Bericht Nr. 85, Februar 1997

Arbeitspapiere des
Sonderforschungsbereichs 340

ISSN 0947-6954/97

SPRACHTHEORETISCHE GRUNDLAGEN FÜR DIE COMPUTERLINGUISTIK

Frank Morawietz & Tom Cornell :

On the Recognizability of Relations over a Tree definable in a Monadic Second Order Tree Description Language

Universität Stuttgart Universität Tübingen
IBM Deutschland Informationssysteme GmbH

Arbeitspapiere des SFB 340
Bericht Nr. 85, Februar 1997

ON THE RECOGNIZABILITY OF RELATIONS OVER A TREE DEFINABLE IN A MONADIC SECOND ORDER TREE DESCRIPTION LANGUAGE

FRANK MORAWIETZ AND TOM CORNELL

ABSTRACT. In this technical report we describe both the theoretical background and the implementation of the prototype MoSeL, a theorem prover based on the monadic second order tree description language $\mathcal{L}^2_{K,P}$ (Rogers 1994). We make use of the results by Thatcher and Wright (1968) and Doner (1970) proving that the (weak) monadic theory of two successor functions is decidable. They show that formulas in monadic second order logic on trees are satisfiable if and only if the satisfying variable assignments can be recognized by a tree automaton. In the paper we provide the necessary background on tree automata and their constructions, redo the decidability proof and describe its realization in the Prolog implementation MoSeL.

1. Introduction

Rogers (1994) presents $\mathcal{L}^2_{K,P}$, a monadic second-order language for describing properties of trees. Aside from being very powerful and flexible, $\mathcal{L}^2_{K,P}$ is proved to be decidable. The proof is rather indirect. First $\mathcal{L}^2_{K,P}$ is shown to be intertranslatable with the language of SnS, the theory of multiple successor functions. SnS is known, in turn, to be decidable, and this is shown by proving that any relation which can be defined in that language is *recognizable*, that is, definable by a tree automaton of a particular sort. In particular, it can be shown (see Rabin 1969, Thatcher and Wright 1968, Doner 1970) that from any formula in the language of

The research presented in this report was sponsored by Teilprojekt A8 of the Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft. We want to thank Hans-Peter Kolb, Uwe Mönnich and Jim Rogers for helpful discussion. Needless to say, all errors and infelicities that remain are our own.

ISSN 0947-6954/97

SnS one can construct a tree automaton which recognizes all and only trees whose labels encode instances of the defined relation.

In the first section of this report we redo the decidability proof for a language essentially like Rogers' $\mathcal{L}^2_{K,P}$ (or more accurately, $\mathcal{L}^2_{\emptyset,\emptyset}$) in a more direct manner, essentially skipping the translation to the language of SnS entirely. In fact this is quite simple to do: we merely need to show that the relations defined by the atomic formulas of $\mathcal{L}^2_{K,P}$ are recognizable, by presenting automata that recognize them. Then the inductive step from the proofs in Rabin (1969), Thatcher and Wright (1968), and Doner (1970) can be applied unchanged. From this we get the more general result that *any* monadic second order tree description language whose atomic formulas can be converted to automata will be decidable.

These are perhaps not very interesting results on their own. We present them here as a foundation for further investigations in the relation of automata to logic, which in turn is an interesting new angle on automatic theorem proving and ultimately, we hope, on constraint logic programming. As a step in this direction, in the second section we present MoSeL, a prototype for a tool for automated theorem proving over monadic second order tree description languages. The implementation is done in Prolog and follows the inductive decidability proof almost to the letter. Our program serves as a translator from logical formulae to tree automata. The interaction of minimization and conditions on reachability of states ensures that we generate non empty automata only if the formula is satisfiable, i.e., the existential closure of the formula represents a theorem of our language.

2. Theoretical Background

2.1. **The Language.** We focus in this report on one particular tree description language, the language $\mathcal{L}^2_{K,P}$ introduced in Rogers (1994). $\mathcal{L}^2_{K,P}$ is a monadic second order language built up from the following relation symbols.[1]

$$\begin{aligned} x \lhd^+ y &\quad \text{— } x \text{ properly dominates } y, \\ x \lhd^* y &\quad \text{— } x \text{ dominates } y, \\ x \lhd y &\quad \text{— } x \text{ immediately dominates } y, \\ x \prec y &\quad \text{— } x \text{ properly precedes } y. \end{aligned}$$

[1] The K and P subscripts refer to arbitrary sets of individual and predicate constant symbols, with which the language can be parameterized. They do not play a role in this paper since, without loss of generality, constants can be treated as free variables.

That is, atomic formulas are built up out of the relation symbols $\lhd^+$, $\lhd^*$, $\lhd$, and $\prec$ together with the logical constants $\approx$ and $\in$, the boolean connectives $\wedge$ and $\neg$, and the existential quantifier $\exists$. Quantifiers may bind variables which range over individuals and also over sets of individuals. We write individual variables as $x, y, z, x_1, x_2, \ldots$ and set variables as $X, Y, Z, X_1, X_2, \ldots$. As usual we make free use of the "abbreviations" $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$, and $(\forall x)\phi$ for $\neg(\neg\phi \wedge \neg\psi)$, $\neg(\phi \wedge \neg\psi)$, $\neg(\phi \wedge \neg\psi) \wedge \neg(\neg\phi \wedge \psi)$, and $\neg(\exists x)\neg\phi$, respectively.

As an example of the language in action, the following formula defines a relation which is, in fact, the $\lhd^*$ relation.

$$D(x, y) \iff_{\text{def}}$$
$$(\forall X)[(\forall x_1, x_2)[x_1 \in X \wedge x_2 \lhd x_1 \Rightarrow x_2 \in X] \Rightarrow (y \in X \Rightarrow x \in X)]$$

So we note in passing that our language is not in fact as spare as it could be: $\lhd^*$ can be defined in terms of $\lhd$, and then $\lhd^+$ can be defined in terms of $\lhd^*$ and (the negation of) $\approx$. This is only true for the second order language $\mathcal{L}^2_{K,P}$. Were we to restrict ourselves to the first order language $\mathcal{L}^1_{K,P}$ (as in, e.g., Backofen et al. 1995), then we would need these second-order definable relations back again as primitives.

2.2. Trees and Tree Automata.

2.2.1. *The Structure and Language of Multiple Successor Functions.* The fundamental results on which we build here (Rabin 1969, Doner 1970, Thatcher and Wright 1968) involve the monadic second order theory of multiple successors. These generalize Peano's approach to generating the natural numbers. Instead of looking at the structure $\langle N, 0, s\rangle$, wherein every natural number is either zero or some number of applications of s to zero, we look now at, for example, $\langle N, 0, r_1, r_2\rangle$. Thinking about the natural numbers, we could define $xr_1 = 2x + 1$ and $xr_2 = 2x + 2$, thereby generating the odd numbers by applications of r_1 and the evens—besides zero—by applications of r_2. (We write the successor functions *postfix* to make the generalization of automata on strings to automata on trees a little clearer.) Note in particular that each natural number is still the value of a unique term in the language of two successor functions; the numbers are this way arranged in a tree, rather than in the familiar number line (which is now seen to be just a degenerate tree). In general, for finite p, $xr_i = px + i, 1 \leq i \leq p$.

Obviously, we are interested far more in trees and less in number theory. But the logic of multiple successors gives us what is in many ways the simplest and most fundamental tree logic. Other logics, like Rogers' $\mathcal{L}^2_{K,P}$, may be more convenient for grammatical applications, but everything is grounded here.

For the rest of this paper we will be concerned with the structure of two successor functions, $\mathcal{N}_2 = \langle N_2, 0, r_1, r_2 \rangle$. The domain of this structure, N_2, is an infinite, strictly binary branching tree. As the *language* of $\mathcal{N}_2$, we take for our terms the set of strings $\{1,2\}^*$. The empty string ϵ is our sole constant and is taken to denote 0. The string $wi, w \in \{1,2\}^*, i \in \{1,2\}$ is interpreted as the application of r_i to the number denoted by w. So for example the string 21 denotes the value of $0r_2r_1$, that is, 5. For our purposes, though, it is easier to think of 21 as being the specification of a *path* through a tree which starts at the root, takes in the right daughter of the root, and ends at that node's left daughter.

The monadic second order theory of n successor functions, conventionally termed SnS, is the set of all sentences involving quantification over sets of individuals as well as individuals, based on the logical constants $\approx$ and $\in$ and the successor functions, which are true of $\mathcal{N}_n$, the structure of n successor functions. Doner (1970) and Thatcher and Wright (1968), as well as Rabin (1970), deal with the so-called *weak* interpretation of that theory, where set variables are constrained to vary over *finite* subsets of N_2. Rabin (1969), on the other hand, deals with the strong interpretation of the theory, in which set variables can range over arbitrary subsets of N_2, and indeed beyond $S2S$ to $S\omega S$, the structure of countably many successor functions. It turns out that even in these wild and wooly cases, the theory remains decidable.

We are primarily interested in linguistic applications where we can count on the sets of nodes we wish to isolate being finite. Accordingly, we focus on the weak interpretation of $S2S$, and generalizations only to systems of finitely many successors. For practical purposes, this means that the tree automata we construct run over finite trees; we do not need to consider here automata on infinite objects, though it is important to note that Rogers himself follows Rabin's work on the strong theories of SnS and $S\omega S$ most closely.

Note that because we are dealing with monadic second order logic for trees, we can always pick out finite subsets of N_2 as our trees of interest, simply relativizing all of our formulas to a chosen set variable whose members are all and only the nodes of our tree of choice. So the fact that N_2 is itself an infinite tree of fixed branching factor is not a problem for us.

2.2.2. *Tree Domains, Term Algebras and Automata.* The structure of n successor functions is the mathematical foundation of what are often called in computer science *tree domains*. A tree domain can be based on any linearly ordered set K; it is defined as a subset $T \subseteq K^*$ such that, for $u, w \in K^*, i, j \in K$,

1. $uw \in T \Rightarrow u \in T$ (*prefix closure*),
2. $uj \in T$ and $i < j \Rightarrow ui \in T$ (*closure under left sister*).

Note that the branching factor of a tree presented as a tree domain is not necessarily fixed.

A labeled tree can be thought of as a function $t : T \to \Sigma$, where T is a tree domain and Σ is an alphabet from which we draw our labels. This is the conception of a tree which is used in Doner (1970), for example. It stays close to the ultimate structure of interest, and so makes the model theory a little easier, but the automata theory is slightly harder to develop in this framework.

Thatcher and Wright (1968), on the other hand, adopt a more algebraic approach, which makes the automata theory more elegant, but requires some footwork when it comes time to focus on the model theory. In this algebraic approach (ordered, labeled) trees are defined as *terms* over a particular alphabet. Notationally, given an alphabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \cdots \Sigma_p$, where each Σ_i is a set of i-place function symbols, we refer to the set of Σ-terms as T_Σ. A tree t over Σ is any member of T_Σ.

Thatcher and Wright (1968) define an *automaton* (over finite trees) as an algebra $\mathcal{A} = (A, \alpha)$, with A the set of states and α a function mapping alphabet symbols—symbols from Σ—to functions over A. Because T_Σ is a free algebra and $\alpha \restriction \Sigma_0$ is a mapping from its generators to A, we can extend α to a homomorphism $\mathrm{h}_\mathcal{A}$ from arbitrary terms of T_Σ to A. For example, for $\lambda \in \Sigma_0$ and $f \in \Sigma_2$, we have the following.

$$\begin{aligned} \mathrm{h}_\mathcal{A}(\lambda) &= \alpha(\lambda) \\ \mathrm{h}_\mathcal{A}(ft_1t_2) &= \alpha(f)(\mathrm{h}_\mathcal{A}(t_1), \mathrm{h}_\mathcal{A}(t_2)) \end{aligned}$$

If we think of the $\alpha(\Sigma)$ as our direct transition functions, then $\mathrm{h}_\mathcal{A}$ is just the standard generalization of a direct transition function to inputs of arbitrary size. The process of computing the value of $\mathrm{h}_\mathcal{A}(t)$ is clearly the same process as that of determining what state to assign to t's root given an assignment of states to its leaves. That is, computing $\mathrm{h}_\mathcal{A}(t)$ is just running a bottom-up automaton on t.

Thatcher and Wright (1968) use an algebra $\mathcal{A}$ to pick out a set of trees by selecting a set of final states $A_F \subseteq A$ and defining the language of the automaton to be

$$\{t \mid \mathrm{h}_\mathcal{A}(t) \in A_F\}.$$

They refer to this set as the *behavior* of $\mathcal{A}$ with respect to A_F, and write

$$\mathrm{bh}_{\mathcal{A}}(A_F) = \{t \mid \mathrm{h}_{\mathcal{A}}(t) \in A_F\}.$$

We find that once one focuses on constructions on automata and operations on the forests they recognize it is easier to reserve the word *automaton* for the pairing of an underlying state-algebra (A, α) with a particular set of final states A_F. Here we will refer to the pair $\mathcal{A} = (A, \alpha)$ as the "underlying state-algebra," or more often just "algebra", and refer to an automaton $\mathfrak{A}$ as the triple $\mathfrak{A} = (A, \alpha, A_F)$.[2] Accordingly, we will speak only of the behavior of an automaton, the intended set of final states being understood. We will write $T(\mathfrak{A})$ for the set of trees recognized by $\mathfrak{A}$, the *language* $\mathrm{bh}_{\mathfrak{A}}(A_F)$ of $\mathfrak{A}$. However, since the assignment of a state to a tree is independent of any choice of final states, we will retain the notation $\mathrm{h}_{\mathcal{A}}(t)$, refering only to the underlying algebra, when it is useful to do so.

We can extend this approach easily to deal with *nondeterministic* automata, as follows. We allow our direct transition functions $\alpha(\Sigma)$ to map tuples of states to *sets* of states. Technically, since our transition functions are no longer operations, *strictu senso*, then our "algebras" are also no longer algebras. Nonetheless all of the essential concepts generalize to these "nondeterministic algebras". We can still extend the direct transition functions to a homomorphism so that given a nondeterministic algebra $\mathcal{A} = (A, \alpha)$, it still makes sense to denote the set of states assigned to a tree t as $\mathrm{h}_{\mathcal{A}}(t)$. Since $\mathrm{h}_{\mathcal{A}}(\mathrm{t})$ is now a set of states, the behavior of a nondeterministic automaton such as $\mathfrak{A}$ is defined as follows.

$$T(\mathfrak{A}) = \{t \mid \mathrm{h}_{\mathcal{A}}(t) \cap A_F \neq \emptyset\}$$

It is often convenient to think of α as a transition *relation* over A^{n+1}, rather than as a function from $A^n \to \wp A$.

Note that, just as for finite automata on strings, there is an equivalence (for bottom-up automata only!) between deterministic and nondeterministic automata. In particular, for every nondeterministic tree automaton we can effectively construct a deterministic automaton which recognizes the same set of trees.

2.2.3. *The Emptiness Problem for Tree Automata.* Suppose we are given an automaton $\mathfrak{A} = (A, \alpha, A_F)$, and we wish to know whether or not $T(\mathfrak{A})$ is empty. One way of deciding this is to systematically generate and test

[2] This is essentially the approach in Gécseg and Steinby (1984), except that they assume an alphabet distinguishing constants and variables, so their definition of an automaton includes a variable assignment function.

all of the Σ-trees of depth less than or equal to $|A|$. It is a standard theorem in the study of tree automata that the language of $\mathfrak{A}$ is non-empty if and only if this (finite) subset is non-empty.

We use another approach, which we adapt from Rabin (1970). (He is concerned with automata on infinite trees; we adapt his algorithm to finite trees.) We define inductively the set of nodes which can be reached from the initial states; if any final state can be reached from some initial states, then the language is non-empty.

