Efficient Inclusion Checking for Deterministic Tree Automata and DTDs

Jérôme Champavère [a,b], Rémi Gilleron [a,b], Aurélien Lemay [a,b] and Joachim Niehren [a,*]

[a] *Inria, Lille, France*

[b] *Université de Lille, France*

Abstract

We present a new algorithm for testing language inclusion $L(A) \subseteq L(B)$ between tree automata in time $O(|A|*|B|)$ where B is deterministic. We extend this algorithm for testing inclusion between automata for unranked trees A and deterministic DTDs D in time $O(|A| * |\Sigma| * |D|)$. Previous algorithms were less efficient.

1 Introduction

Language inclusion for tree automata is a basic decision problem that is closely related to universality and equivalence [1,2,3]. Tree automata algorithms are generally relevant for XML document processing [4,5,6,7]. Regarding inclusion checking, a typical application is inverse type checking for tree transducers [8]. Another one is schema-guided query induction [9], the motivation for the present study. There, candidate queries produced by the learning process are to be checked for consistency with deterministic DTDs, such as for HTML.

We investigate language inclusion $L(A) \subseteq L(B)$ for tree automata A and B under the assumption that B is (bottom up) deterministic, not necessarily A. Without this assumption the problem becomes DEXPTIME complete [3]. Deterministic language inclusion still subsumes universality of deterministic tree automata $L(B) = T_\Sigma$ up to a linear time reduction, as well as equivalence

* Corresponding author.

Email addresses: `www.grappa.univ-lille3.fr/~champavere` (Jérôme Champavère), `www.grappa.univ-lille3.fr/~gilleron` (Rémi Gilleron), `www.grappa.univ-lille3.fr/~lemay` (Aurélien Lemay), `www.grappa.univ-lille3.fr/~niehren` (Joachim Niehren).

June 11, 2008

of two deterministic automata $L(A) = L(B)$. The converse might be false, i.e., we cannot rely on polynomial time equivalence tests, as for instance, by comparing number of solutions [2] or minimal deterministic tree automata.

In the case of standard tree automata for ranked trees, the well-known naive algorithm for inclusion goes through complementation. It first computes an automaton B^c that recognizes the complement of the language of B, and then checks whether the intersection automaton for B^c and A has a nonempty language. The problematic step is the completion of B before complementing its final states, since completion might require to add rules for all possible left-hand sides. The overall running time may thus become $O(|A|*|\Sigma|*|B|^n)$, which is exponential in the maximal rank n of function symbols in the signature Σ.

The next idea is to reduce the maximal arity of function symbols in ranked trees to 2. It is folklore that one can transform ranked trees into binary trees, and automata correspondingly. The problem, however, is to preserve determinism, while the size of automata should grow at most linearly. We can solve this problem by using Currying for encoding of unranked to binary trees, as proposed for stepwise tree automata [10,1]. Thereby we obtain an inclusion test for the ranked case in time $O(|A| * |\Sigma| * |B|^2)$. This is still too much in practice with DTDs, where A and B may be of size 500 and Σ of size 100.

Our first contribution is a more efficient algorithm for stepwise tree automata on binary trees that tests inclusion in time $O(|A| * |B|)$ if B is deterministic. This upper bound is independent of the size of the signature, even though we do not assume Σ to be fixed. By using Currying, the result can be lifted to standard tree automata for ranked trees over arbitrary signatures. It equally lifts to inclusion between stepwise tree automata for unranked trees.

As a second contribution, we show how to test inclusion between stepwise tree automata A for unranked trees and deterministic DTDs D in time $O(|A| * |\Sigma| * |D|)$. Determinism is required by the XML standards. Our algorithm first computes the Glushkov automata of all regular expressions of D in time $O(|\Sigma| * |D|)$, which is possible for deterministic DTDs [11]. The second step is more tedious. We would like to transform the collection of Glushkov automata to a deterministic stepwise tree automaton of the same size. Unfortunately, this seems difficult to achieve, since the usual construction of Martens & Niehren [12] eliminates ϵ-rules on the fly, which may lead to a quadratic blowup of the number of rules (not the number of states).

We solve this problem by introducing deterministic *factorized tree automata*. These are stepwise tree automata with ϵ-rules, which represent deterministic stepwise tree automata more compactly, and in particular the collection of Glushkov automata of a DTD with linear size. Factorized automata have two sorts of states, which play the roles of hedge states and tree states in alterna-

tive automata notions for unranked trees [13,4]. The difficulty is to define an appropriate notion of determinism for factorized tree automata, and to adapt the inclusion test when B is a deterministic factorized tree automaton.

