a_0, a_1, a_2, a_3, a_4, a_5\} \\
\Sigma &= \{[x,y], [x,\neg y], [\neg x, y], [\neg x, \neg y]\} \\
F &= \{a_4\},
\end{aligned}
$$

$$
\begin{array}{ll}
\alpha(a_0,a_0,[\neg x,\neg y]) = a_0 & \alpha(a_0,a_0,[\ x,\neg y]) = a_3 \\
\alpha(a_0,a_0,[\neg x,\ y]) = a_1 & \alpha(a_0,a_1,[\neg x,\neg y]) = a_2 \\
\alpha(a_0,a_2,[\neg x,\neg y]) = a_2 & \alpha(a_0,a_4,[\neg x,\neg y]) = a_4 \\
\alpha(a_1,a_0,[\neg x,\neg y]) = a_2 & \alpha(a_2,a_0,[\neg x,\neg y]) = a_2 \\
\alpha(a_3,a_2,[\neg x,\neg y]) = a_4 & \alpha(a_4,a_0,[\neg x,\neg y]) = a_4 \\
\multicolumn{2}{c}{\text{all other transitions are to } a_5}
\end{array}
$$

On closer examination of the transitions, we note that we just percolate the initial state as long as we find only nodes which are neither x nor y. From the initial state on both the left and the right subtree we

[1] It has to be pointed out immediately that the translation from formulas to automata, while effective, is of non-elementary complexity. In the worst case, the number of states can be given as a function of the number of variables in the input formula with a stack of exponents as tall as the number of quantifier alternations in the formula.

[2] This relation is not monadic, but reducible via syntactic substitution to an MSO signature. In fact, we can define and use relations of any arity as long as they are explicitly presentable in MSO logic, but we cannot quantify over them.

can either go to the state denoting "found x" (a_3) if we read symbol $[x, \neg y]$ or to the state denoting "found y" (a_1) if we read symbol $[\neg x, y]$. After finding a dominating node—which switches the state to a_2—we can then percolate a_2 as long as the other branch does not immediately dominate x. When we have a_3 on the left subtree and a_2 on the right one, we go to the final state a_4 which again can be percolated as long as empty symbols are read. Clearly, the automaton recognizes all trees which have the desired c-command relation between the two nodes. It compactly represents the (infinite) number of possible satisfying assignments.[3] The formula given for AC-Com is far from the most compact and elegant definition of that relation. It was chosen because it is a straightforward formalization of the prose definition of the relation. Nonetheless, the automaton compiled from a much cleverer formalization would still be the same.[4] That is, minimized automata are normal forms, and, since they allow for the direct and efficient recovery of particular solutions, they make appropriate solved forms for use in constraint programming.

Approximating P&P Grammars. If one is interested in putting formalized theories of grammatical competence to use in processing (e.g., parsing), an obvious question is whether it would be possible to generate a tree automaton representing an entire grammar. If we could formalize a P&P grammar in an MSO tree logic, we could apply the compilation techniques to construct an automaton which recognizes all and only the valid parse trees. In this setting, the parsing problem is—as standardly for satisfiability based grammars—the conjunction of a formula characterizing the input with a grammar formula. Compiling this conjoined formula would result in an automaton which recognizes the parse forest. The recognition problem would just be the problem of determining whether or not the resulting automaton recognizes a nonempty language. Since the automaton would represent the parse forest, we could run it to generate parse trees.

Unfortunately this cannot work, in principle, since in the language of WS2S we can only define *recognizable* sets, which yield only the context-free string languages. Since natural language is known not to be context-free—at least for such languages as Swiss-German or Dutch—we cannot write a grammar formula for these languages. Nonetheless, we can still use such a formalism to deal with practical problems in theoretical and computational linguistics. Most obviously, even if the entire grammar is undefinable in an MSO tree logic, it is still the case that very many grammatical properties besides global well-formedness may be definable using only MSO means. Our approach is inspired by the standard approach to verification of linguistic theories. Theory verification can be thought of ideally as getting a theorem prover to enumerate the particular consequences of a theory of well-formedness, which we presumably would compare theorem-by-theorem to the judgments of native speakers. Such a scenario is of course a complete idealization. In fact, the best we can do in practice is to probe the theory with a certain small set of examples, either grammatical or ungrammatical, which are believed to be crucial.

In more detail, while the set of well-formed trees is not in general recognizable, it seems reasonable to assume that the parse forest for any given natural language sentence is recognizable (indeed, finite). So we should be able to write an MSO theory which correctly recognizes or fails to recognize that sentence. The question is whether that theory bears any detectable relationship to the underlying set of grammatical principles and parameters. The beauty of a nice modular theory of grammar is that the parts we cannot state once for all sentences but must state and restate for each individual sentence are easy to isolate from the rest of the grammar, i.e., we have to separate those parts of the grammar which actually contain information regarding the analysis of non-context-free structures and those that don't.