(1)
$$Q^0 = \alpha(\Sigma_0)$$
$$Q^{n+1} = Q^n \cup \{a|(\exists\sigma \in \Sigma, a_1, a_2 \in Q^n)[\alpha(\sigma)(a_1, a_2) = a]\}$$
$$(\text{i.e., } Q^n \cup \alpha(\Sigma_2)(Q^n \times Q^n))$$
$$Q = \bigcup_{n<\omega} Q^n$$

Provably, the construction of Q reaches a fixpoint at the latest when $n = |A|$. At that point, the emptiness of the language of $\mathfrak{A}$ reduces to the question whether or not

$$A_F \cap Q = \emptyset.$$

In fact we can be even more hard-nosed and efficient: the minute a final state appears in one of the Q^n we know that $T(\mathfrak{A})$ is non-empty, and the minute $Q^{n+1} = Q^n$ we know we have decided the question, one way or the other. This algorithm is polynomial (cubic) in the number of states in the underlying state-algebra.

2.2.4. *Constructions on Tree Automata.* The *recognizable sets*, that is, the sets of trees that can be recognized by some tree automaton, enjoy numerous useful closure properties. The proofs of these properties are well-known and easily accessible (see, e.g., Gécseg and Steinby 1984), but we summarize here the key automaton-constructions which are used in these closure proofs as witnesses to the recognizability of some derived tree-set. These constructions form the foundations of our computational investigations into the use of tree automata in theorem provers for tree logics.

Boolean constructions. Suppose we are given an automaton $\mathfrak{A}$ which recognizes a set U of trees, and another automaton $\mathfrak{B}$ which recognizes a set V of trees. That is,

$$U = T(\mathfrak{A}),$$
$$V = T(\mathfrak{B}).$$

It is entirely straightforward to construct an automaton which recognizes $U \cap V$, $U \cup V$, or $U - V$. All of these constructions are based on the cross product of the two algebras $\mathcal{A}$ and $\mathcal{B}$, and differ only in the proper choice of final states.

$$\begin{aligned} \mathcal{A} \times \mathcal{B} &= (A \times B, \gamma) \\ \gamma(\lambda) &= (\alpha(\lambda), \beta(\lambda)) \\ \gamma(f)((a_1, b_1), (a_2, b_2)) &= (\alpha(f)(a_1, a_2), \beta(f)(b_1, b_2)) \end{aligned} \tag{2}$$

Then

$$U \cap V = \mathrm{bh}_{\mathcal{A}\times\mathcal{B}}(A_F \times B_F) \tag{3}$$

$$U \cup V = \mathrm{bh}_{\mathcal{A}\times\mathcal{B}}((A_F \times B) \cup (A \times B_F)) \tag{4}$$

$$U - V = \mathrm{bh}_{\mathcal{A}\times\mathcal{B}}(A_F \times (B - B_F)) \tag{5}$$

Note that in the particular case where $U = T_\Sigma$ the difference $U - V$ (i.e., the complement of V) can be handled without the cross-product construction. The relevant automaton is just $(B, \beta, B - B_F)$.

Note further that in the case of union, if the state-sets of the input automata are disjoint then a simple union construction on the input automata suffices. That is, $U \cup V$ can be recognized by an automaton $\mathfrak{C} = (A \cup B, \alpha \cup \beta, A_F \cup B_F)$. Since the state sets (and hence the domains of α and β) are disjoint, $\alpha \cup \beta$ is certain to be well-defined. This construction corresponds directly to the notion that any tree in $U \cup V$ can be recognized by running $\mathfrak{A}$ or else running $\mathfrak{B}$. The cross-product construction corresponds to running two automata simultaneously and in the essentially disjunctive union operation there is clearly no need to do this.

Projection. The boolean constructions assume that U and V are both subsets of T_Σ, for some fixed Σ. We need also a construction which takes us from a set of trees in one alphabet Σ to another, call it Ω. Assume that we are given a mapping $P : \Sigma \to \Omega$; we extend it to a mapping $\bar{P}$ from $T_\Sigma \to T_\Omega$ in the natural way. That is:

$$\begin{aligned} \bar{P}(\lambda) &= P(\lambda) \\ \bar{P}(ft_1t_2) &= P(f)\bar{P}(t_1)\bar{P}(t_2) \end{aligned}$$

Now suppose that $U \subseteq T_\Sigma$, and that we are given $\mathfrak{A} = (A, \alpha, A_F)$ such that $U = T(\mathfrak{A})$. We construct a nondeterministic automaton $\mathfrak{B}$ working on the new "projected" alphabet as presented in (6). But first we have to introduce some notation. Let F be a set of functions. Then $F(x, y)$ is meant to denote $\{z \mid (\exists g \in F)[g(x,y) = z]\}$.

$$\begin{aligned} \text{(6)} \qquad \mathfrak{B} &= (A, \beta, A_F) \\ \beta(\lambda) &= \alpha(P^{-1}(\lambda)) \\ \beta(f)(a_1, a_2) &= \alpha(P^{-1}(f))(a_1, a_2) \end{aligned}$$

Then

$$\bar{P}(U) = T(\mathfrak{B})$$

So, considering the remark on notation above, $\alpha(P^{-1}(f))(a_1, a_2)$ denotes the set of a such that, for some g with $P(g) = f$, the following equation holds: $\alpha(g)(a_1, a_2) = a$.

Inverse projection. We can also define an inverse-projection construction. Suppose we are again given a projection $P : \Sigma \to \Omega$ and now a set $V \subseteq T_\Omega$, and V is known to be recognizable. That is, for some Ω-automaton $\mathfrak{A} = (A, \alpha, A_F)$, $V = T(\mathfrak{A})$. We construct an automaton $\mathfrak{B}$ as follows.

$$\begin{aligned} \text{(7)} \qquad \mathfrak{B} &= (A, \beta, A_F) \\ \beta(\sigma) &= \alpha(P(\sigma)) \text{ for all symbols } \sigma \in \Sigma \end{aligned}$$

Then

$$\bar{P}^{-1}(V) = T(\mathfrak{B})$$

Determinization. As with finite state automata, there is an equivalence in tree automata between nondeterministic and deterministic automata. (Note, however, that this only holds for bottom-up automata!) The crucial step in the proof, which shows that from any nondeterministic automaton we can construct a deterministic one which recognizes the same set, is witnessed by a construction which we present here. This is, like most of these constructions, a simple generalization of the corresponding construction for finite state automata, in this case the *subset construction.*

To see why this construction is necessary, one may note simply that there are nondeterministic automata which, after the application of the complementation construction, still recognize the same set. For example,

$$\begin{gathered} \mathfrak{A} = (\{a_0, a_1\}, \alpha, \{a_0\}) \\ \alpha(\lambda) = \{a_0, a_1\} \\ \alpha(\sigma)(a_0, a_0) = \{a_0\} \text{ for all } \sigma \in \Sigma_2 \\ \text{All other transitions are to } a_1 \end{gathered}$$

recognizes all of T_Σ: all we have to do is assign a_0 to every leaf, and the tree is guaranteed to be assigned a_0.

Applying the complement construction directly to $\mathfrak{A}$ gives us

$$\mathfrak{B} = (\{a_0, a_1\}, \alpha, \{a_1\}).$$

But $T(\mathfrak{B})$ is also T_Σ! All we have to do is to assign every leaf a_1, and any arbitrary tree will be assigned a_1. If on the other hand we determinize $\mathfrak{A}$ first, then in the resulting automaton every node will be assigned $\{a_0, a_1\}$, which is a final state. The final states of the complement automaton will be just those states which do not contain a_0, that is, $\{\emptyset, \{a_1\}\}$. No tree can be assigned either state, so indeed our complemented automaton recognizes the complement of T_Σ, as we would have hoped.

Suppose we are given a nondeterministic Σ-automaton $\mathfrak{A} = (A, \alpha, A_F)$. We construct a new deterministic automaton $\mathfrak{B}$ from $\mathfrak{A}$ as follows.

(8)
$$\begin{gathered}
\mathfrak{B} = (\wp A, \hat{\alpha}, B_F) \\
\hat{\alpha}(\lambda) = \alpha(\lambda) \\
\text{For } Q_1, Q_2, Q \subseteq A, \\
\hat{\alpha}(\sigma)(Q_1, Q_2) = \bigcup \{Q | (\exists a_1 \in Q_1, a_2 \in Q_2)[\alpha(\sigma)(a_1, a_2) = Q]\} \\
(\text{i.e., } \bigcup \alpha(\sigma)(Q_1 \times Q_2)\ ) \\
B_F = \{Q \subseteq A | Q \cap A_F \neq \emptyset\}
\end{gathered}$$

Then it is a standard result of tree automata theory that

$$T(\mathfrak{A}) = T(\mathfrak{B}).$$

In mathematical work this means that we can ignore the difference between deterministic and nondeterministic automata. In applying these ideas to computation, however, it is important to note that, for example, some of the boolean constructions *only work for deterministic automata*, and therefore we must always assume that the input to such a construction has been filtered through the determinization construction if necessary.

Minimization. Since the process which determinizes an automaton assumes the presented subset construction—which leads in the worst case to $2^{|A|}$ states—we have to find a way to keep the resulting automaton as small as possible. The standard solution to this is the minimization of an automaton as found for example in Gécseg and Steinby (1984). The goal of the minimization construction is to find an automaton which has a smaller state set and fewer transitions without altering the recognized language. Minimal automata always exist and are unique up to isomorphism of their transition graph. Unfortunately, however, the minimal automaton can still be very big; nothing prevents the occasional appearance of the worst-case scenario. Still, in the normal case only a few of the state-sets output by the determinization construction are actually needed, and we can reduce the resulting automaton.

Assume that we are given a tree automaton $\mathfrak{A} = (A, \alpha, A_F)$, $k \leq |A|$. (In the following we are blurring the distinction between an automaton and the underlying algebraic structure.) A simple source of non-minimality is the occurrence of states and transitions which are never used. We defined the set of reachable states already in our discussion of the emptiness problem in section 2.2.3, (equation (1)). In the following we refer to that set as $\mathcal{R}_A$. Obviously we can eliminate all other states from the state set without altering the language recognized by the automaton.

Another source of non-minimality is the occurrence of equivalent states, i.e., states which can be substituted for each other in any computation without affecting the result. To be able to achieve the elimination of those equivalent states, we need some auxiliary definitions. Note that we presuppose that indeed all states are reachable.

The following definition ensures that the sequence of equivalence relations which will be defined later is indeed a sequence of congruence relations. This is so since congruences are invariant with respect to unary algebraic functions.[3]

We are interested in evaluating the behavior of a pair of states a_1 and a_2 against a fixed context. That is, given a transition function $\alpha(\sigma)$, for some σ, and a particular state c, we consider a_1 to be equivalent to a_2 just in case $\alpha(\sigma)(a_1, c) = \alpha(\sigma)(a_2, c)$ and $\alpha(\sigma)(c, a_1) = \alpha(\sigma)(c, a_2)$, for all choices of σ and c. That is, assigning a_1 or a_2 to a given node never makes a difference in the state that will be assigned to that node's parent. More formally:

DEFINITION 1. *A mapping $f : A \to A$ is called an elementary translation of the automaton $\mathfrak{A}$, if there exists $m > 0$, $\sigma \in \Sigma_m$, $1 \leq j \leq m$ and elements $c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_m \in A$ such that*

$$f(a) = \alpha(\sigma)(c_1, \ldots, c_{j-1}, a, c_{j+1}, \ldots, c_m) \text{ for all } a \in A.$$

We denote the set of all elementary translations by $\mathcal{ET}(\mathfrak{A})$.

[3] Lemma I.3.16 in Gécseg and Steinby (1984): An equivalence relation is a congruence relation iff it is invariant with respect to all elementary translations.

The effect of this definition is that f always refers to a certain argument position and leaves the other arguments fixed. Additionally this makes it possible to talk about the behaviour of just two arguments, i.e. states, separately without having to refer to the special context of operations, i.e. elements, from Σ and A.

Any two states whose behaviour is identical can be collapsed into a single one. We define indistinguishability inductively as follows:

DEFINITION 2. *Two states $p, q \in A$ are called k-indistinguishable iff*

(i) *both are final or both are non-final, and*

(ii) *if $k > 0$, then for all $f \in \mathcal{ET}(\mathfrak{A})$ both a and b and $f(a)$ and $f(b)$ are $k-1$-indistinguishable.*

We say that two states are equivalent iff they are k-indistinguishable for all k.

The process has to terminate since our state set is finite, i.e., we can have at most $|A|$ different states and therefore equivalence classes. Equivalence concerning states is determined by demanding indistinguishability concerning all k.

Since we are only interested in the case of $m = 2$, we can be a little more concrete about the definition of elementary translations and the congruence relations they induce. We obtain an ordered series of congruence relations $\equiv_i$ on states $a_1, a_2 \in A$ with $a_1 \equiv_0 a_2$ iff both are final or both are non-final and $a_1 \equiv_{k+1} a_2$ iff $a_1 \equiv_k a_2$ and for all $\sigma \in \Sigma_2$ and for all $c \in A$

$$
\begin{aligned}
\alpha(\sigma)(a_1, c) &\equiv_k \alpha(\sigma)(a_2, c) \\
\alpha(\sigma)(c, a_1) &\equiv_k \alpha(\sigma)(c, a_2)
\end{aligned}
\tag{9}
$$

Since $\equiv_i = \equiv_{i+1}$ for some $i < |A|$, this process stops with a largest congruence relation which we call $\sim_A$.[4]

Since we want to use this congruence relation to minimize the automaton, we have to define what it means to reduce an automaton by a congruence relation. The quotient automaton $\mathfrak{A}/_\rho$ of an automaton $\mathfrak{A}$ with respect to the congruence relation ρ is

$$
\begin{gathered}
\mathfrak{A}/_\rho = (A/_\rho, \alpha_\rho, A_F/_\rho) \\
\alpha_\rho(\sigma)([a_1]_\rho, [a_2]_\rho) = [\alpha(\sigma)(a_1, a_2)]_\rho \\
\text{for all } a_1, a_2 \in A \text{ and } \sigma \in \Sigma_2
\end{gathered}
\tag{10}
$$

[4]The proof that $\sim_A$ is a congruence relation is in Gécseg and Steinby (1984), Theorem II.6.10.

The resulting equivalent automaton represents a real reduction in the number of states in case the congruence was non-trivial. Obviously the greatest congruence achieves the greatest reduction. The recognized forest is preserved under the quotient construction.

COROLLARY 1 (Gécseg and Steinby (1984) II.6.7). *If ρ is a congruence relation, then* $T(\mathfrak{A}/_\rho) = T(\mathfrak{A})$.

Now we can define what it means to be a minimal automaton.

DEFINITION 3. *An automaton $\mathfrak{A}$ is called*

(i) *connected if $A = \mathcal{R}_A$, i.e., all states are reachable*
(ii) *reduced if $\sim_A = id_A$, i.e., the state set cannot be reduced any further*
(iii) *minimal if it is connected and reduced.*

We know that construction of the quotient automaton under a congruence relation preserves the language, but we still have to show that our minimization process indeed finds the least automaton. This is stated in the next theorem.

THEOREM 2 (Gécseg and Steinby (1984) II.6.13). *For every recognizable forest T there exists a minimal tree automaton which is unique up to isomorphism. If $\mathfrak{A}$ is a connected recognizer of T, then the minimal automaton is* $\mathfrak{A}/_{\sim_A}$.

This suggests the following algorithm for minimization. Minimize an automaton $\mathfrak{A}$ by the following steps:

1. Discard all unreachable states and the transitions on them. This results in a connected automaton $\mathfrak{A}'$ with $T(\mathfrak{A}') = T(\mathfrak{A})$.
2. Reduce $\mathfrak{A}'$ by constructing the greatest congruence $\sim_A$ and the corresponding quotient automaton $\mathfrak{A}'/_{\sim_A}$.