Our results equally apply if A is a hedge automaton [1,14,15], with finite word automata for horizontal languages, since such hedge automata can be translated in linear time to stepwise tree automata. Note however, that the notion of determinism for hedge automata is unsatisfactory [12] so that we cannot choose B to be a deterministic hedge automaton even if the horizontal language is defined by a deterministic finite word automaton.

The situation becomes slightly different if A is a tree automaton recognizing firstchild-nextsibling encodings of unranked trees, and D a DTD. The problem is that the conversion of A into a stepwise tree automaton may lead to a quadratic size increase. In this case, however, we can encode DTDs into top-down deterministic tree automata that recognize firstchild-nextsibling encodings of unranked trees, and reduce the inclusion problem to the case of inclusion in deterministic finite word automata. This yields a worst case running time of $O(|A] * |\Sigma| * |D|)$, too. As we show, the same algorithm applies if D is a deterministic extended DTD with restrained competition [16].

Related Work. Compared to the conference version [17], we have added complete proofs, new results on incrementality including early failure detection, and experimental results. Meanwhile, the inclusion test presented here has been integrated into a system for schema-guided query induction [9], where it proves efficient in practice. Furthermore, we have added the alternative algorithm for inclusion in top-down deterministic automata and restrained competition extended DTDs (modulo firstchild-nextsibling encodings).

Heuristic algorithms for inclusion between non-deterministic schemas that avoid the high worst-case complexity were proposed by Tozawa & Hagiya [18]. The complexity of inclusion for various fragments of DTDs and extended DTDs was studied by Martens, Neven & Schwentick [19]. They assume the same types of language definitions on both sides. When applied to deterministic DTDs, the same complexity results seem obtainable. Our algorithm permits richer left-hand sides without increasing in complexity.

Outline. In Section 2, we reduce inclusion for ranked tree automata to the binary case. An efficient incremental algorithm for binary tree automata is given in Section 3. In Section 4, we introduce deterministic factorized tree automata and lift the algorithm for inclusion testing. In Section 5 we apply it to testing inclusion of automata for unranked trees in deterministic DTDs. Section 6 presents experimental results. Section 7 considers inclusion in top-down deterministic tree automata and restrained competition extended DTDs. Appendix A completes a remaining proof and B details the implementation.

2 Standard Tree Automata for Ranked Trees

We reduce the inclusion problem of tree automata for ranked trees [1] to the case of binary trees with a single binary function symbol.

A ranked signature Σ is a finite set of function symbols $f \in \Sigma$, each of which has an arity $n \geq 0$. A constant $a \in \Sigma$ is a function symbol of arity 0. A tree $t \in T_\Sigma$ is either a constant $a \in \Sigma$ or a tuple $f(t_1, \ldots, t_n)$ consisting of a function symbol f of arity n and n trees $t_1, \ldots, t_n \in T_\Sigma$.

A *tree automaton* (possibly with ϵ-rules) A over Σ consists of a finite set $sta(A)$ of states, a subset $fin(A) \subseteq sta(A)$ of final states, and a set $rul(A)$ of rules of the form $f(p_1, \ldots, p_n) \to p$ or $p' \xrightarrow{\epsilon} p$ where $f \in \Sigma$ has arity n and $p_1, \ldots, p_n, p, p' \in sta(A)$. We write $p' \xrightarrow{\epsilon}_A p$ iff $p' \xrightarrow{\epsilon} p \in rul(A)$, $\xrightarrow{\epsilon}{}^*_A$ for the reflexive transitive closure of $\xrightarrow{\epsilon}_A$, and $\xrightarrow{\epsilon}{}^{\leq 1}_A$ for the union of $\xrightarrow{\epsilon}_A$ and the identity relation on $sta(A)$.

The *size* $|A|$ of A is the sum of the cardinality of $sta(A)$ and the number of symbols in $rul(A)$, i.e., $\sum_{f(p_1,\ldots,p_n)\to p \in rul(A)} (n+2)$. The cardinality of the signature can be ignored, since our algorithms will not take unused function symbols into account. Every tree automaton A defines an evaluator $eval_A : T_{\Sigma \cup sta(A)} \to 2^{sta(A)}$ such that

$$eval_A(f(t_1, \ldots, t_n)) = \{p \mid p_1 \in eval_A(t_1), \ldots, p_n \in eval_A(t_n), \\ f(p_1, \ldots, p_n) \to p' \in rul(A),\ p' \xrightarrow{\epsilon}{}^*_A p\}$$

and $eval_A(p) = \{p\}$. A tree $t \in T_\Sigma$ is *accepted* by A if $fin(A) \cap eval_A(t) \neq \emptyset$. The *language* $L(A)$ is the set of trees accepted by A.

