Arbeitspapiere des SFB 340
Bericht Nr. 00, Xxxx 19??

MONADIC SECOND-ORDER LOGIC, TREE AUTOMATA AND CONSTRAINT LOGIC PROGRAMMING

FRANK MORAWIETZ

ABSTRACT. In this paper we present a first step toward the development of a constraint logic programming (CLP) language $\mathcal{R}(\mathcal{MSO})$ based on monadic second-order (MSO) logic. We apply the scheme proposed by Höhfeld and Smolka (1988) to obtain a relational extension of MSO logic with a corresponding sound and complete operational semantics. The solutions to constraints expressed in monadic second-order logic are represented by tree automata recognizing the variable assignments satisfying the formulas. Since this allows a compact and efficient representation of constraints, we use the tree automata as a constraint store and MSO theorem provers as constraint solvers for the CLP interpreter. Building on the detailed investigation of MSO logic as a tree description language in the Principles and Parameter (P&P) paradigm in Rogers (To appear), we present as motivation and applications details from the realm of P&P-based parsing.

1. INTRODUCTION

Model-theoretic techniques are the underlying principle of a number of grammar formalisms for natural language. In this paper, we focus on the use of monadic second-order (MSO) logic for formalizations of Principle and Parameter (P&P) approaches in the Government and Binding tradition (Chomsky, 1981). The suitability of this formalism for P&P grammars has been shown in a recent monograph by Rogers (To appear).

The research presented in this report was sponsored by Teilprojekt A8 of the SFB 340 of the DFG. I want to thank Tom Cornell, Laura Kallmeyer, Hans-Peter Kolb, Uwe Mönnich, Stefan Riezler and Jim Rogers for helpful discussion. Needless to say, all errors and infelicities that remain are my own.

ISSN 0947-6954/??

Constraint-based formalisms characterize objects with logical description languages declaratively., i.e., without the specification of how to generate admissible structures. To be able to use them in applications, computational linguistics has to provide a connection between model theory and theorem proving on the one hand, and natural language processing on the other. We can bridge the gap between the two by exploiting the connection between constraints in MSO logic on trees and tree automata.

Our purpose in this paper is twofold. Firstly we discuss the operational interpretations of MSO logic. Since the solutions to constraints expressed in MSO logic are represented by tree automata which recognize the assignment trees satisfying the formulas, we can directly use the automata as the operational interpretation of our formalism; the aim of the paper being the exploitation of Rogers' formalization in relation to tree automata. And secondly we discuss possible extensions of MSO logic via the addition of inductive definitions with respect to both the operational interpretation and the generative capacity, since MSO logic characterizes only the context-free languages, but natural languages are generally believed to be higher in the Chomsky hierarchy.

More concretely, since inductive definitions can be seen as special cases of constraint logic programming (CLP) clauses and the specification of a CLP interpreter re-introduces the operational concept of derivations, we propose the embedding of MSO logic into the constraint logic programming scheme of Höhfeld and Smolka (1988) to define an even more powerful and flexible language. This CLP language allows a stepwise compilation of tree automata guided by appropriate programs and dependent on an initial goal.

The paper is organized as follows. In the first section we review the results on MSO logic and tree automata briefly discussing advantages and disadvantages of the formalism. The next section is centered on constraint logic programming. We apply the Höhfeld and Smolka scheme to derive a relational extension, discuss its properties and present some examples. In the last section, we argue for the usefulness of the approach with examples from P&P-based parsing.

2. Trees, Tree Automata, and MSO Logic

2.1. **Trees and Tree Automata.** For convenience, we present short definitions of (binary) trees and tree automata here. A proper introduction to tree automata can be found in Gécseg and Steinby (1997).

We actually give two definitions here, suitable for different aspects of the following discussions. Assume an alphabet $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{\lambda\}$

$$\mathfrak{A} = \langle \{a_0, a_1\}, \{A, B\}, a_0, \{a_0\}, \alpha \rangle$$

$$\begin{array}{ll} \alpha(a_0, a_0, A) = a_0 & \alpha(a_0, a_1, A) = a_1 \\ \alpha(a_1, a_0, A) = a_1 & \alpha(a_1, a_1, A) = a_1 \\ \alpha(a_0, a_0, B) = a_1 & \alpha(a_0, a_1, B) = a_1 \\ \alpha(a_1, a_0, B) = a_1 & \alpha(a_1, a_1, B) = a_1 \end{array}$$

FIGURE 1. A simple tree automaton

and i the arity of the elements in Σ_i. The set of binary trees T_Σ over Σ is the set of terms built by letting λ denote the empty tree and by letting $\sigma(t_1, t_2)$, $\sigma \in \Sigma_2$, $t_1, t_2 \in T_\Sigma$ denote the tree with label σ and subtrees t_1, t_2. This algebraic perspective facilitates the automata theory. In the model theory underlying MSO logics it is more convenient to assume a tree t to be a function from the addresses of a binary tree domain $\mathcal{T}_2$ to labels Σ. Intuitively a tree domain is a way to canonically label the nodes in an infinite tree. More formally, a tree domain $\mathcal{T}$ is a subset of strings over a linearly ordered set which is closed under prefix and left sister. $\mathcal{T}_2$ results from an alphabet of strings over, for example, $\{\underline{0}, \underline{1}\}$.

A deterministic (bottom-up) tree automaton $\mathfrak{A}$ on binary trees is a tuple $\langle A, \Sigma, a_0, F, \alpha \rangle$ with A the (finite) set of states, Σ as defined above, $a_0 \in A$ the initial state, $F \subseteq A$ the final states and $\alpha : (A \times A \times \Sigma_2) \to A$ the transition function. We omit the transitions on Σ_0 here and in the following. Instead, we just designate the initial state. Formally we would need a transition $\alpha(\lambda) = a_0$. We can extend the transition function homomorphically to trees by inductively defining $h_\alpha(\lambda) = a_0$ and $h_\alpha(\sigma(t_1, t_2)) = \alpha(h_\alpha(t_1), h_\alpha(t_2), \sigma)$, $t_1, t_2 \in T_\Sigma, \sigma \in \Sigma_2$. An automaton $\mathfrak{A}$ accepts a tree $t \in T_\Sigma$ iff $h_\alpha(t) \in F$. The language recognized by $\mathfrak{A}$ is denoted by $T(\mathfrak{A}) = \{t \mid h_\alpha(t) \in F\}$. Emptiness of the language $T(\mathfrak{A})$ is decidable by a fixpoint construction on the reachable states. Intuitively, the automaton creeps up the tree from the frontier to the root using the labels as symbols for the transitions and assigning a state to each subtree. Recognition of a given tree t is just trivially achieved by running an automaton on an input tree, i.e., computing $h_\alpha(t)$.

As an example, the automaton in Fig. 1 recognizes trees from an alphabet with $\Sigma_2 = \{A\}$. This admittedly very simple tree automaton recognizes all binary trees whose interior nodes are labeled with A by staying in the initial and, at the same time, final state a_0. As soon as we encounter a node labeled B, we go into a sink state (a_1). We will eliminate the transitions to the sink state in the remainder of the paper since they do not contribute knowledge on acceptable structures.

Bottom-up tree automata are closed under complementation, conjunction, disjunction, projection and cylindrification of alphabets, determinization and minimization. Most of the constructions are adaptions from the corresponding ones on finite state automata. We will use the constructions during the compilation from formulas to automata to parallel connectives and quantifiers in MSO formulas. More details on both the specification and implementation of tree automata and their constructions can be found in Morawietz and Cornell (1997) and Klarlund (1998).

2.2. **MSO Logic on Trees.** We choose MSO logic on trees since it is decidable, flexible and has a direct operational interpretation via its decidability proof. Both theorem proving and inductive definitions are well defined. Furthermore, MSO logic has a descriptive complexity result: it characterizes exactly the trees whose yields are context-free languages.

An MSO logical language is like predicate logic extended with variables ranging over sets and quantifiers ranging over these MSO variables. More specifically, it has a syntax of both first and (monadic) second-order quantification, all the usual logical connectives, first-order variables ranging over nodes and a (countably infinite) set of monadic second-order variables ranging over finite sets. Terms and formulas are built as usual.

The techniques we are using come originally from an old result in logic, namely that the weak MSO theory of two successor functions (WS2S) is decidable (Thatcher and Wright, 1968; Doner, 1970). A "weak" second-order theory is one in which the set variables are allowed to range only over finite sets. There is a more powerful result available: it has been shown by Rabin (1969) that the strong second-order theory (variables range over infinite sets) of even countably many successor functions is decidable. However, in our linguistic applications we need only to quantify over finite sets, so the weaker theory is enough, and the techniques correspondingly simpler. In fact, since we are interested in using the technique of the decidability proof for natural language processing and the proof works by showing a correspondence between formulas in the language of WS2S and tree automata and there does not exist an efficient minimization algorithm for the corresponding class of Rabin automata on infinite sets, using strong SnS is not an option open to us. All of these are generalizations to trees of a result on strings originally due to Büchi (1960). The applications we mention here could be adapted to strings with finite-state automata replacing tree automata.

Informally, we create a tree description logic by fixing the domain of the interpretation to "trees" and adding binary relations to the syntax which will be interpreted as the successor functions. So, for the structure of WS2S, we are going to assume a tree domain with the extension of (at least) the two successor relations. These correspond intuitively to the relations of left and right daughter and are used to navigate through the tree. And the structure can be extended with interpretations of other definable relations we may want to use. We will call this basic structure of WS2S $\mathcal{N}_2$.

DEFINITION 2.1. *The **structure of WS2S** ($\mathcal{N}_2$) is a tuple $\langle \mathcal{T}_2, \varepsilon, s_0, s_1 \rangle$ such that $\mathcal{T}_2$ is a binary tree domain with root ε and s_0, s_1 the left and right successor relations respectively.*

We will overload the term WS2S to mean the structure of two successor functions as well as its MSO language.

Intuitively, MSO predicates, i.e., monadic second-order variables, pick out sets of nodes. We can think of the predicates as features labeling the nodes. A tree, then, is just a rooted, dominance connected subset T of the domain of $\mathcal{N}_2$. A labeled tree is a $k+1$-tuple $\langle T, F_1, \ldots, F_k \rangle$ of the tree T and k features. Therefore, MSO formulas with the underlying interpretation on $\mathcal{N}_2$ are constraints on trees. And a grammar in this setting becomes just the specification of a $k+1$-ary relation picking out the well-formed trees.

Formally, each MSO formula represents a constraint on the valuation of its free variables which is determined by the assignment of the variables to (sets of) nodes.

DEFINITION 2.2. *Let $\mathcal{T}$ be a tree domain and VAR a set of (MSO) variables. A **variable assignment** is a total function $\alpha : \mathsf{VAR} \to \wp(\mathcal{T})$. We call the set of all those mappings ASS.*

Obviously, satisfaction is relative to these assignments. We will write satisfaction as $\mathcal{N}_2 \models \phi[\alpha]$ for ϕ an MSO formula, α a variable assignment. Since these assignments are such that they map variables to nodes in a tree, i.e., the assignments together with the domain of interpretation form a (labeled) tree, we will also speak of assignment trees.

Since the proofs of the decidability results are inductive on the structure of MSO formulas, we can choose our particular tree description language rather freely, knowing (a) that the resulting logic will be decidable and (b) that the translation to automata will go through as long as the atomic formulas of the language represent relations which can be translated (by hand if necessary) to tree automata recognizing the "right" assignments to their free variables. We will see how this is done in the next section. Note that it requires further proof that these languages have the full power of WS2S, though.

Therefore, the use of the decidability result is not fixed to a particular area of natural language formalisms. For example, Ayari et al. (1998) have investigated the usefulness of these techniques in dealing with record-like feature trees which unfold feature structures; there the attributes of an attribute-value term are translated to distinct successor functions. On the other hand, Rogers (To appear) has developed a language rich in long-distance relations (dominance and precedence) which is more appropriate for work in GB theory. Compact automata can be easily constructed to represent dominance and precedence relations. One can imagine other possibilities as well: as we will see in Sec. 2.3, the automaton for Kayne-style asymmetric, precedence-restricted c-command (Kayne, 1994) is also very compact, and makes a suitable primitive for a description language along the lines developed by Frank and Vijay-Shanker (1998).

2.2.1. *An Example Language:* $\mathcal{L}^2_{K,P}$. In this paper, we draw our examples from tree description logics used in the P&P paradigm. In particular $\mathcal{L}^2_{K,P}$, the logic proposed in Rogers (To appear), will serve as our main source. Note that $\mathcal{L}^2_{K,P}$ has been demonstrated to offer concise and well founded formalizations of concepts involved in P&P approaches. In fact, Rogers encodes in his monograph most of the proposals made in *Relativized Minimality* by Rizzi (1990). Although Rogers has shown that $\mathcal{L}^2_{K,P}$ is intertranslatable with SnS and therefore not limited to either binary or finite trees, we use it only in the weak sense over finite binary trees.