Some examples for the process of minimization can be found in Appendix A.3.

Note that we can incorporate the reachability test into the algorithms for the other automata constructions. For example, the determinization algorithm constructs the states just blindly as the power set of the previous states. But by constructing the state set incrementally via the transitions from the initial state, we can coroutine parts of the two algorithms. A new state is only introduced into the state set in case it is constructed as the value of some transition on existing states. This leaves just the construction of the congruences and the quotient automaton as a separate step.

2.3. **Coding Relations on Finite Sets of Nodes as Sets of Labeled Trees.** The decidability of $S2S$ follows from the fact that each relation definable in the underlying language corresponds to a recognizable set of trees. The key to that correspondence is the use of a certain family of alphabets with a particular interpretation—a *coding scheme*—according to which a tree can represent a tuple of sets of nodes in N_2. In this section we set out the essentials of that coding scheme. This scheme is somewhat easier to present if we think of trees as tree domains together with a labeling function than if we think of them as terms.

For an n-ary relation we choose an alphabet

$$\Sigma^n = \{\underline{0}, \underline{1}\}^n,$$

that is, the set of n-ary strings of bits. Then, given a tree $t : \{1,2\}^* \to \{\underline{0}, \underline{1}\}^n$, if a node $w \in \{1,2\}^*$ has a label $t(w) = \langle k_1, \ldots, k_n \rangle$, then wherever $k_i = \underline{1}$, we interpret w as a member of set X_i, the ith set variable. In this way we can interpret a Σ^n tree as encoding an assignment function for variables in the (monadic second-order) language of $S2S$. The tree t is interpreted as a function assigning to each X_i the set of nodes in which the ith bit is "on".

In our algebraic approach, $\Sigma^n = \Sigma^n_0 \cup \Sigma^n_2$, and Σ^n_0 contains only the single symbol λ. To really pursue the coding scheme in this framework we would have to set up a correspondence between terms in T_{Σ^n} and labeled subsets of N_2. The best way to do this is probably to define an *addressing* function **addr** which assigns to any subterm of a T_{Σ^n}-term the string in $\{1,2\}^*$ which is its root's "address". Then **addr** assigns the empty string to the whole term, and given an arbitrary subterm $t = ft_1t_2$, $\mathbf{addr}(t_1) = (\mathbf{addr}(t))1$ and $\mathbf{addr}(t_2) = (\mathbf{addr}(t))2$.[5] Given this it is relatively easy to view terms as specifying functions from nodes to label symbols. For example, the T_{Σ^2}-tree $\underline{10}\lambda\lambda$ encodes a pair of sets, the first of which contains only the root node, and the second of which is empty.

The construction we embark upon has the following goal, then. Given a formula ϕ in n free set variables, we wish to construct a tree automaton $\mathfrak{A}_\phi$ which recognizes all and only the trees encoding satisfying assignments for ϕ. Supposing for a moment that we could write $\phi[t]$ for the interpretation of ϕ according to the assignment function encoded by t, we would wish to define an automaton constructor $\mathfrak{aut}$ such that

$$t \in T(\mathfrak{aut}(\phi)) \iff \mathcal{N}_2 \models \phi[t].$$

[5] Recalling that $\mathcal{N}_2$ is primarily a structure of the natural numbers, we can alternatively define **addr** as follows. The value of **addr** for the whole term is zero, and given an arbitrary subterm $t = ft_1t_2$, $\mathbf{addr}(t_1) = 2 \cdot \mathbf{addr}(t) + 1$ and $\mathbf{addr}(t_2) = 2 \cdot \mathbf{addr}(t) + 2$.

Technically, we can only encode tuples of sets this way, so we can only encode assignments to sequences of set variables. However, it is relatively easy to deal with individual variables as well. At very least, we can translate to a language in which all individual variables have been replaced one-for-one with set variables which are constrained to denote singleton sets. We can do this because singletonhood is definable in a language without individual variables (in which $\subseteq$ replaces both $\approx$ and $\in$ as the sole 'built-in' relation symbol).

$$Sing(X) \iff_{\mathrm{def}} (\forall Y)[Y \subseteq X \Rightarrow (X \subseteq Y \vee (\forall Z) Y \subseteq Z)] \wedge (\exists Y)[X \not\subseteq Y]$$

This formula looks rather forbidding, but the minimal automaton corresponding to it is quite simple. (In fact it can be constructed by hand without any difficulty.) For the case of Σ^1, the automaton is as follows.

$$\begin{aligned}
\mathfrak{A} &= (\{a_0, a_1, a_2\}, \alpha, \{a_1\}), \\
\alpha(\lambda) &= a_0, \\
\alpha(\underline{0})(a_0, a_0) &= a_0, \\
\alpha(\underline{0})(a_1, a_0) &= a_1, \\
\alpha(\underline{0})(a_0, a_1) &= a_1, \\
\alpha(\underline{1})(a_0, a_0) &= a_1, \\
&\text{all other transitions are to } a_2
\end{aligned} \tag{11}$$

The set of trees encoding $Sing(X_1)$ is then just $T(\mathfrak{A})$. (It is easily seen that this automaton is just counting the number of occurrences of $\underline{1}$ in the number system "zero, one, (too) many.") Where several variables are required to be singletons, the conjunction construction can be applied to multiple copies of this automaton.

There is yet a third solution, which is one we will often employ in practice. It is quite easy to assure as part of the construction of primitive automata that certain sets can contain only one member. In effect, this amounts to constructing the relevant cross product automaton, optimizing it and renaming the states. We will see a number of examples of this technique in §2.5.

Note that the set of trees which represent a certain tuple of sets will necessarily be infinite. To see this note that the n-tuple $\underline{0}^n$ labels a node which plays no direct role in the sets being picked out. Now suppose we have a tree t which represents the desired tuple. All of its leaves are labeled λ, our sole zero-place function symbol, which plays no role in the representation of the tuple. Clearly we can take any of these λ-labeled nodes and replace it with $\underline{0}^n\lambda\lambda$ without either adding, removing or displacing any node marked for membership in one of the sets. Furthermore we can repeat this process as often as we like. Any tree we derive in this manner will represent the same tuple of sets as the tree with which we started. The same procedure can go in the other direction as well: any subtree of the form $\underline{0}^n\lambda\lambda$ can be replaced with λ, and we can do this as often as we like without affecting the tuple of sets which the tree represents.

Two courses are open to us. If we want to be able to say "the tree" that officially encodes tuple K, then we must take steps to isolate one member of this infinite set for special treatment. Both Doner (1970) and Thatcher and Wright (1968) ultimately follow this course. On the other hand, as Doner (1970) shows, given any set of trees which represent a given tuple, both its minimal member and its "closure" under extension with and removal of filler-trees $\underline{0}^n\lambda\lambda$ form recognizable sets. In spite of historical precedent, it seems to us that the encoding via the closure is simpler to implement than the encoding via minimal representatives, and that is the one we will pursue here. From now on we will distinguish the following two notions:

- a set of trees *represents* the relation made up of the tuples the trees encode;
- a set of trees *encodes* a relation just in case it contains *all* of the trees that encode each tuple of the relation.

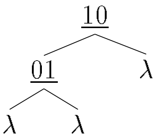
Note that encoding is a strictly stronger notion than representation; a set which encodes a relation necessarily represents it, but a set which (merely) represents a relation need not encode it. We will be at some pains to ensure that the automata we construct for a given formula recognize that formula's encoding.

So now we can say what it means for a relation on $\wp_{fin}N_2$ to be *recognizable*. An n-ary set-valued relation is a subset of $(\wp_{fin}N_2)^n$. Each member of $R \subseteq (\wp_{fin}N_2)^n$ is thus an n-tuple of (finite) sets of nodes, and can be encoded by a (finite) Σ^n-labeled tree. The relation R is, then, encodable by a set of trees, namely the full set of trees which encode its members. So we have the following.

DEFINITION 4. *An n-ary relation R is* recognizable *if and only if it can be encoded by a recognizable set of Σ^n-trees.*

2.3.1. *Some More Constructions.* Now that we have our coding scheme in place, we will need some more constructions in order to implement it. We need in particular a construction to deal with the following case.

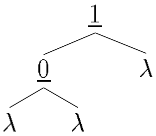
Suppose we have an automaton $\mathfrak{A}$ which recognizes the following Σ^2-tree, together with all of its extensions by $\underline{00}\lambda\lambda$.



Now take the projection function P defined as follows:

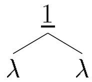
$$P : \Sigma^2 \to \Sigma^1 = \langle k_1, k_2 \rangle \mapsto \langle k_1 \rangle$$

Then applying $\bar{P}$ to the tree above yields



So the automaton we derive by applying the projection construction to $\mathfrak{A}$ will recognize this tree and all of its extensions by $\underline{0}\lambda\lambda$.

Now, our original automaton, $\mathfrak{A}$, recognized the official encoding of a certain relation (essentially the relation holding between nodes which are the root and nodes which are the first successor of the root). But this is no longer true of the projected automaton! In particular, it will not recognize the following tree, obtained by replacing the subtree $\underline{0}\lambda\lambda$ with λ, even though this tree encodes the same 1-tuple of sets as the tree above.



Therefore we need to do a little clean-up work if we want our automata to maintain the official encoding. This is not just a matter of pointless bookkeeping! If $T(\mathfrak{A})$ encodes a given relation, then its complement encodes the complement of the relation. But now if we complemented the projected automaton (even after making it deterministic!), we would wind up with a set of trees containing the tree above, which therefore does not even *represent* the complement of the projection of $T(\mathfrak{A})$, since that tree encodes a tuple in $T(\mathfrak{A})$.

First, let us define the set V of trees as being those subtrees of trees in $\bar{P}(T(\mathfrak{A}))$ which are labeled only with λ and $\underline{0}$. We need to compute the set of states A_V which are assigned by $\mathrm{h}_{\mathcal{A}}$ to members of V. This is easy to do: we need only a slight variant of the procedure of section §2.2.3 for checking the emptiness of an automaton's recognized forest. There we allowed node-labels to range freely over the alphabet. All we need to do now is to fix our attention on transitions for nodes labeled with $\underline{0}$'s. We proceed as follows.

$$A_V^0 = \alpha(\Sigma_0) \tag{12}$$
$$A_V^{m+1} := A_V^m \cup \{a | (\exists a_1, a_2 \in A_V^m)[\alpha(\underline{0}^n)(a_1, a_2) = a]\}$$
$$\text{i.e., } A_V^m \cup \alpha(\underline{0}^n)(A_V^m \times A_V^m)$$
$$A_V := \bigcup_{m<\omega} A_V^m$$

Once again, the construction is guaranteed to reach a fixpoint in a number of steps bounded by the size of A. At that point, $A_V = A_V^m$.

Now we construct from $\mathfrak{A}$ a nondeterministic automaton $\mathfrak{A}'$, as follows.

$$\mathfrak{A}' = (A, \rho, A_F), \tag{13}$$
$$\rho(\lambda) = A_V,$$
$$\rho(\sigma)(a_1, a_2) = \{\alpha(\sigma)(a_1, a_2)\}$$

(Note that the resulting automaton $\mathfrak{A}'$ is only truly nondeterministic at the leaves of a tree.) In describing the construction we tacitly assumed that the input was a deterministic automaton; we will in general be applying it to nondeterministic automata, but this does not present us with anything like the kind of problems we have with complementation. We simply need to modify the procedure for constructing A_V to read

$$A_V^{m+1} := A_V^m \cup \bigcup\{Q | (\exists a_1, a_2 \in A_V^m)[\alpha(\underline{0}^n)(a_1, a_2) = Q]\} \tag{14}$$

and we need to remove the set braces from the definition of the transition functions, so that it now reads as follows.

$$\rho(\sigma)(a_1, a_2) = \alpha(\sigma)(a_1, a_2) \tag{15}$$

This gives us a "roll-up" or *truncation* construction. For completeness we should provide also a *closure* construction which guarantees that the entire extension got by replacing λ-trees with $\underline{0}^n\lambda\lambda$-trees is recognized. Note however that in practice, while we need the truncation construction, we do not need the closure construction, since its behavior can be easily built into the primitive automata and is inherited under the various constructions. (The curious reader is referred to the proof of Lemma 11 in Thatcher and Wright (1968).)

2.4. **The Decidability Proof (Inductive Step).** In this report we wish to emphasize the "modular" nature of the decidability proof, by tackling the inductive step first. This step is common to proofs of the decidability of both the language of $S2S$ and $\mathcal{L}^2_{K,P}$, and indeed it assures the decidability of any monadic second order language with "automaton-definable" atomic formulas. So we will here assume an arbitrary monadic second-order language $\mathcal{L}$, assuming only that all relations definable by $\mathcal{L}$'s atomic formulas alone are recognizable.

First, all sentences of $\mathcal{L}$ are either of the form $(\exists X)\phi$ or of the form $\neg(\exists X)\phi$. We know that

$$\mathcal{N}_2 \models ((\exists X)\phi) \text{ iff there is an } A \subseteq N_2 \text{ and } \mathcal{N}_2 \models \phi(A)$$

Such an A exists just in case the (unary) relation defined by ϕ is non-empty. (Each member of that unary relation is a potential candidate for X.) By the same token, $\mathcal{N}_2 \models \neg(\exists X)\phi$ just in case the relation defined by ϕ *is* empty. So the decidability of $\mathcal{L}$ reduces to the decidability of emptiness for the $\mathcal{L}$-definable relations.

This indicates our strategy. If we can show that all $\mathcal{L}$-definable relations are *recognizable* then, knowing that the emptiness of recognizable sets is decidable, we have immediately that the emptiness of $\mathcal{L}$-definable relations is decidable, and so membership of $\mathcal{L}$-sentences in the $\mathcal{L}$-theory of $\mathcal{N}_2$ is decidable as well.

We prove this by induction. In the next section we will prove that the atomic wffs of $\mathcal{L}^2_{K,P}$ define recognizable relations, by the brute force approach of actually exhibiting the automata which witness this fact. To prove the inductive step, we must prove the following claim (which is essentially Lemma 19 in Thatcher and Wright (1968), and the second half of Theorem 3.7 in Doner (1970)):

THEOREM 3. *If formulas ϕ and ψ define recognizable relations, then so do*

1. $\phi \wedge \psi$,
2. $\neg\phi$,
3. $(\exists X)\phi$.

The proof is a matter of exhibiting the constructions by which we arrive at $\mathfrak{aut}(\phi \wedge \psi)$, $\mathfrak{aut}(\neg\phi)$ and $\mathfrak{aut}((\exists X)\phi)$, given $\mathfrak{aut}(\phi)$ and $\mathfrak{aut}(\psi)$.

Case 1: Conjunction. Unsurprisingly, the construction which proves the conjunction case is based on the construction for the intersection of two recognizable sets ((3), p. 8). However, simple intersection is not quite enough, since ϕ and ψ may differ on their variables. Suppose that ϕ has free variables X_1 and X_2, while ψ has free variables X_2 and X_3. That is, ϕ defines relation $R(X_1, X_2)$ while ψ defines relation $S(X_2, X_3)$. Our first problem is to get the relations to "line up". The solution is as follows. Clearly, R is the projection of some relation $R'(X_1, X_2, X_3)$, under a projection P_3 which maps $\langle k_1, k_2, k_3\rangle \mapsto \langle k_1, k_2\rangle$. Similarly, S is the projection of a relation $S'(X_1, X_2, X_3)$ under a projection P_1 which maps $\langle k_1, k_2, k_3\rangle \mapsto \langle k_2, k_3\rangle$. Since the recognizable sets are closed under inverse projection ("cylindrification"), and since by assumption both R and S are recognizable, it must be the case that R' and S' are recognizable as well.