A tree automaton is (bottom-up) *deterministic* if it has no ϵ-rules, and if no two rules have the same left-hand side. It is complete if there are rules for all potential left-hand sides. It is well-known that deterministic complete tree automata can be complemented in linear time, by switching the final states.

Deterministic inclusion. We will study the deterministic inclusion problem for tree automata. Its input consists of a ranked signature Σ, a possibly non-deterministic tree automaton A with ϵ-rules, and a deterministic tree automaton B, both with signature Σ, and its output is the truth value of $L(A) \subseteq L(B)$.

We can deal with this problem by restriction to stepwise signatures $\Sigma_@$, which consist of a single binary function symbol @ and a finite set of constants $a \in \Sigma$. A stepwise tree automaton over binary trees [10] is a tree automaton over a stepwise signature. The alternative interpretation over unranked trees can be

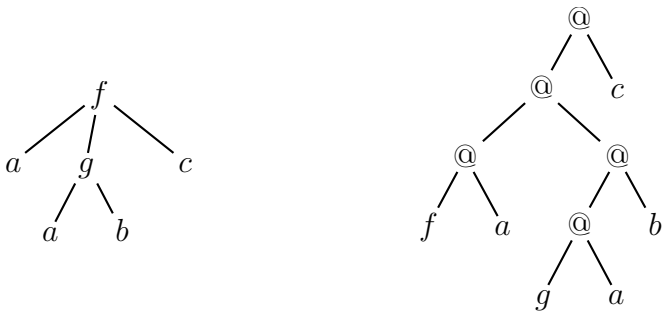

Figure 1. Currying the ranked tree $f(a, g(a, b), c)$ into the binary tree $f@a@(g@a@b)@c$.

$$\frac{f(q_1, \ldots, q_n) \to q \in rul(A) \qquad 1 \leq i < n}{\begin{array}{c} f \to f \in rul(step(A)) \\ fq_1 \ldots q_{i-1}@q_i \to fq_1 \ldots q_i \in rul(step(A)) \\ fq_1 \ldots q_{n-1}@q_n \to q \in rul(step(A)) \end{array}} \qquad \frac{a \to q \in rul(A)}{a \to q \in rul(step(A))}$$

Figure 2. Transforming ranked tree automata into stepwise tree automata.

obtained either algebraically or *via* binary encoding.

Proposition 1 *The deterministic inclusion problem for standard tree automata over ranked trees can be reduced in linear time to the deterministic inclusion problem for stepwise tree automata over binary trees.*

We first encode ranked trees into binary trees via Currying. Given a ranked signature Σ we define the corresponding signature $\Sigma_@ = \{@\} \uplus \Sigma$ whereby all symbols of Σ become constants. Currying is defined by a function $curry : T_\Sigma \to T_{\Sigma_@}$ which for all trees $t_1, \ldots, t_n \in T_\Sigma$ and $f \in \Sigma$ satisfies:

$$curry(f(t_1, \ldots, t_n)) = f@curry(t_1)@ \ldots @curry(t_n)$$

For instance, $f(a, g(a, b), c)$ is mapped to $f@a@(g@a@b)@c$, which is infix notation with left-most parenthesis for the tree $@(@(@(f, a), @(@(g, a), b)), c)$, as shown in Figure 1.

Now we encode tree automata A over Σ into stepwise tree automata $step(A)$ over $\Sigma_@$, such that the language is preserved up to Currying, i.e., such that $L(step(A)) = curry(L(A))$. The states of $step(A)$ are the prefixes of left-hand sides of rules in A, i.e., words in $\Sigma(sta(A))^*$:

$$sta(step(A)) = \{fq_1 \ldots q_i \mid f(q_1, \ldots, q_n) \to q \in rul(A),\ 0 \leq i \leq n\} \uplus sta(A)$$

The rules of $step(A)$ are given in Figure 2. They extend prefixes step by step by states q_i according to the rules of A. Since constants cannot be extended, we need to distinguish two cases.