The language of $\mathcal{L}^2_{K,P}$ is designed to express relationships between nodes in trees representing linguistic structures. There are local relations on nodes such as the immediate dominance relation as well as nonlocal ones such as the reflexive closure of immediate dominance, simply called dominance. Various other theory independent relations for reasoning about relations between nodes can be defined in WS2S and added, e.g., proper precedence to express ordering information. We parameterize the language with both individual and predicate constants.

EXAMPLE 2.3. *Let* $\mathcal{L}^2_{K,P}$ *be defined by a set* K *of countably many individual constants; a set* P *of countably many predicate constants; a countably infinite set* $\mathbf{X} = \mathbf{X}_0 \cup \mathbf{X}_1$ *of first-order and monadic second-order* *variables;* $\wedge, \vee, \neg$ *– logical connectives;* $\forall, \exists$ *– quantifiers;* $(,), [,]$ *– brackets;* $\triangleleft, \triangleleft^*, \triangleleft^+$ *– immediate, reflexive, and proper dominance;* $\prec$ *– proper precedence; and* $\in, \approx$ *– set membership and equality.*

In the remainder of the paper we will mostly use $\mathcal{L}^2_{\emptyset,\emptyset}$ and refer to it as $\mathcal{L}^2$. The change does not limit the power of the language since both the predicate and individual constants correspond to free (global) variables. In fact, if one uses a tool to compile the automata from the formulas, one must use free variables since the compilation process cannot handle constants. We deviate slightly from Rogers' definition of the language in that we include $\in$ explicitly and use $X(x)$ and $x \in X$ interchangeably.

We will freely use appropriate symbols for standard combinations of the boolean operators (for example $\Rightarrow$, $\not\approx$, $\Leftrightarrow$, …), uppercase letters to denote set variables and lowercase ones to denote individual ones. As an example for relations definable in this language, take the subset relation in (1), the constraint for a set to contain exactly one member in (2) and, more linguistically motivated, the formula denoting the relation of directed asymmetric c-command[1] in the sense of Kayne (1994), see (3).

(1) $$X \subseteq Y \overset{def}{\Longleftrightarrow} (\forall x)[x \in X \Rightarrow x \in Y]$$

(2) $$\begin{aligned}\mathsf{Sing}(X) \overset{def}{\Longleftrightarrow}\ & (\forall Y, Z)[Y \subseteq X \Rightarrow (X \subseteq Y \vee Y \subseteq Z)] \wedge \\ & (\exists Y)[X \not\subseteq Y]\end{aligned}$$

(3) $$\begin{aligned}\mathsf{AC\text{-}Com}(x,y) \overset{def}{\Longleftrightarrow}\ & \quad x \prec y \\ & \wedge (\forall z)[z \triangleleft^+ x \Rightarrow z \triangleleft^+ y] \wedge \neg(x \triangleleft^* y) \\ & \wedge \neg((\forall z)[z \triangleleft^+ y \Rightarrow z \triangleleft^+ x] \wedge \neg(y \triangleleft^* x))\end{aligned}$$

The formula in (1) is simple enough to be self-explanatory, whereas the one in (2) demonstrates how to use second-order quantification. Therefore we will explain it briefly. A set is a singleton iff all its subsets are either equal to it or the empty set and it itself is nonempty. The definition in (3) specifies that two nodes asymmetrically c-command each other from "left-to-right" iff x c-commands y, y does not c-command x and x precedes y. We depict an example tree in Fig. 2. The dashed arrow represents the c-command relation. Note that x also c-commands a and b, but does not asymmetrically c-command a.

[1]This relation is not monadic, but reducible via syntactic substitution to an MSO signature. We can define and use these explicitly definable relations freely, but we

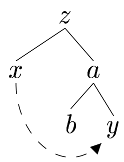

FIGURE 2. An example for asymmetric directed c-command

A structure for $\mathcal{L}^2_{K,P}$ consists of an appropriate set of nodes – a tree domain – with relations interpreting the elements from the signature. We limit the structure to dominance, parent and precedence since proper dominance can be defined in terms of them. This structure is a more specific alternative for the one given in Def. 2.1.

EXAMPLE 2.4. *A model for $\mathcal{L}^2_{K,P}$ is a tuple $\langle \mathcal{T}, \mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{L}, \mathcal{R}_p \rangle_{p \in P}$ with:*

$\mathcal{T}$	*a nonempty tree domain*
$\mathcal{I}$	*a function from K to $\mathcal{T}$*
$\mathcal{P}$, $\mathcal{D}$, $\mathcal{L}$	*binary relations on $\mathcal{T}$ which interpret $\triangleleft$, $\triangleleft^*$ and $\prec$*
$\mathcal{R}_p$	*are relations of appropriate arity on $\mathcal{T}$ interpreting $p \in P$.*

A model for $\mathcal{L}^2$ is then just a tuple $\langle \mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{L} \rangle$.

We will use this type of models whenever we speak of $\mathcal{L}^2$ or $\mathcal{L}^2_{K,P}$ and the more general one of $\mathcal{N}_2$ otherwise.

2.3. **Representing MSO Formulas with Tree Automata.** In this section we take a closer look at the connection between MSO formulas and tree automata. It should 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 free variables in the input formula with a stack of exponents as tall as the number of the formula's quantifier alternations. However, there is a growing body of work in the computer science literature on computer hardware and system verification (e.g., Kelb et al. (1997), Klarlund (1998), Basin and Klarlund (To appear)) suggesting that in practical cases the extreme explosiveness of this technique can be effectively controlled, and our own experiments suggest that this should be the case in most linguistic applications as well. Experiences with a software tool are documented in Morawietz and Cornell (To appear).

---

cannot quantify over them. A relation r is explicitly MSO-definable iff there exists an MSO formula ϕ such that $\mathsf{r}(X_1, \dots, X_n) \iff \phi(X_1, \dots, X_n)$.

The attraction of the proposed approach is obvious: the connection allows the linguist the specification of ideas about natural language in a concise manner in logic and at the same time provides a representation of the constraints which can be efficiently used for processing.

The decidability proof works by showing a correspondence between formulas in the language of WS2S and tree automata, developed in such a way that the formula is satisfiable iff the set of trees accepted by the corresponding automaton is nonempty.

More specifically, Doner (1970) and Thatcher and Wright (1968) show that each formula in MSO logic can be represented by a tree automaton which recognizes the relevant assignment trees. One step in the proof is to encode the variable assignments as the inverse of the labeling function, another the inductive construction of automata. Then WS2S is decidable since the emptiness problem for tree automata is decidable.

Remember that assignment functions map first order variables to nodes and second-order variables to sets of nodes in $\mathcal{N}_2$. The labeling functions on the trees the automaton accepts do the converse: they map nodes to the set of free variables to which that node is assigned. Concretely, we can think of nodes as being labeled with bit-strings such that bit i is on at node n if and only if n is to be assigned to variable X_i.

More formally, let again $\mathcal{N}_2$ be the structure of WS2S and $\phi(\vec{X})$ a formula with free variables $\vec{X}$. We can safely ignore individual variables since they can be encoded as properly constrained set variables with the formula for Sing given in (2). For each relation $r(\vec{X})$ from the signature of the MSO logic, we have to define a tree automaton $\mathfrak{A}^r$ recognizing the proper assignment trees. Then we can inductively build a tree automaton $\mathfrak{A}^\phi$ for each MSO formula $\phi(\vec{X})$, such that

$$\mathcal{N}_2 \models \phi[\tau] \Longleftrightarrow t \in T(\mathfrak{A}^\phi) \tag{4}$$

where τ assigns sets of nodes to the variables $\vec{X}$ and t is the corresponding assignment tree. The automaton compactly represents the (potentially infinite) number of valuations, i.e., the solutions to constraints.

Consider as an example the tree automaton corresponding to Kayne's asymmetric c-command relation from (3), see Fig. 3. For readability, we denote the alphabet of node labels as tuples indicating for each free variable whether a node is assigned to it or not. In this case we have only free variables x and y, so the alphabet consists of the tuples $\langle x, y\rangle$, $\langle \neg x, y\rangle$, $\langle x, \neg y\rangle$ and $\langle \neg x, \neg y\rangle$.

On closer examination of the transitions, we note that we just percolate the initial state as long as we find nodes which are neither x nor y. From

$$\begin{aligned}
\mathfrak{A} &= \langle A, \Sigma, a_0, F, \alpha \rangle, \\
A &= \{a_0, a_1, a_2, a_3, a_4, a_5\}, \\
\Sigma &= \{\langle x, y\rangle, \langle x, \neg y\rangle, \langle \neg x, y\rangle, \langle \neg x, \neg y\rangle\} \\
F &= \{a_4\},
\end{aligned}$$

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

FIGURE 3. The automaton for $\mathsf{AC\text{-}Com}(x, y)$

the initial state on both the left and the right subtree we can either go to the state denoting "found x" (a_3) if we read symbol $\langle x, \neg y\rangle$ or to the state denoting "found y" (a_1) if we read symbol $\langle \neg x, y\rangle$. After finding a dominating node while being in a_1 – which switches the state to a_2 – we can percolate a_2 as long as the other branch does not immediately dominate x. If we come into the situation that 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. But in addition to the arbitrary number of intermediate nodes, we can also have an arbitrary number of $\langle \neg x, \neg y\rangle$ labeled nodes dominating the constellation or being dominated by it such that we recognize the closure of the (AC-Com) relation in the sense of Doner. The relation can now serve as a new primitive in further MSO formulas.

To keep the automata as small as possible, we employ the suitably adapted minimization of finite state automata. The minimization procedure also gives us an alternative way to decide emptiness since the canonical empty automaton has only one state which is initial but not final.

The non-elementary complexity is caused by alternation between quantifier blocks. Quantification results in nondeterministic automata, but negation needs deterministic ones as input. So for each universal quantifier followed by an existential one, we need the subset construction which has an exponential worst case complexity: the number of states can grow according to a function with a stack of exponents whose height is determined by the number of $\forall$–$\exists$ quantifier alternations.

2.4. **Automata-based Theorem Proving.** Automated theorem proving is the standard approach to processing with constraint-based grammars. The decidability proof gives us a procedure to do theorem proving since the compilation produces an automaton recognizing the empty language iff the formula is unsatisfiable. Practically this means that a formula is satisfiable if the resulting automaton has a non-empty set of final states and valid if the resulting automaton has exactly one state which is initial and final such that all assignments are recognized. So, theorem proving and compilation are in fact one and the same process.

An obvious goal for the use of the discussed approach is the (offline) generation of a tree automaton representing an entire grammar. Then parsing would be just proving that the grammar implies the input. Unfortunately, as mentioned above, the problem to generate the tree automata from arbitrary MSO formulas is highly exponential because of the necessary determinization. And even worse: the alphabet space is an additional source for combinatorial explosion. Features are free variables and therefore alphabet symbols for the automata. Since there is no a priori limit on any combination not occurring, we have to represent all their permutations. In extreme cases, this dominates even the state space. As an example, consider a naïve specification of the lexicon:

$$
\begin{aligned}
\mathsf{Lexicon}(x) \overset{def}{\iff} \quad & (Sees(x) \wedge V(x) \wedge 3rd(x) \wedge \dots) \\
\vee\, & (Jan(x) \wedge N(x) \wedge Masc(x) \wedge \dots) \\
\vee\, & (Lena(x) \wedge N(x) \wedge Fem(x) \wedge \dots) \\
& \vdots
\end{aligned}
\tag{5}
$$

We have defined a relation called Lexicon via a disjunctive specification of lexical labels, realized as for example the second-order variable $Sees$, and the appropriate combination of features, e.g., V. The way we presented it, Lexicon looks like a monadic predicate with just one argument. But considering the automaton to be generated from the formula, one has to know that the alphabet of the automata has to contain every free variable to encode the right assignments. Therefore all the free variables contained in the formula are implicitly present: Lexicon is a relation. Remember that since constants are free variables to the compiler, using them does not change the argument. And since every feature and every lexical entry is represented with a new free set variable, the alphabet size will be 2^{n+m}, n the number of lexical entries and m the number of features. Nevertheless we will continue to leave these "global" variables implicit.

For tests, we used the tool MONA (Klarlund and Møller, 1998). The major improvement of MONA against our own prototype (Morawietz and Cornell, 1997) is the use of binary decision diagrams (BDDs) for the representation of large alphabets. The graph structure of BDDs allows compression of the transitions by removing redundant tests and nodes. But the amount of compression which can be achieved is very much dependent on ordering the free variables and finding an optimal ordering is NP-hard. An introduction to BDDs can, for example, be found in Andersen (1996). It is still an open problem whether the generation of an entire grammar can be handled. The question is not efficiency of the compilation, but rather feasibility since the size of an automaton representing a grammar is still indeterminate. And it is unlikely that even the use of BDDs helps sufficiently with the representation of a realistic lexicon.

