Tree Set Automata for Prolog Mode Analysis

Nathaniel Wesley Filardo and Jason Eisner

No Institute Given

Abstract. We present our work towards a formalism useful for static analysis of pure Prolog programs. Semantically, such programs reason about trees, but the solvers actually manipulate *sets* of trees. Thus, at compile time we wish to circumscribe the set of possibilities—which trees may arise semantically ("type analysis"), but also which tree sets may arise procedurally ("mode analysis"). Our proposed solution is a class of automata in which each accepting path represents a *set* of trees. We discuss the automaton operations that arise during static analysis. We build atop rigid tree automata, which we regard as adequate for a simple Prolog type system: they permit disjunction and recursion (for algebraic data types on trees) as well as limited forms of equality constraints (enforced by repeated variables in Prolog). We then "lift" this formalism to handle tree sets. Disjunction and recursion can now capture uncertainty about the tree set, and equality encodes which Prolog variables are identical.

1 Prolog

A Prolog program—like a tree automaton—is a description of a set of trees, called the provable **terms**. (We will use "tree" and "term" interchangeably.) Each line of the program is a proof rule that justifies the acceptance of one tree (the **head**) if a finite set of other trees (the **body**) are all accepted.[1] A proof rule may apply in infinitely many settings if it contains **variables**, denoted by capitalized identifiers, which are universally quantified over all possible terms.

The set of "possible terms" $\mathcal{TF}$ is called the **Herbrand universe** (or **term algebra**), and is built up from a signature, $\mathcal{F}$, of symbols-with-arity.[2] E.g., , if $\mathcal{F} = \{\texttt{nil}/0, \texttt{cons}/2, \texttt{length}/2, \ldots\}$, then `cons(nil,length(nil,nil))` is a possible term. That said, a program that treats `cons` as a list constructor should never construct this term—something that we would wish to verify statically.

Each proof rule in a Prolog program is a **Horn clause** (with variables) of the form `Head :- Subgoals` or simply `Head`. The `:-` symbol is read "if." For example, one may define a $\texttt{length}\langle \texttt{X}, \texttt{Y} \rangle$ relation between "cons" lists and natural numbers by two rules:[3] `length([],0)` and `length(L,N) :- L = [H|T], length(T,M), N is M+1`. The first simply relates the empty list and the number `0`, while the other can be read inductively: the list `L` has length `N` if `L` begins

[1] In this paper we focus on "pure" Prolog. Our static analysis techniques would also apply to full Prolog, or to our own pure logic language Dyna [3].

[2] Often, $\mathcal{F}$ is defined to have primitive constants, e.g., numbers, as symbols of arity 0.

[3] A **cons list** is a term built up from the symbols `nil`/0 and `cons`/2; the first child of a `cons` is the first element of the list and the other is the rest of the list. For brevity, $[] \stackrel{\text{def}}{=} \texttt{nil}/0$, $[h|t] \stackrel{\text{def}}{=} \texttt{cons}\langle h, t \rangle$, $[a, b|t] \stackrel{\text{def}}{=} \texttt{cons}\langle a, \texttt{cons}\langle b, t \rangle \rangle$, etc.

with `H` and has tail `T`, and `N` is one more than the length `M` of `T`. The two rules above define the set of provable terms to include

$$S = \{\texttt{length}\langle[],0\rangle, \texttt{length}\langle[h|t],n+1\rangle \mid h \in \mathcal{TF}, \texttt{length}\langle t,n\rangle \in S\}$$
$$= \{\texttt{length}\langle[],0\rangle, \texttt{length}\langle[x_1],1\rangle, \texttt{length}\langle[x_1,x_2],2\rangle, \ldots \mid x_i \in \mathcal{TF}\}.$$

Each term $t \in S$ can be justified by instantiating variables in one of the two rules to obtain t or t `:-` $t_1, t_2, \ldots$ where $t_1, t_2, \ldots$ are themselves provable.[4]

