Bericht Nr. 86, Februar 1997

Arbeitspapiere des
Sonderforschungsbereichs 340

ISSN 0947-6954/97

Sprachtheoretische Grundlagen für die Computerlinguistik

Frank Morawietz:

Monadic Second Order Logic, Tree Automata and Constraint Logic Programming

Universität Stuttgart Universität Tübingen
IBM Deutschland Informationssysteme GmbH

Arbeitspapiere des SFB 340
Bericht Nr. 86, Februar 1997

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 assignment trees satisfying the formulas. Since this allows a compact and efficient representation of constraints, we can use mso theorem provers as constraint solvers for the clp interpreter. Building on the detailed investigation of mso logic in Rogers (1994) as a tree description language in the Principles and Parameter (P&P) paradigm, we present as motivation and applications details from the realm of P&P-based parsing.

1. Introduction

Model theoretic techniques are used in formalizations of a number of approaches toward theories about natural language. Maybe most notably in feature logics (King; 1989; Carpenter; 1992) for Head-Driven Phrase Structure Grammar (Pollard and Sag; 1994); modal logic (Blackburn and Meyer-Viol; 1994) for Lexical-Functional Grammar (Kaplan and Bresnan; 1983); or full first order predicate logic (Stabler; 1992), monadic second order logic (Rogers; 1994) or again modal logic (Kracht; 1995) for the

---

To appear in: Kolb, H.-P. & U. Mönnich (eds.), *The Mathematics of Syntactic Structure*.

The research presented in this report was sponsored by Teilprojekt A8 of the Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft. 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 0947-6954/97

Principle and Parameter (P&P) approaches in the Government and Binding tradition (Chomsky; 1981).[1] All of these constraint-based formalisms characterize objects with logical description languages declaratively., i.e., without the specification of details on the generation of the 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.

In this paper we bridge the gap between the two by exploiting the connection between constraints in monadic second order (mso) logic on (finite binary) trees and tree automata. Our purpose is twofold. Firstly we discuss the operational interpretations of mso logic. Since the solutions to constraints expressed in monadic second order logic are represented by tree automata recognizing the assignment trees satisfying the formulas, we can use the automata as the operational interpretation of out formalism. 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. We want to extend the computational power since mso logic characterizes the strongly context-free languages, but natural languages are generally believed to be higher in the Chomsky hierarchy (Shieber; 1985; Huybregts; 1984). Inductive definitions can be seen as special cases of constraint logic programming (clp) clauses. The specification of a clp interpreter reintroduces the operational concept of derivation of a (representation of a) structure. More concretely, we propose the embedding of mso logic into the constraint logic programming scheme of Höhfeld and Smolka (1988).[2] The extension gives us an even more powerful and flexible language and allows us to guide the compilation process with appropriate programs.

The paper is organized as follows. In the first section we review the results on mso logic and tree automata 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 the relational extension, discuss its properties and present some examples. In the last section, we argue for the usefulness of the approach with examples from the realm of P&P-based parsing.

2. Trees, Tree Automata, and MSO Logic

[1] We do not claim that this enumeration is in any way complete or representative.

[2] We are aware that a constraint logic programming approach toward P&P has been taken in Hathout (1992), too. But unfortunately the thesis is in French and therefore not completely accessible to us.

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 (1984).

Assume an alphabet $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{\lambda\}$ and i the arity of the elements in Σ_i. The set of binary trees T_Σ over Σ is the set of terms build 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. 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 $\{\underline{0}, \underline{1}\}$ only.

As an example consider the alphabet with $\Sigma_2 = \{A\}$. Some trees derived from that alphabet are shown in figure 1. Note that λ serves as

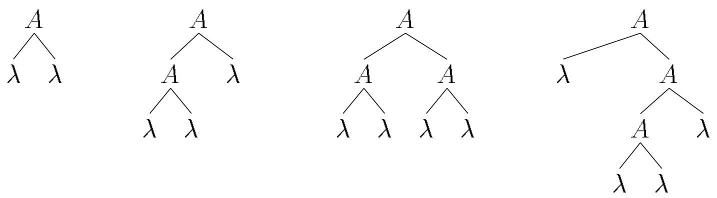

FIGURE 1. Some example trees

a terminator and has no content.

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 set of states, $a_0 \in A$ the initial state, $F \subseteq A$ the final states and $\alpha : (A \times A \times \Sigma) \to A$ the transition function. We can extend the transition function homomorphically to trees by inductively defining $h_\alpha(\lambda) = a_0$ and $h_\alpha(\sigma(t_1, t_2)) = \alpha(h_\alpha(t_1), h_\alpha(t_2), \sigma)$, $t_1, t_2 \in T_\Sigma$. 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 $h_\alpha(t)$.

Since tree automata will be used later on, we present an example here. The automaton in figure 2 recognizes, among others, the trees given in

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

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

FIGURE 2. A simple tree automaton

figure 1. This admittedly very simple tree automaton recognizes all binary trees whose interior nodes are labeled with A. 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, determinization and minimization. Most of the constructions are adaptions from the corresponding ones on finite state automata. We use them to parallel the connectives and quantifiers from our logic. More details on both the specification and implementation of the tree automata and their constructions can be found in either Morawietz and Cornell (1996) or Henriksen et al. (1995) and Biehl et al. (1996).

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 an interesting descriptive complexity result. It characterizes exactly the strongly context-free languages.

The techniques we are using come originally from an old result in logic, 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.[3]

[3] 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.

The 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.

The proofs of these decidability results are inductive on the structure of mso formulas. Therefore 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. We will see how this is done in the next section, but the point can be appreciated immediately. For example, Niehren and Podelski (1992) have investigated the usefulness of these techniques in dealing with 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 (1994) 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, 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 (1995).

A mso logical language has a syntax of both first and second order quantification, 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 build as usual. So we deviate from first order logic by the use of monadic second order variables and quantifiers. To create a tree description logic we fix the domain of the interpretation to trees and add constant binary relations to the syntax which will be interpreted as the successor functions. Due to the inductive proof we can also add any desired mso definable relation.

For the structure of WS2S, we are going to assume a tree domain with the extension of (at least) the two successor functions. These correspond intuitively to the relations of left and right daughter. Then the structure can be extended with the interpretations of the extra relations we want to use. We will call the structure of WS2S $\mathcal{N}_2$.

Intuitively, mso predicates pick out sets of nodes. We can think of them as labeling the nodes. Therefore, mso logic on trees does give us the power to specify constraints on labeled trees. Each formula represents a constraint on the valuation of its free variables which is determined by the assignment of the variables to (sets of) nodes. 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 (sets of) nodes in a tree, we will sometimes call them assignment trees.

2.2.1. *An Example Language: $\mathcal{L}^2_{K,P}$.* We draw our examples from tree description logics used in the P&P paradigm. In particular $\mathcal{L}^2_{K,P}$, a logic proposed in Rogers (1994), 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 thesis most of the proposals made in *Relativized Minimality* by Rizzi (1990).[4]

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 and added, e.g., proper precedence to express ordering information. We parameterize the language with both individual and predicate constants.

Example 2.1. *Let $\mathcal{L}^2_{K,P}$ be defined by a set K of countably many individual constants; a set P of countably many propositional constants; a countably infinite set $\mathbf{X}$ of 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.*

Occasionally we 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 we can always encode both the predicate and individual constants as free variables. Since $\mathcal{L}^2_{K,P}$ has set valued variables only, we have to enforce that the variables denoting individuals are of cardinality 1 as given as an example below in equation (2). We deviate slightly from Rogers' definition of the language in that we include $\in$ explicitly. Again, nothing substantial is changed by this.