Lemma 2 *The encoding of tree automata A over Σ into stepwise tree automata $step(A)$ over $\Sigma_@$ preserves determinism, the tree language modulo Currying, and the automata size up to a constant factor of* 3*.*

As a consequence, $L(A) \subseteq L(B)$ is equivalent to $L(step(A)) \subseteq L(step(B))$, and can be tested in this way modulo a linear time transformation. Most importantly, the determinism of B carries over to $step(B)$.

3 Stepwise Tree Automata for Binary Trees

We present our new algorithm for testing deterministic inclusion in the case of stepwise tree automata over binary trees. We start with a characterization of deterministic inclusion, express it by a Datalog program [20], and then turn it into an efficient algorithm. The idea for making the last step work is nontrivial.

3.1 Ground Datalog

The following efficiency theorem for ground Datalog will be fundamental to all what follows. Given a Datalog program P (without negation), we write $lfp(P)$ for its least fixed point semantics.

Theorem 3 (Efficiency of Ground Datalog [20]) *For every ground Datalog program P, the least fixed point semantics $lfp(P)$ can be computed in linear time $O(|P|)$ where the size $|P|$ is the number of symbols in P.*

This result holds even without any bound on the arity of the relation symbols of P, which will be very useful later on. If relation symbols of higher arities are used, the number of their arguments is accounted for by the size of P.

3.2 Characterization of Inclusion

Let A be a tree automaton over Σ. We call a state $p \in sta(A)$ *accessible* and write $A \models \mathsf{acc}(p)$ if there exists a tree $t \in T_\Sigma$ such that $p \in eval_A(t)$, and *co-accessible* if there exists a tree $C[p] \in T_{\Sigma \cup \{p\}}$ with a unique occurrence of p (i.e., a context with hole marker p) such that $eval_A(C[p]) \cap fin(A) \neq \emptyset$. For every term $s \in T_\Sigma$, we denote by $C[s] \in T_\Sigma$ the term obtained by replacing the unique occurrence of p in $C[p]$ by s.

We call A *productive* if all its states are accessible and co-accessible. Note that A can be rendered productive in linear time without altering its language.

$$\frac{a \to q \in \mathit{rul}(A)}{\mathsf{acc}(q).} \qquad \frac{q_1@q_2 \to q \in \mathit{rul}(A)}{\mathsf{acc}(q) :\!-\ \mathsf{acc}(q_1), \mathsf{acc}(q_2).} \qquad \frac{q \xrightarrow{\epsilon} q'}{\mathsf{acc}(q') :\!-\ \mathsf{acc}(q).}$$

$$\frac{q \in \mathit{fin}(A)}{\mathsf{coacc}(q).} \qquad \frac{q_1@q_2 \to q \in \mathit{rul}(A)}{\begin{array}{l}\mathsf{coacc}(q_1) :\!-\ \mathsf{coacc}(q), \mathsf{acc}(q_2).\\ \mathsf{coacc}(q_2) :\!-\ \mathsf{coacc}(q), \mathsf{acc}(q_1).\end{array}} \qquad \frac{q \xrightarrow{\epsilon} q'}{\mathsf{coacc}(q) :\!-\ \mathsf{coacc}(q').}$$

Figure 3. Accessible and co-accessible states of A.

This can be done by computing the least fixed point of the ground Datalog program in Figure 3. The number of rules of this program is linear in the size of A, so the least fixed point can be computed in linear time by Theorem 3. States that are not accessible or not co-accessible and all rules using them can be safely removed from A.

Let B be another automaton over Σ but without ϵ-rules. The *product* $A \times B$ has state set $\mathit{sta}(A) \times \mathit{sta}(B)$, and rules inferred as follows:

$$\frac{\begin{array}{c}a \to p \in \mathit{rul}(A)\\ a \to q \in \mathit{rul}(B)\end{array}}{a \to (p,q)} \qquad \frac{\begin{array}{c}p_1@p_2 \to p \in \mathit{rul}(A)\\ q_1@q_2 \to q \in \mathit{rul}(B)\end{array}}{(p_1,q_1)@(p_2,q_2) \to (p,q)} \qquad \frac{\begin{array}{c}p' \xrightarrow{\epsilon} p \in \mathit{rul}(A)\\ q \in \mathit{sta}(B)\end{array}}{(p',q) \xrightarrow{\epsilon} (p,q)}$$

We do not care about final states of $A \times B$ since these are useless in our characterization of inclusion.