So the first step is to identify the relevant projections and invert them, giving us automata $\mathfrak{A}'$ and $\mathfrak{B}'$ recognizing R' and S', respectively. Now $R' \cap S'$ is a 3-ary relation which is just

$$\mathrm{bh}_{\mathcal{A}'\times\mathcal{B}'}(A_F \times B_F).$$

(Recall that the inverse projection construction preserves the states and final states of its inputs; for example, $\mathfrak{A}' = (A, \alpha', A_F)$, where only the transition function differs from $\mathfrak{A}$.)

The same construction can be extended to handle formulas that overlap on any number of variables.

Case 2: Negation. This case is relatively straightforward, as long as we can make the following assumption. If we are attempting to construct $\mathfrak{aut}(\neg\phi)$, we must be able to assume that $\mathfrak{aut}(\phi)$ is deterministic. The conjunction construction, and the complementation construction itself, preserve this property of their inputs, and all of the atomic automata are deterministic, so we need only worry about this constraint in the case of existential quantification, to be considered below.

As long as our determinism constraint is satisfied, the construction for $\mathfrak{aut}(\neg\phi)$ is just complementation. If

$$\mathfrak{aut}(\phi) = (A, \alpha, A_F),$$

then

$$\mathfrak{aut}(\neg\phi) = (A, \alpha, A - A_F)$$

and that is that.

Case 3: Quantification. The core of this construction is the projection construction from §2.2.4. We wish to construct $\mathfrak{aut}((\exists X_i)\phi)$, given $\mathfrak{aut}(\phi)$. Suppose ϕ defines a relation

$$R(X_1, \ldots, X_i, \ldots, X_n).$$

Then $\mathfrak{aut}(\phi)$ is a Σ^n automaton that recognizes trees each of which encodes a tuple of subsets $\langle S_1, \ldots, S_n\rangle$. $((\exists X_i)\phi)(\bar{S})$ is satisfiable by any sequence of sets $\bar{S}$ whose i-th member can be replaced by a set S'_i such that $\phi(\bar{S}[\frac{S'_i}{X_i}])$ is satisfiable. That formula is satisfiable if the set of tuples of sets which satisfy it is not empty. That set of tuples is the relation ϕ defines. So $(\exists X_i)\phi$ defines a relation which is a projection of the relation defined by ϕ, where the projection essentially removes or suppresses the ith member of the elements of Σ^n to yield Σ^{n-1}.

The projection function corresponding to the quantifier prefix $(\exists X_i)$ is the following.

(16)

$$P_i : \Sigma^n \to \Sigma^{n-1} = \langle k_1, \ldots, k_i, \ldots, k_n \rangle \mapsto \langle k_1, \ldots, k_{i-1}, k_{i+1}, \ldots, k_n \rangle$$

In section §2.3.1 we observed that the output of this construction need not recognize an encoding, even when its input does. It was for just this purpose that we developed the truncation operation, and we apply it here. But we are not done yet! Both projection and truncation introduce nondeterminacy into the automaton, and we must therefore determinize the output in case we later apply complementation.[6] So the sequence of steps we must go through are as follows.

1. Apply the projection construction.
2. Apply the truncation construction.
3. Apply the subset construction.

This gives us a deterministic automaton which recognizes the projection of the encoding of $T(\mathfrak{aut}(\phi))$.

2.5. **The Decidability Proof (Various Base Steps).** The base of the induction involves proving that the relations definable by atomic formulas of the relevant language are recognizable. In the case of $\mathcal{L}^2_{K,P}$, then, we must prove five cases.

THEOREM 4. *The relations defined by the following formulas are all recognizable.*

1. $x \lhd^+ y$,
2. $x \lhd y$,
3. $x \prec y$,
4. $x \approx y$,
5. $x \in X$.

We prove theorem 4 by brute force, exhibiting five automata which recognize the encodings of the five relations defined in the theorem.

Keep in mind, however, that our encoding scheme is defined as if all our variables were set variables. We will see below that in all of these proofs

[6] As a matter of efficiency, it may be better to delay determinization until a negation is actually encountered, at which time we can also minimize the resulting automaton.

the assignments to individual variables appearing in these formulas are constrained to be unique. Put another way, it is as if we translated our formulas to a language containing only set variables, but where certain of these sets were in addition constrained to be singletons. (That is, we appear to adopt a typographical convention which requires all lower-case variables to denote singleton sets. Note that we still write $x_i \in X_j$, not $x_i \subseteq X_j$.)

Case 1: Dominance. We define here a Σ^2-automaton $\mathfrak{A}$ such that $T(\mathfrak{A})$ encodes the relation defined as $x_1 \lhd^+ x_2$.

(17)
$$\begin{gathered}\mathfrak{A} = (\{a_0, a_1, a_2, a_3\}, \alpha, \{a_2\}),\\ \alpha(\lambda) = a_0,\\ \alpha(\underline{01})(a_0, a_0) = a_1,\\ \alpha(\underline{10})(a_1, a_0) = a_2,\\ \alpha(\underline{10})(a_0, a_1) = a_2,\\ \alpha(\underline{00})(a_i, a_0) = a_i, a_i \in \{a_0, a_1, a_2\},\\ \alpha(\underline{00})(a_0, a_i) = a_i, a_i \in \{a_0, a_1, a_2\},\\ \text{all other transitions are to } a_3.\end{gathered}$$

Note that for any tree recognized by $\mathfrak{A}$, only a single node can be labeled $\underline{10}$ and only a single node can be labeled $\underline{01}$. That is, variables x_1 and x_2 are assigned to unique nodes by any variable assignment encoded by a tree in $T(\mathfrak{A})$. This is assured by the fact that all subtrees peripheral to a path containing denotata for x_1 and x_2 must be assigned a_0 by $\mathfrak{A}$. But no subtree containing either $\underline{01}$ or $\underline{10}$ (or for that matter $\underline{11}$) can be assigned a_0. So nodes for x_1 and x_2 must be found on a single path to the root, and, clearly, they can only be found once on any given path. Informally, the automaton $\mathfrak{A}$ is in state a_0 while it is looking for an occurrence of a node labeled $\underline{01}$. Once it has found such a node it changes to a_1, which means that it is currently searching for a node labeled $\underline{10}$. Once it has found such a node it switches to the final state a_2.

As long as $\mathfrak{A}$ sees only nodes labeled $\underline{00}$, it remains in whatever state it is currently in, so the node denoted by x_1 can be arbitrarily far above the node denoted by x_2, and both can be embedded arbitrarily deeply in the tree, and arbitrarily far above the frontier. This last point assures that $\mathfrak{A}$ recognizes a true encoding. Since the tree λ and any tree containing only zeroes and λ's are all assigned a_0 (only), they can be freely substituted for each other and so we have the effects of both truncation and closure built into the automaton.

Case 2: Immediate Dominance. The immediate dominance automaton is a simple variant of the dominance automaton of the last paragraph. We need only disallow $\underline{00}$ from intervening between the node labeled $\underline{01}$ and the node labeled $\underline{10}$. That is, we can remain in state a_1 only momentarily.

(18)
$$\begin{gathered}
\mathfrak{A} = (\{a_0, a_1, a_2, a_3\}, \alpha, \{a_2\}), \\
\alpha(\lambda) = a_0, \\
\alpha(\underline{01})(a_0, a_0) = a_1, \\
\alpha(\underline{10})(a_1, a_0) = a_2, \\
\alpha(\underline{10})(a_0, a_1) = a_2, \\
\alpha(\underline{00})(a_i, a_0) = a_i, a_i \in \{a_0, a_2\} \\
\alpha(\underline{00})(a_0, a_i) = a_i, a_i \in \{a_0, a_2\} \\
\text{all other transitions are to } a_3.
\end{gathered}$$

Case 3: Precedence. Essentially, $x_1 \prec x_2$ just in case there is a pair of sisters the leftmost one of which (reflexively) dominates x_1 and the rightmost one of which (reflexively) dominates x_2. The automaton encodes this fairly directly.

(19)
$$\begin{gathered}
\mathfrak{A} = (\{a_0, a_1, a_2, a_3, a_4\}, \alpha, \{a_3\}), \\
\alpha(\lambda) = a_0, \\
\alpha(\underline{10})(a_0, a_0) = a_1, \\
\alpha(\underline{01})(a_0, a_0) = a_2, \\
\alpha(\underline{00})(a_1, a_2) = a_3, \\
\alpha(\underline{00})(a_i, a_0) = a_i, a_i \in \{a_0, a_1, a_2, a_3\}, \\
\alpha(\underline{00})(a_0, a_i) = a_i, a_i \in \{a_0, a_1, a_2, a_3\}, \\
\text{all other transitions are to } a_4.
\end{gathered}$$

Once again, the set of nodes labeled $\underline{10}$ is a singleton, and likewise for $\underline{01}$. Also, $\mathfrak{A}$ does recognize an official encoding of the precedence relation, though this fact is a little trickier to establish here. A simple inductive argument suffices, however, to show that $\mathrm{h}_{\mathcal{A}}(\lambda) = a_0$ and, whenever $\mathrm{h}_{\mathcal{A}}(t_1) = a_0$ and $\mathrm{h}_{\mathcal{A}}(t_2) = a_0$, then $\mathrm{h}_{\mathcal{A}}(\underline{00}t_1t_2) = a_0$. So, in essence, all representations of the tuple of empty sets are treated equivalently.

Case 4: Equality. The automaton for $x_1 \approx x_2$ is designed to scan an input tree for a single node labeled $\underline{11}$.

$$\begin{aligned}
(20) \qquad \mathfrak{A} &= (\{a_0, a_1, a_2\}, \alpha, \{a_1\}),\\
\alpha(\lambda) &= a_0,\\
\alpha(\underline{11})(a_0, a_0) &= a_1,\\
\alpha(\underline{00})(a_i, a_0) &= a_i, a_i \in \{a_0, a_1\},\\
\alpha(\underline{00})(a_0, a_i) &= a_i, a_i \in \{a_0, a_1\},\\
&\text{all other transitions are to } a_2.
\end{aligned}$$

Again, only a single node labeled $\underline{11}$ is tolerated. All other nodes must be labeled $\underline{00}$ or λ. As usual, the automaton does not distinguish subtrees $\underline{00}\lambda\lambda$ from subtrees λ.

Case 5: Set membership (a.k.a. 'singleton subset'). The automaton for $x_1 \in X_2$ (properly, $x_1 \subseteq X_2 \wedge Sing(x_1)$) is essentially just like the automaton for equality, except that now only x_1 is constrained to appear uniquely. Nodes labeled $\underline{01}$ may appear freely; in fact they are indistinguishable from $\underline{00}$, reflecting that the relation here defined is completely indifferent to the size of X_2, as long as it is at least 1.

$$\begin{aligned}
(21) \qquad \mathfrak{A} &= (\{a_0, a_1, a_2\}, \alpha, \{a_1\}),\\
\alpha(\lambda) &= a_0,\\
\alpha(\underline{11})(a_0, a_0) &= a_1,\\
\alpha(\underline{0}k)(a_i, a_0) &= a_i, k \in \{\underline{0}, \underline{1}\}, a_i \in \{a_0, a_1\},\\
\alpha(\underline{0}k)(a_0, a_i) &= a_i, k \in \{\underline{0}, \underline{1}\}, a_i \in \{a_0, a_1\},\\
&\text{all other transitions are to } a_2.
\end{aligned}$$

This completes the "brute force" proof of the base case for decidability of our language.

So far, we have been content to restrict our attention to the most convenient alphabet (that containing only the two variables mentioned in the formulas). We pause here to show briefly how to rigorously describe the more general case. We take as our example the equality automaton, which has the smallest transition table. We assume an arbitrary (finite) value of n, giving us as our alphabet $\Sigma^n = \Sigma_0^n \cup \Sigma_2^n = \{\lambda\} \cup \{\underline{0}, \underline{1}\}^n$. We assume also the n projection functions $\pi_1, \ldots, \pi_n$, such that $\pi_i := (k_1 \ldots k_n \mapsto k_i)$. Then $\mathfrak{aut}(x_i \approx x_j)$ can be described by the following assertions.

$$\begin{aligned}
(22) \qquad \mathfrak{aut}(x_i \approx x_j) &= (\{a_0, a_1, a_2\}, \alpha, \{a_1\}), \\
\alpha(\lambda) &= a_0, \\
\alpha(\sigma)(a_0, a_0) &= a_1, \text{ iff } \pi_i(\sigma) = \underline{1} \text{ and } \pi_j(\sigma) = \underline{1}, \\
\alpha(\sigma)(a_i, a_0) &= a_i, \text{ iff } \pi_i(\sigma) = \underline{0} \text{ and } \pi_j(\sigma) = \underline{0}, \\
\alpha(\sigma)(a_0, a_i) &= a_i, \text{ iff } \pi_i(\sigma) = \underline{0} \text{ and } \pi_j(\sigma) = \underline{0}, \\
&\text{all other transitions are to } a_2.
\end{aligned}$$

Note that doing things in this abstract way might simplify some of our constructions. In particular, in considering the proper construction to correspond to conjunction, we had to consider the case that the two input formulas did not share all of their variables. As a result we had to invoke the inverse projection construction. However, once we translate our automata to this more abstract form, we free them to a great extent from the underlying alphabet, and in particular the inverse projection step is no longer required as part of the conjunction construction. But we will not implement this idea for the constructions. For a discussion of this issue, see section 3.2.

3. Implementational Issues

3.1. **Introduction.** In this part of the report we describe the prototype MoSeL and its implementation in some detail. MoSeL is a compiler which transforms $\mathcal{L}^2_{K,P}$ formulas to tree automata. Our implementation (in SICStus Prolog) follows point-for-point the decidability proof described in the previous chapter. Therefore its correctness is almost trivially established.

In a sense MoSeL represents a case study in a slightly unusual approach to automated theorem proving. We do not employ the standard techniques of resolution based theorem proving, model construction or tableau methods. Instead we translate the formulas to tree automata, which are themselves neither models nor, properly speaking, syntactic theories. Nonetheless, the results on which our research is based establish that the language recognized by a tree automaton is non-empty iff the formula is satisfiable. Tree automata recognize an encoding of the assignment under which the formula is satisfiable. Models for the formula can be generated by recording the generation process for these assignment trees, i.e., the automata encode the models allowed by the formula. In a certain sense the automata with non empty languages are a normal form. In this respect our work is similar to MONA (Henriksen et al. 1995). MONA, too, is a tool to compile formula of a monadic second order logic – but on strings – into finite state automata. The connection to our work on the monadic second order tree description logic $\mathcal{L}^2_{K,P}$ is straightforward. It corresponds to the step from Büchi's (Büchi 1960) result on $S1S$ to Rabin's (Rabin 1969) result on $S2S$.[7]

Automated theorem proving gains its importance for computational linguistics from the fact that processing with grammar formalisms couched in logic can be seen as such. So, one goal of our approach is to provide a tool to verify and use descriptions about natural language presented in $\mathcal{L}^2_{K,P}$.

The chapter proceeds as follows. We start by outlining the data-structure for the tree automata. Then we have a section on some work which can be done before the actual processing starts. The fourth section presents the compiler and the implementations of the constructions discussed in the first chapter. We conclude with a discussion of strengths and weaknesses of MoSeL.

3.2. **The Data-Structure.** For the Presentation of the data structure for tree automata it seems easiest to take an example and explain the data-structure from there. Consider again the automaton for equality as presented in (20). In Figure 1 we depict the automaton for equality of two singleton sets ($x_1 \approx x_2$) as a list of trees. We assume that the current alphabet ranges over two variables, i.e., there exist four different alphabet symbols, corresponding to the cases where a given node has been assigned to x_1, to x_2, to both, or to neither. We will transform this most basic representation step by step to the one we are actually going to use.