However, the tests have shown that we can compile modules and by this gain insight into parts of the formalization (e.g., see Morawietz and Cornell (To appear) for a report on the compilation of Rogers' entire X-Bar theory). As an example consider the c-command relation between categories instead of nodes following the presentation of Rizzi's ideas from *Relativized Minimality* in Rogers' monograph. This is a nontrivial part of the theory, but turns out to have a compact automaton.

Informally, a category is a node which has been split into segments through adjunction. Categories are used in P&P approaches since for example in a WH-question the auxiliary is supposed to c-command its trace. Following Rizzi, this is explained by the fact that the node is adjoined to its head and therefore not dominated by it. In Fig. 4 one can see that I_j is indeed not dominated by the *category* formed by the two nodes labeled C. I_j c-commands its trace t_j since all *categories* which dominate it, also dominate t_j.

The definition of categories can be found in (6) and (7). In the formulas, we assume some additional predicates: Feq ensures agreement on some set of relevant features; Path can be understood as a dominance-connected ordered subgraph, and *Adj* is the feature marking an adjoined node.

(6)
$$\begin{aligned}\mathsf{Component}(X) \stackrel{def}{\Longleftrightarrow}\ & \mathsf{Path}(X) \wedge \\ & (\forall x, y)[(x \in X \wedge y \in X) \Rightarrow \mathsf{Feq}(x, y)] \wedge \\ & (\forall x, x')(\exists y)(\forall z)[(x \in X \wedge x' \in X \wedge x \lhd x') \Rightarrow \\ & \quad (x' \notin Adj \wedge x \lhd y \wedge y \not\approx x' \wedge y \in Adj \wedge \\ & \quad (x \lhd z \Rightarrow (z \approx x' \vee z \approx y)))]\end{aligned}$$

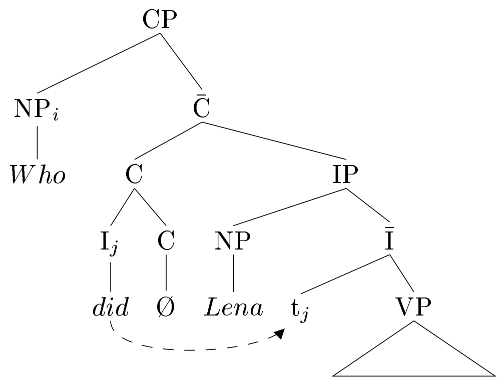

FIGURE 4. A syntax tree from Rogers (To appear)

So, a component is a set of nodes which (a) form a path in the tree; (b) agree on a set of particular features; and (c) have exactly one node in every local tree which is the result of an adjunction.

A category is then just a maximal component.

$$
\begin{aligned}
\mathsf{Category}(X) \overset{def}{\iff} & \ \mathsf{Component}(X) \land \\
& (\forall Y)[(X \subseteq Y \land Y \not\subseteq X) \Rightarrow \neg\mathsf{Component}(Y)]
\end{aligned} \tag{7}
$$

Note that a number of additional constraints on the distribution of adjuncts is necessary to achieve the "right" behaviour.

Building on the given definition, we can define primitives of domination, precedence and c-command in terms of categories rather than nodes. We can relativize all of them to nodes, e.g. $\mathsf{Category}(x, y)$ means that there exists a category such that x and y are both members of it, thereby avoiding the third-order notions.

As might be suspected considering (6) and (7), the complete presentation of the formulas encoding category based c-command is fairly complex – although c-command itself is just a lifted version of what we saw in (3) using the new category-based primitives for domination instead of the ones from the signature – and therefore beyond the scope of this paper. The reader is encouraged to check the full formalization in Rogers' monograph. But actually, after existentially binding the global variables, the resulting automaton is small. It has only 5 states and 16 transitions, see Fig. 5. Since c-command on categories is a complicated relation, we cannot give a detailed explanation of the automaton.

$$\mathfrak{A} = \langle A, \Sigma, a_0, F, \alpha \rangle$$
$$A = \{a_0, a_1, a_2, a_3, a_4\}$$
$$\Sigma = \{\langle x, y\rangle, \langle x, \neg y\rangle, \langle \neg x, y\rangle, \langle \neg x, \neg y\rangle\}$$
$$F = \{a_3\}$$

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

FIGURE 5. The automaton for category based c-command

One can see from the discussions in this section that the approach of pure theorem proving, i.e., using only the automata as the operational interpretation of our formalism, is computationally not without problems. But we can define new primitives and work with them since we can represent MSO constraints, i.e., knowledge about variable bindings on trees, efficiently with tree automata. In the following sections we are going to use constraint logic programming to reintroduce a notion of derivation into the purely model-theoretic presentation. By this we provide a way to use the previously compiled parts or primitives of formalizations separately under the control of programs and gain the desired extension of the generative capacity.

3. Constraint Logic Programming

Logic programming (LP) reflects nicely the separation of the actual logic of a program and the control of its execution (Lloyd, 1984). Programs in the LP paradigm are sets of logical clauses. These programs are interpreted within Herbrand models, i.e., sets of ground atoms. One could be content with this declarative interpretation. But on the other hand one wants to compute with them. The added operational interpretation used in most LP languages is based on SLD-resolution (Linear resolution with a Selection function for Definite clauses). Then the semantics is reformulated in procedural terms of least models and fixpoints such that the languages have provably equivalent denotational and operational semantics. Two problems with this approach – the generate and test paradigm to find the solutions and the use of uninterpreted structures – can be overcome by the addition of constraints to the standard logic programming scheme.

Intuitively, with the constraint logic programming scheme we develop a way to give definite clause specifications over MSO formulas. We will use this extended language to constrain the variables by incrementally adding suitable constraints to a constraint store. The store can be viewed as a tree automaton representing the possible valuations. In fact, we control the generation of the tree automaton we are interested in. By this we can focus on just those parts of the grammar which are relevant to the actual computation instead of having to consider – and precompile – all possibilities inherent in the grammar at once. A standard interpreter (left to right, depth first search) is sufficient for the execution of one of our programs. The answer of a search branch of such a program will be a satisfiable MSO constraint represented by a tree automaton. Backtracking then allows the search for more answers. Furthermore, the embedding allows us to apply a wealth of techniques developed in general for (constraint) logic programming and the corresponding interpreters to optimize the behaviour of the programs.

In logic programming enhanced with constraints over a specific domain, we need a specialized mechanism to solve the constraints, i.e., decide their satisfiability. Although this mechanism is treated as a black box in the actual definitions of the constraint language, it is an important aspect in the effectiveness to have a suitable constraint solver. In Sec. 3.2.5 we will discuss some desiderata for an efficient constraint solver for our purposes.

The CLP language serves both the facilitation of the use of MSO formalizations in computational linguistics and the extension of the generative capacity of MSO-based grammar formalisms. Our main motivation for the necessity of the extension are the arguments on non-context-freeness of natural languages. For example, Swiss German (Shieber, 1985) or Dutch (Huybregts, 1984) have overt case marking on objects and allow cross-serial dependencies in subordinate clauses. Cross-serial dependencies cannot be formalized in MSO-based approaches since their underlying structure is comparable to the copy language – $\mathcal{L}_{copy} = \{ww \mid w \in \Sigma^*\}$ – which is not context-free.

3.1. **The Constraint Language.** In this section we apply the general scheme developed in Höhfeld and Smolka (1988) for the definition of a constraint language to MSO logic, fixing the domain of computation to trees. We use this scheme as opposed to the more standard one of Jaffar and Maher (1994) since it abstracts from the actual details of the syntax of the constraint language and obviates some technical requirements on the language, most notably *solution compactness*. Remember that formally we do not need individual variables.

Following Höhfeld and Smolka, we can turn any MSO logic on trees or strings into a constraint language. $\mathcal{MSO}$ will be used as a place holder for any such constraint language. We already presented the definition of an interpretation for WS2S in Def. 2.1. We generalize it by the inclusion of a solution mapping $[\![-]\!]^{\mathcal{I}}$ which in essence amounts to our compilation procedure.

DEFINITION 3.1. *A **constraint language** $\mathcal{MSO} = \langle \mathsf{VAR}, \mathsf{CON}, \mathsf{V}, \mathsf{INT} \rangle$ is a fourtuple such that*

1. VAR *is a countable (possibly) infinite set of (MSO) variables,*
2. CON *is a (decidable) set of well formed formulas (of MSO logic),*
3. V *is a computable function which assigns to every $\phi \in \mathsf{CON}$ a finite set $\mathsf{V}(\phi)$ of variables, called the variables constrained by ϕ or the free variables of ϕ,*
4. INT *is a nonempty set of $\mathcal{MSO}$-interpretations $\mathcal{I}$ such that each interpretation is a pair $(\mathcal{N}_2, [\![-]\!]^{\mathcal{I}})$ consisting of $\mathcal{N}_2$ and solution or denotation mappings $[\![-]\!]^{\mathcal{I}}$ such that*
   (a) *$[\![-]\!]^{\mathcal{I}}$ is a function mapping every constraint ϕ to a set $[\![\phi]\!]^{\mathcal{I}}$ of assignments, $[\![-]\!]^{\mathcal{I}} : \mathsf{CON} \longrightarrow \wp(ASS)$. We call that set the solutions of ϕ (in $\mathcal{I}$).*
   (b) *$\phi \in \mathsf{CON}$ constrains only $\mathsf{V}(\phi)$, i.e., if $\alpha \in [\![\phi]\!]^{\mathcal{I}}$ and β is an assignment witch agrees with α on $\mathsf{V}(\phi)$, then $\beta \in [\![\phi]\!]^{\mathcal{I}}$.*

Satisfiability and validity are defined as usual. Most conditions are trivially satisfied for MSO logics, but two details are left to be specified. Firstly we present the function which computes the free variables.

DEFINITION 3.2. *Let $\mathcal{MSO} = \langle \mathsf{VAR}, \mathsf{CON}, \mathsf{V}, \mathsf{INT} \rangle$ be a constraint language, then the function $\mathsf{V} : \mathsf{CON} \to \wp(\mathsf{VAR})$ is defined as follows (R ranges over the defined relations from the signature, $*$ over binary connectives, $\mathcal{Q}$ over quantifiers):*

$$\mathsf{V}(\mathsf{R}(X_1, \ldots, X_n)) = \{X_1, \ldots, X_n\} \qquad \mathsf{V}(\neg\phi) = \mathsf{V}(\phi)$$
$$\mathsf{V}(\phi * \psi) = \mathsf{V}(\phi) \cup \mathsf{V}(\psi) \qquad \mathsf{V}(\mathcal{Q}X[\phi]) = \mathsf{V}(\phi) - \{X\}$$

And secondly, the definition of the denotation function $[\![-]\!]^{\mathcal{I}}$. This depends on the actual relations in the signature. We use $\mathcal{L}^2$ as our example to present it.

EXAMPLE 3.3. *Let $\mathcal{M} = \langle \mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{L} \rangle$ be a model for $\mathcal{L}^2$ and let $\alpha[X/A]$ be the assignment function just like α except that the variable X gets assigned the value $A \subseteq \mathcal{T}$. Then $\mathcal{L}^2$ is a constraint language $\mathcal{MSO}$ with* VAR *being* $\boldsymbol{X}$*,* CON *being the formulas of $\mathcal{L}^2$,* V *as defined above and with* INT *consisting of interpretations $\mathcal{I} = (\mathcal{M}, [\![-]\!]^{\mathcal{I}})$ such that the denotation function is defined as follows:*

$$
\begin{array}{rcl}
[\![x \lhd y]\!]^{\mathcal{I}} & = & \{\alpha \mid \langle \alpha(x), \alpha(y) \rangle \in \mathcal{P}\} \\
[\![x \lhd^* y]\!]^{\mathcal{I}} & = & \{\alpha \mid \langle \alpha(x), \alpha(y) \rangle \in \mathcal{D}\} \\
[\![x \prec y]\!]^{\mathcal{I}} & = & \{\alpha \mid \langle \alpha(x), \alpha(y) \rangle \in \mathcal{L}\} \\
[\![x \approx y]\!]^{\mathcal{I}} & = & \{\alpha \mid \alpha(x) = \alpha(y)\} \\
[\![x \in X]\!]^{\mathcal{I}} & = & \{\alpha \mid \alpha(x) \in \alpha(X)\} \\
[\![\phi \wedge \psi]\!]^{\mathcal{I}} & = & [\![\phi]\!]^{\mathcal{I}} \cap [\![\psi]\!]^{\mathcal{I}} \\
[\![\phi \vee \psi]\!]^{\mathcal{I}} & = & [\![\phi]\!]^{\mathcal{I}} \cup [\![\psi]\!]^{\mathcal{I}} \\
[\![\neg \phi]\!]^{\mathcal{I}} & = & ASS - [\![\phi]\!]^{\mathcal{I}} \\
[\![\exists X[\phi]]\!]^{\mathcal{I}} & = & \{\alpha \mid \alpha[X/A] \in [\![\phi]\!]^{\mathcal{I}} \textit{ for some } A \subseteq \mathcal{T}\} \\
[\![\forall X[\phi]]\!]^{\mathcal{I}} & = & \{\alpha \mid \alpha[X/A] \in [\![\phi]\!]^{\mathcal{I}} \textit{ for all } A \subseteq \mathcal{T}\}
\end{array}
$$

3.1.1. *Properties of $\mathcal{MSO}$.* In this section we recall some of the properties of $\mathcal{MSO}$. In part these are properties directly derived from MSO logic. The other part concerns properties which are relevant in the proofs concerning the further development of the constraint language and its relational extension.