Proposition 4 *Inclusion $L(A) \subseteq L(B)$ for productive stepwise tree automata A with ϵ-rules and deterministic stepwise tree automata B fails iff:*

`fail`$_0$*: there exists a rule $a \to p \in \mathit{rul}(A)$ but no state $q \in \mathit{sta}(B)$ such that $a \to q \in \mathit{rul}(B)$, or*

`fail`$_1$*: there are accessible states (p_1, q_1) and (p_2, q_2) of $A \times B$ and a rule $p_1@p_2 \to p \in \mathit{rul}(A)$ but no $q \in \mathit{sta}(B)$ such that $q_1@q_2 \to q \in \mathit{rul}(B)$, or*

`fail`$_2$*: some accessible state (p, q) of $A \times B$ satisfies $p \in \mathit{fin}(A)$ and $q \notin \mathit{fin}(B)$.*

Proof. For soundness, we suppose that one of the failure conditions holds, and show that some tree $t \in L(A)$ witnesses inclusion failure, i.e., $t \notin L(B)$.

`fail`$_0$. Let us consider a rule $a \to p \in \mathit{rul}(A)$ such that no rule $a \to q \in \mathit{rul}(B)$ exists. Since A is productive, state p is co-accessible, i.e., there exists a term $C[p] \in T_{\Sigma \cup \{p\}}$ with a single occurrence of p such that $\mathit{eval}_A(C[p]) \cap \mathit{fin}(A) \neq \emptyset$. Hence $C[a] \in L(A)$. But $C[a] \notin L(B)$ because there is no rule $a \to q \in \mathit{rul}(B)$.

`fail`$_1$. There exists $t_1 \in T_\Sigma$ such that $(p_1, q_1) \in \mathit{eval}_{A \times B}(t_1)$ by accessibility of (p_1, q_1) and there exists $t_2 \in T_\Sigma$ such that $(p_2, q_2) \in \mathit{eval}_{A \times B}(t_2)$ by accessi-

bility of (p_2, q_2). Since $p_1@p_2 \to p \in rul(A)$ we also get $p \in eval_A(t_1@t_2)$ by definition of $eval_A$. Furthermore since A is productive there exists a term $C[p] \in T_{\Sigma\cup\{p\}}$ with a single occurrence p such that $C[t_1@t_2] \in L(A)$. Since B is deterministic it follows that $q_1 \in eval_B(t_1)$ and $q_2 \in eval_B(t_2)$ are unique. By hypothesis $\nexists q.q_1@q_2 \to q \in rul(B)$ so that $C[t_1@t_2] \notin L(B)$.

$\texttt{fail}_2$**.** There are $p \in \mathit{fin}(A)$ and $q \notin \mathit{fin}(B)$ such that (p, q) is accessible. Thus, there exists $t \in T_\Sigma$ such that $(p, q) \in eval_{A\times B}(t)$. The state p is final in A thus $t \in L(A)$. Since B is deterministic $q \in eval_B(t)$ is unique but q not final in B implies $t \notin L(B)$.

For completeness, we assume that there exists a tree $t \in L(A)$ such that $t \notin L(B)$, and show that some failure condition holds. There are two cases to be considered, depending on $eval_B(t)$.

(i) Assume $eval_B(t) = \emptyset$. There exists a minimal subtree t' of t such that $eval_B(t') = \emptyset$, too. If $t' = a$ is a leaf then $eval_A(a) \neq \emptyset$, since $t \in L(A)$, and $eval_B(a) = \emptyset$, thus $\texttt{fail}_0$ holds. If $t' = t_1@t_2$, then there exist $p_1 \in eval_A(t_1)$, $p_2 \in eval_A(t_2)$ and $p_1@p_2 \to p \in rul(A)$, since $t \in L(A)$. Since t' is defined as a minimal subtree and B is deterministic, $eval_B(t_1) = \{q_1\}$, $eval_B(t_2) = \{q_2\}$, and since $eval_B(t') = \emptyset$, there is no rule $q_1@q_2 \to q \in rul(B)$. This leads to $\texttt{fail}_1$.