Let us take a moment to introduce notation. We write $t{\downarrow_{2.3}}$ (for example) to denote t's second child's third child; the subscript 2.3 is called an **address** and in general is a finite sequence π of counting numbers. We lift this **projection** operator to sets of trees: $\tau{\downarrow_\pi} \stackrel{\text{def}}{=} \{t{\downarrow_\pi} \mid t \in \tau\}$. One may **specialize** a set of trees τ to a subset satisfying a particular projection: $\tau[\sigma/\pi] \stackrel{\text{def}}{=} \{t \in \tau \mid t{\downarrow_\pi} \in \sigma\}$. A rule R can itself be represented as a set of trees; for example, our recursive `length` rule is $\{\texttt{rule}\langle\texttt{length}\langle l,n\rangle, \texttt{subgoals}\langle\texttt{=}\langle l, \texttt{cons}\langle h,t\rangle\rangle, \texttt{length}\langle t,m\rangle, \texttt{is}\langle n, \texttt{+}\langle m,1\rangle\rangle\rangle\rangle \mid h,l,m,n,t \in \mathcal{TF}\}$. We can now formally state that a Prolog program $\mathcal{P}$ accepts a term t via a rule R with k subgoals, encoded as above, iff there exists $r \in R[\{t\}/1]$ such that $r{\downarrow_{2,i}}$ is also accepted by $\mathcal{P}$ for each $i \in \{1,\ldots,k\}$.

An Prolog solver attempts to determine which provable terms match a given **query**, specified by a set E. Semantically, it finds the **solution** $\Psi(E) = \bigcup_{R\in\mathcal{P}} \Psi_R(E)$, where each $\Psi_R(E) \subseteq E$ gives all terms in E that can be justified by rule R. The solver finds $\Psi_R(E)$ by successive specializations of rule R, which involve recursive queries of R's subgoals. It begins by unifying the rule head with the query, obtaining the specialized rule $R'_0 = R[E/1]$. It then uses the first subgoal $R'_0{\downarrow_{2.1}}$ as a query to obtain the solution $\alpha_1 = \Psi_{R'_0{\downarrow_{2.1}}}$, with which it further refines the rule to obtain $R'_1 = R'_0[\alpha_1/2.1]$; this process continues with the second subgoal, obtaining $\alpha_2 = \Psi_{R'_1{\downarrow_{2.2}}}$, and on, until it has solved all k subgoals and has computed $R'_k = R[E/1][\alpha_1/2.1]\cdots[\alpha_k/2.k]$. With all subgoals specialized to their provable subsets, $R'_k{\downarrow_1}$ is exactly the set of trees justified by R, $\Psi_R(E)$.

The solver strategy described above presupposes that a query E returns a single object. For computational reasons, a traditional Prolog solver actually returns the set $\Psi(E)$ as a stream $\mathcal{S}(E)$ of **packets**—sets whose union is $\Psi(E) \subseteq E$. Traditionally, each packet A is a **non-ground term**, a simple and constrained way to express a set of trees. Non-ground terms are not closed under finite or infinite union, so a stream of packets is more expressive than a single packet. To be precise, the standard **backtracking search** solver returns the stream

$$\mathcal{S}_E = \bigoplus_R \mathcal{S}_R(E) = \bigoplus_R \bigoplus_{\substack{\alpha_1\in\mathcal{S}_{R'_0{\downarrow_{2.1}}} \\ R'_1 = R'_0[\alpha_1/2.1]}} \cdots \bigoplus_{\substack{\alpha_n\in\mathcal{S}_{R'_{n-1}{\downarrow_{2.n}}} \\ R'_n = R'_{n-1}[\alpha_n/2.n]}} R'_n{\downarrow_1} \qquad (1)$$

where $\bigoplus$ denotes stream concatenation and $\mathcal{S}_R(E)$ covers $\Psi_R(E)$. A rule with k subgoals thus creates k nested loops to answer $\Psi_R(E)$.

[4] This definition of `length`/2 is straightforward but limited relative to Prolog standard libraries' `length`/2. Prolog always solves subgoals left to right, so while the query `length([8,9],N)` will succed and bind `N` to 2, the query `length(L,2)` will recurse to `length(X,Y)` and will not terminate, rather than producing $\{\texttt{length}\langle[a,b],2\rangle \mid a,b\}$. One of the motivations of static analysis of Prolog is automated *subgoal reordering* to derive different procedures for differently instantiated queries; see [7].

Some **built-in** queries—for example, queries about arithmetic facts—are handled specially by the solver. The solver in this case returns a stream of results computed by some library procedure rather than by the program rules.