We will freely use appropriate symbols for standard combinations of the boolean operators, for example $\Rightarrow$, $\not\approx$, $\Leftrightarrow$, ... As an example for relations we can define with this language, take the subset or equal relation in

[4] $\mathcal{L}^2_{K,P}$ is not limited to binary finite trees. Rogers actually allows for the full power of Rabin's result to be used.

equation (1), the constraint for a set to contain exactly one member in equation (2) and, more linguistically motivated, the formula denoting the relation of directed asymmetric c-command[5] in the sense of Kayne (Kayne; 1994), see equation (3).

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

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

$$\begin{aligned} AC\text{-}Com(x_1, x_2) \overset{def}{\Longleftrightarrow} & \neg(x_1 \lhd^* x_2) \wedge \neg(x_2 \lhd^* x_1) \wedge x_1 \prec x_2 \wedge \\ & (\forall x_3)[x_3 \lhd^+ x_1 \Rightarrow x_3 \lhd^+ x_2] \end{aligned} \tag{3}$$

The definition in equation (3) specifies that two nodes asymmetrically c-command each other iff neither dominates the other, they are not identical, x_1 precedes x_2 and all x_3 which properly dominate x_1 properly dominate x_2. We depict an example tree in figure 3. The dashed arrow represents the c-command relation. Note that x_1 also c-commands y and

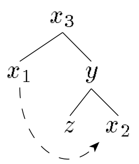

FIGURE 3. An example for asymmetric directed c-command

z.

Models for $\mathcal{L}^2_{K,P}$ are an appropriate set of nodes, a tree domain, with relations interpreting the elements from the signature.[6]

Example 2.2. *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 $\lhd$, $\lhd^*$ 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 $\langle \mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{L} \rangle$.

[5] This relation is not monadic, but reducible via syntactic substitution to a mso signature.

[6] We limit the structure to dominance, parent and precedence since the others can be defined in terms of them.

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 just about as complex as it is possible to be. The number of states can be in the worst case given as a function of the number of variables in the input formula with a stack of exponents as tall as the number of quantifier alternations in the formula. However, there is a growing body of work in the computer science literature (Henriksen et al.; 1995)[7] 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 linguistic applications as well.

Nevertheless, it is attractive to have a principled means of generating automata from a constraint-based theory. 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 variable assignments have to assign sets of nodes to the variables. Doner and Thatcher & Wright 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.

More formally, let again $\mathcal{N}_2$ be the structure of WS2S and $\phi(\vec{X})$ a formula with free variables $\vec{X}$. 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 assignment tree. For each mso formula $\phi(\vec{X})$, we can inductively build a tree automaton $\mathfrak{A}^\phi$ such that

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

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

Consider as an example the corresponding tree automaton to Kayne's c-command relation from equation (3). It is shown in figure 4. Consider the subscripts of the variables to correspond to positions in a bitstring. A $\underline{1}$ in position i means that the node is in set X_i. Roughly, these bitstrings are the alphabet the automata work on. On closer examination of the

[7] The given reference is to the description of the tool MONA on mso logic of strings. There is work in progress to extend MONA to trees (Biehl et al.; 1996).

transitions, we note that we just percolate the initial state as long as we

$$\mathfrak{A} = \langle \{a_0, a_1, a_2, a_3, a_4\}, \{\underline{11}, \underline{10}, \underline{01}, \underline{00}\}, a_0, \{a_3\}, \alpha \rangle$$

$$\begin{array}{ll} \alpha(a_0, a_0, \underline{00}) = a_0 & \alpha(a_0, a_0, \underline{10}) = a_2 \\ \alpha(a_0, a_0, \underline{01}) = a_1 & \alpha(a_0, a_1, \underline{00}) = a_1 \\ \alpha(a_0, a_3, \underline{00}) = a_3 & \alpha(a_1, a_0, \underline{00}) = a_1 \\ \alpha(a_2, a_1, \underline{00}) = a_3 & \alpha(a_3, a_0, \underline{00}) = a_3 \\ \multicolumn{2}{c}{\text{all other transitions are to } a_4} \end{array}$$

FIGURE 4. The automaton for $AC\text{-}Com(x_1, x_2)$

find only nodes which are neither x_1 nor x_2. From the initial state on both the left and the right subtree we can either go to the state denoting "found x_1" (a_2) if we read symbol $\underline{10}$ or to the state denoting "found x_2" (a_1) if we read symbol $\underline{01}$. We can then percolate a_1 as long as the other branch does not immediately dominate x_1. If we come into the situation that we have a_2 on the left subtree and a_1 on the right one, we go to the final state a_3 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 $\underline{00}$ labeled nodes dominating the constellation or being dominated by it. We recognize the closure of the ($AC\text{-}Com$) relation in the sense of Doner. The relation can now serve as a new primitive in further mso formulas.

It is obviously desirable to keep the automata as small as possible. Therefore we employ the suitably adapted minimization of finite state automata. If we think of the automata as representing knowledge of the grammatical structures we want to characterize, then the automata constructions symbolize how to combine or alter the knowledge accordingly. Especially, minimization corresponds to a generalization of knowledge.

2.4. **Theorem Proving.** Automated theorem proving is the standard approach to processing with constraint-based grammars. The effectiveness of 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. And an automaton recognizes the empty language (due to minimization) iff it has only an initial state which is not final. 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 a whole grammar. Unfortunately, as mentioned above, the problem to generate the tree automata from arbitrary mso formulas is highly exponential. Sources of exponentiality are the necessary determinization of automata, the cross product for the representation of the combination of the states in the transitions, and the explosion of the alphabet space. As an example, consider a naïve specification of the lexicon:

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

We have specified a set called *Lexicon* via a disjunctive specification of lexical labels, e.g., *Sees*, and the appropriate combination of features, e.g., V. Since every feature and every lexical entry is represented with a new free set variable, the alphabet size with the encoding as bitstrings will be at least 2^{n+m}, n is the number of lexical entries and m the number of features.

For tests, we used a prototype called MoSeL (Morawietz and Cornell; 1996)[8] which compiles the tree automata from the formulas. But it is an open problem whether the generation of a complete grammar can be handled. The question is not so much efficiency, but rather feasibility since the size of an automaton representing an entire grammar is still indeterminate.

But the tests have shown that we can compile modules and by this gain insight into parts of the formalization. For example, we have successfully compiled the tree automaton representing c-command on categories rather then on single nodes following the presentation of Rizzi's ideas from *Relativized Minimality* in Rogers' thesis. Informally, a category is a node which has been split into segments through adjunction. The definition of categories can be found in equations (6) and (7).[9] A component is a set

[8] As mentioned, we are aware of promising work in progress at the University of Aarhus on the extension of MONA under the Thatcher & Wright semantics. MONA++ tries to minimize the alphabet explosion by the use of binary decision diagrams for the alphabet representation and the quadratic blow-up in the transition table by the use of guided automata.

[9] We are glossing over some details, Feq ensures agreement on the relevant features; $Path$ can be understood as a connected ordered subgraph, and Adj is the feature

of nodes which (a) form a path in the tree; (b) agree on a set of particular features; and (c) one node in every local tree has to be adjoint.

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

A category is a maximal component.

$$(7)\quad Category(X) \stackrel{def}{\Longleftrightarrow} Component(X) \wedge \\ (\forall Y)[(X \subseteq Y \wedge Y \not\subseteq X) \Rightarrow \neg Component(Y)]$$

Building on that, we can define primitives of domination, precedence and c-command in terms of categories rather than nodes.

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 figure 5 one can see that I_j is indeed not dominated by the category formed by the two nodes labeled C. I_j can c-command its trace t_j since all *categories* which dominate it, also dominate t_j.