(ii) If $eval_B(t) \neq \emptyset$ then there exists $q \in eval_B(t)$; B being deterministic this q is necessarily unique. Since $t \notin L(B)$ this yields $q \notin \mathit{fin}(B)$. Moreover, since $t \in L(A)$, there exists $p \in eval_A(t) \cap \mathit{fin}(A)$. Thus, $\texttt{fail}_2$ holds. □

3.3 Testing the Characterization in Ground Datalog

Figure 4 presents a ground Datalog program $D_1(A, B)$ that verifies the characterization of $L(A) \subseteq L(B)$ in Proposition 4. Transformation rules ($\text{acc}_{/1}$), ($\text{acc}_{/2}$), and ($\text{acc}_{/3}$) define clauses for accessibility in $A \times B$ through predicate acc. Clearly, $\mathsf{acc}(p, q) \in \mathit{lfp}(D_1(A, B))$ iff $A, B \models \mathsf{acc}(p, q)$. The clauses produced by transformation rule (frb) define *forbidden states* of $A \times B$ through predicate frb. The semantics of this predicate is:

$$(A, B \models \mathsf{frb}(p, q)) \Leftrightarrow (A, B \models \mathsf{acc}(p, q) \Rightarrow \texttt{fail}_1).$$

The clauses produced by transformation rule (fail_1) guarantee that forbidden states raise $\texttt{fail}_1$ when accessed. The Datalog rules for inferring the other two failure conditions (fail_0) and (fail_2) are as expected.

Proposition 5 *Let A and B be stepwise tree automata for binary trees. If A is productive and B deterministic then:*

$$L(A) \subseteq L(B) \Leftrightarrow \mathit{lfp}(D_1(A, B)) \cap \{\texttt{fail}_0, \texttt{fail}_1, \texttt{fail}_2\} = \emptyset$$

$$(\mathrm{acc}_{/1})\ \frac{a \to p \in rul(A) \qquad a \to q \in rul(B)}{\mathsf{acc}(p,q).}$$

$$(\mathrm{acc}_{/2})\ \frac{p' \xrightarrow{\epsilon}_A p \in rul(A) \qquad q \in sta(B)}{\mathsf{acc}(p,q) :\!-\ \mathsf{acc}(p',q).}$$

$$(\mathrm{acc}_{/3})\ \frac{p_1@p_2 \to p \in rul(A) \qquad q_1@q_2 \to q \in rul(B)}{\mathsf{acc}(p,q) :\!-\ \mathsf{acc}(p_1,q_1), \mathsf{acc}(p_2,q_2).}$$

$$(\mathrm{frb})\ \frac{p_1@p_2 \to p \in rul(A) \qquad \nexists q.q_1@q_2 \to q \in rul(B)}{\begin{array}{c}\mathsf{frb}(p_2,q_2) :\!-\ \mathsf{acc}(p_1,q_1).\\ \mathsf{frb}(p_1,q_1) :\!-\ \mathsf{acc}(p_2,q_2).\end{array}}$$

$$(\mathrm{fail}_0)\ \frac{a \to p \in rul(A) \qquad \nexists q.a \to q \in rul(B)}{\mathtt{fail}_0.}$$

$$(\mathrm{fail}_1)\ \frac{p \in sta(A) \qquad q \in sta(B)}{\mathtt{fail}_1 :\!-\ \mathsf{acc}(p,q), \mathsf{frb}(p,q).}$$

$$(\mathrm{fail}_2)\ \frac{p \in fin(A) \qquad q \notin fin(B)}{\mathtt{fail}_2 :\!-\ \mathsf{acc}(p,q).}$$

Figure 4. Transforming tree automata A and B into a Datalog program $D_1(A,B)$.

Proof. This is an immediate consequence of Proposition 4 and the fact that the Datalog program $D_1(A,B)$ captures the failure conditions. □

The sum of the sizes of the clauses defined by transformation rules ($\mathrm{acc}_{/1}$), ($\mathrm{acc}_{/2}$), ($\mathrm{acc}_{/3}$), (fail_0), (fail_1), and (fail_2) is $O(|A|*|B|)$. The sizes of the clauses defined by transformation rule (frb) sum up to $O(|A| * |sta(B)|^2)$. The overall size of the ground Datalog program $D_1(A,B)$ is $O(|A| * (|B| + |sta(B)|^2))$, which may be $O(|A| * |B|^2)$ in the worst case. Therefore, using Theorem 3, inclusion can be decided in time $O(|A| * |B|^2)$. Unfortunately, this is not yet any better than the naive algorithm.