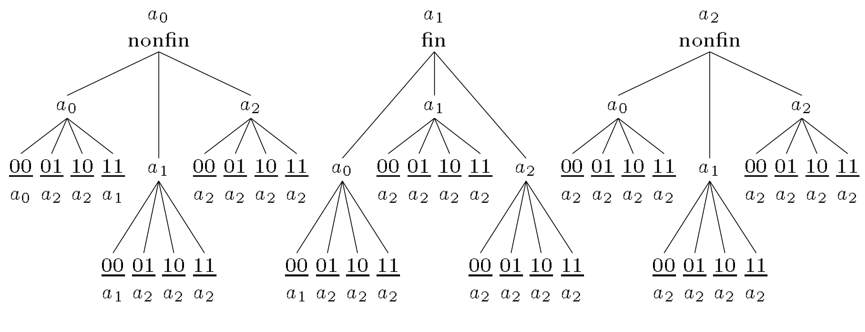


FIGURE 1. The full equality automaton for $x_1 \approx x_2$

[7]There is also work in progress to extend MONA to trees (Biehl et al. 1996).

We keep a list of states on the top level which are marked as final or non-final. From each of these states hangs another list of states. This is to be interpreted as the states from the topmost list being the left state of a transition and the states from the lower one as the right. Each arc ends in a pair with the alphabet symbol being represented with the relevant bit string followed by the resulting state. As an example for a transition suppose we want to find $\alpha(\underline{11})(a_0, a_0)$. We go to a_0 in the topmost layer, find a_0 hanging off of it, then find its daughter labeled $\underline{11}$ and the resulting state a_1.

Obviously this is a rather space consuming representation. For example, note that most transitions are to a_2. But a_2 is a non final state which cannot be left once it is entered. In other words, it is a sink state. So far our example represented a total transition table, but in the first chapter of this report we already simplified the representation of automata considerably by just stating that all other transitions are to a sink state. So the most obvious optimization is to switch to a partial transition function. We can identify the sink state automatically and then remove, or rather not construct, all the transitions resulting in it. If there are no remaining transitions on a pair of states, we can eliminate that pair as well. Note that we still keep the sink state in our topmost list for convenience. This will be explained further in the discussion of the negation construction. Our new automaton looks as presented in Figure 2.

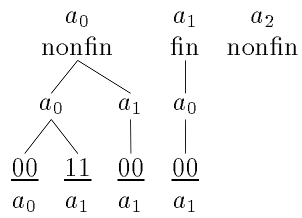


FIGURE 2. The equality automaton without sink transitions

This automaton is already a lot smaller than the previous one, but in the general case we can still do better. To facilitate access to the results of transitions we represent the alphabet with the results not as a plain list but as a balanced binary search tree. A tree is balanced iff for every node the heights of its two subtrees differ at most by one. Access, deletion and insertion become in the worst case $\mathcal{O}(\log n)$ operations. Obviously our bit vector representation of the alphabet corresponds to natural numbers. We use these as the keys to access and sort these balanced trees. The actual implementation uses the library *assoc* provided with the SICStus Prolog distribution (SIC 1995). This makes the nesting deeper, but we do not have to search the whole list if we want to find the result of a specific transition. Again in our example this does not result in a real change since we only have at most two elements in the lowest lists left. Nevertheless, the new automaton is depicted in Figure 3. We marked the balanced tree which is relevant for the pair of states (a_0, a_0). Accessing the result of the transition means now to find the relevant key in the balanced tree and to extract the corresponding value. Note that t denotes the empty balanced tree and serves as a terminator. We also identify the sink state with the atom *sink*. Keeping the sink state unique helps in reducing the state space and saves some of the labor of minimization.

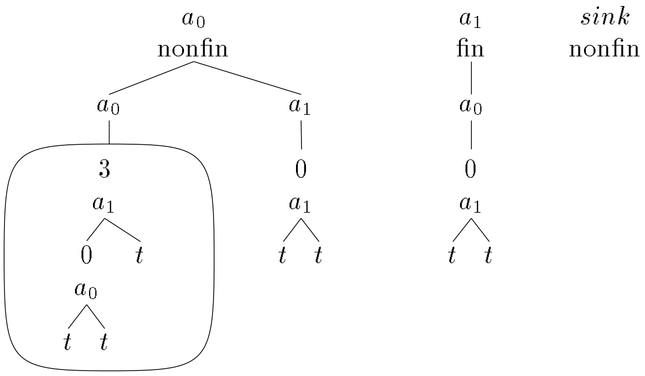


FIGURE 3. The equality automaton with balanced trees

To translate this representation into Prolog we use explicit lists of states with dependent substructures, either again a list or an association tree. We present the actual data-structure in EBNF-like notation in Table 1. In the table we allow ourselves the use of standard Prolog notation for lists, i.e., in the expression `[ H | T ]` `H` stands for the head of the list and `T` for the tail. (This can create some confusion since | is also the EBNF symbol for disjunction or alternation.)

The first argument of the automaton contains the initial state(s), the second one a list of state trees. The initial state has to be unique in principle, but can become a non-singleton list under the quantification construction. Each state tree consists of two parts: a header, containing the actual state-name and an indication of its final/nonfinal status, followed by an association list of states. Each element of that list consists

```
        AUT  →  aut( NESTATELIST, NETREELIST )
NESTATELIST  →  [ STATE | STATELIST ]
  STATELIST  →  []  |  NESTATELIST
 NETREELIST  →  [ STATETREE | TREELIST ]
  STATETREE  →  HEADER - ASSOCLIST
   TREELIST  →  []  |  NETREELIST
  ASSOCLIST  →  []  |  [ STATE - NEASSOC | ASSOCLIST ]
    NEASSOC  →  t( NUM, STATE, NUM, ASSOC, ASSOC ) |
                t( NUM, NESTATELIST, NUM, ASSOC, ASSOC )
      ASSOC  →  t  |  NEASSOC
     HEADER  →  ( STATE, MARKER )
     MARKER  →  fin  |  nonfin
        NUM  →  n ∈ ℕ
      STATE  →  atom
```

TABLE 1. The data-structure for tree automata ported to Prolog

again of the actual state and a balanced tree. Concerning the association tree note that `t` denotes the empty balanced tree and `t( K, A, N, T1, T2 )` stands for a balanced tree with key `K`, content `A`, difference in depth of the subtrees `N` and left subtree `T1` and right subtree `T2`. The content of a particular node can be either a state if the automaton is deterministic or a list of states if the automaton is nondeterministic. State names are arbitrary Prolog atoms with `sink` being the distinguished atom representing the sink state. By forcing non-empty lists on both the list of initial states and the list of state trees we capture the restriction that we have to have at least one (initial) state. Similarly, if we have a state in our association tree list it has to have a non-empty association tree. Otherwise all transitions are to *sink* and the state would have had to be removed.

In Figure 4 we show as an example the corresponding "pretty printed" version of the equality automaton in Prolog.

This data structure uses alphabet symbols as keys in the association trees. This means that the data structure for $x_1 \approx x_2$ is different from the data structure for $x_{23} \approx x_{17}$. Considering that we want to maintain a library of already compiled automata, it seems problematic to have such a concrete representation of automata only. Therefore we employ the projections presented in Section 2.5 to allow a generic representation of automata in the database. There we do not have this tree like presentation but rather Prolog clauses, each representing a transition on variable

```
aut( [ a0 ],
     [ (a0,nonfin)-
         [ a0-t(3,a1,-1,t(0,a0,0,t,t),t),
           a1-t(0,a1,0,t,t) ],
       (a1,fin)-[ a0-t(0,a1,0,t,t ) ],
       (sink,nonfin)-[] ]
   )
```

FIGURE 4. The equality automaton in Prolog

```
states( e(_,_), [q0,q1,sink] ).
finals( e(_,_), [q1] ).

m( e(_,_), lambda, q0 ).

m( e( A, B ), S, Q1, Q2, Q ) :-
      pi( A, S, IBit ),
      pi( B, S, JBit ),
      m0( e, IBit, JBit, Q1, Q2, Q ).

m0( e, 0, 0, q1, q0, q1 ).
m0( e, 0, 0, q0, q1, q1 ).
m0( e, 1, 1, q0, q0, q1 ).
m0( e, 0, 0, q0, q0, q0 ).
```

FIGURE 5. Prolog clauses for the generic equality automaton.

alphabet symbols by using the projections as an intermediate layer. (See Figure 5.)

We identify the relevant automaton by the use of an appropriate tag. In the example in Figure 5, the tag is the Prolog term `e(A,B)`, where the variables `A` and `B` are assigned the names of the particular variables being equated. Since these names correspond to bit-positions in the alphabet symbols, we can use them in calling the projection functions `pi/3` to test the relevant bits. More on these tags will be found in Section 3.4.1. With these abstract representations of automata for particular relations we do not have to have new automata in the database for each new alphabet constellation. When compiling an actual formula, we use these abstract automata as source code and compile them to our tree data-structure which is then used by the theorem prover. This is one of the tasks of the preprocessor, to which we now turn.

3.3. **The Preprocessor.** The formula preprocessor serves multiple purposes. First, we allow the input of formulas in a fairly "user-friendly" form, i.e., with lowercase letters for singleton variables and uppercase letters for set valued variables, so we have to convert the input from this format to our internal representation, that is, bit vectors whose bit positions correspond to particular variable names. So one purpose of the preprocessor is to count the number of variables appearing in the input and replace them with natural numbers representing their position in the bit vector. Currently this is done from the outside in, that is, the outermost variable gets assigned the lowest number. Doing things this way simplifies the projection construction. In projection we work our way from the inside out, so we can always just discard the highest numbered variable. More on this in section 3.4.4.

The second purpose of the preprocessor is to introduce the necessary singletonhood constraints for the appropriate variables. As mentioned previously, the basic automata contain the necessary constraints on singletonhood for their variables. But since this is just an abbreviation, we have to be very careful under negation. Consider the formula $\neg x \approx y$. Given the automaton in Figure 5, this is satisfied under an assignment which maps x, for example, to two distinct nodes. That is, the equality automaton will fail in this case, because it is designed to enforce singletonhood, and so the automaton for the entire formula will succeed. That is, if we do nothing else, this formula yields the same automaton as

$$\neg(Sing(x) \wedge Sing(y) \wedge x \approx y).$$

We wish our automata to respect the typing of the variables in the input formula. That is, the automaton we really want is the automaton for

$$Sing(x) \wedge Sing(y) \wedge \neg x \approx y.$$

Therefore we have to add the necessary singletonhood statements, and in the right places. For bound variables, this just means converting quantifiers over individual variables to restricted quantifiers over singleton set variables. For free individual variables, the singleton constraints must appear conjoined with the entire formula.

The third purpose of the preprocessor is to do cylindrification. Since for example conjunction just works for relations on the same set of variables we have to make sure that we actually know the scope of the alphabet at each formula. Suppose we have two formulas $x_1 \approx x_3$ and $x_2 \lhd^+ x_3$. To be able to conjoin them, they both have to be compiled to automata scanning for all three variables, although for each component automaton only two are relevant. To achieve this, the preprocessor counts the numbers of free variables for each subformula and annotates each construction with them. The process which makes the automata concrete can then take this information and create the appropriate bit-strings.

Again we are going to illustrate this with an example. Consider the formulas in (23)–(26). We use ^ for quantification and ~ for negation. The tag e stands for the equality automaton, rd for reflexive dominance and $sing$ for singletonhood.

(23) $$(\exists x_1)[x_1 \not\approx x_2 \wedge x_1 \triangleleft^* x_2]$$

(24)
```
(x^(~(e(x,y)) /\ rd(x,y)))
```

(25)
```
(sing(1)*1 /\ 2^((((~e(2,1)*2)*2 /\
        rd(2,1)*2)*2 /\ sing(2)*2)*2)*1)*1
```

(26) $$(Sing(x_2)^1 \wedge ((\exists x_1)(((\neg(x_1 \approx x_2)^2)^2 \wedge (x_1 \triangleleft^* x_2)^2)^2 \wedge Sing(x_1)^2)^2]^2)^1)^1$$

The first line (23) shows the logical formula and the second one (24) the corresponding input to our program. Line (25) shows the converted formula and the last one (26) the output pretty printed in logic with the number of free variables annotating the relevant brackets as powers. We have two basic automata with the alphabet size being two variables. After the first atomic formula has been negated, the two are conjoined. Before we leave the scope of the quantifier on 2, we have to ensure that it remains a singleton set. After the quantification, the scope of the alphabet has been reduced by one. It just remains to constrain 1 to be a singleton. The resulting formula provides all the information for the compiler to work efficiently.

3.4. **The Compiler.** In general the compiler works just like the inductive proof. We take our formula and generate inside out the appropriate tree automata. This means that after the preprocessing we again recursively traverse the formula. The base case means finding a tag such that a corresponding generic automaton is contained in our library. Then we create a concrete automaton with the right alphabet space and variable labeling in the described data-structure. On return of an automaton we can proceed with the boolean constructions. In this way we build the automaton recognizing the relation which the formula defines. After completion of the compilation we create a new generic automaton and output it. Additionally we allow the specification of a switch which triggers renaming of states. For debugging purposes it seems important to keep as much information as possible. Therefore the constructions can keep the complex state names formed, for example, in the various cross-product constructions or the subset construction. In the case of conjunction this means that the new automaton has pairs of the old states as state names. This enables us to see which conjunct provides which information to the overall automaton. On the other hand leads this bookkeeping to fairly large state names. If we want to work most efficiently it seems best to keep the state names small. In this case we rename the complex names to new atoms.

3.4.1. *The Base Case.* We start with the base case of the induction. In the proof we stated that we have automata for all the basic relations. The generic representations of those form the core of our library. And furthermore the proof allows us to add any compiled relation, i.e., tree automaton, to our signature. Transfered to the compiler, this means that we can add all the compiled automata to the library, too. Therefore the output of our program is a tree automaton in its generic representation.

To be able to identify the appropriate automaton, we use Prolog terms which represent the formula which is the source for the automaton, with their argument slots for the formula's variables. As an example, consider the equality automaton in Figure 6 (repeated from Figure 5). Our tag for identifying the automaton is `e(X,Y)`. Note that in case of a concrete query, the preprocessor produces concrete numbers instead of the variables, see (25). The automaton has separate clauses for the states, the final states, an initial transition encoding the initial state and the transition table.

When processing a formula we get the information from the preprocessor how big our alphabet has to be and which positions are occupied in the bit string by the variables. With this information and the generic automaton we are able to construct our tree-like representation by feeding the appropriate bitstrings to the clauses and recording all resulting transitions. The tree representation is then used in the inductive steps.

3.4.2. *The Reachability Construction.* It turns out that the central construction for the theorem prover is the reachability construction described in section 2.2.3. As we presented in the discussion on minimization, we can discard those states which are not reachable from the initial state. The reachability construction finds exactly the set of reachable states, so we use a variant of it in all the other constructions. The effect of this is that we never construct states which are not reachable. The details of

```
states( e(_,_), [a0,a1,sink] ).
finals( e(_,_), [a1] ).

m( e(_,_), lambda, a0 ).

m( e( X, Y ), S, A1, A2, A ) :-
      pi( X, S, IBit ),
      pi( Y, S, JBit ),
      m0( e, IBit, JBit, A1, A2, A ).

m0( e, 0, 0, a1, a0, a1 ).
m0( e, 0, 0, a0, a1, a1 ).
m0( e, 1, 1, a0, a0, a1 ).
m0( e, 0, 0, a0, a0, a0 ).
```

FIGURE 6. The generic equality automaton in Prolog

how to incorporate the this construction into the other constructions is presented in the respective subsections.