As might be suspected considering equations (6) and (7), the complete presentation of the formulas encoding category based c-command is fairly complex and therefore beyond the scope of this paper.[10] But actually, the resulting automaton is small. It has only 5 states and 15 transitions, see figure 6.

One can see from the discussions in this section that the approach of pure theorem proving 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, so far, purely model-theoretic presentation. By this we provide a way to use parts or primitives of formalizations separately by the specification of programs and gain the desired extension of the generative capacity.

---

denoting an adjoint node. A number of additional constraints on the distribution of adjuncts is necessary to achieve the right behaviour.

[10] The reader is encouraged to check the details in Rogers' thesis.

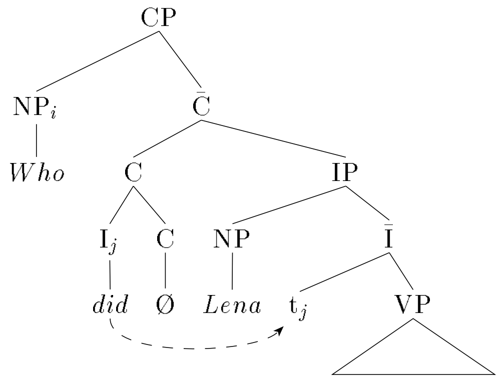

FIGURE 5. A syntax tree from Rogers (1994)

$$\mathfrak{A} = \langle \{a_0, a_1, a_2, a_3, a_4\}, \{\underline{11}, \underline{10}, \underline{01}, \underline{00}\}, a_0, \{a_3\}, \alpha \rangle$$

$$\begin{array}{ll}
\alpha(a_0, a_0, \underline{00}) = a_0 & \alpha(a_0, a_0, \underline{10}) = a_2 \\
\alpha(a_0, a_0, \underline{01}) = a_1 & \alpha(a_0, a_1, \underline{00}) = a_1 \\
\alpha(a_0, a_1, \underline{10}) = a_3 & \alpha(a_0, a_2, \underline{00}) = a_2 \\
\alpha(a_0, a_2, \underline{01}) = a_3 & \alpha(a_0, a_3, \underline{00}) = a_3 \\
\alpha(a_1, a_0, \underline{00}) = a_1 & \alpha(a_1, a_0, \underline{10}) = a_3 \\
\alpha(a_1, a_2, \underline{00}) = a_3 & \alpha(a_2, a_0, \underline{00}) = a_2 \\
\alpha(a_2, a_0, \underline{01}) = a_3 & \alpha(a_2, a_1, \underline{00}) = a_3 \\
\multicolumn{2}{c}{\alpha(a_3, a_0, \underline{00}) = a_3} \\
\multicolumn{2}{c}{\text{all other transitions are to } a_4}
\end{array}$$

FIGURE 6. The automaton for category based c-command

3. Constraint Logic Programming

Logic programming reflects nicely the separation of the actual logic of a program and the control of its execution (a standard reference to logic programming is Lloyd; 1984). The execution method used in most logic programming (lp) languages is based on SLD-resolution. Programs in the lp paradigm are sets of clauses. One could be content with a logical interpretation. But on the other hand one wants to compute with them. So the semantics is reformulated in terms of least models and fixpoints. The programs are interpreted within Herbrand models, i.e., sets of ground atoms. The two main problems[11] with this approach, the generate and test paradigm to find the solutions and the necessary use of uninterpreted structures only, can be overcome by the addition of constraints to the standard logic programming scheme. There are two ways to enhance logic programming languages with constraints. One is to provide specialized interpreted domains of computation, as for example done in Jaffar and Lassez (1986). The other one is concerned with the application of consistency techniques (Frühwirth et al.; 1993). In this paper we are concerned mainly with the first of these issues. The guiding of the search behaviour of the programs will be sketched briefly, but left as a topic for further research.

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 focus on just those parts of the grammar which are relevant to the actual computation instead of having to consider – and precompile – all possibilities at once. A standard interpreter, as for example the one used in Prolog (left to right, depth first search), is sufficient for the execution of one of our programs. The answer of a 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 section 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

[11]There are more problems – one which deserves attention is the one of infinite computations. We do not address this issue here, but propose an approach using domain theory to remedy that (Carlson; 1991).

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 can not be formalized in mso-based approaches since their underlying structure is comparable to the copy language[12] 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. We fix the domain of computation to trees. We use this scheme as opposed to the more standard one of Jaffar and Lassez (1986) 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. If we want to use them, we can use sets with the additional constraint that they are singleton. This is the case in both mso theorem provers (MoSeL and MONA) we know of. Nevertheless, we will use as abbreviations lowercase letters $x, y, \ldots$ for singletons and uppercase letters $X, Y, \ldots$ for set valued variables. $\wp(X)$ denotes the power set of X.

Abstracting away from the actual syntactic details, we can turn any mso logic into a constraint language. $\mathcal{MSO}$ will be used as a placeholder for any logic based on WS2S.

Definition 3.1. *The 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* $\mathcal{I}$*.*

As usual, assignments are relative to an interpretation.

Definition 3.2. *Let* $\mathcal{T}$ *be a tree domain. An* $\mathcal{I}$*-assignment is a total function* $\alpha : \mathsf{VAR} \to \wp(\mathcal{T})$*. We call the set of all those mappings* $ASS^{\mathcal{I}}$*.*

We already presented the definition of an interpretation in example 2.2. We generalize it by the inclusion of a solution mapping $[\![-]\!]^{\mathcal{I}}$.

[12] $\mathcal{L}_{copy} = \{ww \mid w \text{ a word}\}$

Definition 3.3. *An interpretation $\mathcal{I}$ of the constraint language $\mathcal{MSO}$ is a pair $(\mathcal{M},[\![-]\!]^{\mathcal{I}})$ consisting of a nonempty model $\mathcal{M}$ and solution mappings $[\![-]\!]^{\mathcal{I}} : \mathsf{CON} \longrightarrow \wp(ASS^{\mathcal{I}})$ such that:*

1. *$[\![-]\!]^{\mathcal{I}}$ is a function mapping every constraint ϕ to a set $[\![\phi]\!]^{\mathcal{I}}$ of $\mathcal{I}$-assignments. We call that set the solutions of ϕ in $\mathcal{I}$.*
2. *a constraint ϕ constrains only $\mathsf{V}(\phi)$, i.e., if $\alpha \in [\![\phi]\!]^{\mathcal{I}}$ and β is an $\mathcal{I}$-assignment witch agrees with α on $\mathsf{V}(\phi)$, then $\beta \in [\![\phi]\!]^{\mathcal{I}}$.*

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

Definition 3.4. *The function $\mathsf{V} : \mathsf{CON} \rightarrow \wp(\mathsf{VAR})$ is defined as (R ranges over the defined relations, $*$ over binary connectives, $\mathcal{Q}$ over quantifiers):*

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

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 3.5. *Let $\mathcal{I} = (\mathcal{M},[\![-]\!]^{\mathcal{I}})$ with $\mathcal{M} = \langle \mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{L}\rangle$ a model for $\mathcal{L}^2$ be an interpretation and let $\alpha_{\frac{X}{A}}$ 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 $\boldsymbol{X}$ being VAR, CON, V and INT as defined. The denotation function $[\![-]\!]^{\mathcal{I}}$ is defined as follows:*

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

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 respectively.