Renamings and variants are relevant in the proof of the soundness and completeness of the operational semantics in Sec. 3.2.3. The following definition of a renaming is based on the denotational properties of the formulas involved. It is not a purely syntactic substitution. For variants we demand that two formulas agree on their free variables and denote the same assignment functions modulo a renaming. A renaming is the bijection $\rho : \mathsf{VAR} \to \mathsf{VAR}$ such that it is the identity except for finitely many variables. If ρ is a renaming, a constraint ϕ' is called a ρ-variant of ϕ if (a) ϕ' is a syntactic variant of ϕ, i.e., ϕ' results from simultaneously replacing every occurrence of a variable X in ϕ with $\rho(X)$ and (b) if $\mathsf{V}(\phi') = \rho(\mathsf{V}(\phi))$ and $[\![\phi]\!]^{\mathcal{I}} = [\![\phi']\!]^{\mathcal{I}}\rho$ with $[\![\phi']\!]^{\mathcal{I}}\rho := \{\alpha\rho^{-1} \mid \alpha \in [\![\phi]\!]^{\mathcal{I}}\}$ for every interpretation $\mathcal{I}$. A constraint ϕ' is called a variant of ϕ if there exists a renaming ρ such that it is a ρ-variant under that renaming. Note that by this definition renamings preserve satisfiability and validity.

DEFINITION 3.4. *A constraint language $\mathcal{MSO} = \langle \mathsf{VAR}, \mathsf{CON}, \mathsf{V}, \mathsf{INT} \rangle$ is*

1. ***closed under renaming*** *if every constraint $\phi \in \mathsf{CON}$ has a ρ-variant $\psi \in \mathsf{CON}$ for every renaming ρ,*
2. ***closed under intersection*** *if for every two constraints ϕ and ψ in CON there exists a constraint $\theta \in \mathsf{CON}$ such that for every interpretation $\mathcal{I} \in \mathsf{INT}$: $[\![\phi]\!]^{\mathcal{I}} \cap [\![\psi]\!]^{\mathcal{I}} = [\![\theta]\!]^{\mathcal{I}}$,*
3. ***decidable*** *if the satisfiability of its constraints is decidable.*

LEMMA 3.5. *$\mathcal{L}^2$ is a decidable constraint language closed under renaming and intersection.*

3.2. **Relational Extension.** We add distinguished relations to the constraint language $\mathcal{MSO}$ to derive a relational extension. Recall that we take the step of adding special relations to a (new) language for two reasons. Firstly we can now give an operational semantics to inductive definitions in $\mathcal{R}(\mathcal{MSO})$ and by this explicitly separate the original signature and these defined relations. And secondly we extend both generative capacity and computational power. It is an important question which of the properties of $\mathcal{MSO}$ we can preserve and which we loose.

DEFINITION 3.6. *A constraint language $\mathcal{MSO} = \langle \mathsf{VAR}, \mathsf{CON}, \mathsf{V}, \mathsf{INT} \rangle$ is extended with a decidable set of relations $\mathcal{R}$ and a corresponding arity function $ar : \mathcal{R} \longrightarrow \mathbb{N}$ to a* ***relational constraint language*** *$\mathcal{R}(\mathcal{MSO}) = \langle \mathsf{VAR}, \mathcal{R}(\mathsf{CON}), \mathcal{V}, \mathcal{R}(\mathsf{INT}) \rangle$ as follows:*

1. *The set of $\mathcal{R}(\mathcal{MSO})$-**constraints** $\mathcal{R}(\mathsf{CON})$ is defined inductively such that*
   (a) *$\phi \in \mathcal{R}(\mathsf{CON})$ for all $\phi \in \mathsf{CON}$.*
   (b) *$r(X_1, \ldots, X_n) \in \mathcal{R}(\mathsf{CON})$ for all tuples of pairwise distinct variables $X_1, \ldots, X_n$ and relation symbols $r \in \mathcal{R}$ with $ar(r) = n$. We call these $\mathcal{R}(\mathcal{MSO})$-atoms.*
   (c) *if F and G are constraints in $\mathcal{R}(\mathsf{CON})$, then so are $F \,\&\, G$, $F \rightarrow G$ and the empty conjunction $\emptyset$.*
   (d) *Nothing else is in $\mathcal{R}(\mathsf{CON})$.*
2. *The function $\mathcal{V}$ computing the free variables of an $\mathcal{R}(\mathcal{MSO})$-constraint is again defined inductively:*
   (a) *$\mathcal{V}(\phi) := \mathsf{V}(\phi)$ for $\phi \in \mathsf{CON}$*
   (b) *$\mathcal{V}(r(X_1, \ldots, X_n)) := \{X_1, \ldots, X_n\}$ for $r \in \mathcal{R}$ with $ar(r) = n$*
   (c) *$\mathcal{V}(\emptyset) := \emptyset$*

(d) $\mathcal{V}(F \,\&\, G) := \mathcal{V}(F) \cup \mathcal{V}(G)$
(e) $\mathcal{V}(F \to G) := \mathcal{V}(F) \cup \mathcal{V}(G)$

3. *An $\mathcal{R}(\mathcal{MSO})$-**interpretation** $\mathcal{A}$ is the extension of an $\mathcal{MSO}$-interpretation $\mathcal{I}$ on the same domain to include the new relational symbols by choosing relations $r^{\mathcal{A}} \subseteq \wp(\mathcal{T}) \times \cdots \times \wp(\mathcal{T})$ for each $r \in \mathcal{R}$, and by extending the solution mapping as follows:*
   (a) $[\![\phi]\!]^{\mathcal{A}} := [\![\phi]\!]^{\mathcal{I}}$ *for* $\phi \in \mathsf{CON}$
   (b) $[\![r(X_1, \ldots, X_n)]\!]^{\mathcal{A}} := \{\alpha \in ASS \mid \alpha(X_1, \ldots, X_n) \in r^{\mathcal{A}}\}$
   (c) $[\![\emptyset]\!]^{\mathcal{A}} := ASS$
   (d) $[\![F \,\&\, G]\!]^{\mathcal{A}} := [\![F]\!]^{\mathcal{A}} \cap [\![G]\!]^{\mathcal{A}}$
   (e) $[\![F \to G]\!]^{\mathcal{A}} := (ASS - [\![F]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}}$
   $\mathcal{R}(\mathsf{INT})$ *is the set of all $\mathcal{R}(\mathcal{MSO})$-interpretations $\mathcal{A}$.*

3.2.1. *Definite Clauses.* To be able to use the relational extensions we presented, we have to formalize the notion of computing with the language. The simplest approach – as has been presented for logic programs in Lloyd's book – is to define definite clauses and definite clause specifications, also called programs.

In this section we follow again Höhfeld and Smolka in applying their general scheme to our special case. The key is that, as in standard logic programming, we can always find a minimal model for a given program. Firstly, we define what a program in the formal sense is.

DEFINITION 3.7. *Let $\mathcal{R}(\mathcal{MSO}) = \langle \mathsf{VAR}, \mathcal{R}(\mathsf{CON}), \mathcal{V}, \mathcal{R}(\mathsf{INT})\rangle$ be a relational constraint language. A **definite clause specification** $\mathcal{S}$ is a finite set of definite clauses of the form*

$$H \leftarrow \phi \,\&\, B_1 \,\&\, \ldots \,\&\, B_n$$

where H and the B_i are $\mathcal{R}(\mathcal{MSO})$-atoms, $n \geq 0$ and $\phi \in \mathsf{CON}$.

As usual, we will refer to the single atom as the head of the clause and to the conjunction of the atoms with the constraint as the body. The $\mathcal{MSO}$-constraint in the body will also be called a guard. A definite clause with just the $\mathcal{MSO}$-constraint and no atoms in the body is called a fact. A goal is a possibly empty conjunction of $\mathcal{MSO}$-constraints and $\mathcal{R}(\mathcal{MSO})$-atoms.

DEFINITION 3.8. *Given a definite clause specification $\mathcal{S}$, an $\mathcal{R}(\mathcal{MSO})$-interpretation $\mathcal{A}$ is a **model of** $\mathcal{S}$ if every variable assignment is a solution for every definite clause, i.e., $\forall \alpha \in ASS\ \forall \sigma \in \mathcal{S} : \alpha \in [\![\sigma]\!]^{\mathcal{A}}$.*

Höhfeld and Smolka prove that indeed there exists a least model $\mathcal{A}$ for every program which can be constructed from an interpretation $\mathcal{I}$ of the underlying constraint language as a limit of the application of the clauses in the following way.

The sets of interpretations $\mathcal{A}_i$ agree on their base interpretation $\mathcal{I}$. Therefore we can order them according to the order imposed by the relational symbols in $\mathcal{R}$, i.e., $\mathcal{A}_i \subseteq \mathcal{A}_j$ iff $r^{\mathcal{A}_i} \subseteq r^{\mathcal{A}_j}$ for all $r \in \mathcal{R}$. Note that the denotation function actually preserves this order, i.e., for G an $\mathcal{R}(\mathcal{MSO})$-atom, $[\![G]\!]^{\mathcal{A}_i} \subseteq [\![G]\!]^{\mathcal{A}_j}$ if $\mathcal{A}_i \subseteq \mathcal{A}_j$. Intersection and union of such interpretations are again defined via their relations, i.e., $\mathcal{A}_i \cap \mathcal{A}_j = \mathcal{A}_k$ iff $r^{\mathcal{A}_i} \cap r^{\mathcal{A}_j} = r^{\mathcal{A}_k}$ and $\mathcal{A}_i \cup \mathcal{A}_j = \mathcal{A}_k$ iff $r^{\mathcal{A}_i} \subseteq r^{\mathcal{A}_j} = r^{\mathcal{A}_k}$ for all $r \in \mathcal{R}$. Note that a set of interpretations extending $\mathcal{I}$ forms a complete lattice.

THEOREM 3.9 (Definiteness Theorem, H&S theo. 4.4). *Let $\mathcal{S}$ be a definite clause specification in $\mathcal{R}(\mathcal{MSO})$, $\mathcal{I}$ an $\mathcal{MSO}$-interpretation and B ranging over bodies of clauses from $\mathcal{S}$. Then we can inductively define a chain of $\mathcal{R}(\mathcal{MSO})$-interpretations $\mathcal{A}_i \subseteq \mathcal{A}_{i+1}$, $i \geq 0$, extending $\mathcal{I}$ by*

(i) $r^{\mathcal{A}_0} := \emptyset$
(ii) $r^{\mathcal{A}_{i+1}} := \{\alpha(\vec{X}) \mid (r(\vec{X}) \leftarrow B) \in \mathcal{S} \text{ and } \alpha \in [\![B]\!]^{\mathcal{A}_i}\}$
(iii) $r^{\mathcal{A}} := \bigcup_{i \geq 0} r^{\mathcal{A}_i}$

The least model $\mathcal{A}$ of $\mathcal{S}$ extending $\mathcal{I}$ is the union $\mathcal{A} := \bigcup_{i \geq 0} \mathcal{A}_i$.

The proof of this theorem is via the monotonicity of the definitions and the corresponding fixpoint theorem.

3.2.2. *An $\mathcal{R}(\mathcal{MSO})$ Example.* As an example for the relational language we defined, we present the definition of append on lists in $\mathcal{R}(\mathcal{MSO})$. Notational conventions for the definite clauses are that the associated $\mathcal{MSO}$-constraints are enclosed in curly brackets, lowercase letters denote node variables, uppercase letters or words starting with them set variables and relations in $\mathcal{MSO}$ are uppercase sans-serif and $\mathcal{R}(\mathcal{MSO})$ ones are lowercase sans-serif strings.