A Prolog solver may refuse to answer certain built-in or user-defined queries, typically because the answer streams would be infinitely long and would cause the solver to not terminate. Consider executing the subgoal sequence `B=2, C=1, A is B+C`: by the time the solver invokes the `is`/2 subgoal query, that query has been specialized to `A is 2+1`, which easily returns $\alpha_3 = \{$`3 is 2+1`$\}$. If, however, the subgoal sequence had been reversed, the *unspecialized* `is`/2 query would have been obligated to produce an infinite answer summarizing all addition facts. Encoding that answer into finitely many packets (so that the solver terminates) would require a highly expressive representation for packets (an approach adopted in constraint logic programming [2]). Most Prolog implementations simply generate an **instantiation fault** (a runtime exception) if this kind of problematic query arises at runtime. It would be desirable to statically exclude the possibility of instantiation faults.

2 A Simple Static Analysis of Prolog Programs

To statically analyse a program is to construct or check upper bounds on its runtime behavior. Prolog static analysis has a rich history stretching back decades: [1] (1993) claims that "modes" as we know them are due to [6] (1981). Our effort follows most closely that of the Mercury project [7] (2003) and can be thought of largely as an automata-theoretic re-development of that work, using tree automata to track variables' types (not supported in Mercury) and handling repeated variables via equality checks in the tree automaton.

The input to our static analysis is a Prolog program $\mathcal{P}$ augmented with **declarations** that describe the queries and packets that may arise at runtime: $\mathcal{E}$ is a set of **supported queries**, and $\mathcal{A}$ is a set of **possible packets** that might answer them. These declarations claim that for each $E \in \mathcal{E}$, the runtime engine will produce an answer stream $\mathcal{S}(E)$ consisting of packets $A \in \mathcal{A}$ (where each $A \subseteq E$). Note that $\mathcal{E}$ and $\mathcal{A}$ are both (possibly infinite) sets of (possibly infinite) sets of trees. They represent uncertainty about which tree sets—that is, which queries or packets—will arise at runtime.[5] We will represent them by tree set automata.

With that information at hand, we seek to certify $(\mathcal{P}, \mathcal{E}, \mathcal{A})$ as **well-moded**. That is, for any query $E \in \mathcal{E}$ and any rule $R \in \mathcal{P}$, the stream $\mathcal{S}_R(E)$ should consist entirely of packets in $\mathcal{A}$. To prove this, we assume inductively that it is true for the subgoal queries that are used to generate this stream.

In practice, $\mathcal{P}$ may be augmented at runtime with additional facts that form the input to the program. A **fact** (or axiom) is a rule with an empty body ($k = 0$ subgoals). For example, an input graph may be specified at runtime by a set of

[5] Thus, they resemble the **instantiation states (insts)** of Mercury [7] (although Mercury in effect specifies a separate pair of insts $(\mathcal{E}_i, \mathcal{A}_i)$ for each operational procedure in the compiled program). By contrast, a stream is a set of sets of trees that *all* arose at runtime. Static analysis of determinism would attempt to determine the possible cardinalities of a stream at compile time.

$\texttt{edge}\langle x, y\rangle$ facts. A fact H is only legal as input if it satisfies $(\forall E \in \mathcal{E})E \cap H \in \mathcal{A}$, ensuring that the augmented program remains well-moded.

We lift our tree-set operators to insts: $T_1[T_2 /\!\!/ \pi] \stackrel{\text{def}}{=} \{\tau_1[\tau_2/\pi] \mid \tau_1 \in T_1, \tau_2 \in T_2\}$ represents at compile-time the possible results of a runtime specialization of a rule, and $T{\downarrow_\pi} \stackrel{\text{def}}{=} \{\tau{\downarrow_\pi} \mid \tau \in T\}$ projects through both set levels. Those two operations, and singleton formation and a few decision procedures, form the tooling needed for our analysis. Concretely, for each rule $R \in \mathcal{P}$ with k subgoals,

- Check that $(\forall i \in [1, k])\{R\}[\mathcal{E} /\!\!/ 1][\mathcal{A} /\!\!/ 2.1] \cdots [\mathcal{A} /\!\!/ 2.(i-1)]{\downarrow_{2.i}} \subseteq \mathcal{E}$. That is, check that the i^{th} subgoal query will always be a legal query.
- Finally, check that $\{R\}[\mathcal{E} /\!\!/ 1][\mathcal{A} /\!\!/ 2.1] \cdots [\mathcal{A} /\!\!/ 2.k]{\downarrow_1} \subseteq \mathcal{A}$. That is, check that the packet obtained as the head of the fully specialized rule will always be a legal packet.