Renamings and variants are relevant in the proof of the soundness and completeness of the operational semantics in section 3.2.3. The following definition of a renaming is based on the semantic 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 a bijection $\rho : \mathsf{VAR} \to \mathsf{VAR}$ that is the identity except for finitely many exceptions. If ρ is a renaming, we call a constraint ϕ' a ρ-variant of ϕ if 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 $\mathsf{V}(\phi') = \rho(\mathsf{V}(\phi))$ 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 is called a variant 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 3.6. *Let $\phi, \psi, \theta \in \mathsf{CON}$. A constraint language is*

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

Lemma 3.7. *$\mathcal{L}^2$ is a decidable constraint language closed under renaming and intersection.*

In the following we blur the distinction between $\mathcal{L}^2$ and $\mathcal{MSO}$, but we always assume that CON is closed under 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 in a (new) language for two reasons. Firstly we can now give an operational semantics to the inductive definitions of $\mathcal{R}(\mathcal{MSO})$ 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 3.8. *We extend $\mathcal{MSO}$ with a decidable set of relations $\mathcal{R}$ and a corresponding arity function to a relational constraint language $\mathcal{R}(\mathcal{MSO})$ on the same set of variables $\boldsymbol{X}$ by building the $\mathcal{R}(\mathcal{MSO})$-constraints $\mathcal{R}(\mathsf{CON})$ with the function $\mathcal{V}$ computing the free variables and $\mathcal{R}(\mathcal{MSO})$-interpretations $\mathcal{R}(\mathsf{INT})$ from $\mathcal{MSO}$.*

Definition 3.9. *The $\mathcal{R}(\mathcal{MSO})$-constraints $\mathcal{R}(\mathsf{CON})$ are defined inductively such that*

1. *$\phi \in \mathcal{R}(\mathsf{CON})$ for all $\phi \in \mathcal{MSO}$.*
2. *$r(X_1, \ldots, X_n) \in \mathcal{R}(\mathsf{CON})$ for all tuples of pairwise distinct variables $X_1, \ldots, X_n$ and relation symbols r of rank n. We call these atoms.*
3. *if F and G are constraints in $\mathcal{R}(\mathsf{CON})$, then so are $F \,\&\, G$ (this includes the empty conjunction $\emptyset$) and $F \rightarrow G$.*
4. *Nothing else is in $\mathcal{R}(\mathsf{CON})$.*

Definition 3.10. *The set of free variables $\mathcal{V}$ of an $\mathcal{R}(\mathcal{MSO})$-constraint is again defined inductively:*

1. *$\mathcal{V}(\phi) := \mathsf{V}((\phi)$ if ϕ is an $\mathcal{MSO}$-constraint*
2. *$\mathcal{V}(r(X_1, \ldots, X_n)) := \{X_1, \ldots, X_n\}$*
3. *$\mathcal{V}(\emptyset) := \emptyset$*
4. *$\mathcal{V}(F \,\&\, G) := \mathcal{V}(F) \cup \mathcal{V}(G)$*
5. *$\mathcal{V}(F \rightarrow G) := \mathcal{V}(F) \cup \mathcal{V}(G)$*

Definition 3.11. *An $\mathcal{R}(\mathcal{MSO})$-interpretation is the extension $\mathcal{A}$ of the $\mathcal{MSO}$-interpretation $\mathcal{I}$ 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 extending the solution mapping as follows:*

1. *$[\![\phi]\!]^{\mathcal{A}} := [\![\phi]\!]^{\mathcal{I}}$ if ϕ is an $\mathcal{MSO}$-constraint*
2. *$[\![r(X_1, \ldots, X_n)]\!]^{\mathcal{A}} := \{\alpha \in ASS^{\mathcal{A}} \mid \alpha(X_1, \ldots, X_n) \in r^{\mathcal{A}}\}$*
3. *$[\![\emptyset]\!]^{\mathcal{A}} := ASS^{\mathcal{A}}$*
4. *$[\![F \,\&\, G]\!]^{\mathcal{A}} := [\![F]\!]^{\mathcal{A}} \cap [\![G]\!]^{\mathcal{A}}$*
5. *$[\![F \rightarrow G]\!]^{\mathcal{A}} := (ASS^{\mathcal{A}} - [\![F]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}}$*

We call the set of all $\mathcal{A}$-interpretations $\mathcal{R}(\mathsf{INT})$.

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. This has been done for logic programs in Lloyd (1984) by defining 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 3.12. *A definite clause specification $\mathcal{S}$ is a set of definite clauses of the form*

$$B \leftarrow \phi \,\&\, A_1 \,\&\, \ldots \,\&\, A_n$$

where B and the A_i are 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. 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 3.13. *Given a definite clause specification $\mathcal{S}$, a model of $\mathcal{S}$ is an $\mathcal{R}(\mathcal{MSO})$ interpretation $\mathcal{A}$ such that every variable assignment α is a solution for every definite clause $\delta \in \mathcal{S}$.*

Höhfeld and Smolka prove that indeed there exists a least model $\mathcal{A}$ for every program which can be constructed from any $\mathcal{MSO}$-interpretation $\mathcal{I}$ as a limit of the application of the clauses.

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., $[\![A]\!]^{\mathcal{A}_i} \subseteq [\![A]\!]^{\mathcal{A}_j}$ if $\mathcal{A}_i \subseteq \mathcal{A}_j$. Intersection and union of such interpretations are again defined via the relations. Note that a set of interpretations extending $\mathcal{I}$ forms a complete lattice.

Theorem 3.14 (H&S theo. 4.4)**.** *Let $\mathcal{S}$ be a definite clause specification in $\mathcal{R}(\mathcal{MSO})$ and $\mathcal{I}$ an $\mathcal{MSO}$-interpretation. Then we can inductively define a chain of $\mathcal{R}(\mathcal{MSO})$-interpretations $\mathcal{A}_i$ extending $\mathcal{I}$ by*

1. $r^{\mathcal{A}_0} := \emptyset$
2. $r^{\mathcal{A}_{i+1}} := \{\alpha(\vec{X}) \mid (r(\vec{X}) \leftarrow G) \in \mathcal{S}$ *and* $\alpha \in [\![G]\!]^{\mathcal{A}_i}\}$
3. $r^{\mathcal{A}} := \bigcup_{i \geq 0} r^{\mathcal{A}_i}$

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

We have already stated informally what we expect the answer 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 $\phi \rightarrow G$ is valid in every model of $\mathcal{S}$.

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 finite lists in $\mathcal{R}(\mathcal{MSO})$. Notational conventions are that the $\mathcal{MSO}$-constraints are enclosed in curly brackets, lowercase letters denote node variables, uppercase letters or words starting with them set variables and relations both in $\mathcal{MSO}$ and $\mathcal{R}(\mathcal{MSO})$ are lowercase 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 linear precedence. With the constraints *min* and *max* we can pick out the least and the greatest element in this set. The empty list is just an empty set, and a nonempty list has a minimal and a maximal element, and all the elements which are neither are totally ordered by $\prec$. The head of such a list is just the smallest element, the tail all the remaining elements. The exact $\mathcal{MSO}$-definitions can be found in in equations (8) to (14).

$$min(x, X) \stackrel{def}{\Longleftrightarrow} x \in X \wedge (\forall y)[(y \in X \wedge y \not\approx x) \Rightarrow x \prec y] \tag{8}$$

$$max(x, X) \stackrel{def}{\Longleftrightarrow} x \in X \wedge (\forall y)[(y \in X \wedge y \not\approx x) \Rightarrow y \prec x] \tag{9}$$

$$elist(X) \stackrel{def}{\Longleftrightarrow} \neg(\exists x)[x \in X] \tag{10}$$

$$\begin{aligned} nelist(X) \stackrel{def}{\Longleftrightarrow} & (\exists x, y)[min(x, X) \wedge max(y, X)] \wedge \\ & (\forall x, y)[(\neg min(x, X) \wedge \neg min(y, X) \\ & \qquad \wedge \neg max(x, X) \wedge \neg max(y, X) \\ & \qquad \wedge x \in X \wedge y \in X) \Rightarrow \\ & \quad (x \prec y \vee y \prec x)] \end{aligned} \tag{11}$$

$$list(X) \stackrel{def}{\Longleftrightarrow} elist(X) \vee nelist(X) \tag{12}$$

$$hd(X, x) \stackrel{def}{\Longleftrightarrow} min(x, X) \tag{13}$$

$$tl(X, Y) \stackrel{def}{\Longleftrightarrow} Y \subseteq X \wedge (\forall x)[(\neg min(x, X) \wedge x \in X) \Rightarrow x \in Y] \tag{14}$$

The resulting $\mathcal{R}(\mathcal{MSO})$ definition for *append* is almost like the corresponding definition in standard logic programming. We just have to make the desired unifications between the variables explicit in the constraints, see figure 7.