3.4 Efficient Algorithm

The square factor in the size of $D_1(A,B)$ is raised by inference rule (frb). We will introduce a new predicate frb^c in order to group frb literals. Its semantics is as follows:

$$A, B \models \mathsf{frb}^c(p,Q) \Leftrightarrow \forall q \in sta(B) \setminus Q,\ A, B \models \mathsf{frb}(p,q)$$

$$(\mathrm{frb}^c)\ \frac{p_1@p_2 \to p \in rul(A) \quad q_1 \in sta(B)}{\mathsf{frb}^c(p_2, Q_2^B(q_1)) :\!-\ \mathsf{acc}(p_1, q_1).} \qquad \frac{p_1@p_2 \to p \in rul(A) \quad q_2 \in sta(B)}{\mathsf{frb}^c(p_1, Q_1^B(q_1)) :\!-\ \mathsf{acc}(p_2, q_2).}$$

$$Q_2^B(q') = \{q_2 \mid q'@q_2 \to q \in rul(B)\},\ Q_1^B(q') = \{q_1 \mid q_1@q' \to q \in rul(B)\}$$

Figure 5. Grouping (frb) transformations.

where $Q \subseteq sta(B)$ and, for a fixed order $<$ on $sta(B)$, we define $\mathsf{frb}^c(p, \{q_1, \ldots, q_n\})$ as $(n+1)$-ary literals $\mathsf{frb}^c(p, q_{i_1}, \ldots, q_{i_n})$ such that $\{q_{i_1}, \ldots, q_{i_n}\} = \{q_1, \ldots, q_n\}$ and $q_{i_1} < \ldots < q_{i_n}$.

In Figure 5, we propose two transformations rules for inferring frb^c clauses, both of which group (frb) transformations. Still, we may infer a whole collection of literals $\mathsf{frb}^c(p, Q)$ for the same state p.

Let us consider the transformation of tree automata A and B into a ground Datalog program $D_2(A, B)$ defined by transformation rules ($\mathrm{acc}_{/1}$), ($\mathrm{acc}_{/2}$), ($\mathrm{acc}_{/3}$), (frb^c), (fail_0), and (fail_2). This program remains incomplete, in that frb^c literals are never used in order to infer $\mathtt{fail}_1$. This part remains to be done *a posteriori* (and is problematic for complexity).

Lemma 6 *The size of $D_2(A, B)$ is in $O(|A| * |B|)$.*

Proof. The clauses producing acc, $\mathtt{fail}_0$, and $\mathtt{fail}_2$ of $D_1(A, B)$ and $D_2(A, B)$ are identical and their number is in $O(|A| * |B|)$. The number of frb^c clauses introduced by rule (frb^c) is in $O(|A| * |sta(B)|)$ but the size of each such clause is $n + 1$ which in the worst case could be $|sta(B)| + 1$, and symmetrically for (frb^c). The overall size of all frb^c clauses, however, is bounded by the overall number of acc clauses, which in turn is bounded by $O(|A| * |B|)$, too! To see this, we can rewrite the first rule of (frb^c) as shown in Figure 6, such that the corresponding ($\mathrm{acc}_{/3}$) clauses are inferred simultaneously (and these don't overlap). □

The Datalog programs $D_1(A, B)$ and $D_2(A, B)$ have the same clauses for inferring literals $\mathsf{acc}(p, q)$, $\mathtt{fail}_0$, and $\mathtt{fail}_2$, so their least fixed points coincide in that respect. In particular, we can decide $A, B \models \mathtt{fail}_0 \vee \mathtt{fail}_2$ in time $O(|A| * |B|)$ by testing membership of $\mathtt{fail}_0$ and $\mathtt{fail}_2$ in $lfp(D_2(A, B))$. It remains to relate both programs with respect to forbidden states and $\mathtt{fail}_1$.

Lemma 7 $\mathsf{frb}(p, q) \in lfp(D_1(A, B))$ *if and only if* $\exists Q \subseteq sta(B) \setminus \{q\}$ *such that* $\mathsf{frb}^c(p, Q) \in lfp(D_2(A, B))$.

Proof. Straightforward by induction on the definitions of the least fixed points, and the correspondence of rules (frb) and (frb^c). □

Lemma 7 yields the following algorithm for testing inclusion $L(A) \subseteq L(B)$:

$$\frac{p_1@p_2 \to p \in rul(A) \qquad \left.\begin{array}{c} q_1@q_2^1 \to q^1 \in rul(B) \\ \vdots \\ q_1@q_2^n \to q^n \in rul(B) \end{array}\right\} \text{all the rules for } q_1}{\begin{array}{c} \mathsf{frb}^c(p_2, \{q_2^1, \ldots, q_2^n\}) :- \mathsf{acc}(p_1, q_1). \\ \mathsf{acc}(p, q^1) :- \mathsf{acc}(p_1, q_1), \mathsf{acc}(p_2, q_2^1). \\ \vdots \\ \mathsf{acc}(p, q^n) :- \mathsf{acc}(p_1, q_1), \mathsf{acc}(p_2, q_2^n). \end{array}}$$

Figure 6. Rewriting the first grouped rule for complexity analysis of (frb^c) clauses.