We now turn our attention to the implementation of these operations and tests. We remark that it would be possible to apply similar techniques to more sophisticated solvers, which might dynamically reorder or restructure subgoals or might go beyond backtracking search to (also) use forward chaining.

3 Tree Set Automata

Regular Tree Set Automata We begin with a notion of Regular Tree Set Automata, built using two disjoint sets of states, **tree states** Q and **set states** $\hat{Q}$, where $\hat{Q}_F \subseteq \hat{Q}$ are the **final states**. A tree state accepts trees, while a set state accepts sets of trees (queries or packets, which in traditional Prolog take the form of non-ground terms; our formalism is more expressive). Transition rules in a TSA come in 6 forms (where $q, q_i \in Q$ and $\hat{q}, \hat{q}_i \in \hat{Q}$):

- $\texttt{f}\,\langle q_1, \ldots, q_n\rangle \to q_0$ is an ordinary TA rule, which asserts that $\mathcal{L}(q_0) \supseteq \{\texttt{f}\langle t_1, \ldots, t_n\rangle \mid t_i \in \mathcal{L}(q_i)\}$, a set of trees.
- $\textsc{free}\, q \to \hat{q}$ asserts that $\mathcal{L}(\hat{q}) \supseteq \{\mathcal{L}(q)\}$, a set of tree sets. Thus, $\hat{q}$ accepts the "free variable" of type q (a query or packet representing all of $\mathcal{L}(q)$). We will require that $\hat{q}$ and rules (transitively) contributing to its definition describe a *top-down deterministic* automaton.
- $\textsc{ground}\, q \to \hat{q}$ asserts that $\mathcal{L}(\hat{q}) \supseteq \{\{t\} \mid t \in \mathcal{L}(q)\}$. Thus, $\hat{q}$ accepts any ground term of type q (a query or packet representing a specific tree in $\mathcal{L}(q)$, although *which* term will not be known until runtime). We need no restriction on the shape of the automaton description of q.
- $\textsc{subtype}\, q \to \hat{q}$ asserts that $\mathcal{L}(\hat{q}) \supseteq \wp(\mathcal{L}(q))$. Thus, $\hat{q}$ accepts *any* specialization of the free variable of type q (which must also, as with $\textsc{free}$, be top-down deterministic). It may occasionally be necessary to include such sets in $\mathcal{A}$, when the program $\mathcal{P}$ is not well-moded under any tighter declaration that is expressible in our formalism.
- $\textsc{bound}\, \texttt{f}\,\langle \hat{q}_1, \ldots, \hat{q}_n\rangle \to \hat{q}_0$ asserts that $\mathcal{L}(\hat{q}_0) \supseteq \{\{\texttt{f}\langle t_1, \ldots, t_n\rangle \mid t_i \in \tau_i\} \mid \tau_i \in \mathcal{L}(\hat{q}_i)\}$. The name "bound" comes from [7] and signifies an instantiation state between free and ground: the runtime packet may represent one term or many, subject to $\hat{q}_1, \ldots, \hat{q}_n$, but all terms in the packet have root symbol $\texttt{f}$.
- $\textsc{bound}_\subset\, \texttt{f}\,\langle \hat{q}_1, \ldots, \hat{q}_n\rangle \to \hat{q}_0$ is necessary for specialization closure; it asserts that $\mathcal{L}(\hat{q}_0) \supseteq \{\alpha' \mid \alpha \in \{\{\texttt{f}\langle t_1, \ldots, t_n\rangle \mid t_i \in \tau_i\} \mid \tau_i \in \mathcal{L}(\hat{q}_i)\}, \emptyset \subsetneq \alpha' \subseteq \alpha\}$.

Primitive types can be handled via "built-in" tree states, e.g., define $\mathcal{L}(q_{\text{int}}) = \mathbb{Z}$. Recognizable tree *sets* then include $\{\{\mathtt{f}\,\langle n\rangle : n \in \mathbb{Z}\}, \{\mathtt{g}\,\langle n\rangle : n \in \mathbb{Z}\}\}$ (allowing two possible packets, via two BOUND transitions to a final state); $\{\{\mathtt{f}\,\langle n\rangle : n \in \mathbb{Z}\} \cup \{\mathtt{g}\,\langle n\rangle : n \in \mathbb{Z}\}\}$ (allowing only one possible packet—which includes both `f` and `g` terms—via one FREE transition); and $\{\{\mathtt{f}\,\langle n\rangle\} : n \in \mathbb{Z}\} \cup \{\{\mathtt{g}\,\langle n\rangle\} : n \in \mathbb{Z}\}$ (allowing infinitely many possible packets, via one or two GROUND transitions).