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 $\phi \in \mathsf{CON}$.

$$\begin{array}{lll} append(X,Y,Z) & \leftarrow & \\ & \{elist(X) \wedge Y \approx Z\} & \\ append(X,Y,Z) & \leftarrow & \\ & \{hd(X,x) \wedge hd(Z,x) \wedge tl(X,U) \wedge tl(Z,V)\} & \& \\ & append(U,Y,V) & \end{array}$$

FIGURE 7. The definition of *append* in $\mathcal{R}(\mathcal{MSO})$

Intuitively, we can substitute the body of a clause if the head represents a renamed variant of the 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. Informally, 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.

Definition 3.15. *Let ϕ, ψ, θ be satisfiable $\mathcal{MSO}$-constraints and $\mathcal{A}$ a $\mathcal{R}(\mathcal{MSO})$-interpretation. $\mathcal{S}$-goal reduction ($\xrightarrow{r}_{\mathcal{S}}$) is defined to be*

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

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 $\mathcal{V}(G) \cap \mathcal{V}(F) \subseteq \mathcal{V}(A)$.

Since the language is closed under intersection, we can treat the conjunction of the two constraints $\phi \,\&\, \psi$ as a single new constraint θ.

The 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. We need some technical preliminaries before we can state the theorem. First we generalize the notion of a solution to a set of constraints. If Φ is a set of constraints and $\mathcal{I}$ an interpretation, the solutions of Φ in $\mathcal{I}$ are defined to be $[\![\Phi]\!]^{\mathcal{I}} := \bigcup_{\phi \in \Phi} [\![\phi]\!]^{\mathcal{I}}$. Note that this interprets the constraints disjunctively, as opposed to conjunctively in the definition of the denotation function. So, solutions for a set of constraints are the union of the sets of assignment functions of the elements of the set.

Theorem 3.16 (H&S prop. 5.1). *If $G \xrightarrow{r}_{\mathcal{S}} F$, then $[\![F]\!]^{\mathcal{A}} \subseteq [\![G]\!]^{\mathcal{A}}$ for every model $\mathcal{A}$ of $\mathcal{S}$.*

Lemma 3.17 (Soundness). *If $G \xrightarrow{r}{}^{*}_{\mathcal{S}} \phi$, then ϕ is an $\mathcal{S}$-answer.*

Proof. ϕ is an $\mathcal{S}$-answer just in case $(ASS^{\mathcal{A}} - [\![\phi]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}} = ASS^{\mathcal{A}}$ for all $\mathcal{A}$ which are a model of $\mathcal{S}$. This is true by theorem 3.16 since $[\![\phi]\!]^{\mathcal{A}} \subseteq [\![G]\!]^{\mathcal{A}}$ holds. □

We can now proceed to show the completeness of the reduction rule. Since the proof relies on a rather involved complexity measure, we rely on the reader to supply his or her own intuition on what a (procedural) complexity reduction means.[13] Completeness in this sense can be paraphrased as follows. For every assignment function in the denotation, given an atom, a reduction is possible and it will still be in the denotation after the reduction.

Theorem 3.18 (H&S theo. 5.2)**.** *Let G be goal, A an atom in G, $\mathcal{A}$ be a minimal model of $\mathcal{S}$, and $\alpha \in [\![G]\!]^{\mathcal{A}}$. Then there exists a clause C in $\mathcal{S}$ such that*

1. *$\mathcal{S}$-goal reduction of G on A using a variant of C is possible,*
2. *if G_1 is obtained from G by $\mathcal{S}$-goal reduction on A using a variant of C then $\alpha \in [\![G_1]\!]^{\mathcal{A}}$ and the complexity of the goal is reduced.*

The preceding theorem actually stated what happened when we commit to a particular choice (don't care nondeterminism). But in general we are not interested which atoms and clauses are used in achieving the result (don't know nondeterminism). This is reflected in the next corollary.

Corollary 3.19 (Completeness, H&S cor. 5.3)**.** *Let $\mathcal{A}$ be a minimal model of $\mathcal{S}$, G a goal, A an atom in G and $\alpha \in [\![G]\!]^{\mathcal{A}}$. Then there exists an $\mathcal{S}$-answer ϕ of G such that $G \xrightarrow{r}{}^{*}_{\mathcal{S}} \phi$ and $\alpha \in [\![\phi]\!]^{\mathcal{A}}$.*

Intuitively, if there exists a solution to a goal, then we can find a constraint representing it via goal reduction.

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 the definition of the relational extension. 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 $P = \{(l_i, r_i) \mid i = 1 \ldots m\}$ with l_i and r_i words over Σ. A solution for such a system S is a sequence of indices

[13]The formal details are in Höhfeld and Smolka (1988).

$1 \leq i_1, \ldots, i_n \leq m$ iff $l_{i_1} \cdots l_{i_m} = r_{i_1} \cdots r_{i_m}$. It is undecidable whether there exists a solution for a given system (if the alphabet has more than two symbols).

To be able to present the encoding into $\mathcal{R}(\mathcal{MSO})$ we have to sugar our notation. Instead of using the constraint part to deal with our decomposition of linearly ordered sets or better lists, we use the "." to encode list concatenation ($(X.Y)$ for $hd(Z, X) \wedge tl(Z, Y)$) directly in the atom. 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 equations (8) to (14) and in figure 7 to lists of lists.

The encoding adapts the one from Hanschke and Würtz (1993). Words are represented as lists, e.g., $a_1 a_2 a_3 = (a_1.a_2.a_3)$. 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 figure 8. Considering the query in the first line, this

$$\leftarrow pcp((l_1).((l_2).(\cdots.(l_m))\ldots),\ (r_1).((r_2).(\cdots.(r_m))\ldots))$$

$$\begin{aligned}
&pcp((L.X),\ (L.Y)) \leftarrow \\
&pcp((L.X),\ (R.Y)) \leftarrow \\
&\qquad\qquad append(X,\ (l_1.L).\cdots.(l_m.L),\ X1) \\
&\qquad\qquad append(X,\ (r_1.L).\cdots.(r_m.L),\ Y1) \\
&\qquad\qquad pcp(X1,\ Y1)
\end{aligned}$$

FIGURE 8. The PCP predicate

encodes all sequences of length 1, in the next step we use the sequence 1 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 figure 8 has a SLD refutation iff a given Post correspondence system has a solution. Therefore we can state the next lemma.

Lemma 3.20. *Satisfiability in the relational extension $\mathcal{R}(\mathcal{MSO})$ is undecidable.*

Furthermore, we can achieve the same by only considering a smallest binary program (see Hanschke 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. But, as yet, we do not have a proof for that.