Firstly we need some explicit definitions of $\mathcal{MSO}$ constraints for the data type List on nodes. We do this by ordering a set of nodes by dominance. With the constraints Min and Max we can pick out the least and the greatest element in this set (the least one being the lowest one in the tree). The empty list is just an empty set, and a nonempty list has at least one element, is connected and all its elements stand in the reflexive dominance relation. The head of such a list is just the "highest" element, the tail all the remaining elements. It would have been possible equally well to assume a list to be a yield of a tree, i.e., a set of nodes ordered by precedence. We leave this definition to the reader. The exact definitions can be found in in (8) to (14).

$$\begin{aligned}
&\mathsf{append}(X, Y, Z) \leftarrow \\
&\qquad\qquad \{\mathsf{Elist}(X) \wedge Y \approx Z\} \\
&\mathsf{append}(X, Y, Z) \leftarrow \\
&\qquad\qquad \{\mathsf{Hd}(x, X) \wedge \mathsf{Hd}(x, Z) \wedge \mathsf{Tl}(X, U) \wedge \mathsf{Tl}(Z, V)\}\ \& \\
&\qquad\qquad \mathsf{append}(U, Y, V)
\end{aligned}$$

FIGURE 6. The definition of append in $\mathcal{R}(\mathcal{MSO})$

$$\mathsf{Min}(x, X) \overset{def}{\Longleftrightarrow} x \in X \wedge (\forall y \in X)[y \not\approx x \Rightarrow y \triangleleft^{+} x] \tag{8}$$

$$\mathsf{Max}(x, X) \overset{def}{\Longleftrightarrow} x \in X \wedge (\forall y \in X)[y \not\approx x \Rightarrow x \triangleleft^{+} y] \tag{9}$$

$$\mathsf{Elist}(X) \overset{def}{\Longleftrightarrow} \neg(\exists x)[x \in X] \tag{10}$$

$$\begin{aligned}\mathsf{Nelist}(X) \overset{def}{\Longleftrightarrow}\ & (\exists x)[x \in X] \wedge \mathsf{Connected}(X) \\ & \wedge (\forall x, y \in X)[x \triangleleft^{*} y \vee y \triangleleft^{*} x]\end{aligned} \tag{11}$$

$$\mathsf{List}(X) \overset{def}{\Longleftrightarrow} \mathsf{Elist}(X) \vee \mathsf{Nelist}(X) \tag{12}$$

$$\mathsf{Hd}(x, X) \overset{def}{\Longleftrightarrow} \mathsf{Nelist}(X) \wedge \mathsf{Max}(x, X) \tag{13}$$

$$\begin{aligned}\mathsf{Tl}(X, Y) \overset{def}{\Longleftrightarrow}\ & Y \subseteq X \wedge \mathsf{Nelist}(X) \wedge \mathsf{List}(Y) \\ & \wedge (\forall x \in X)[\neg\mathsf{Hd}(x, X) \Leftrightarrow x \in Y]\end{aligned} \tag{14}$$

The resulting $\mathcal{R}(\mathcal{MSO})$ definition for *append* is almost like the definition in standard logic programming. We just have to make what corresponds to unifications between variables explicit in the constraints, see Fig. 6.

3.2.3. *Operational Semantics.* The standard approach for an operational semantics of logic programs is to move stepwise from goals to answers. In this section we adapt the Höhfeld and Smolka definitions of their generalization of SLD-resolution to $\mathcal{R}(\mathcal{MSO})$. In the following $\mathcal{S}$ is always a definite clause specification, A an atom, F and G goals and $\phi \in \mathsf{CON}$.

Intuitively, we can substitute the body of a clause if the head represents a renamed variant of an atom occuring in the goal. To limit the search for the right clause, we incorporate constraint solving. This prevents the execution of a body of a clause if the requirements stated in the constraint are not met, i.e., we apply the rule only if the combination of the information in the constraint store and the constraint in the clause does not result in a contradiction. To provide an example, in the case of Prolog only clauses whose head unifies with the selected atom are tried for further resolution.

Definition 3.10. *Let $\mathcal{S}$ be a definite clause specification, V a finite set of $\mathcal{MSO}$-variables, A an $\mathcal{R}(\mathcal{MSO})$-atom, F and G goals, $\mathcal{A}$ an $\mathcal{R}(\mathcal{MSO})$-interpretation and ϕ, ψ, θ satisfiable $\mathcal{MSO}$-constraints.* ***$\mathcal{S}$-goal reduction*** *($\xrightarrow{r}_{\mathcal{S},V}$) is defined to be*

$$A \,\&\, G \,\&\, \phi \xrightarrow{r}_{\mathcal{S},V} F \,\&\, G \,\&\, \theta$$

if $A \leftarrow F \,\&\, \psi$ is a variant of a clause in $\mathcal{S}$ such that $[\![\phi]\!]^{\mathcal{A}} \cap [\![\psi]\!]^{\mathcal{A}} = [\![\theta]\!]^{\mathcal{A}}$ and $(V \cup \mathcal{V}(G)) \cap \mathcal{V}(F) \subseteq \mathcal{V}(A)$. Let $\xrightarrow{r}^{*}_{\mathcal{S},V}$ be the reflexive, transitive closure of $\xrightarrow{r}_{\mathcal{S},V}$.

Note that the condition on the free variables ensures that those common to F and G have to be "carried over" by A. The set V can be used to distinguish between variables introduced by the query and the ones introduced by the computation if V corresponds to the free variables of the initial goal. Since the language is closed under intersection, we can treat the conjunction of the two constraints $\phi \,\&\, \psi$ as a single new constraint θ.

We have already stated informally what we expect the answer to a search branch to one of our programs to be. More formally, if $\mathcal{S}$ is a definite clause specification, then an $\mathcal{S}$-answer of a goal G is a satisfiable $\mathcal{MSO}$-constraint ϕ such that $\phi \to G$ is valid in every model of $\mathcal{S}$.

The reduction rule we defined is actually sound, i.e., if we reduce a goal we are at least as specific about the answer as we were before.

Theorem 3.11 (H&S prop. 5.1). *Let $\mathcal{S}$ be a definite clause specification and G, F goals. If $G \xrightarrow{r}_{\mathcal{S},V} F$, then $[\![F]\!]^{\mathcal{A}} \subseteq [\![G]\!]^{\mathcal{A}}$ for every model $\mathcal{A}$ of $\mathcal{S}$.*

Lemma 3.12 (Soundness). *Let $\mathcal{S}$ be a definite clause specification, G a goal and ϕ an $\mathcal{MSO}$-constraint. If $G \xrightarrow{r}^{*}_{\mathcal{S},V} \phi$, then ϕ is an $\mathcal{S}$-answer.*

Proof. ϕ is an $\mathcal{S}$-answer just in case $(ASS - [\![\phi]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}} = ASS$ for all models $\mathcal{A}$ of $\mathcal{S}$. This is true by Theorem 3.11 since $[\![\phi]\!]^{\mathcal{A}} \subseteq [\![G]\!]^{\mathcal{A}}$ holds. □

Given a finite set of variables V as above, the notion of a solution is extended accordingly, i.e., $[\![\phi]\!]^{\mathcal{A}}_V = \{\alpha\restriction_V \mid \alpha \in [\![\phi]\!]^{\mathcal{A}}\}$; $\alpha\restriction_V$ being the restriction of α to the variables occurring in V.

We state in the following corollary that reduction is also complete. Intuitively, if there exists a solution to a goal, then we can find a constraint representing it via goal reduction.

COROLLARY 3.13 (Completeness, H&S cor. 5.3). *Let G be a goal, A an atom in G, $\mathcal{A}$ a minimal model of $\mathcal{S}$ and $\alpha \in [\![G]\!]^{\mathcal{A}}_{V}$. Then there exists an $\mathcal{S}$-answer ϕ of G such that $G \xrightarrow{r}{}^{*}_{\mathcal{S},V} \phi$ and $\alpha \in [\![\phi]\!]^{\mathcal{A}}_{V}$.*

3.2.4. *Properties of $\mathcal{R}(\mathcal{MSO})$.* In this section we investigate which properties of $\mathcal{MSO}$ are preserved by the extension to $\mathcal{R}(\mathcal{MSO})$. Höhfeld and Smolka establish the fact that if $\mathcal{MSO}$ is closed under renaming, so is $\mathcal{R}(\mathcal{MSO})$. Closure under intersection is covered by definition. But what about generative capacity and decidability?

Unfortunately, we have to answer the last point in the negative. We can indeed define undecidable problems. Consider the Post Correspondence Problem (PCP) (Post, 1946). A Post correspondence system over an alphabet Σ is a finite nonempty set $P = \{(l_i, r_i) \mid i = 1 \dots m \;\&\; l_i, r_i \in \Sigma^*\}$. A solution for such a system P is a sequence of indices $1 \leq i_1, \dots, i_n \leq m$ such that $l_{i_1} \cdots l_{i_m} = r_{i_1} \cdots r_{i_m}$. It is undecidable whether there exists a solution for a given system (if the alphabet has more than two symbols).

To be able to facilitate the presentation of the encoding into $\mathcal{R}(\mathcal{MSO})$ we sugar our notation. Instead of using the constraint part to deal with identity of sets (unification) and the decomposition of linearly ordered sets or better lists, we use the "$\bullet$" to encode list concatenation ($[X \bullet Y]$ for $\mathsf{Hd}(X, Z) \wedge \mathsf{Tl}(Z, Y)$) directly in the atom and not in an extra constraint in the body. Furthermore we assume that all our variables are constrained to be lists and that there exists an appropriate extension of our List and append examples in (8) to (14) and in Fig. 6 to lists of lists.

The encoding adapts the one from Hanschke and Würtz (1993). Words are represented as lists, e.g., $a_1a_2a_3 = [a_1 \bullet a_2 \bullet a_3]$. The empty lists denotes the empty word. The solution to the PCP will be encoded in two lists (of lists). We will successively generate all possible sequences of indices using the program in Fig. 7.

Considering the query in the first line, this encodes all sequences of length 1, in the next step we use the first sequence as a suffix to generate all sequences of length 2. In general, we select a sequence s of length j and use it as a suffix to generate the sequences of length $j + 1$. This is a fair strategy to generate all possible sequences. In fact, the program in Fig. 7 has an SLD refutation iff a given Post correspondence system has a solution. Therefore we can state the next lemma.

LEMMA 3.14. *Satisfiability in $\mathcal{R}(\mathcal{MSO})$ is undecidable.*

$$\leftarrow \mathsf{pcp}([[l_1] \bullet [l_2] \bullet \cdots \bullet [l_m]], [[r_1] \bullet [r_2] \bullet \cdots \bullet [r_m]])$$

$$\begin{aligned}
\mathsf{pcp}([L \bullet L1], [L \bullet R1]) &\leftarrow \\
\mathsf{pcp}([L \bullet X], [R \bullet Y]) &\leftarrow \\
&\quad \mathsf{append}(X, [[l_1 \bullet L] \bullet \cdots \bullet [l_m \bullet L]], X1) \,\&\\
&\quad \mathsf{append}(Y, [[r_1 \bullet R] \bullet \cdots \bullet [r_m \bullet R]], Y1) \,\&\\
&\quad \mathsf{pcp}(X1, Y1)
\end{aligned}$$

FIGURE 7. The PCP predicate

Furthermore, we can achieve the same by only considering a smallest binary program (see Hanschke and Würz for a definition), since we can eliminate the calls to append by the use of difference lists. Since smallest binary programs (on terms) are Turing equivalent (Devienne et al., 1994), it seems that our extension has the same power.

There are two directions we can take from here. Firstly, we can accept the extension as it is. In that case we defined a general purpose programming language on trees. Compared with Prolog, the difference is that we do not interpret terms over a Herbrand universe, but sets of nodes in $\mathcal{N}_2$. And secondly, we can search for (syntactic) characterizations of decidable subclasses of these inductive definitions. We pursue this last proposal in the next paragraphs.

Obviously we can use all relations which are explicitly definable in $\mathcal{MSO}$. An example for an explicit definition is the one of lists given in (8) to (14) or the definition of an arbitrary fact like the base case of append in Fig. 6, i.e., those relations which reduce via a syntactic substitution to an MSO signature.

Moschovakis (1974) provides the notions of first and second-order inductive definability to designate those relations which are recursively definable.[2] The following paragraphs relate the model-theoretic notions with the operational ones we used in the preceding sections.