Finite tree automata can only keep track of a bounded number of distinct objects. For P&P theories, as Rogers shows, what pushes them beyond the boundary of the recognizable sets is the ability, via Free Indexing, to keep track of unboundedly many distinct chains. It seems rather plausible that for each particular input we can find an upper bound of the number of indices which can appear in its parse trees, though working this out for any particular theory may be quite tricky.[5] Given a bounded set of indices which we can use on a particular parse, we can list them schematically in a disjunction as in equation (3) and specify co-indexation as in equation (4). Since in MSO logic constants are equivalent to free variables, we have to be careful to represent them in both the head of the formula and the resulting

[3] The automaton was constructed by a simple implementation of a compiler developed at the University of Tübingen (Morawietz and Cornell 1997).

[4] The structure of the formula *does* often have an effect on the time required by the compiler.

[5] Note that we want to be able to make very accurate predictions here. The alphabet our automata use must distinguish all possible combinations of features, and alphabet size can be a source of explosiveness in the size of the automata.

automata to preserve their satisfying assignments.[6] Here, $\vec{I}_n$ stands for the n indices we have to create depending on the input.

$$\text{(3)} \quad \mathsf{Index}(X, \vec{I}_n) \stackrel{def}{\Longleftrightarrow} (X = I_1) \vee (X = I_2) \vee \cdots \vee (X = I_n)$$

$$\text{(4)} \quad \mathsf{Co\_Idx}(x, y, \vec{I}_n) \stackrel{def}{\Longleftrightarrow} (\exists X)[\mathsf{Index}(X, \vec{I}_n) \wedge x \in X \wedge y \in X \wedge (\forall Y)[\mathsf{Index}(Y, \vec{I}_n) \Rightarrow (X = Y \vee \mathsf{Disjoint}(X, Y))]]$$

In theory, at least, we can compile the independent parts of the grammar into a tree automaton. After the instantiation of the schematic representation, we can compile an automaton for the input and index formulas as well. Now the construction of a grammar-and-input automaton becomes possible, only now it is a *ternary* conjunction: input description, partial grammar specification and index theory (e.g., ECP, Binding Theory, Index, Co_Idx, ...). As long as our index-estimator function is accurate (or at worst generous), a series of sentence-verifications gives results equivalent to verifying a fully specified grammar formula against the same sentences. Note that if we could treat the indices as real constants, i.e., we find a type of automaton which can handle constants, the modularization ensures that only the clause for Index is dependent on the input such that our partial grammar automaton would also contain the rest of the index theory. Alas, although it would be a far more elegant solution, this has to be left for further research.

MSO Logic and CLP. We still have a profound engineering problem to overcome, namely the non-elementary complexity of the formula-to-automaton compilation. This casts some doubt on whether we can (pre)compile the partial grammar automaton, even to the extent that we can define it, and therefore we cannot construct a parse-forest automaton simply by conjoining the input automaton, partial grammar automaton and index automaton. It makes sense to search for ways to incrementally construct the (much smaller!) parse-forest automaton for a particular input. We propose the embedding of the MSO constraint language into a CLP scheme. Intuitively, the constraint base is an automaton which represents the incremental accumulation of knowledge about the possible valuations of variables constructed by applying and solving clauses with their associated constraints. That is, we actually use the compiler *on line* as the constraint solver. Some obvious advantages include that we can still use our succinct and flexible constraint language with negation and monadic second order quantification, but gain (a) a more powerful language, since we now can include inductive definitions of relations, and (b) a way of guiding the compilation process by the specification of programs. An additional benefit of (b) is that we thereby automate human-machine interaction which is used in some approaches to speed up or enable (higher-order) theorem proving.

We define a relational extension $\mathcal{R}$(WS2S) of our constraint language following Höhfeld and Smolka (1988). From their scheme we get a sound and complete, but now only semi-decidable, operational interpretation of a definite clause based derivation process (see Morawietz 1997). The resulting structure is an extension of the underlying constraint structure with the new relations defined via fixpoints. For simplicity we assume a standard left-to-right, depth-first interpreter for the execution of the programs. The solution to a search branch of a program is a satisfiable constraint, represented in solved form as an automaton.

To demonstrate how we split the work between compiler and CLP interpreter, we present as a simple example the following naïve specification of a lexicon:

$$\text{(5)} \quad \mathsf{Lexicon}(x, Sees, V, \ldots) \stackrel{def}{\Longleftrightarrow} (x \in Sees \wedge x \in V \wedge \ldots) \vee (x \in John \wedge x \in N \wedge \ldots) \vee \ldots$$

Naïvely, at least, every feature we use (e.g., *Sees*), has to be coded into the alphabet Σ of the automaton. So, $|\Sigma|$ with an encoding as bitstrings will be 2^n (n the number of free variables). It is immediately clear that the compilation of such an automaton is extremely unattractive, if at all feasible. But we can avoid having to compile the whole lexicon by having the obvious clauses for each lexical entry in the CLP extension.

$$\text{(6)} \quad \begin{array}{l} \mathsf{lexicon}(x, Sees, \ldots) \longleftarrow \\ \quad \{x \in Sees \wedge x \in V \wedge \ldots\} \\ \mathsf{lexicon}(x, John, \ldots) \longleftarrow \\ \quad \{x \in John \wedge x \in N \wedge \ldots\} \\ \quad \vdots \end{array}$$

This shifts the burden of handling disjunctions to the interpreter, but it is no longer the case that every

[6] If one is not interested in the information provided by a particular variable, it can be bound existentially at the top level of the grammar-and-input formula.

constraint in the grammar has to be expressed in one single tree automaton. We only compile those into the constraint store which are needed.