We use for the reachability construction the "semi-naïve method" as presented in Ullman (1988) (and see O'Keefe (1990) for a Prolog version on which our code is actually based). This method is helpful when one needs to close a given set under some two-place operation. The essential idea is to keep track of just the truly new data items which are added to the output set on each pass. Supposing we have a current output set S, to which we have just added new data items ΔS. Our specification says that the next iteration should compute $S \otimes S$. Naïvely this can be computed by a double recursion on S. The semi-naïve method uses the observation that this equals

$$(S \otimes \Delta S) \cup (\Delta S \otimes S) \cup (\Delta S \otimes \Delta S)$$

which can be computed by an outer recursion on ΔS and an inner recursion on S, for example. When ΔS is typically much smaller than S, we can get very significant savings this way.

3.4.3. *The Boolean Constructions.* The implementation of the boolean constructions is, apart from the one for negation, straightforward. Both conjunction and disjunction are easily coupled to the reachability construction since they just represent cross products of the component automata.

The outer loop of the cross product construction is presented in Figure 7. Intuitively, the most basic version of this just takes two state lists initialized with the initial state and then finds the states that can be reached from these via the transition function and collects the results in a new list. If the list is non-empty, we continue with the newly added states plus the old ones and continue doing this until no new states have been added. We keep ordered lists of states since this facilitates the necessary operations. When initialized with the initial state, both state lists are identical. In further iterations of the predicate the first argument contains the states found during the last recursive step whereas the second argument contains the states collected so far in all iterations. The inner

```
reach_cross_and( Agenda, States, UsedStates, TransA, TransB,
                 InTrans, OutTrans ) :-
      a_and_b( Agenda, States, TransA, TransB, Reachables, [],
               InTrans, NewTrans ),
      sort( Reachables, SortedRs ),
      merge( SortedRs, States, NewStates, NewAgenda ),
      ( NewAgenda == [] ->
          UsedStates = States,
          OutTrans = NewTrans
      ;
          reach_cross_and( NewAgenda, NewStates, UsedStates,
                           TransA, TransB, NewTrans, OutTrans )
      ).
```

FIGURE 7. Cross-product construction for conjunction, outer loop.

loop searches for the transitions in the respective automata and adds the resulting pairs. For example if automaton $\mathfrak{A}$ has initial state a_0 and automaton $\mathfrak{B}$ b_0 then we start with the initial pair (a_0, b_0). Suppose there is a transition in $\mathfrak{A}$ from a_0 and a_0 to a_1 using $\underline{0}$. And there is a transition in $\mathfrak{B}$ from b_0 and b_0 to b_1 using $\underline{0}$. Then we can construct the new pair (a_1, b_1) which enters the construction itself on the next recursive step. Note that now we have to care for the alphabet symbol we use. In the reachability construction it does not matter by which alphabet symbol a new state was introduced, but now we are essentially running two automata in parallel so that a new state pair can only be constructed if both automata have a valid, i.e., non-sink transition on the same alphabet symbol. The implementation has to construct these pairs by generating

```
% a_and_b( +NewStatePairs, +OldStatePairs,
%          +TransitionsAutA, +TransitionsAutB,
%          -AccReachables, +AccReachables,
%          +AccNewTransitions, -AccNewTransitions )
a_and_b( Set1, Set2, TransA, TransB, R0, R, In, Out ) :-
      ( Set1 = [] ->
          R0 = R,
          Out = In
      ;
          Set1 = [A1|RestSet1],
          a1_and_b( Set2, A1, TransA, TransB, R0, R1, In, Out1 ),
          a_and_b( RestSet1, Set2, TransA, TransB, R1, R, Out1,
                   Out)
      ).
```

FIGURE 8. The predicate `a_and_b/8`.

all transitions on two particular input state pairs and recording the successful ones. We do this by fetching the two balanced trees—one from each input automaton—and merging them. Essentially, from the pair of trees we extract a tree of pairs. We can improve on this by catching early on a number of special cases which entail that we do not have to do the pairing at all. An example of this is the case where one of the balanced trees is empty.

This describes only how to find the necessary new states, but in fact we also do all the work necessary to create the new transition table at the same time. Whenever we find a new pair of states we know which states and alphabet symbols contributed. By recording this information we can incrementally build the new conjunction automaton. For this we merge each newly created transition into our transition tree. The merging is accomplished by successively checking both levels of the respective trees for the occurrence of the relevant transition. So our loop has as arguments the transitions of the old automata, the state pairs generated in the last step of the loop and all the pairs generated previously, and accumulator pairs in which we construct the new pairs and the transition tree. The actual predicate can be seen in Figures 8–10.

The new final states are then just those pairs where both components were final states in the original automata. In the previous construction

```
a1_and_b( Set2, (A1,A2), TransA, TransB, R0, R, In, Out ) :-
      ( Set2 = [] ->
          R0 = R,
          Out = In
      ;
          % Step 1:
          Set2 = [(B1,B2)|RestSet2],
          access( TransA, A1, B1, TupleA ),
          access( TransB, A2, B2, TupleB ),
          ( TupleA == t ->
              R1 = R0,
              merge_tree( ((A1,A2),_Marker)-[], In, Out1 )
          ;
              ( TupleB == t ->
                  R1 = R0,
                  merge_tree( ((A1,A2),_Marker)-[], In, Out1 )
              ;
                  pair_and( TupleA, TupleB, OutTuple, States ),
                  ( OutTuple == t ->
                      R1 = R0,
                      merge_tree( ((A1,A2),_Marker)-[],
                                  In, Out1 )
                  ;
                      append( States, R1, R0 ),
                      merge_tree(
                         ((A1,A2),_Marker)-[(B1,B2)-OutTuple],
                         In, Out1 )
                  )
              )
          ),
```

FIGURE 9. The inner loop of the conjunction construction: Step 1.

of the new transitions we constructed the appropriate markers for membership in the set of final states by reference to the information of the conjoined automata.

The disjunction construction works similarly. The obvious difference being that now we are essentially running one or the other automaton alternatively. The implementation changes only in respect to which states

```
% Step 2:
access( TransA, B1, A1, TupleC ),
access( TransB, B2, A2, TupleD ),
( TupleC == t ->
    R2 = R1,
    merge_tree( ((B1,B2),_Marker1)-[], Out1, Out2 )
;
    ( TupleD == t ->
        R2 = R1,
        merge_tree( ((B1,B2),_Marker1)-[], Out1, Out2 )
    ;
        pair_and( TupleC, TupleD, OutTuple1, States1 ),
        ( OutTuple1 == t ->
            R2 = R1,
            merge_tree( ((B1,B2),_Marker1)-[],
                        Out1, Out2 )
        ;
            append( States1, R2, R1 ),
            merge_tree(
               ((B1,B2),_Marker1)-[(A1,A2)-OutTuple1],
               Out1, Out2 )
        )
    )
),
a1_and_b( RestSet2, (A1,A2), TransA, TransB, R2, R,
          Out2, Out )
).
```

FIGURE 10. The inner loop of the conjunction construction: Step 2.

are now recognized as being valid, i.e., any state pair where we have at least one valid transition.

The only problem is the negation automaton. The problem is due to the partial character of our automaton representation. In principle all negation does is to flip the markers of the states: final becomes nonfinal and nonfinal becomes final. This means that all the transitions which are not present are valid since *sink* is now a final state. The result is that we have to identify the new sink state, if it exists, and remove all transitions involving it, and also add explicit transitions to the old sink state. This is the price we have to pay for our data-structure. With a total transition function we just have to complement the markers and are done.

Again we use a variant of the reachability construction to find our new sink state(s). We compute both the bottom up fixpoint as before and the top down fixpoint. This means we start from the final states and see which states are reachable working backwards. The implementation of the top down reachability contains nothing different from what we presented so far, therefore we skip it here. The intersection of those two lists of states are the states which can actually be used in a computation with our automaton. The ones which are not used are the new sink states. Since we want to keep the sink state unique, we eliminate all of them and add a new sink state. Since the old atom sink is now a final state, we rename it to a new atom and use *sink* for the new sink state. With our list of used states we can then reconstruct our transition table. This is fairly simple and involves just one traversal of our transition function since we just have to go through the old table and add those transitions which are not present and eliminate those which contain one of the new sink states. We think that the saving in space is worth this complication.

3.4.4. *Other Constructions.* The constructions we deal with in this section are projection and the auxiliary ones of minimization, determinization and truncation.

As mentioned before we arrange our variable numbering in such a way that we always project the biggest (remaining) number. We have to traverse our transition table once to alter the alphabet symbols. This is done with the following formula (see Figure 11).

```
% project( +AlphSymb, +PosToProject, -AlphSymb )
project( Symb1, Pos, Symb2 ) :-
      Symb2 is Symb1 /\ ( exp(2,(Pos-1)) - 1 ).
```

FIGURE 11. The projection predicate

Input is the old key and the position to be projected, output the new key. Intuitively what happens is that we zero out the relevant position.

If the projection leads to nondeterminism, we actually have a set of new states as the result of a transition. Consider the following example.

$$\alpha(\underline{00})(a_0)(a_1) = a_1 \tag{27}$$

$$\alpha(\underline{01})(a_0)(a_1) = a_0 \tag{28}$$

$$\alpha(\underline{0})(a_0)(a_1) = \{a_0, a_1\} \tag{29}$$

The transitions in (27) and (28) collapse to the single transition in (29) when we project the rightmost bit. Internally this is represented as a list of states in the argument slot reserved for the result in the balanced tree. This means it is necessary to find all transitions on a pair of states before assigning the result into our balanced tree. But again, this does not seem harmful since it just means a reordering of work which has to be done anyway.

To cope with quantification in our logical language we need two more constructions, truncation and determinization. Truncation is a straightforward variant of the reachability construction. Recall that all we need is to collect all states which are reachable by transitions on alphabet symbols consisting of only $\underline{0}$'s to close the language recognized by the automaton under our encoding. After addition of those states to the initial state our automaton recognizes the right encoding. All that changes internally is that now we have to keep a list of initial states.

In fact, we have two kinds of automata. The standard deterministic ones and the nondeterministic ones resulting from the previously described constructions. But as discussed in section 2.4, complementation does not work for nondeterministic automata. It is an as yet open question when we do absolutely have to determinize our automata or when we can leave them nondeterministic. To do it only when necessary saves much work since the worst case complexity of the underlying subset construction is exponential. Considering the formulas to be in prefix normal form,[8] we only have to determinize in case there is an alternation in the color of the quantifier, i.e., universal versus existential. We did not implement this yet. So far the program determinizes after every quantification. We are currently investigating how to work with nondeterministic automata as long as possible, i.e., for all constructions apart from negation and only to determinize the automaton in case we have to complement it.

The subset construction will again be tied to the reachability construction. But note that this does not overcome the worst case complexity.

Again the reachability construction is initialized with those subsets of the state set which contain the initial states. Since in our determinized automaton sets of states serve as states, i.e., in a certain sense we use the subsets as complex meaningful names, these form the new initial state. From there reachability takes over and constructs the reachable subsets and the new transition trees. The final states are then just those subsets which contain a final state. The result is a new deterministic automaton with more complex names, but recognizing the same language. Internally

[8] Although we do not want to imply that this has to be the case. Rather the contrary, since pushing the quantifiers in as far as possible helps in reducing the size of the automata.

the subsets are kept as ordered lists of states, so a little care is required to distinguish real lists of states and these new complex names.

The last construction we have to present is minimization. We do this in two steps. The first one finds those states which behave identically and the second one constructs the appropriate quotient automaton. Note that we coded some minimization into the other constructions by keeping the sink state unique. Practical tests have shown that most of the states which are removed by minimization are indeed copies of the sink state. So they are reachable, but they do not contribute in finding solutions. We talked about this only for negation, but we have incorporated heuristics to eliminate some of them into all the other constructions as well. Note that in all cases a unique state *sink* is only necessary if we are not dealing with a transition function without a sink state. In our data-structure this corresponds to a total transition function, i.e., transition trees in which nothing was removed. The worst case for testing is a traversal of the tree structure with checks whether all states appear at all levels and all alphabet symbols are apparent. This is implemented in such a way that we stop looking for a total transition function if we can find a sublist or balanced tree which is not of the appropriate length or size respectively. If this predicate fails, i.e., all states appear at all levels and all balanced trees are of the complete size, then we are dealing with a total function. Having *sink* in all transition trees regardless would mean that we can have a unreachable state in the automaton. Since most of the automata we deal with have a sink state, we almost never have to deal with the worst case sketched above. Therefore we think that it is better to test for the totality of the transition representation than to have unreachable states.

Conjunction produces a number of easily identified sink states since every pair which contains sink does not represent a valid transition. We did not even construct those. So the reachability construction does not add any identifiable sink state at all. Nevertheless there might be pairs of states which turn out to be sink states. Disjunction does produce just one sink state namely the pair containing both of the old sink states.

Projection and truncation do not affect the sink state.[9] But determinization might require a new unique sink state if the transition tree is not total. So far we do not have a heuristic to eliminate subsets which might be sink states. Here we really need minimization.

[9] If there is a $\underline{0}$ labeled transition into the sink state, then truncation may add the sink state to the initial states, but that does not change it being a sink state.

So we come back to the first step necessary to minimize an automaton. We have to partition the state set into equivalence classes. Note that it is important in which position a state is used with which result. States that are equivalent have to behave identically both as a right and a left state. We start out by separating the final from the non-final states. Then we have to refine each of those partitions further. We do this by assuming that all states in our partition are equal and testing them against the whole state set. If we can find a behavioral difference, we eliminate this possibility. The crucial predicate is shown in Figure 12.

```
% equiv( +StateList, +EquivPair, +Transitions, +PreviousEquivs )
equiv( [], _, _ , _ ).
equiv( [ QNX | States ], Q1-Q2, Trans, Equivs ) :-
      access( Trans, QNX, Q1, Tuple1 ),
      access( Trans, QNX, Q2, Tuple2 ),
      equiv_tuple( Tuple1, Tuple2, Equivs ),
      access( Trans, Q1, QNX, Tuple3 ),
      access( Trans, Q2, QNX, Tuple4 ),
      equiv_tuple( Tuple3, Tuple4, Equivs ),
      equiv( States, Q1-Q2, Trans, Equivs ).
```

FIGURE 12. The central equivalence predicate

In this predicate we recurse on the state list against a pair of states which is assumed to be equivalent. Then we access the relevant balanced trees both with the states as the right and the left state and check those against the equivalence statements constructed so far. If we find a difference, the predicate fails and we are done with this hypothesis. We have to keep on doing this until no more changes in our equivalence partitioning can be discovered. Again we implement it by a loop using the semi-naïve method (see Figure 13).

After we fully partitioned our state set, we form our equivalence classes with the smallest state being the representative, i.e., the mother node of an ordered tree with all the daughters being equivalent states. The only exception is the sink state where not the smallest, but the atom *sink* is the representative. This representation allows a better access of the representative and the current state.

The second step of forming the quotient automaton is achieved by traversing the transition tree and putting in the relevant representatives into the transitions headed by a representative and removing all the other transitions headed by equivalent states.

```
% equiv_states(+EquivPairs,+StateList,+Transitions,-EquivStates)
equiv_states( Pairs, States, Trans, Equivs ) :-
      equiv_states_aux(Pairs,States,Trans,Pairs,RemPairs,Diff ),
      ( Diff == [] ->
          Equivs = Pairs
      ;
          equiv_states( RemPairs, States, Trans, Equivs)
      ).
```

FIGURE 13. The equivalence loop

3.4.5. *Properties of the Compiler.* After this description of the compiler we want to present some of its properties. We already noted that the worst case complexity is non-elementary. But by tying all constructions to the reachability constructions we achieve a much better behaviour in the average case.

The coupling of the reachability construction and minimization means at the same time that our automata are as compact as possible. This is especially interesting in case we recognize no trees or all trees. In the first case that we recognize no trees, we have only the sink state and no final states. In this sense the compiler represents a theorem prover. If the resulting automaton is empty, the (existential closure of the) formula is not a theorem of our language. On the other hand the automaton which recognizes all trees has just one state which is initial and final at the same time. If the compilation generates such an automaton it means that the (existential closure of the) formula represents a tautology in our language. In all other cases the resulting automaton means that the (existential closure of the) formula is a theorem of our language.