We are going to define the relations recursively via an operator which computes all the tuples of sets contained in it. Since we limit ourselves to positive occurrences of the relation symbol, the operator is monotone and a least fixed point exists. An inductive definition is not satisfiable by a

[2]A detailed analysis of these notions in relation to $\mathcal{L}^2_{K,P}$ can be found in Rogers (To appear).

single relation but by a whole set of them. These are the fixed points of the operator. Intuitively, an inductive definition corresponds to a recursive definite clause definition where the relation we define is to be added to the signature of our model. In general, the name of the predicate in the definite clause corresponds to a third-order variable. For example in a fact like the base case of $\mathsf{append}(X, Y, Z)$, we can view append as a third-order variable with monadic second-order arguments. In the following all the X_i and X are meant to be (monadic) second-order variables. $\mathbf{X}$ is a third-order variable with monadic arguments, but naturally we do not allow quantification over the non-monadic and third-order ones. We assume the extension of the variable assignments to these second- and third-order variables such that they assign sets of nodes, i.e., relations on individuals or sets of sets of nodes, i.e., relations on sets. Recall that $\vec{X}$ denotes a tuple of distinct variables; $\vec{X}_n$ stands for a tuple of length n.

DEFINITION 3.15. *Let $R \subseteq \wp(\mathcal{T})^n$ and ϕ be a formula with $\mathcal{V}(\phi) = \{\vec{X}_n, \mathbf{X}\}$ such that $\mathbf{X}$ has arity n and occurs only positively. Then we define the operator $\Gamma_\phi : \wp(\wp(\mathcal{T})^n) \to \wp(\wp(\mathcal{T})^n)$ as*

$$\Gamma_\phi(R) := \{\alpha(\vec{X}_n) \,|\, \mathcal{N}_2 \models \phi[\alpha] \text{ and } \alpha(\mathbf{X}) = R\}.$$

The corresponding least fixpoint of $\Gamma_\phi^\omega = \Gamma_\phi\left(\bigcup_{n<\omega} \Gamma_\phi^n\right)$, i.e., the least Γ_ϕ^ω such that $\Gamma_\phi(\Gamma_\phi^\omega) = \Gamma_\phi^\omega$ will be called $F(\Gamma_\phi)$.

*The relation R is called **(second-order) inductively definable** iff there exists such a formula $\phi(\vec{X}_n, \mathbf{X})$ such that $R = F(\Gamma_\phi)$.*

Recall from Def. 3.6 that the relations we define there are sets of sets and computed as fixpoints, too. And $\mathcal{R}(\mathcal{MSO})$ does not have negation so that the relational symbols occur only positively. In case we have multiple clauses for one relation we assume this to correspond to a disjunction of its clauses. We can classify the $\mathcal{R}(\mathcal{MSO})$-relations as follows.

LEMMA 3.16. *Let $\mathcal{R}(\mathcal{MSO})$ be a relational constraint language with $r \in \mathcal{R}$, $ar(r) = n$.*

(i) *An $\mathcal{R}(\mathcal{MSO})$-relation $r(\vec{X}_n)$ is explicitly monadic second-order definable if it is a fact only or defined with clauses which do not contain inductive definitions and which are not directly or indirectly inductive themselves.*
(ii) *An $\mathcal{R}(\mathcal{MSO})$-relation $r(\vec{X}_n)$ is second-order definable if it uses only first-order induction, i.e., there appear only first order variables in the definition of its clauses. In particular r is monadic second-order definable if $n = 1$.*

(iii) *An $\mathcal{R}(\mathcal{MSO})$-relation $r(\vec{X}_n)$ is inductively second-order definable if it uses second-order induction.*

Proof. Let ϕ be the formula corresponding to the definite clause definition of the $\mathcal{R}(\mathcal{MSO})$-relation r, i.e., for all definite clauses $r(\vec{X}_n) \leftarrow \psi_i \,\&\, G_i$ in $\mathcal{S}$ with all ψ_i being $\mathcal{MSO}$-constraints, all G_i being (possibly empty) definite clause bodies and $1 \leq i \leq n$, in general $\phi(\vec{X}_n, \mathbf{X}) := \mathbf{X}(\vec{X}_n) \leftarrow (\psi_1 \,\&\, G_1) \vee \cdots \vee (\psi_n \,\&\, G_n)$ and $\alpha(\mathbf{X}) = r$.

(i) For facts and other non-recursively defined relations we simply give an explicit formula $r(\vec{X}_n) \Longleftrightarrow (\psi_1 \,\&\, G_1) \vee \cdots \vee (\psi_n \,\&\, G_n)$.

(ii) If the definition of the $\mathcal{R}(\mathcal{MSO})$-relation r uses only first-order induction, ϕ is of the form $\phi(\vec{X}_n, X)$ with each $X_i \in \vec{X}_n$ being only a singleton set and $\Gamma_\phi(r) = \{\alpha(\vec{X}_n) \mid \mathcal{N}_2 \models \phi[\alpha] \text{ and } \alpha(X) = r\}$ with fixpoint $F(\Gamma_\phi) = r$. The fixpoint exists since X occurs only positively. The relation is explicitly second-order definable by the Π^1_1-formula $r(\vec{X}_n) \Longleftrightarrow (\forall Z)[(\forall \vec{y}_n)[\phi(\vec{y}_n, Z) \Rightarrow \vec{y}_n \in Z] \Rightarrow \vec{X}_n \in Z]$ since it defines the property to be the least relation fulfilling ϕ. Note that if $n = 1$ then the preceding formula quantifies only over monadic second-order variables such that the relation is monadic second-order definable.

(iii) If the definition of the $\mathcal{R}(\mathcal{MSO})$-relation r uses second-order induction ϕ is of the form $\phi(\vec{X}_n, \mathbf{X})$ with $\vec{X}_n$ being arbitrary (monadic) second-order variables. Then the fixpoint $F(\Gamma_\phi)$ exists and equals r since $\mathbf{X}$ occurs only positively.

□

We have shown above that in general inductive second-order definable relations are undecidable. But the lemma gives us one (unattractive) way to ensure the decidability of the relations we define with our programs. We have to alter the Def. 3.6 such that only one singleton variable is allowed to appear in the atoms in all directly or indirectly recursive predicates. The new relations we introduce are then limited to $r^{\mathcal{A}} \subseteq \mathcal{T}$. Obviously this gives us only the re-introduction of the notion of derivation, but not more expressive power. On the other hand, there are binary inductive definitions which are decidable. Maybe the most obvious example is a definition of proper domination via recursion on immediate domination. We have to leave this point open for further research. But results from logic programming and higher order unification related to properties such as groundness or linearity of the involved programs or queries (Devienne et al., 1994; Levy, 1996) suggest that we might be able to find an interesting subclass. Other interesting subclasses of definable relations are identified as the binary *matching* relations (Lautemann et al., 1995) and the binary relations recognizable by tree-walking automata (Bloem and Engelfriet, 1997). It is a question open to further research how to relate these definability results to the inductive case.

3.2.5. *The Interpreter.* In this section on the interpreter we discuss the constraint solver and possible optimizations. For simplicity, we choose a basic left-to-right and depth-first selection of the goals.

The desiderata for a constraint solver are twofold. Firstly it is supposed to be complete and secondly efficient. Complete means just that it can decide the satisfiability of the constraints in the domain. Efficiency is mainly achieved by being able to work incrementally. The advantages of $\mathcal{MSO}$ theorem provers, despite their exponential worst case behaviours, are that

- they represent assignments to the variables efficiently as tree automata
- they can be used incrementally since we can view a tree automaton as a constraint store where the conjunction with a new constraint yields a new store;
- they are efficient since goal reduction uses only conjunction (which is only polynomial);
- emptiness, i.e., satisfiability, is trivially decidable after automata construction;
- we can create a library of constraints to avoid recomputation and view this as an extension of the signature of $\mathcal{MSO}$;
- the automata are a normal form.

We present the working of the desired interpreter in more detail. First of all, we single out a particular subclass of formulas by designating the atomic satisfiable formulas as *normal* constraints. Note that this includes all formulas which can be compiled into a minimal, non-empty tree automaton.

In the constraint solving step we actually compute the conjunction of the input automata to a new tree automaton and decide the emptiness of its language. This ensures that if the clause is applicable we can continue to work with normal constraints.

To ensure normality through the entire computation we demand that both goals and definite clause specifications introduce and utilize only normal constraints. A normal goal G is a goal which contains only one normal $\mathcal{MSO}$-constraint and a normal definite clause specification $\mathcal{S}$ is one in which all constraints in the clauses of $\mathcal{S}$ are normal. This does not impose any additional burden on the programmer since we can turn any definite clause specification into a normal one by compiling the necessary constraint if it is complex and by just eliminating the clauses containing unsatisfiable constraints.

The interpreter tries to solve a conjunction of normal $\mathcal{MSO}$-constraints immediately to test each goal reduction step. If it can compute a new normal $\mathcal{MSO}$-constraint the clause might be applicable. If not, we can discard this clause since it will not produce any solutions.

We only have a very preliminary prototype to test our ideas. We had to use our own implementation of a compiler since MONA can – at the moment at least – not be used inside other programs. Therefore we cannot make any claims to the practicality of the proposed interpreter.

A basic depth-first, left-to-right interpreter using the process sketched above does not use any number of optimizations which are available in the literature on constraint logic programming. We kept it deliberately simple, so that its behaviour is easily understandable. Nevertheless it is necessary to extend the scheme we proposed to get more benefits from using constraint logic programming. In the following paragraphs we will sketch some of the possible directions this might take.

Keeping within the Höhfeld and Smolka scheme, we can actually augment the system by including garbage collection and a type discipline. Garbage collection could be integrated since the scheme allows existential quantification in the relational extension. We could bind variables in places where they are not needed any more.

The second extension inside the scheme allows the typing of variables with $\mathcal{MSO}$ constraints as in standard imperative programming languages. The major advantage of the typing is that we can infer whether the specification contains errors by a well-typedness check. The definitions in Höhfeld and Smolka are general enough that they subsume our particular case.

Another possible optimization is the use of charts to record computations. In connection with the library of compiled relations, this might save the recomputation of some constraints. Obviously it involves checking whether the current constraint solving has been done before. But if we store the computations in a table, the lookup is not costly and therefore preferable to the recomputations involved in checking satisfiability.

A major change giving us a nice optimization would be the change to a concurrent framework. Maybe the most complete presentation of this can be found as the *ask* and *tell* framework developed in Saraswat (1993). Instead of only *telling* constraints, i.e., adding them to the constraint store, we can also *ask* whether a constraint is entailed by it. This amounts to checking the satisfiability of the constraint store conjoined with the negation of the constraint. Obviously this extension can be handled by the constraint solver. But to our knowledge there are no denotational semantics for this framework. We have to give up the nice properties of logic programs to coincide on their operational and declarative semantics. On the other hand, the framework allows to delay goals until the variables are sufficiently instantiated. Since we are considering P&P-based parsing, this seems to be a desirable property of our formalism.

A completely different source for optimizations can be found in the direction of static or offline program transformations. These techniques try to modify the program to optimize the runtime behaviour under a given selection strategy. Possible tasks include a data flow analysis of the program with the succeeding reordering of the literals in the bodies of clauses according to their determinacy (Strzalkowski, 1994). Another technique involves the addition of new predicates to allow bottom up execution of the program, called magic compilation (Ramakrishnan, 1991). Again, those approaches are not of immediate concern, but might be applicable to further improve execution of our programs.

4. Some Applications of $\mathcal{R}(\mathcal{MSO})$

In this section we are going to sketch some applications of $\mathcal{R}(\mathcal{MSO})$. As noted in the introduction a number of natural language formalisms adopt the constraint-based view. A grammar is treated as a (finite) set of constraints which have to be satisfied. In particular, in $\mathcal{MSO}$ the grammar is a $k+1$-ary relation describing a tree T and its k labels. In general, the parsing problem is to find the well formed trees spanning the input string. In our context, if we can generate an automaton covering an entire grammar, parsing means the relativization of the grammar automaton to the actual input sentence, i.e., conjoining a description of the input with the grammar. The input has been recognized iff the resulting automaton is non empty. The automaton represents the parse forest, but it does not output any trees. To achieve this, we have to run the automaton as a generator and record the structure of the run and the labels used. Remember that due to our encoding of the relations as a closure instead of having a unique minimal representative, the number of trees we can generate may be infinite. The same is true for the answers produced by $\mathcal{R}(\mathcal{MSO})$.

The following discussion should not be misunderstood as a serious attempt to define a parser. We are not concerned with efficiency or the presentation of even a complete algorithm. Rather we want to show the suitability of $\mathcal{R}(\mathcal{MSO})$ to linguistic processing by exploiting facets of the parsing problem. Note that an explicit definition corresponds directly to a fact.

4.1. **Pure Parsing.** As discussed in Sec. 2.3, one could use a complete formalization of the grammar in $\mathcal{MSO}$ for parsing directly via theorem proving. Assume for example that $\mathsf{Gram}(X)$ encodes Roger's formalization of Rizzi's ideas. Then the parsing problem can be stated as a simple $\mathcal{R}(\mathcal{MSO})$ goal, too. We name this approach *pure* parsing. Remember that we ignore the global variables denoting features/labels in the head of the formula.