We can also use offline compiled modules in a $\mathcal{R}$(WS2S) parsing program. As a simple example, we present the definition of P&P parsing from Johnson (1995), see equation (7). We have to define any of the subgoals in the body either via precompiled automata (so they are essentially treated as facts), or else with more standard definite clause definitions.

$$\begin{aligned}(7)\qquad & \mathsf{parse}(Words, Parse, \vec{V}) \longleftarrow \\ & \quad \{\mathsf{Tree}(Parse)\}\ \& \\ & \quad \mathsf{yield}(Words, Parse, \vec{V})\ \& \\ & \quad \mathsf{xbar}(Parse, \vec{V})\ \& \\ & \quad \mathsf{ecp}(Parse, \vec{V})\end{aligned}$$

$\vec{V}$ stands for the free variables which are added to the constraint store during the program resolution. *Words* denotes a set of nodes labeled according to the input description. Our initial constraint base—which can be automatically generated from a list of input words—is the corresponding tree automaton. The associated constraint Tree is easily compilable and serves as the initialization for our parse tree. The yield predicate can be explicitly defined, compiled and treated as a fact. The xbar predicate, on the other hand, is a disjunctive specification of licensing conditions depending on different configurations, which is more intuitively expressed as several separate rules. In fact, since we want the lexicon to be represented as clauses (and xbar anchors the projections in the lexicon), we cannot have xbar as a pure MSO constraint. Since the ecp is dependent upon the actual number of indices, it has to be schematically expressed. But it can be realized as either a fact or a number of recursive rules which traverse the structure of the parse tree as in Johnson's proposal.

The approach interleaves the intersection of the grammar and the input so that only the minimal amount of information needed to determine the parse is incrementally stored in the constraint base. The extension to the formalism retains most of the properties of MSO but avoids some of the explosiveness of the compilation. We have to admit though, that there are still many (engineering) problems left. At the moment, at least, the question of which grammatical properties can be compiled in a reasonable time is largely empirical. It is made even more difficult by the lack of high quality software tools,[7] since our own prototype is sufficient only for the most basic needs.

Acknowledgements. This work has been supported by the project A8 of the SFB 340 of the Deutsche Forschungsgemeinschaft. We wish especially to thank Uwe Mönnich and Jim Rogers for discussions and advice and three anonymous reviewers for comments. Needless to say, any errors and infelicities which remain are ours alone.

References

Doner, J. (1970). Tree acceptors and some of their applications, *J. of Comp. and Sys. Sci.* **4**: 406–451.

Gécseg, F. and Steinby, M. (1984). *Tree Automata*, Akadémiai Kiadó, Budapest.

Gurevich, Y. (1985). Monadic second-order theories, *in* J. Barwise and S. Feferman (eds), *Model-Theoretic Logics*, Springer, Heidelberg, pp. 479–506.

Henriksen, J. G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T. and Sandhol, A. (1995). MONA: Monadic second-order logic in practice, *in* Brinksma et al. (eds.), TACAS '95, LNCS 1019, Springer, pp. 89–110.

Höhfeld, M. and Smolka, G. (1988). Definite relations over constraint languages, *LILOG Report 53*, IBM Deutschland, Stuttgart, Germany.

Johnson, M. (1995). Constraint-based natural language parsing, ESSLLI '95, Barcelona.

Morawietz, F. (1997). Monadic second order logic, tree automata and constraint logic programming, *Arbeitspapiere des SFB 340 86*, SFB 340, Universität Tübingen.

Morawietz, F. and Cornell, T. L. (1997). On the recognizability of relations over a tree definable in a monadic second order tree description language, *Arbeitspapiere des SFB 340 85*, SFB 340, Universität Tübingen.

Rogers, J. (1994). *Studies in the Logic of Trees with Applications to Grammar Formalisms*, PhD thesis, University of Delaware. Published as CS-Technical Report No. 95-04.

Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata theory with an application to a decision problem of second-order logic, *Math. Sys. Theory* **2**(1): 57–81.

[7] This situation should be alleviated when work on MONA^{++} at the University of Aarhus is completed; the usefulness of its older sister MONA (Henriksen et al. 1995), which works on strings and FSA's, has been well demonstrated in the computer science literature.