An interesting open question is the connection between types of formulas and the size of the corresponding automata. In the paragraph above we noted that tautologies and contradictions have very small automata. But what about the other cases. Are there particular sets of formulas that produce small automata? One advantage of specifying syntax via logic is the conciseness of the representations. When we switch with our representation to automata, under which conditions do we lose this property? And since we use Prolog as our implementation language, what happens due to our representation of these automata in (a special form of) logic?

Last let us note that the automata, after they have been generated, specify a very efficient means of processing natural language since they are deterministic finite state devices. The ideal scenario allows the specification of a grammar along the lines of Roger's formalization of most of Rizzi's Relativized Minimality (Rizzi 1990). Then we can compile this grammar in an offline process into a tree automaton. If this works – we come back to the problems involved in section 3.5 – we can use the automaton as a very efficient means to process further queries relative to the grammar. The obvious task to address here is parsing. Recognition of a given tree is trivial since this just means running the automaton on a specific tree. For recognition of a string, (which amounts to parsing in this setting) we have to conjoin our grammar with the formula representing the input string. In case it is a theorem we get a new non-empty tree automaton in return. Running this tree automaton as a generator then results in assignment trees which allow the construction of a parse analysis.[10] So in fact, we can use these automata not just as recognizers but as parsers as well.

3.5. **Summary.** In this paper we presented MoSeL, a compiler from monadic second order logic on trees to tree automata. We hope that it helps to use $\mathcal{L}^2_{K,P}$ for natural language processing. We can compile (parts of) our grammar offline into a deterministic finite state tree automaton which can be used efficiently as a recognition and generation device. And in fact, the compiler does theorem proving already. Due to the incorporation of reachability and minimization, any generated automaton which does have a final state represents a theorem of our language.

Despite its apparently staggering complexity bound, we think that its average behavior and the fact that we want to use it offline, make it into an interesting tool for natural language processing. But we are well aware that so far MoSeL only represents a prototype. It was designed mainly with a focus on correctness, although it does allow the specification of predicates and libraries. But it does not incorporate all possible optimizations. For example, looking at the implementation of MONA, we do not fully incorporate their ideas of using BDDs and their corresponding operations.

[10]We have to be a bit careful since the number of resulting assignment trees can be infinite. Remember that we did not define a unique minimal tree, but the closure under our relation. In generation we have to employ something like iterative deepening on the $\underline{0}$ labeled transitions, so that we generate first the trees without $\underline{0}$ labeled nodes, then the trees with just one $\underline{0}$ labeled node, etc. We then can stop at any point in the generation.

Apart from these technical points there is a real issue left. Since our main application is to be sought in the realm of natural language processing, we have to be able to cope with alphabets of large size. Considering that each lexical entry and each feature in a formalization has to be represented by a new free variable, we face for a grammar of realistic coverage real problems since – looking at the way our alphabet is coded – this means that we have to consider too many possibilities to be able to get automata of any reasonable size. Even the data-structure proposed for MONA faces pointers of an immense depth. It is an open question whether these large amounts of free variables can be handled efficiently. In the following, we will point out some possible solutions to this problem.

One direction for optimization seems to lie in using generic automata throughout all the constructions. This depends on whether an appropriate data structure for the projections on the transitions can be found and implemented.

A different solution – and in fact the third use of MoSeL – consists in embedding it as a constraint solver into a constraint logic programming scheme. In Morawietz (1997) we describe our first attempt using a standard Höhfeld and Smolka scheme (Höhfeld and Smolka 1988). Thus it becomes possible to consider one lexical entry at a time. The idea is that we try to split the work into components in such a way that our constraint logic programming component handles the control structure of the compilation process by parameterizing it to the particular input.

The embedding into an constraint logic programming scheme suggests that we might be able to handle the problem caused by features and lexical entries by employing a different constraint solver when dealing with these. We would have a hierarchy of constraint solvers depending on the level of the problem we are working on. Outermost would be the clp interpreter, followed by MoSeL which in turn would use another solver in the appropriate places. This corresponds roughly to a layering of logics discussed for example in Blackburn and de Rijke (1994). But the exploration of this is beyond the scope of the paper and left for further research.

Appendix A. Additional Examples

A.1. **Some Example Constructions.** Given the nature of the decidability proof, we observe immediately that we can extend our language with any relation symbols which can be defined using our tree automata.

For example, we can define the subset relation directly by the following automaton construction. Here is the automaton definition for $\mathfrak{aut}(X_1 \subseteq X_2)$.

$$\mathfrak{aut}(X_1 \subseteq X_2) = \mathfrak{A} = (\{a_0, a_1\}, \alpha, \{a_0\})$$
$$\alpha(\lambda) = a_0$$
$$\alpha(\sigma)(a_0, a_0) = a_0, \text{for } \sigma \in \{\underline{00}, \underline{01}, \underline{11}\}$$
$$\text{everything else maps to } a_1$$

We consider now, in light of this definition, the automata for the following simple formulas.

(30) $$(\exists X_2) \neg X_1 \subseteq X_2$$

(31) $$\neg(\exists X_2)\, X_1 \subseteq X_2$$

(32) $$\neg(\exists X_2) \neg X_1 \subseteq X_2$$

This example may help to elucidate how our automata constructions take care of the rather tricky problem of scope.

Tackling the automaton for (30) first, we apply the complementation construction to $\mathfrak{A}$ and then we apply the projection construction to it, removing the second position of the node labels. The complement construction simply inverts the final states. So the relation defined by $\neg X_1 \subseteq X_2$ is just

$$T((\{a_0, a_1\}, \alpha, \{a_1\})).$$

Since all the leaf nodes are assigned state a_0, the only way to get into state a_1 is to encounter somewhere in the tree a node labeled $\underline{10}$, that is, which is assigned to the denotation of X_1 but not to X_2. Note that this means that some node must be labeled with a $\underline{1}$ in the first position. This is what we would expect: since the empty set is a subset of every set, it cannot be the first term in any tuple in the not-subset relation.

We now construct the projection automaton. Transitions on $\underline{10}$ and $\underline{11}$ now collapse together, as do transitions on $\underline{00}$ and $\underline{01}$ (which were never in fact distinct).

$$\mathfrak{A}_2 = (\{a_0, a_1\}, \beta, \{a_1\})$$
$$\beta(\lambda) = \{a_0\}$$
$$\beta(\underline{0})(a_0, a_0) = \{a_0\}$$
$$\beta(\underline{1})(a_0, a_0) = \{a_0, a_1\}$$
$$\text{everything else maps to } \{a_1\}$$

The formula defines the set of sets which are not subsets of everything, that is, the set of sets distinct from the empty set. The implementation is correct, then, just in case only trees which have some $\underline{1}$-labeled nodes in them are accepted: these are the representatives of the non-empty sets. This is obviously the case: the only way into state a_1 is via a node labeled $\underline{1}$. The trivial tree λ is assigned a_0, and so, by a simple induction, is every tree labeled only with $\underline{0}$'s. Any tree with at least one $\underline{1}$ in it obviously defines a non-empty set of nodes, and, correspondingly, there is an accepting run of $\mathfrak{A}_2$ for every such tree.

According to our account of quantification we should now determinize this automaton, but since we are done and, in particular, we are not under the scope of a negation, we need not construct this further automaton which is, after all, equivalent. This step is needed in the next construction, however, and we discuss it there.

We now turn to the second formula, (31). Here we first construct the projection, and then we must complement it; but we cannot complement it until we have first determinized it. The projection automaton has the same underlying algebra as $\mathfrak{A}_2$; only the final states are different.

$$\mathfrak{A}_3 = (\{a_0, a_1\}, \beta, \{a_0\})$$

The formula which we have just implemented, $(\exists X_2)\, X_1 \subseteq X_2$, defines the set of all (finite) sets of nodes (under the weak interpretation of the theory), and it is clear that the automaton does in fact implement this behavior. First, by the reasoning above, any tree which represents the empty set will have a run assigning a_0 to the tree. The rest of the trees in T_{Σ^1} are in the set by the following reasoning. Every node labeled $\underline{1}$ and dominating only nodes labeled $\underline{0}$ or λ can be assigned a_0. This furnishes the base of a simple inductive argument which shows that we can always find a run assigning a_0 to any tree.

When we apply the complementation construction to this automaton, then, we had better derive an automaton which recognizes no trees at all. And this is in fact what happens. First, the determinized automaton can be described compactly by the following rules:

$$\mathfrak{A}_4 = (\{\emptyset, \{a_0\}, \{a_1\}, \{a_0, a_1\}\}, \hat{\beta}, \{\{a_0\}, \{a_0, a_1\}\})$$
$$\hat{\beta}(\lambda) = \{a_0\}$$

$$\begin{aligned}
\hat{\beta}(\underline{0})(Q_1, Q_2) = Q \iff & \\
& (a_1 \in Q_1 \vee a_1 \in Q_2 \Rightarrow a_1 \in Q) \wedge \\
& (a_0 \in Q_1 \wedge a_0 \in Q_2 \Rightarrow a_0 \in Q) \\
\hat{\beta}(\underline{1})(Q_1, Q_2) = Q \iff & \\
& (a_1 \in Q) \wedge \\
& (a_0 \in Q_1 \wedge a_0 \in Q_2 \Rightarrow a_0 \in Q)
\end{aligned}$$

The final states of the complement automaton are those states of $\mathfrak{A}_4$ which do not have a_0 as a member, that is,

$$\{\emptyset, \{a_1\}\}$$

Clearly, the trivial tree λ is not accepted by this automaton. The tree $\underline{0}\lambda\lambda$ is assigned state $\{a_0\}$, so it is not in the set, and the tree $\underline{1}\lambda\lambda$ is assigned $\{a_0, a_1\}$, so it too is not in the set. It should be clear that there is now no way to get a_0 out of the set assigned by the transition function, for any tree. So indeed, the language of this automaton is empty as desired.

We turn now to our final formula, (32). Note that this is equivalent to the universally quantified formula

$$(\forall X_2)\, X_1 \subseteq X_2$$

This formula defines the set containing only the empty set, so the automaton we construct ought to accept only trees representing the empty set, that is, the trivial tree λ and any tree containing only $\underline{0}$-labeled interior nodes.

We have already constructed $\mathfrak{aut}((\exists X_2)\, \neg X_1 \subseteq X_2)$, so all we have to do is negate it. But of course to do this we must first determinize it. Once again, we have already done the work: the desired transition function is just $\hat{\beta}$. But now the final states are just those states which do not contain a_1. Clearly the tree λ is accepted. Then it is easy to show that every tree labeled only with $\underline{0}$ will have a_0 in the state it is assigned. Finally, any tree with at least one node labeled $\underline{1}$ will be assigned a state containing a_1, so it will be rejected. So indeed these constructions seem to work as we would hope.

A.2. **More Detailed Example Constructions.** We consider here a rather more involved example. In (33) we see the axiom of rootedness.

(33) $$(\exists x_1)(\forall x_2)[x_1 \lhd^* x_2]$$

		00	01	10	11
a_0,	a_0	a_0	a_1	$\perp$	b_1
	a_1	a_1	$\perp$	a_2	$\perp$
	a_2	a_2	$\perp$	$\perp$	$\perp$
	b_1	b_1	$\perp$	$\perp$	$\perp$
a_1,	a_0	a_1	$\perp$	a_2	$\perp$
a_2,	a_0	a_2	$\perp$	$\perp$	$\perp$
b_1,	a_0	b_1	$\perp$	$\perp$	$\perp$

Everything else goes to $\perp$

TABLE 2. Transition table for α.

Converting individual variables to singleton set variables, we get (34).

(34) $$(\exists X_1)[Sing(X_1) \wedge (\forall X_2)[Sing(X_2) \Rightarrow X_1 \lhd^* X_2]]$$

Finally, unpacking our various abbreviations, we get (35).

(35) $$(\exists X_1)[Sing(X_1) \wedge \neg(\exists X_2)[Sing(X_2) \wedge \neg(X_1 \lhd^+ X_2 \vee X_1 \approx X_2)]]$$

Let us suppose that we have automata $\mathfrak{A}$ and $\mathfrak{B}$ corresponding to the $\lhd^+$ and $\approx$ formulas. We construct the automaton for the disjunction by the (disjoint) union operation, giving us the following combined automaton.[11]

$$\mathfrak{A} = (A, \alpha, A^F)$$
$$A = \{a_0, a_1, a_2, b_1, \perp\}$$
$$A^F = \{a_2, b_1\}$$
$$\alpha(\lambda) = a_0$$

The transition function α is displayed in Table 2.

This automaton is deterministic, so the automaton for

$$\neg(X_1 \lhd^+ X_2 \vee X_1 \approx X_2)$$

is got by simply inverting the final states:

(36) $$\sim A^F_{\lhd^*} = \{a_0, a_1, \perp\}$$

[11]Note that the initial states of the two automata are equivalent, as are the sink states, and we have collapsed them into a_0 and $\perp$, respectively. The transitions are now really given as a table. If the left column of the state pair is empty, it is supposed to be filled with the previous left state, e.g., in the transition Table 2 a_0.

The cross product automaton which we get from the conjunction

$$Sing(X_2) \wedge \neg(X_1 \lhd^+ X_2 \vee X_1 \approx X_2)$$

contains a large number of useless states. This is because $\mathfrak{A}_{\lhd^+}$ has a certain amount of counting behavior built into it; in particular, it makes some definite response to each non-zero label. So, in particular, it makes no sense for the cross-product automaton to pretend that it can be in a state in which one of its component automata has responded to the presence of a label $\underline{01}$ while the other has not. So, assuming that $\mathfrak{A}_{Sing}$ is defined as follows (with the transition table as given in (11) on page 15),

$$\mathfrak{A}_{Sing} = (\{0, 1, \perp\}, \beta, \{1\})$$

we get a combined automaton like the following.

$$\begin{aligned}
\mathfrak{A}_2 &= (A_2, \alpha_2, A_2^F) \\
A_2 &= \{(0, a_0), (1, a_1), (1, a_2), (1, b_1), (1, \perp), (\perp, \perp)\} \\
A_2^F &= \{(1, a_1), (1, \perp)\} \\
\alpha_2(\lambda) &= (0, a_0)
\end{aligned}$$

with the transitions as in Table 3.

We now construct the "raw" projected automaton, corresponding to

$$(\exists X_2)[Sing(X_2) \wedge \neg(X_1 \lhd^+ X_2 \vee X_1 \approx X_2)]$$

This is very easy to construct: its state set $A_3 = A_2$, and its final states are also the same: $A_3^F = A_2^F$. Its transition table can be got very simply from the table for α_2 by collapsing the $\underline{00}$ and $\underline{01}$ columns into a $\underline{0}$ column and by collapsing the $\underline{10}$ and $\underline{11}$ columns into a 1 column. We derive the following transitions, see Table 4.