So we can indeed define problematic relations. 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 trees. 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 equations (8) to (14) or the definition of an arbitrary fact like the base case of *append* in figure 7. Rogers provides the notions of first and second order inductive definability to designate those relations which are recursively definable without leaving $\mathcal{MSO}$. His refers to Moschovakis (1974) as the source of the definitions. The following paragraphs relate the model-theoretic notions with the operational ones we used in theorem 3.14.

We are going to define the relations recursively via an operator which computes all the tuples of sets contained in it. An inductive definition is not satisfiable by a single relation but by a whole set of them. These are the fixed points of the operator. Since we limit ourselves to positive occurrences of the relation symbol, the operator is monotone and a least fixed point exists. 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. The name of the predicate in the definite clause corresponds to a second order variable. In the following all the X_i and X are meant to be monadic second order variables. $\mathbf{X}$ is a second order variable with monadic arguments. Recall that $\vec{X}$ and $\vec{x}$ denote tuples of distinct variables.

Definition 3.21. *Let ϕ be a formula with $\mathcal{V}(\phi) = \{\vec{X}, \boldsymbol{X}\}$ such that $\boldsymbol{X}$ 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*

$$\Gamma_\phi(r) := \{\alpha(\vec{X}) \mid \mathcal{M} \models \phi[\alpha] \text{ and } \alpha(\boldsymbol{X}) = r\}.$$

It is a fact along the lines of theorem 3.14 that this operator has a fixpoint. If a formula exists such that there is a least fixed point for an operator computing the relation, we call the relation (positive) inductive second order definable. Recall from definition 3.11 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. We have shown above that inductive second order definable relations might be undecidable.

But if we limit ourselves to just singleton variables, we stay within the power of $\mathcal{MSO}$.

Definition 3.22. *Let ϕ be a formula with $\mathcal{V}(\phi) = \{\vec{x}, X\}$ such that X has arity n and occurs only positively. Then we define the operator $\Lambda_\phi : \wp(\mathcal{T}^n) \to \wp(\mathcal{T}^n)$ as*

$$\Lambda_\phi(r) := \{\alpha(\vec{x}) \mid \mathcal{M} \models \phi[\alpha] \text{ and } \alpha(X) = r\}.$$

The free variable X stands for the (monadic) relation on nodes we are defining. Again, if a formula exists such that there is a least fixed point for such the relation, we call it (positive) inductive first order definable. Those relations are definable since they can be explicitly second order defined with the following formula. In fact, this seems to be a variant of Beth's theorem.[14]

$$r(\vec{y}) \overset{def}{\Longleftrightarrow} (\forall X)[(\forall \vec{x})[\phi \Rightarrow \vec{x} \in X] \Rightarrow \vec{y} \in X] \tag{15}$$

Now we know one way to ensure the decidability of the relations we define with our programs. We have to alter the definitions 3.9 – 3.11 such that only singleton variables are allowed to appear in the atoms in all directly or indirectly recursive predicates. The new relations we introduce are then limited to $r^{\mathcal{A}} \subseteq \mathcal{T} \times \cdots \times \mathcal{T}$. Unfortunately this gives us only the reintroduction of the notion of derivation, but not more expressive power. We have to leave this point open for further research. But results from logic programming[15] and higher order unification[16] suggest that we might be able to find an interesting subclass.

[14] Beth's theorem (as in Makowsky (1992)): Let Ω be a set of first order formulas over a signature Σ and P be an n-ary relation symbol not in Σ. We say a formula $\phi(P)$ over $\Sigma \cup \{P\}$ defines P *implicitly* using Ω, if in each model $\mathcal{M}$ of Ω there is at most one interpretation of P. We say that the predicate implicitly defined by ϕ using Ω has an *explicit* definition for which there is a formula $\psi(\vec{x})$ over Σ such that

$$\Omega \cup \phi(P) \models \forall \vec{x}[\psi(\vec{x}) \Longleftrightarrow P(\vec{x})].$$

The theorem then says that for Ω a set of first-order formulas and for $\phi(P)$ an implicit definition of P using Ω, there exists an explicit definition of P using Ω.

[15] The results are related to properties such as groundness or linearity of the involved programs and queries, see Devienne et al. (1994).

[16] Levy's result states that linear unification is decidable (Levy; 1996).

3.2.5. *The Interpreter.* In this section on the interpreter we specify a search strategy, a constraint solver and discuss 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 (Frühwirth et al.; 1993). 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 knowledge about the variables very efficiently as tree automata
- they can be used incrementally since the nature of the proof allows us to view a tree automaton as a constraint store where the conjunction with a new constraint yields a new store;
- emptiness, i.e., satisfiability, is trivially decidable after automata construction;
- we can even create a library of constraints to avoid recomputation and view this as an extension of the signature of $\mathcal{MSO}$;
- we can define what it means for an $\mathcal{MSO}$ constraint to be normal.

To make some of the statements more precise we present the working of the desired interpreter in more detail. First of all, we single out a particular subclass of formulas.

Definition 3.23. *A normal $\mathcal{MSO}$-constraint is a formula whose corresponding minimized tree automaton recognizes a nonempty language.*

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.

Lemma 3.24. *For every pair of normal $\mathcal{MSO}$-constraints we can compute a new $\mathcal{MSO}$-constraint and decide whether it is normal.*

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 a definite clause specification such that the 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.

Lemma 3.25. *Every definite clause specification can be converted to a normal definite clause specification.*

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. By this testing of the applicability of the clauses, we achieve something like Prolog's term unification to find matching clauses.

The outline of the interpreter 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 a type discipline. This allows the typing of variables with $\mathcal{MSO}$ constraints as it is done 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 it subsumes 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.

The third possible aspect which we do not address is the use of consistency techniques like constraint propagation. It involves narrowing the domain of values the variables can take by inferring the information between constraints. For more information on constraint propagation, see Le Provost and Wallace (1993).

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). Similar techniques, called guards or residuation have also been proposed by Smolka (1993). Instead of only *telling* constraints, i.e., adding them to the constraint store, we can also *ask* whether a constraint is entailed by the constraint store. This amounts to checking the satisfiability of the constraint store conjoined with the negation of the constraint. Obviously this is an extension that the constraint solver can handle. 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 allows the framework 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. We think that this is a major point for further research.

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 (1991), Minnen and Gerdemann (1993)). Another technique involves the addition of new predicates to allow bottom up execution of the program, called magic compilation (Beeri and Ramakrishnan (1987), 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 the case of P&P-based theories formulated in $\mathcal{MSO}$, the linguistic structures which are subject to the constraints of the grammar are tree like sets of nodes. 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. 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 the 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 $\mathcal{R}(\mathcal{MSO})$. In $\mathcal{R}(\mathcal{MSO})$ we can specify a parsing program.

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. We provide examples for both the limited and the full extension of $\mathcal{MSO}$. Note that an explicit definition corresponds directly to a fact.

4.1. **Pure Parsing.** As discussed in section 2.3, one could use a complete formalization of the grammar in $\mathcal{MSO}$for parsing directly via theorem proving. Assume for example that $gram(X)$ 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.