Inner-Rigid Tree Set Automata We now extend Regular Tree Set Automata to handle global equality constraints, borrowing from Rigid Tree Automata [5]. A particular tree t in a set τ accepted by a TSA can be given a run labeling: every position is associated with at most one Q and at most one $\hat{Q}$; only the nodes at transitions (e.g., FREE) will have both. We extend the above definition with **inner-rigid states**, both tree, $Q_R \subseteq Q$, and set, $\hat{Q}_R \subseteq \hat{Q}$, to obtain Inner-Rigid Tree Set Automata (IRTSA). The sets accepted by an IRTA are of trees whose runs obey the rigidity requirement: for each $q \in Q_R \cup \hat{Q}_R$, all nodes in a run labeled with q are the roots of equal subtrees. These rigid states allow us to encode the reuse of variables within a Prolog rule.

3.1 Operations on (IR)TSAs

Outer-Union and Outer-Intersection Given a (IR)TSA A, we seek to form two (R)TAs (resp.) that describe the union and intersection of the sets accepted by A, denoted $\bigcup A$ and $\bigcap A$. Union is straightforward as the resulting automata has the same shape as the input (IR)TSA. Intersection is a little more complicated, but tractable because set states cyclic through BOUND have empty intersection.

Emptiness Testing (IR)TSA inherit emptiness testing from (R)TAs: a depth-first traversal state-marking algorithm suffices to label each state (be it either tree or set) as (non-)empty. The only tweak is that FREE q *always* accepts a set, regardless of q's emptiness, and GROUND q accepts a set iff q does.

Acceptence of the Empty Set (IR)TSA may use a very similar algorithm to that of emptiness testing to determine the acceptance of the empty set: FREE $q \ni \emptyset$ iff q is empty, and BOUND $\mathtt{f}\,\langle q_1, \ldots\rangle \ni \emptyset$ iff any of its q_i does.

Projection Projection is straightforward: having removed BOUND rules which accept no sets, start from the root states of a TSA, and walk the projection path to find the new root states of the projected automaton.

Unification In order to implement $A_1[A_2/\!\!/\pi]$, it suffices to implement **unification**, $\mathcal{L}(A_1 \between A_2) \stackrel{\text{def}}{=} \{\tau_1 \cap \tau_2 \mid \tau_i \in \mathcal{L}(A_i)\}$. See § A for a sketch of the algorithm.

Subset Testing Testing whether $\mathcal{L}(A) \subseteq \mathcal{L}(A')$ for two regular TSAs can proceed by a straightforward recursive algorithm, relying on the subset tests from regular TAs. For IRTSAs, however, no decidable strategy exists as RTAs do not have decidable subset tests. We hope to develop a sufficiently powerful *3-way* test which, if it reports yes or no, is correct, but may declare defeat.

4 Extensions and Future Work

Generalized Free Variable Types Relaxing the top-down determinism requirement on free variables, and/or generalizing beyond RTA types, requires restruc-

turing of the TSA. If, for example, we wished to have types from (a subclass of) TAs with local equality constraints (TAEC), we would need to equip BOUND with local equality constraints along the same lines. As Rigid Tree Automata are not a perfect fit for our needs (they are, e.g., not Kleene closed and do not have decidable subset testing), we are actively casting around for other classes.

Aggregation The authors are working on a logic programming language, Dyna, with semantics more general than (pure) Prolog's [3]. One of the core aspects of this language is that programs describe a *map* from terms (keys) to terms (values), rather than a set of provable terms. The values are derived from associative-commutative **aggregation** across Horn *expressions* which give an aggregand for each provable specialization of the rule. This need to aggregate values implies a need to reshape packets; the passive flow of results to answers, as in Equation 1, no longer holds. Modeling this behavior will require new operations of our TSAs.

5 Related Work

Mercury Our go-to guide for all things pertaining to expressive mode analysis of Prolog, Mercury, specifically [7], is slightly less expressive in what insts its machinery can describe, but it tracks additional data about the computation. Its insts vocabulary seems to be borne of two key restrictions: the use of top-down-deterministic inner-rigid tree set automata—though it does not discuss its formalism in terms of automata—and the requirement that all FREE variables accept $\mathcal{TF}$—leaving type analysis to another component of the system. It does not have our notion of SUBTYPE because the underlying Mercury runtime uses non-ground terms at runtime. On the other hand, its analysis tracks *determinism* of queries—capturing bounds on the size of the $\mathcal{S}_E$ streams—as well as *uniqueness* of variables—enabling elision of data copies in favor of efficient destructive mutation, at runtime. We hope to be able to lift these features into further refined automata classes.