We have to add one detail to connect pure parsing and theorem proving. We have to relate the input description to the set $Tree$ encoding the parse tree. This is done with the $\mathcal{MSO}$-constraint Yield and an $\mathcal{MSO}$ set variable labeling nodes with an empty phonology ($Empty$). Naturally, its distribution has to be constrained further.

$$\begin{aligned} \mathsf{Yield}(S,T) \stackrel{def}{\iff} \quad & (\forall x \in T)[((\neg(\exists y \in T)[x \lhd y]) \Leftrightarrow \\ & \qquad (x \in S \vee x \in Empty)) \wedge \\ & \qquad \neg(x \in S \wedge x \in Empty)] \\ & \wedge S \subseteq T \end{aligned} \tag{15}$$

A set of nodes forms the yield of a (finite) tree if it has as members exactly the nodes from its frontier with a nonempty phonology. We combine the Yield and the Gram constraints to obtain the final one.

$$\mathsf{Parse}(In, Tree) \stackrel{def}{\iff} \mathsf{Yield}(In, Tree) \wedge \mathsf{Gram}(Tree) \tag{16}$$

The corresponding definite clause parse is trivially identical with the definition in the preceding equation.

We treat the input string In as a set of nodes labeled with the words as monadic properties and linearly ordered by precedence. Additionally, we exclude any domination relations within In and further nodes with a nonempty phonology. This is clearly definable and serves as the initial constraint in our goal. Suppose we want to recognize the input *Jan sees*

Lena. Then the (simplified) goal looks as follows:

$$
(17)\qquad
\begin{array}{l}
\leftarrow \mathsf{parse}(In, Tree)\,\&\\
\quad \{(\exists x, y, z \in In)[Jan(x) \wedge Sees(y) \wedge Lena(z) \wedge x \prec y \prec z \wedge\\
\qquad\qquad (\neg\exists u, v \in In)[u \lhd^{+} v] \wedge\\
\qquad\qquad (\forall w \in In)[(w \not\approx x \wedge w \not\approx y \wedge w \not\approx z) \Rightarrow\\
\qquad\qquad\qquad Empty(w)]]\}
\end{array}
$$

Naturally, we can generate these descriptions of the input from a more standard representation, e.g., lists of words, automatically.

4.2. **Parsing with $\mathcal{R}(\mathcal{MSO})$ as a Programming Language.** The other extreme would be not to use any $\mathcal{MSO}$-based grammar formalizations, but to define data types and programs as for any other logic programming language. As a simple example, recall the definition of finite lists and append (see Sec. 3.2.4). With these datatypes we can then use $\mathcal{R}(\mathcal{MSO})$ and its interpreter as a programming language and implement (standard) parsing algorithms.

4.2.1. *Johnson (1995).* Using $\mathcal{R}(\mathcal{MSO})$ as a programming language, we can capture some of Johnson's proposals (Johnson, 1995) more or less directly by specifying the modules in $\mathcal{R}(\mathcal{MSO})$ and using the underlying tree as our datastructure for recursion. His parse relation holds of a string and a tree in case it obeys the xbar theory, the ecp[3] and the string is the yield of the tree, see Fig. 8. We take the liberty of presenting the constraints not as a single normal constraint but rather in a more informative manner representing all necessary conditions. But one could compile this conjunction into a new normal constraint on two variables without problems, i.e., $\mathsf{String\_and\_Tree}(X, Y)$. We assume that all our MSO variables, with the exception of the input which is constrained to be a string, i.e., a list of the leaves of a tree, are constrained to be trees, i.e., dominance-connected, rooted sets.

As an example, we will work through the definition of the xbar predicate. First, we have to define some new $\mathcal{MSO}$-constraints though. The definition of a root node is relative to a specific set, x is the root of X.

$$
(18)\qquad \mathsf{Root}(X, x) \stackrel{def}{\Longleftrightarrow} (x \in X) \wedge (\forall y)[y \in X \Rightarrow x \lhd^{*} y]
$$

[3]The ecp deals with the distribution of the empty nodes in the tree. We leave out its definition here since it can be captured using similar techniques as in the examples we will give, for more details see Morawietz and Cornell (To appear).

$$\begin{aligned}\mathsf{parse}(Words, Tree) \leftarrow &\\ &\{\mathsf{String}(Words) \wedge \mathsf{Tree}(Tree)\}\ \&\\ &\mathsf{xbar}(Tree)\ \&\\ &\mathsf{ecp}(Tree)\ \&\\ &\mathsf{yield}(Words, Tree)\end{aligned}$$

FIGURE 8. parse from Johnson (1995)

$$\begin{aligned}\mathsf{lexicon}(x) &\longleftarrow \{Sees(x) \wedge V(x) \wedge 3rd(x) \wedge \dots\}\\ \mathsf{lexicon}(x) &\longleftarrow \{Jan(x) \wedge N(x) \wedge Masc(x) \wedge \dots\}\\ \mathsf{lexicon}(x) &\longleftarrow \{Lena(x) \wedge N(x) \wedge Fem(x) \wedge \dots\}\\ &\vdots\end{aligned}$$

FIGURE 9. An $\mathcal{R}(\mathcal{MSO})$-definition of the lexicon

The definition of daughters in (19) just picks those nodes which are the immediate daughters from the model. Since it is defined for models consisting of binary branching trees, it specifies the sole two daughters. By the use of $\prec$ between y and z we do not only order them, but force them to be different nodes, too.

(19) $$\mathsf{Dtrs}(x, y, z) \overset{def}{\iff} x \triangleleft y \wedge x \triangleleft z \wedge y \prec z$$

(20) $$\mathsf{Subtree}(X, y, Y) \overset{def}{\iff} y \in X \wedge (\forall x \in X)[y \triangleleft^* x \Leftrightarrow x \in Y]$$

The subtree predicates are such that in the first one Y is the subtree of X rooted in y. In the second one x is the root of X with daughters y and z which are the roots of Y and Z respectively.

(21) $$\begin{aligned}\mathsf{Subtrees}(X, x, Y, y, Z, z) \overset{def}{\iff} &\mathsf{Root}(X, x) \wedge \mathsf{Dtrs}(x, y, z) \wedge\\ &\mathsf{Subtree}(X, y, Y) \wedge \mathsf{Subtree}(X, z, Z)\end{aligned}$$

Coming back to an issue raised in Sec. 2.3 in (5), let us note that in the CLP extension, we can avoid having to compile the lexicon as one constraint by having facts for each lexical entry separately (see Fig. 9). Naturally this is highly nondeterministic and shifts the burden to the CLP interpreter. But at least the approach makes the problem feasible.

$$
\begin{aligned}
&\mathsf{xbar}(X) \leftarrow \\
&\qquad \{\mathsf{Sing}(X)\}\ \& \\
&\qquad \mathsf{lexicon}(X) \\
&\mathsf{xbar}(X) \leftarrow \\
&\qquad \{\mathsf{Subtrees}(X, x, Y, y, Z, z)\}\ \& \\
&\qquad \mathsf{local\_xbar}(x, y, z)\ \& \\
&\qquad \mathsf{xbar}(Y)\ \& \\
&\qquad \mathsf{xbar}(Z)
\end{aligned}
$$

FIGURE 10. xbar following Johnson (1995)

$$
\begin{aligned}
\mathsf{local\_xbar}(x, y, z) &\leftarrow CP(x) \wedge Wh(y) \wedge C1(z) \wedge y \prec z \\
\mathsf{local\_xbar}(x, y, z) &\leftarrow CP(x) \wedge C(y) \wedge IP(z) \wedge y \prec z \\
\mathsf{local\_xbar}(x, y, z) &\leftarrow C1(x) \wedge C(y) \wedge IP(z) \wedge y \prec z \\
&\ \ \vdots
\end{aligned}
$$

FIGURE 11. local_xbar following Johnson (1995)

Now we come back to the definition of the X-Bar principle itself. The first clause of xbar in Fig. 10 allows nodes which appear in the lexicon. This includes those leaves of the parse tree with an empty phonology under the assumption that we include traces, PRO and other phonologically empty nodes with entries in the lexicon. The second one forces each local tree to satisfy some local_xbar constraints and the immediate subtrees to satisfy xbar. For simplicity, the local_xbar predicate consists just of phrase structure rules (see Fig. 11).

The predicate Yield was already defined in (15). We could have used the yield clause before the other predicates to reduce the search space, but since we are not really concerned with efficiency here, we stuck as closely as possible to Johnson's definition.

Furthermore, the general approach of constraining variables to a range of particular values allows the specification of data types. We can define them in the constraint language without having to explicitly include a type discipline. We simply attach additional conditions to every variable appearing in a clause. Consider as an example again the definition of

$$
\begin{aligned}
&\mathsf{lex}(X) \leftarrow \\
&\qquad \{\mathsf{Elist}(X)\} \\
&\mathsf{lex}(X) \leftarrow \\
&\qquad \{\mathsf{Hd}(x, X) \wedge \mathsf{Tl}(X, Y)\}\ \& \\
&\qquad \mathsf{lexicon}(x)\ \& \\
&\qquad \mathsf{lex}(Y)
\end{aligned}
$$

FIGURE 12. An $\mathcal{R}(\mathcal{MSO})$-definition of lex

the lexicon. Instead of anchoring xbar with it, we could have used as an alternative an atom $\mathsf{lex}(Words)$ in the definition of parse in Fig. 8 which would recurse on the input string, see Fig. 12.

In the definition one can see that we split the work into subtasks making sure that each terminal node is labeled with a lexical word and the necessary lexical features. This presupposes that we alter the definition of lists we gave in (12) such that they are precedence ordered instead of being ordered by dominance.

The advantage of guiding the search by the use of available data structures is readily apparent. What we did here for lists and for the set of nodes representing the tree avoids the need to compile the entire available information into one automaton which might be too big for present day computers to handle.

Additionally, we can use this implicit typing to choose a more specific interpretation for the variables. During the presentation of $\mathcal{MSO}$ we already discussed following Rogers and Rizzi (and others) in their assumption that the basic entities a P&P theory deals with are not longer nodes, but categories. Now we can constrain the variables in $\mathcal{R}(\mathcal{MSO})$ to range over categories so that they become the basic components of a theory.

An even better way to guide the search would be to check for the information content of the variables to be constrained and make the further action dependent on that. But in our present framework we cannot check for entailment and then act appropriately on the result. We can reject the alternative, but this might not be what we want to do. It would be more appropriate to keep the task in a queue and execute it when the variable becomes "specific" enough. This can be done in a concurrent framework which we cannot discuss further here.

$$
\begin{aligned}
&\mathsf{xp}(x) \leftarrow \\
&\qquad \{\mathsf{Max}(x) \wedge \mathsf{Proj}(x,y) \wedge (\exists X)[\mathsf{Dts}(x,X) \wedge \mathsf{Hd}(y,X) \wedge \mathsf{Tl}(X,Z)]\} \ \& \\
&\qquad \mathsf{x1}(y) \ \& \\
&\qquad \mathsf{specs}(y,Z)
\end{aligned}
$$

FIGURE 13. xp following Fong (1991)

4.2.2. *Fong (1991).* The next example is taken from Fong (1991). He uses phrase structure rules with restrictions attached to them. We can state his rule that an XP rewrites to an X1 followed by its specifiers if XP is a maximal projection and X1 immediately projects to XP as in Fig. 13. Again, we need some auxiliary definitions. We follow Fong in them, although we think they are pretty ad hoc. Max identifies those nodes which are maximal projections, Proj expresses that two nodes are projections of each other and Dts just collects all daughters of x in a set ordered by precedence.

(22) $\mathsf{Max}(x) \overset{def}{\Longleftrightarrow} x \in NP \vee x \in VP \vee \ldots$

(23) $\mathsf{Proj}(x,y) \overset{def}{\Longleftrightarrow} (x \in NP \Leftrightarrow y \in N1) \vee (x \in VP \Leftrightarrow y \in V1) \vee \ldots$

(24) $\mathsf{Dts}(x,X) \overset{def}{\Longleftrightarrow} (\forall y)[x \triangleleft y \Leftrightarrow y \in X]$

Since we are dealing with rules on nodes here, we do not get an accepted tree in the end. If we want to have an explicit parse tree, we have to either follow standard parsing techniques and record the rule applications or alternatively to include a particular subset in all the predicates such that quantification is always relative to it. It represents the parse tree in the end.