After truncation, we derive the initial states.

$$\{(0, a_0), (1, a_1), (\perp, \perp)\}$$

We are now ready to apply the subset construction. Before we do so, however, it is very helpful to rename our states. The state-names of the subset automaton will be large enough as is! Accordingly, we apply the following mapping.

$$\begin{array}{lclclcl}
(0, a_0) & \mapsto & c_0 & \qquad & (0, \perp) & \mapsto & d_0 \\
(1, a_1) & \mapsto & c_1 & & (1, \perp) & \mapsto & d_1 \\
(1, a_2) & \mapsto & c_2 & & (\perp, \perp) & \mapsto & d_2 \\
(1, b_1) & \mapsto & c_3 & & & &
\end{array}$$

		$\underline{00}$	$\underline{01}$	$\underline{10}$	$\underline{11}$
$(0, a_0)$,	$(0, a_0)$	$(0, a_0)$	$(1, a_1)$	$(0, \perp)$	$(1, b_1)$
	$(0, \perp)$	$(0, \perp)$	$(1, \perp)$	$(0, \perp)$	$(1, \perp)$
	$(1, a_1)$	$(1, a_1)$	$(\perp, \perp)$	$(1, a_2)$	$(\perp, \perp)$
	$(1, a_2)$	$(1, a_2)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(1, b_1)$	$(1, b_1)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(1, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
$(0, \perp)$,	$(0, a_0)$	$(0, \perp)$	$(1, \perp)$	$(0, \perp)$	$(1, \perp)$
	$(0, \perp)$	$(0, \perp)$	$(1, \perp)$	$(0, \perp)$	$(1, \perp)$
	$(1, a_1)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(1, a_2)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(1, b_1)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(1, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
$(1, a_1)$,	$(0, a_0)$	$(1, a_1)$	$(\perp, \perp)$	$(1, a_2)$	$(\perp, \perp)$
	$(0, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
$(1, a_2)$,	$(0, a_0)$	$(1, a_2)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(0, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
$(1, b_1)$,	$(0, a_0)$	$(1, b_1)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(0, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
$(1, \perp)$,	$(0, a_0)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$
	$(0, \perp)$	$(1, \perp)$	$(\perp, \perp)$	$(1, \perp)$	$(\perp, \perp)$

Everything else goes to $(\perp, \perp)$

TABLE 3. Transition table for α_2.

Our automaton now looks as follows.

$$\mathfrak{A}_4 = (A_4, \alpha_4, A_4^F)$$
$$A_4 = \{c_0, c_1, c_2, c_3, d_0, d_1, d_2\}$$
$$A_4^F = \{c_1, d_1\}$$

The transition function α_4 is described in Table 5.

There are 2^7 states in the "raw" subset automaton. Fortunately, almost all of these sates are not reachable from the initial state. Considering only the reachable states, we cut things down to 4 states! They are the following.

$$\{\{c_0, c_1, d_2\}, \{c_2, c_3, d_0, d_2\}, \{c_2, c_3, d_0, d_1, d_2\}, \{d_0, d_1, d_2\}\}$$

		$\underline{0}$	$\underline{1}$
$(0, a_0)$,	$(0, a_0)$	$\{(0, a_0), (1, a_1)\}$	$\{(0, \perp), (1, b_1)\}$
	$(0, \perp)$	$\{(0, \perp), (1, \perp)\}$	$\{(0, \perp), (1, \perp)\}$
	$(1, a_1)$	$\{(1, a_1), (\perp, \perp)\}$	$\{(1, a_2), (\perp, \perp)\}$
	$(1, a_2)$	$\{(1, a_2), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(1, b_1)$	$\{(1, b_1), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(1, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
$(0, \perp)$,	$(0, a_0)$	$\{(0, \perp), (1, \perp)\}$	$\{(0, \perp), (1, \perp)\}$
	$(0, \perp)$	$\{(0, \perp), (1, \perp)\}$	$\{(0, \perp), (1, \perp)\}$
	$(1, a_1)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(1, a_2)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(1, b_1)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(1, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
$(1, a_1)$,	$(0, a_0)$	$\{(1, a_1), (\perp, \perp)\}$	$\{(1, a_2), (\perp, \perp)\}$
	$(0, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
$(1, a_2)$,	$(0, a_0)$	$\{(1, a_2), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(0, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
$(1, b_1)$,	$(0, a_0)$	$\{(1, b_1), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(0, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
$(1, \perp)$,	$(0, a_0)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$
	$(0, \perp)$	$\{(1, \perp), (\perp, \perp)\}$	$\{(1, \perp), (\perp, \perp)\}$

Everything else goes to $\{(\perp, \perp)\}$

TABLE 4. Transition table for α_3.

The initial state is $\{c_0, c_1, d_2\}$, and the final states are all the states containing either c_1 or d_1, that is,

$$\{\{c_0, c_1, d_2\}, \{c_2, c_3, d_0, d_1, d_2\}, \{d_0, d_1, d_2\}\}$$

The transition table is given as the table for α_5, in Table 6.

Now we can complement this automaton to give us the automaton for the formula

$$\neg(\exists X_2)[Sing(X_2) \wedge \neg(X_1 \triangleleft^+ X_2 \vee X_1 \approx X_2)]$$

All we have to do is invert the final states, giving us

$$\sim A_5^F = \{\{c_2, c_3, d_0, d_2\}\}$$

¿From Table 6 we see immediately that the final state is only reachable from the initial state $\{c_0, c_1, d_2\}$, So states $\{c_2, c_3, d_0, d_1, d_2\}$ and $\{d_0, d_1, d_2\}$ are both sink states of this automaton. It should be clear

		$\underline{0}$	$\underline{1}$
$c_0,$	c_0	$\{c_0, c_1\}$	$\{d_0, c_3\}$
	d_0	$\{d_0, d_1\}$	$\{d_0, d_1\}$
	c_1	$\{c_1, d_2\}$	$\{c_2, d_2\}$
	c_2	$\{c_2, d_2\}$	$\{d_1, d_2\}$
	c_3	$\{c_3, d_2\}$	$\{d_1, d_2\}$
	d_1	$\{d_1, d_2\}$	$\{d_1, d_2\}$
$d_0,$	c_0	$\{d_0, d_1\}$	$\{d_0, d_1\}$
	d_0	$\{d_0, d_1\}$	$\{d_0, d_1\}$
	c_1	$\{d_1, d_2\}$	$\{d_1, d_2\}$
	c_2	$\{d_1, d_2\}$	$\{d_1, d_2\}$
	c_3	$\{d_1, d_2\}$	$\{d_1, d_2\}$
	d_1	$\{d_1, d_2\}$	$\{d_1, d_2\}$
$c_1,$	c_0	$\{c_1, d_2\}$	$\{c_2, d_2\}$
	d_0	$\{d_1, d_2\}$	$\{d_1, d_2\}$
$c_2,$	c_0	$\{c_2, d_2\}$	$\{d_1, d_2\}$
	d_0	$\{d_1, d_2\}$	$\{d_1, d_2\}$
$c_3,$	c_0	$\{c_3, d_2\}$	$\{d_1, d_2\}$
	d_0	$\{d_1, d_2\}$	$\{d_1, d_2\}$
$d_1,$	c_0	$\{d_1, d_2\}$	$\{d_1, d_2\}$
	d_0	$\{d_1, d_2\}$	$\{d_1, d_2\}$

Everything else goes to d_2

TABLE 5. Transition table for α_4.

		$\underline{0}$	$\underline{1}$
$\{c_0, c_1, d_2\},$	$\{c_0, c_1, d_2\}$	$\{c_0, c_1, d_2\}$	$\{c_2, c_3, d_0, d_2\}$
	$\{c_2, c_3, d_0, d_2\}$	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
$\{c_2, c_3, d_0, d_2\},$	$\{c_0, c_1, d_2\}$	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
	$\{c_2, c_3, d_0, d_2\}$	$\{d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
$\{c_2, c_3, d_0, d_1, d_2\},$	$\{c_0, c_1, d_2\}$	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
	$\{c_2, c_3, d_0, d_2\}$	$\{d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$
	$\{c_2, c_3, d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$	$\{d_0, d_1, d_2\}$

Everything else goes to $\{d_0, d_1, d_2\}$

TABLE 6. Transition table for α_5.

$$\mathfrak{A} = (A, \alpha, A_F)$$
$$A = \{\emptyset, \{a_0\}, \{a_s\}, \{a_0, a_s\}\}$$
$$A_F = \{\emptyset, \{a_s\}\}$$
$$\alpha(\lambda) = \{a_0\}$$

$\alpha(\underline{0})(\emptyset, \emptyset) = \emptyset$	$\alpha(\underline{1})(\emptyset, \emptyset) = \{a_s\}$
$\alpha(\underline{0})(\emptyset, \{a_0\}) = \emptyset$	$\alpha(\underline{1})(\emptyset, \{a_0\}) = \{a_s\}$
$\alpha(\underline{0})(\emptyset, \{a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\emptyset, \{a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\emptyset, \{a_0, a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\emptyset, \{a_0, a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_0\}, \emptyset) = \emptyset$	$\alpha(\underline{1})(\{a_0\}, \emptyset) = \{a_s\}$
$\alpha(\underline{0})(\{a_s\}, \emptyset) = \{a_s\}$	$\alpha(\underline{1})(\{a_s\}, \emptyset) = \{a_s\}$
$\alpha(\underline{0})(\{a_0, a_s\}, \emptyset) = \{a_s\}$	$\alpha(\underline{1})(\{a_0, a_s\}, \emptyset) = \{a_s\}$
$\alpha(\underline{0})(\{a_0\}, \{a_0\}) = \{a_0\}$	$\alpha(\underline{1})(\{a_0\}, \{a_0\}) = \{a_0, a_s\}$
$\alpha(\underline{0})(\{a_0\}, \{a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\{a_0\}, \{a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\}$	$\alpha(\underline{1})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\}$
$\alpha(\underline{0})(\{a_s\}, \{a_0\}) = \{a_s\}$	$\alpha(\underline{1})(\{a_s\}, \{a_0\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\}$	$\alpha(\underline{1})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\}$
$\alpha(\underline{0})(\{a_s\}, \{a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\{a_s\}, \{a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_s\}, \{a_0, a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\{a_s\}, \{a_0, a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_0, a_s\}, \{a_s\}) = \{a_s\}$	$\alpha(\underline{1})(\{a_0, a_s\}, \{a_s\}) = \{a_s\}$
$\alpha(\underline{0})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\}$	$\alpha(\underline{1})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\}$

FIGURE 14. A determinized but unminimized automaton.

that this automaton does indeed only recognize sets X_1 which contain only the root. Since the root is unique in any case, the extra call to *Sing* is superfluous here. The set of singleton sets containing only the root is nonempty, and so the original sentence

$$(\exists x_1)(\forall x_2)\, x_1 \lhd^* x_2$$

is determined to be true, as we would have hoped!

A.3. **Some Minimization Examples.**

A.3.1. $\neg(\exists X_2)[X_1 \subseteq X_2]$. Consider the determinized automaton $\mathfrak{A}$ for the formula $\neg(\exists X_2)\, X_1 \subseteq X_2$, given in Figure 14. If we limit automaton $\mathfrak{A}$ to the reachable states, we get the automaton $\mathfrak{A}'$ exhibited in Figure 15.

$$\mathfrak{A}' = (A', \alpha', A'_F)$$
$$A' = \{\{a_0\}, \{a_0, a_s\}\}$$
$$A'_F = \emptyset$$
$$\alpha'(\lambda) = \{a_0\}$$

$$\begin{array}{ll}
\alpha'(\underline{0})(\{a_0\}, \{a_0\}) = \{a_0\} & \alpha'(\underline{1})(\{a_0\}, \{a_0\}) = \{a_0, a_s\} \\
\alpha'(\underline{0})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\} & \alpha'(\underline{1})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\} \\
\alpha'(\underline{0})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\} & \alpha'(\underline{1})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\} \\
\alpha'(\underline{0})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\} & \alpha'(\underline{1})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\}
\end{array}$$

FIGURE 15. Restriction to reachable states.

The congruences are very simple ($f \in \mathcal{ET}(\mathfrak{A}')$):

$$\{a_0\} \equiv_0 \{a_0, a_s\} \tag{37}$$
since both are final.
$$\{a_0\} \equiv_1 \{a_0, a_s\} \tag{38}$$
since (37) holds and $f(\{a_0\})$ and $f(\{a_0, a_s\})$ are in $\equiv_0$.

So we cannot distinguish $\{a_0\}$ and $\{a_0, a_s\}$, and the resulting automaton is $\mathfrak{A}''$, see Figure 16.

$$\mathfrak{A}'' = (A'', \alpha'', A''_F)$$
$$A'' = \{[\{a_0\}]\}$$
$$A''_F = \emptyset$$
$$\alpha''(\lambda) = [\{a_0\}]$$
$$\alpha''(\{\underline{0}, \underline{1}\})([\{a_0\}], [\{a_0\}]) = [\{a_0\}]$$

FIGURE 16. Final output of minimization.

This automaton really reflects the intuitions about a minimal automaton which recognizes the empty language. The only state we need is the sink state and we stay in it, no matter what happens. If we complement it, the only state becomes final and everything is recognized.

A.3.2. $\neg(\exists X_2)\,[\neg X_1 \subseteq X_2]$. Now we consider the same nucleus under universal quantification. Letting $\mathfrak{B}$ be the automaton for

$$\neg(\exists X_2)\,[\neg X_1 \subseteq X_2]$$

we have

$$\begin{aligned} \mathfrak{B} &= (B, \beta, B_F) \\ B &= \{\emptyset, \{a_0\}, \{a_s\}, \{a_0, a_s\}\} \\ B_F &= \{\emptyset, \{a_0\}\} \end{aligned}$$

The full transition table is identical to the one for the automaton in example (A.3.1).

The same states are reachable, i.e., $\mathfrak{B}'$ looks as in Figure 17.

$$\begin{aligned} \mathfrak{B}' &= (B', \beta', B'_F) \\ B' &= \{\{a_0\}, \{a_0, a_s\}\} \\ B'_F &= \{\{a_0\}\} \\ \beta'(\lambda) &= \{a_0\} \end{aligned}$$

$$\begin{array}{ll} \beta'(\underline{0})(\{a_0\}, \{a_0\}) = \{a_0\} & \beta'(\underline{1})(\{a_0\}, \{a_0\}) = \{a_0, a_s\} \\ \beta'(\underline{0})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\} & \beta'(\underline{1})(\{a_0\}, \{a_0, a_s\}) = \{a_0, a_s\} \\ \beta'(\underline{0})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\} & \beta'(\underline{1})(\{a_0, a_s\}, \{a_0\}) = \{a_0, a_s\} \\ \beta'(\underline{0})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\} & \beta'(\underline{1})(\{a_0, a_s\}, \{a_0, a_s\}) = \{a_0, a_s\} \end{array}$$

FIGURE 17. Automaton $\mathfrak{B}$ restricted to its reachable states.

Since $\{a_0\}$ is final, but $\{a_0, a_s\}$ is not, we cannot refine this automaton any further. Again the resulting automaton reflects the intuition that the only set which is a subset of every other set is the empty set. This is achieved by going into the sink state $\{a_0, a_s\}$ as soon as one symbol $\underline{1}$ is read.

REFERENCES

Backofen, R., Rogers, J. and Vijay-Shanker, K. (1995). A first-order axiomatization of the theory of finite trees, *Journal of Logic, Language, and Information* **4**: 5–39.

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

Blackburn, P. and de Rijke, M. (1994). Zooming in, zooming out, *CLAUS report 48*, University of the Saarland.

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

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

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

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.

Morawietz, F. (1997). Monadic second order logic, tree automata and constraint logic programming, *SFB 340-Report 86*, University of Tübingen.

O'Keefe, R. (1990). *The Craft of Prolog*, MIT Press, Cambridge, Mass.

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

Rabin, M. O. (1970). Weakly definable relations and special automata, *in* Y. Bar-Hillel (ed.), *Mathematical Logic and Foundations of Set Theory*, North-Holland, Amsterdam, pp. 1–23.

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.

SIC (1995). *SICStus Prolog User's Manual*, release 3 # 2 edn.

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.

Ullman, J. D. (1988). *Principles of Database and Knowledge-base systems*, Vol. 1, Computer Science Press.

Seminar für Sprachwissenschaft, SFB 340, Projekt A8, Universität Tübingen,
Wilhelmstrasse 113, D-72074 Tübingen, Germany
E-mail address: {frank,cornell}@sfs.nphil.uni-tuebingen.de