TATA Tree Set Automata The canonical text on Tree Automata [4, ch. 5] introduces a concept termed a "generalized tree set automaton". These automata recognize "E-valued $\mathcal{F}$-generalized tree sets", i.e., maps $g : \mathcal{H} \to E$. The set E generalizes the use of $\mathbf{2}$, which would make the function g an indicator function of a set of $\mathcal{F}$-trees. These automata are not suitable for our use case as they cannot recognize, for example, singleton sets of recursive structure, e.g., what we would denote as GROUND q_l with $\{\texttt{nil}\,\langle\rangle \to q_l, \texttt{cons}\,\langle q, q_l\rangle \to q_l\}$ and $|\mathcal{L}(q)| > 1$. They can, however, recognize FREE and SUBTYPE constructions, as well as proper-subsets.

6 Conclusion

We have presented the beginnings of an automata-theoretic formalism for analysis of Prolog-like logic programs. While many details and extensions remain to be worked out, we hope to interest the tree automata community and welcome any feedback.

A Unification Cases

An algorithm for unification can largely be deduced from its set-theoretic definition on possible cases:

- $(T \cup T') \between S = (T \between S) \cup (T' \between S)$; this forms the basis of handing of multiple rules targeting the same states within TSAs being unified.
- $\textsc{free}\, q \between \textsc{free}\, q' = \textsc{free}\, (q \cap q')$ appeals to intersection of the underlying TA family (e.g., RTA), as do $\textsc{free} \between \textsc{subtype}$ and $\textsc{subtype} \between \textsc{subtype}$.
- $\textsc{bound}\, \mathtt{f}\, \langle \hat{q}_1, \ldots \rangle \between \textsc{bound}\, \mathtt{f}\, \langle h q'_1, \ldots \rangle = \textsc{bound}\, \mathtt{f}\, \langle \hat{q}_1 \between \hat{q}'_1, \ldots \rangle$; for mismatched symbols or arities, the answer is $\textsc{free}\, \emptyset$.
- $\textsc{ground}\, q \between q$ is $\textsc{ground}$ of an automaton recognizing $\mathcal{L}(q) \cap \bigcup \mathcal{L}(q)$ with the addition of $\emptyset$ being accepted if $\mathcal{L}(q) \not\subseteq \bigcup \mathcal{L}(q)$.[6]

The last two cases, of unifying $\textsc{free}\, q$ (resp. $\textsc{subtype}\, q$) with $\textsc{bound}\, \mathtt{f}\, \langle T_1, \ldots \rangle$, relies on q being top-down determinisitc: there is at most one rule targeting q that applies to trees whose roots are labeled with $\mathtt{f}/k$; selecting $\mathtt{f}\, \langle q_1, \ldots, q_k \rangle$ as its LHS, the answer becomes $\textsc{bound}\, \mathtt{f}\, \langle \textsc{free}\, q_1 \between T_1, \ldots \rangle$ (resp. $\textsc{bound}_{\subset}\, \mathtt{f}\, \langle \cdots \rangle$).

References

1. Krzysztof R. Apt and Elena Marchiori. Reasoning about prolog programs: From modes through types to assertions. Technical report, Amsterdam, The Netherlands, The Netherlands, 1993.
2. Alain Colmerauer. Opening the prolog iii universe. *BYTE*, 12(9):177–182, August 1987.
3. Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2011. Longer version available as tech report.
4. Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Loding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*.
5. Florent Jacquemard, Francis Klay, and Camille Vacher. Rigid tree automata and applications. *Information and Computation*, 209(3):486–512.
6. C.S. Mellish and University of Edinburgh. Department of Artificial Intelligence. The automatic generation of mode declarations for prolog programs. 1981.
7. David Overton. *Precise and expressive mode systems for typed logic programming languages*. PhD thesis, University of Melbourne, 2003.

[6] While RTA do not have decidable $\subseteq$ testing, for this case relatively little precision is lost by assuming $\emptyset$ to be accepted: $\emptyset$ does not alter the *semantics* of the Prolog program, as it carries no trees.