4.3. **Nonlocal Constraints.** All of these examples are just supposed to show how powerful $\mathcal{R}(\mathcal{MSO})$ is. We can cope with most of the standard approaches, although it does seem to be a bit cumbersome at times. But the real advantage of $\mathcal{MSO}$ and $\mathcal{R}(\mathcal{MSO})$ lies in the ability to directly express nonlocal constraints. Following an argument of Johnson (1995) on nonlocal constraints, we claim that they are crucial in the P&P-based approaches to natural language and the difference to other theories as for example HPSG (Pollard and Sag, 1994). This is most clearly seen in phrase structure based implementations of these theories, e.g., in ALE (Carpenter and Penn, 1998). We can see the strong locality requirements since all the feature constraints are coupled to phrase structure rules.[4] The constraints are only concerned with a node and its immediate children. They can be checked during the (standard) parse tree construction. Naturally, this requires some encoding of the nonlocal dependencies for example between a filler and its gap via slash features. Since this seems to be more a problem of memory management and implementation techniques than of linguistic analysis, we strive to avoid it.

The relaxation of the locality conditions is realizable in $\mathcal{R}(\mathcal{MSO})$ directly since the variables range over sets of nodes. This is where we gain on Johnson's approach. Where he pursues the alternatives of coroutining and compilation into local constraints, we can use the principles directly on sets of nodes. And furthermore, we have quantification available in the constraint language $\mathcal{MSO}$ so that we can express statements as the following: "there has to exist a trace somewhere down in this subtree".

One approach to parsing which reflects the modular nature of P&P-theories would be the incremental narrowing of the constraint store by passing the set of nodes representing the tree to be constructed through the various modules. This is an improvement on the general generate and test method since we do not have to construct a particular tree which has to be validated. We limit the range of values the variable can take, although, on the surface, it looks much like the standard Prolog approaches to P&P-parsing. But we are not limited to the construction of any local dependencies since we are directly passing on the constraints on the entire tree. In fact, we modify a tree automaton representing the constraint store. Even with a definition as given in Fig. 8, we can almost use the pure $\mathcal{MSO}$ approach, only modified to work incrementally on a particular input. Instead of using the tree for recursion we can just conjoin more an more separately compiled modules. This becomes especially interesting if we consider using an approach to the lexicon as the one given in Fig. 12 since we do not have to specify `xbar` with definite clauses any more.

As mentioned above, the other area of advantages of $\mathcal{MSO}$-constraints is in the specification of the existence of further information to be found. This is most easily be illustrated in a phrase structure based approach: we can now express the relation between a filler and its gap directly. Assume for example that we have a definite clause which is a phrase structure rule combining a +WH element with a VP. We make sure that indeed the VP contains the appropriate trace without having to specify a passing mechanism like slash features by quantifying over the trace, see Fig. 14.

[4] In HPSG itself they are not bound to the ID schemata, but one still has to express explicit paths of features to find the desired values.

$$
\begin{aligned}
&\mathsf{s}(X) \leftarrow \\
&\qquad \{(\exists x, y, z)[\mathsf{Subtrees}(X, x, Y, y, Z, z)] \wedge \\
&\qquad\ \ (\exists z)[Z(z) \wedge Trace(z) \wedge \mathsf{Feq}(z, Y)]\} \ \& \\
&\qquad \mathsf{np}_{\mathsf{wh}}(Y) \ \& \\
&\qquad \mathsf{vp}(Z)
\end{aligned}
$$

FIGURE 14. Nonlocality in a phrase structure rule

The clause finds and recurses on the subtrees if there exists a trace in Z such that it agrees on the relevant features with the extracted WH-phrase.

An example for the use of quantification in the guard of an $\mathcal{R}(\mathcal{MSO})$-clause which is more suitable for P&P approaches can be found in formalizations of the binding principles. We can state a constraint that for all appropriate nodes in the scope of the current node, a principle has to hold. More concretely, whenever we find a proper noun phrase we stipulate that it may bind all appropriate anaphoric noun phrases if it occurs in their governing category. This is a simplified version of Principle A of the Binding Theory. A very simple formalization of such a constraint – ignoring all the inherent problems of the binding theory – may look as given below. We assume the existence of predicates Anaphor to pick out anaphoric noun phrases, Gov-Cat which encodes the governing category and A-Bind ensuring that x A-binds y.

$$
\begin{aligned}
&NP(x) \Rightarrow \\
&\qquad ((\forall y)[(\mathsf{Anaphor}(y) \wedge (\exists X)[\mathsf{Gov\text{-}Cat}(y, X) \wedge X(x)]) \\
&\qquad\qquad\qquad\qquad \Rightarrow \mathsf{A\text{-}Bind}(x, y)])
\end{aligned}
\tag{25}
$$

5. SUMMARY

We presented in this paper the first step towards $\mathcal{MSO}$-based constraint logic programming. We applied the Höhfeld and Smolka scheme to MSO logic on trees resulting in the the constraint logic programming language $\mathcal{R}(\mathcal{MSO})$. The scheme gives us a sound and complete operational semantics. We presented an outline for a basic interpreter using goal reduction and constraint solving. Since we have specialized constraint solvers for $\mathcal{MSO}$ formulas, e.g., MONA, it is possible to develop a straightforward prototype.

One advantage we see in the logic programming approach consists in the operational interpretation which can be given to the relational extensions of our constraint language. We have shown how to leave the pure model theory towards a more processing oriented view. In fact, we defined a programming language which can directly use linguistic primitives. A further advantage is that we are able to apply techniques from computer science to further develop a useful tool for natural language processing.

We raised some topics and problems in the paper which are yet to be answered. There are three major areas we have to investigate further. The first is concerned with formal issues connected with $\mathcal{R}(\mathcal{MSO})$. If we want to extend our relational constraint language with negation it seems to be necessary to clarify which alternatives there are compared to the standard approaches. Standardly, compactness is a condition to be met if the desideratum of an agreement of operational and denotational semantics for a CLP language with negation is to be achieved. Compactness fails in $\mathcal{MSO}$ since finiteness is definable. Another interesting point seems to be the generative capacity of the relational extension. Considering the discussion on definability in Sec. 3.2.4, the relational extension gains in power since it involves second-order inductive definitions. It is especially interesting whether we can find a mild, decidable extension, e.g., to the indexed languages. In that case the objection that $\mathcal{MSO}$ cannot handle all natural languages, i.e., Swiss German or Dutch, could be invalidated.

The second direction is concerned with the further development and extension of the $\mathcal{R}(\mathcal{MSO})$-interpreter. For example, we are interested in whether there still exists a denotational interpretation for a concurrent framework; and if this is the case whether the denotational and operational semantics still coincide as for standard logic programming. An issue raised briefly in Saraswat (1993) is the use of domain theory in the guise of information systems for the development of the semantics. We believe that this also is an interesting area for further research.

The third direction is probably the most difficult. We have to spend more time on the development of $\mathcal{R}(\mathcal{MSO})$-based formalizations of P&P-theories. We hinted at some details, but naturally this does not provide proof for a conclusive realization of the principles and modules. We do not doubt that a formalization can be found since $\mathcal{R}(\mathcal{MSO})$ seems to be powerful enough. But it also seems to be nontrivial to come up with a proper and complete implementation which uses the features $\mathcal{MSO}$ and $\mathcal{R}(\mathcal{MSO})$ offer. Especially the investigation into the successor of GB, the minimalist program (Chomsky, 1995) seems to be promising, since it also combines procedural and representational techniques.

References

Andersen, H. R. (1996). An introduction to binary decision diagrams, Course Notes, Technical University Denmark.

Ayari, A., Basin, D. and Podelski, A. (1998). LISA: A specification language based on WS2S, *in* M. Nielsen and W. Thomas (eds), *Computer Science Logic, CSL'97*, number 1414 in *LNCS*, Springer.

Basin, D. and Klarlund, N. (To appear). Automata based symbolic reasoning in hardware verification, *The Journal of Formal Methods in Systems Design*.

Bloem, R. and Engelfriet, J. (1997). Characterization of properties and relations defined in monadic second order logic on the nodes of trees, *Technical Report 97-03*, Leiden University.

Büchi, J. (1960). Weak second-order arithmetic and finite automata, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* **6**: 66–92.

Carpenter, B. and Penn, G. (1998). ALE: The attribute logic engine, Version 3.0 Beta, *User manual*, Laboratory for Computational Linguistics, Carnegie Mellon University.

Chomsky, N. (1981). *Lectures on Government and Binding*, Dordrecht: Foris.

Chomsky, N. (1995). *The Minimalist Program*, Vol. 28 of *Current Studies in Linguistics*, MIT Press.

Devienne, P., Lebègue, P., Parrain, A., Routier, J. and Würtz, J. (1994). Smallest horn clause programs, *Journal of Logic Programming* **19/20**: 1–41.

Doner, J. (1970). Tree acceptors and some of their applications, *Journal of Computer and System Sciences* **4**: 406–451.

Fong, S. (1991). *Computational Properties of Principle–Based Grammatical Theories*, PhD thesis, MIT.

Frank, R. and Vijay-Shanker, K. (1998). TAG derivation as monotonic c-command, *Proceedings of the Fourth Workshop on Tree-Adjoining Grammars and Related Frameworks TAG+98*, Philadelphia, PA. Extended Abstract.

Gécseg, F. and Steinby, M. (1997). Tree languages, *in* G. Rozenberg and A. Salomaa (eds), *Handbook of Formal Languages: Beyond Words*, Vol. 3, Springer.

Hanschke, P. and Würtz, J. (1993). Satisfiability of the smallest binary program, *Information Processing Letters* **45**(5): 237–241.

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

Huybregts, R. (1984). The weak inadequacy of context-free phrase structure grammars, *in* de Haan, Trommelen and Zonneveld (eds), *Van Periferie naar Kern*, Foris, Dordrecht, pp. 81–99.

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey, *Journal of Logic Programming* **19/20**: 503–581.

Johnson, M. (1995). Constraint-based natural language parsing, European Summer School in Logic, Language, and Information (ESSLLI '95), Barcelona, Course notes.

Kayne, R. S. (1994). *The Antisymmetry of Syntax*, Vol. 25 of *Linguistic Inquiry Monographs*, MIT Press.

Kelb, P., Margaria, T., Mendler, M. and Gsottberger, C. (1997). MOSEL: A flexible toolset for monadic second-order logic, *in* E. Brinksma (ed.), *TACAS '97 Workshop*, number 1019 in *LNCS*, Springer, pp. 183–202.

Klarlund, N. (1998). MONA & Fido: The logic-automaton connection in practice, *in* M. Nielsen and W. Thomas (eds), *Computer Science Logic, CSL'97*, number 1414 in *LNCS*, Springer.

Klarlund, N. and Møller, A. (1998). *MONA Version 1.2 User Manual*, BRICS Notes Series NS-98-3, Department of Computer Science, University of Aarhus.

Lautemann, C., Schwentick, T. and Thérien, D. (1995). Logics for context-free languages, *in* L. Pacholski and J. Tiuryn (eds), *Computer Science Logic CSL '94*, number 933 in *LNCS*, Springer, pp. 205–216.

Levy, J. (1996). Linear second-order unification, *in* H. Ganzinger (ed.), *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, number 1103 in *LNCS*, Springer, pp. 332–346.

Lloyd, J. (1984). *Foundations of Logic Programming*, Springer Verlag.

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

Morawietz, F. and Cornell, T. (To appear). The logic-automaton connection in linguistics, *Proceedings of LACL 1997*, LNAI, Springer.

Moschovakis, Y. N. (1974). *Elementary Induction on Abstract Structures*, North-Holland, Amsterdam.

Pollard, C. J. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar*, University of Chicago Press and CSLI Publications.

Post, E. (1946). A variant of a recursively unsolvable problem, *Bulletin of the AMS* **52**: 264–268.

Rabin, M. O. (1969). Decidability of second-order theories and automata on infinite trees, *Transactions of the AMS* **141**: 1–35.

Ramakrishnan, R. (1991). Magic templates: A spellbinding approach to logic programs, *Journal of Logic Programming* **11**: 189–216.

Rizzi, L. (1990). *Relativized Minimality*, MIT Press.

Rogers, J. (To appear). *A Descriptive Approach to Language-Theoretic Complexity*, Studies in Logic, Language and Information, CSLI.

Saraswat, V. A. (1993). *Concurrent Constraint Programming*, MIT Press.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language, *Linguistics and Philosophy* **8**: 333–343.

Strzalkowski, T. (1994). A general computational method for grammar inversion, *in* T. Strzalkowski (ed.), *Reversible Grammars in Natural Language Processing*, Kluwer, pp. 175–200.

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

Index

Seminar für Sprachwissenschaft, SFB 340, A8, University of Tübingen, Wilhelmstr. 113, 72074 Tübingen, Germany
E-mail address: `frank@sfs.nphil.uni-tuebingen.de`