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 another$\mathcal{MSO}$-constraint identifying nodes with an empty phonology (e_phon).

$$(16) \qquad yield(S,T) \stackrel{def}{\Longleftrightarrow} (\forall x)[(x \in T \wedge \neg\exists y[x \lhd y] \wedge \neg e\_phon(x)) \Rightarrow \\ x \in S] \wedge \\ \neg(\exists z)[z \in S \wedge z \notin T]$$

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.

$$(17) \qquad gram(In,X) \stackrel{def}{\Longleftrightarrow} yield(In,Tree) \wedge gram(Tree)$$

The definite clause can be found in figure 9.

$$\begin{array}{ll} parse(In,X) & \leftarrow \\ & \{gram(In,X)\} \end{array}$$

FIGURE 9. Pure $\mathcal{MSO}$ *parse*

Treat the input string In as a set of nodes labeled with the words as monadic properties and linearly ordered by precedence. Additionally, we have to exclude 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:

(18)
$$\begin{aligned}
&\leftarrow parse(In, Tree)\,\&\\
&\quad \{(\exists x, y, z)[x, y, z \in In \wedge Jan(x) \wedge Sees(y) \wedge Lena(z) \wedge\\
&\qquad\quad x \prec y \prec z \wedge\\
&\qquad\quad (\forall w)[(w \in IN \wedge w \not\approx x \wedge w \not\approx y \wedge w \not\approx z) \Rightarrow\\
&\qquad\qquad e_phon(w)]]\}
\end{aligned}$$

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 section 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 $\mathcal{R}(\mathcal{MSO})$. His *parse* relation holds of a string and a tree in case it obeys the *xbar* theory, the *ecp* and the string is the *yield* of the tree, see figure 10. We assume that all our mso variables are constraint to be trees.

$$\begin{array}{lll}
parse(Words, Tree) & \leftarrow & \\
 & \{list(Words)\} & \& \\
 & xbar(Tree) & \& \\
 & ecp(Tree) & \& \\
 & yield(Words, Tree) &
\end{array}$$

FIGURE 10. *parse* from Johnson (1995)

As an example, we will work through the definition of the *xbar* predicate. First, we have to define some new $\mathcal{MSO}$-constraints though.

(19)
$$root(X, x) \overset{def}{\iff} (x \in X) \wedge (\forall y)[y \in X \Rightarrow x \triangleleft^* y]$$

The definition of a *root* is relative to a specific set, x is the root of X.

$$dtrs(x, y, z) \overset{def}{\Longleftrightarrow} x \lhd y \wedge x \lhd z \wedge y \prec z \tag{20}$$

The definition of daughters ($dtrs$) 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.

$$subtree(X, y, Y) \overset{def}{\Longleftrightarrow} (\forall x)[(x \in X \wedge y \lhd^* x) \Rightarrow x \in Y] \tag{21}$$

$$\begin{aligned} subtrees(X, x, Y, y, Z, z) \overset{def}{\Longleftrightarrow}\ & root(X, x) \wedge dtrs(x, y, z) \wedge \\ & subtree(X, y, Y) \wedge subtree(X, z, Z) \end{aligned} \tag{22}$$

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.

Coming back to an issue raised in section 2.3 in equation (5), let us note that we can avoid having to compile the lexicon as one constraint by having facts for each lexical entry separately (see figure 11). Naturally this is highly nondeterministic and shifts the burden to the clp interpreter. But at least the approach makes the problem feasible.

$$\begin{array}{lcl} lexicon(x) & \longleftarrow & \{Sees(x) \wedge V(x) \wedge 3rd(x) \wedge \ldots\} \\ lexicon(x) & \longleftarrow & \{Jan(x) \wedge N(x) \wedge Mask(x) \wedge \ldots\} \\ lexicon(x) & \longleftarrow & \{Lena(x) \wedge N(x) \wedge Fem(x) \wedge \ldots\} \\ & \vdots & \end{array}$$

FIGURE 11. An $\mathcal{R}(\mathcal{MSO})$-definition of the lexicon

The first clause of *xbar* in figure 12 allows empty nodes to appear freely. The distribution of those nodes has to be constrained with the *ecp*. The second clause allows nodes which appear in the lexicon while the third 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 figure 13).

$$
\begin{array}{lll}
xbar(X) & \leftarrow & \\
 & \{elist(X)\} & \\[1ex]
xbar(X) & \leftarrow & \\
 & \{sing(X)\} & \& \\
 & lexicon(X) & \\[1ex]
xbar(X) & \leftarrow & \\
 & \{subtrees(X,x,Y,y,Z,z)\} & \& \\
 & local\_xbar(x,y,z) & \& \\
 & xbar(Y) & \& \\
 & xbar(Z) &
\end{array}
$$

FIGURE 12. *xbar* following Johnson (1995)

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

FIGURE 13. *local_xbar* following Johnson (1995)

The *ecp* deals with the distribution of the empty nodes in the tree.[17] The predicate *yield* was already defined in equation (16). 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.

The general approach of constraining variables to particular values allows the specification of data types. We can define them in the constraint language without having to include a type discipline. We attach additional conditions to every variable appearing in a clause. In the case of $\mathcal{MSO}$, we already discussed following Rogers and Rizzi (and others) in their assumption that the basic entities the theory deals with are not longer nodes, but categories. Recall the definitions and discussion from section 2.3. Now we can constrain our variables in $\mathcal{R}(\mathcal{MSO})$ to range

[17] We leave out the definition of the *ecp* predicate. It can be captured using similar techniques as in the preceding examples.

over categories so that they become the basic components of a theory. Again, details can be found in Rogers' thesis.

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 figure 14.[18] Again, we need some auxiliary definitions. We follow Fong in them, although we think they are pretty ad hoc.

$$(23) \qquad max(x) \stackrel{def}{\Longleftrightarrow} x \in NP \vee x \in VP \vee \ldots$$

max denotes those nodes which are labeled maximal.

$$(24) \qquad proj(x,y) \stackrel{def}{\Longleftrightarrow} (x \in NP \Leftrightarrow y \in N1) \vee (x \in VP \Leftrightarrow y \in V1) \vee \ldots$$

proj expresses that two nodes are projections of each other.

$$(25) \qquad dts(x,X) \stackrel{def}{\Longleftrightarrow} (\forall y)[x \lhd y \Leftrightarrow y \in X] \wedge list(X)$$

dts just collects all daughters of x in a list.

$$\begin{array}{lll} xp(x) & \leftarrow & \\ & \{max(x) \wedge proj(x,y) \wedge (\exists X)[dts(x,X) \wedge hd(X,y) \wedge tl(X,Z)]\} & \& \\ & x1(y) & \& \\ & specs(y,Z) & \end{array}$$

FIGURE 14. *xp* following Fong (1991)

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.

[18]We take here 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 three variables without problems, i.e., $proj\_max\_list(x,y,Z)$.

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 $\mathcal{MSO}$ 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 big difference to other theories as for example HPSG. This is most clearly seen in phrase structure based implementations of these theories, e.g., in Shieber (1992) or Carpenter (1993). We can see the strong locality requirements since all the feature constraints are coupled to the phrase structure rules. In HPSG itself they are bound to the ID schemata. 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 (Pollard and Sag; 1994). 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 mso 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. As presented in figure 15, we can almost use the pure $\mathcal{MSO}$ approach, only modified to work incrementally on a particular input. Remember that the resulting tree automaton can represent a number of possibilities to construct trees which are licensed by this particular branch of the search tree.

$$
\begin{array}{lll}
parse(In, X) & \leftarrow & \\
 & \{yield(In, X)\} & \& \\
 & \theta\text{-}theory(X) & \& \\
 & xbar(X) & \& \\
 & case(X) & \& \\
 & move\text{-}\alpha(X) & \& \\
 & lex(In) & \& \\
 & \vdots &
\end{array}
$$

FIGURE 15. A possible $\mathcal{R}(\mathcal{MSO})$-definition of *parse*

Now we can add either $\mathcal{MSO}$ relations which just encode these modules, or we can split them further where suitable. As an example, we present two definitions of *lex* in figure 16. In the first definition one can

$$
\begin{array}{lll}
lex(X) & \leftarrow & \\
 & \{elist(X)\} & \\
lex(X) & \leftarrow & \\
 & \{hd(X, x) \wedge tl(X, Y)\} & \& \\
 & lexicon(x) & \& \\
 & lex(Y) &
\end{array}
$$

or

$$
\begin{array}{ll}
lex(X) & \leftarrow \\
 & \{(\forall x)[x \in X \Rightarrow lexicon(x)]\}
\end{array}
$$

FIGURE 16. Two possible $\mathcal{R}(\mathcal{MSO})$-definition of *lex*

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. We use the list structure of the input to recurse on it. Whereas in the second approach we rely directly on the power of the constraint solver to handle the problem. Note that this second example presupposes the compilation of the whole lexicon into a variant of the *lexicon* definition we presented in equation (5). The $\mathcal{MSO}$-constraint *lexicon* is a disjunction of all the lexical items. This shows the advantage of guiding the search by the use of available data structures. What we did here for lists might be done similarly for the set of nodes representing the tree and the subsets representing the subtrees. By this we can recurse on the tree structure without loosing the ability to express nonlocal constraints. We always constrain the whole set by constraining its members.

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. But as discussed before, this has to be left for further work.

Another area of advantages of $\mathcal{MSO}$-constraints is in the specification of the existence of further information to be found. If we consider some phrase structure based approaches, we can now express the relation between a filler and its gap directly. Assume for simplicity that the definite clauses are just used to encode phrase structure rules. Suppose we have a 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 figure 17. The clause finds and recurses on the subtrees if there exists a trace in

$$
\begin{array}{lll}
s(X) & \leftarrow & \\
 & \{subtrees(X,Y,Z) \wedge (\exists z)[z \in Z \wedge z \in Trace \wedge feq(z,Y)]\} & \& \\
 & np_{wh}(Y) & \& \\
 & vp(Z) &
\end{array}
$$

FIGURE 17. Nonlocality in a phrase structure rule

Z such that it agrees on the relevant features with the extracted WH-phrase. The auxiliary definition identifying the two immediate subtrees is presented in equation (26).

$$
(26) \qquad subtrees(X,Y,Z) \stackrel{def}{\Longleftrightarrow} (\exists x,y,z)[subtrees(X,x,Y,y,Z,z)]
$$

An example for the use of quantification in $\mathcal{R}(\mathcal{MSO})$ 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 Principle. We leave the formal specification to the reader.

5. Summary

We presented in this paper the first step towards $\mathcal{MSO}$-based constraint logic programming. We applied the Höhfeld and Smolka scheme for the definition of the constraint logic programming language $\mathcal{R}(\mathcal{MSO})$ with relational extensions. This 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., MoSeL or 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 certainly 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 can investigate. The first is concerned with formal issues connected with $\mathcal{R}(\mathcal{MSO})$. If we want to extend our constraint language with negation it seems to be necessary to clarify which alternatives we have since $\mathcal{MSO}$ is not compact.[19] Another interesting point seems to be the generative capacity of the relational extension. Considering the discussion on definability in section 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 whether there still exists a denotational interpretation for a concurrent

[19] 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.

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 of 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.

References

Beeri, C. and Ramakrishnan, R. (1987). On the power of magic, *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.*

Biehl, M., Klarlund, N. and Rauhe, T. (1996). Algorithms for guided tree automata, *Proc. WIA '96*, LNCS, Springer-Verlag.

Blackburn, P. and Meyer-Viol, W. (1994). Linguistics, logic and finite trees, *IGLP Bulletin.*

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

Carlson, B. (1991). An approximation theory for constraint logic programs, Licentiate Thesis. Dep. of Computer Science, Uppsala University.

Carpenter, B. (1992). *The Logic of Typed Feature Structures*, Vol. 32 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

Carpenter, B. (1993). ALE the attribute logic engine, user's guide, *User manual*, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213.

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

Devienne, P., Lebg̀ue, 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. (1995). C-command and grammatical primitives, Presentation at the 18th GLOW Colloquium. University of Tromsø.

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E. and Wallace, M. (1993). Constraint logic programming: An informal introduction, *Technical Report ECRC-93-05*, ECRC. Appears in *Logic Programming Summer School*, Zürich, Springer Lecture Notes.

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

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

Hathout, N. (1992). *Theórie du Governement et du Liage et programmation logique avec contraintes: une application à l'analyse automatique du français*, PhD thesis, L'Université Paul Sabatier de Toulouse (Sciences), Toulouse.

Henriksen, J. G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T. and Sandhol, A. (1995). Mona: Monadic second-order logic in practice, *in* Brinksma, Cleaveland, Larsen, Margaria and Steffen (eds), *Tools and Algorithms for The Construction and Analysis of Systems: International Workshop*, TACAS '95, Selected Papers, Lecture Notes in Computer Science 1019, Springer, pp. 89–110.

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 Lassez, J.-L. (1986). Constraint logic programming, *Technical report*, IBM-Thomas J. Watson Research Center, Yorktown Heights, N.Y. 10598.

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

Kaplan, R. M. and Bresnan, J. (1983). Lexical-functional grammar: A formal system for grammatical representation, *in* J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, Mass, pp. 173–381.

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

King, P. J. (1989). *A Logical Formalism for Head–Driven Phrase Structure Grammar*, PhD thesis, University of Manchester.

Kracht, M. (1995). Syntactic codes and grammar refinement, *Journal of Logic, Language, and Information* **4**: 41–60.

Le Provost, T. and Wallace, M. (1993). Generalized constraint propagation over the clp scheme, *Journal of Logic Programming* **16**: 319–359.

Levy, J. (1996). Linear second order unification, *Proceedings of the Conference on Rewriting Techniques and Applications*, Springer.

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

Makowsky, J. A. (1992). *Model Theory and Computer Science: An Appetizer*, Vol. 1 of *Handbook of Logic in Computer Science*, Oxford Science Publications, pp. 763–814.

Minnen, G. and Gerdemann, D. D. (1993). Direct automated inversion of logic grammars, *SFB-Report 35*, SFB 340, University of Tübingen.

Morawietz, F. and Cornell, T. L. (1996). On the recognizability of relations over a tree definable in a monadic second order tree description language, *SFB-Report 85*, University of Tübingen, SFB 340, A8.

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

Niehren, J. and Podelski, A. (1992). Feature automata and recognizable sets of feature trees, *in* M.-C. Gaudel and J.-P. Jouannaud (eds), *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development*, Springer Verlag, LNCS 668, Orsay, France, pp. 356–375.

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 American Mathematical Society* **52**: 264–268.

Rabin, M. O. (1969). Decidability of second-order theories and automata on infinite trees, *Transactions of the American Mathematical Society* **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. (1994). *Studies in the Logic of Trees with Applications to Grammar Formalisms*, PhD thesis, University of Delaware, Dep. of Computer & Information Sciences, Newark, DE 19716. Published as

Technical Report No. 95-04.

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

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

Shieber, S. M. (1992). *Constraint–Based Grammar Formalisms*, MIT Press.

Smolka, G. (1993). Residuation and guarded rules for constraint logic programming, *in* F. Benhamou and A. Colmerauer (eds), *Constraint Logic Programming: Selected Research*, MIT Press, chapter 21, pp. 405–419.

Stabler, E. P. (1992). *The Logical Approach to Syntax: Foundations, Specifications and Implementations*, MIT Press, Cambridge, Massachusetts.

Strzalkowski, T. (1991). A general computational method for grammar inversion, *Proceedings of the Reversible Grammar Workshop*, Berkeley, CA.

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

Seminar für Sprachwissenschaft, SFB 340, A8, University of Tübingen, Wilhelmstr. 113, 72074 Tübingen, Germany

E-mail address: frank@sfs.nphil.uni-tuebingen.de