(1) compute $\mathit{lfp}(D_2(A, B))$ in time $O(|A| * |B|)$ (by Lemma 6);
(2) return false if $\mathtt{fail}_0$ or $\mathtt{fail}_2$ belong to $\mathit{lfp}(D_2(A, B))$;
(3) return false if there exists literals $\mathsf{acc}(p, q), \mathsf{frb}^c(p, Q) \in \mathit{lfp}(D_2(A, B))$ such that $q \in \mathit{sta}(B) \setminus Q$ (this captures $\mathtt{fail}_1$ by Lemma 7);
(4) otherwise return true.

We have to show that step 3 can be done in time $O(|A| * |B|)$ in order to prove the following theorem.

Theorem 8 *Let Σ be a ranked signature, and A and B be standard tree automata where A may have ϵ-rules. If B is deterministic, inclusion $L(A) \subseteq L(B)$ can be decided in time $O(|A| * |B|)$ independently of the size of Σ.*

Proof. By Proposition 5, the above algorithm is correct for productive stepwise tree automata A and deterministic stepwise tree automata B. More general ranked signatures with higher arities can be reduced to this case by Proposition 1. Every tree automaton can be made productive in linear time. The above algorithm is in time $O(|A| * |B|)$ iff step (3) can be done in this time. How to do this will be shown in Section 3.5. □

3.5 Efficient Data Structure

We have to decide step (3) in time $O(|A| * |B|)$. The computation of all literals $\mathsf{frb}(p, q) \in \mathit{lfp}(D_1(A, B))$ induced by some literal $\mathsf{frb}^c(p, Q) \in D_2(A, B)$ could lead to a quadratic blowup when complementing all such Q's.

In a first step, we fix a single state $p \in \mathit{sta}(A)$ and let $\{Q_1, \ldots, Q_n\} = \{Q \mid \mathsf{frb}^c(p, Q) \in \mathit{lfp}(D_2(A, B))\}$. More generally, given a finite set Q and collection of subsets $Q_1, \ldots, Q_n \subseteq Q$ we have to compute the union of complements

Q	1	2	3	4	5	6
Q_1	x		x			
Q_2			x			
Q_3			x			x
$Q_1^c \cup Q_2^c \cup Q_3^c$	x	x		x	x	x
C	1	0	3	0	0	1

Figure 7. Computing union of complements by using a table or counting.

$Q_1^c \cup \ldots \cup Q_n^c$ where complementation is with respect to Q. In Figure 7, we illustrate an instance of the problem where $Q = \{1, \ldots, 6\}$, $Q_1 = \{1, 3\}$, $Q_2 = \{3\}$ and $Q_3 = \{3, 6\}$. The result $Q_1^c \cup Q_2^c \cup Q_3^c$ should be $\{1, 2, 4, 5, 6\}$.

The naive method is to create a table of Booleans $T : Q \times \{1, \ldots, n\} \to \mathbb{B}$ in which $T(q, i) = 1$ iff $q \in Q_i$. In other words, we mark all positions (q, i) in the table such that $q \in Q_i$. The union of the complements can then be computed by enumerating the set $\{q \mid \exists i.T(q, i) = 0\}$. This requires to iterate once over the whole table, which can be done in time $O(n*|Q|)$. Unfortunately, this time may be quadratically larger than the size of the input $|Q| + |Q_1| + \ldots + |Q_n|$. The problem is to avoid the inspection of the set $\{(q, i) \mid q \notin Q_i,\ 1 \leq i \leq n\}$ which contains the negative information. Indeed, a more efficient solution can be obtained by counting positive information.

Lemma 9 *Given a finite set Q with subsets $Q_1, \ldots, Q_n \subseteq Q$, one can compute $Q_1^c \cup \ldots \cup Q_n^c$ in time $O(|Q| + |Q_1| + \ldots + |Q_n|)$.*

Proof. For every element $q \in Q$, we count the number $C(q) = \#\{i \mid q \in Q_i\}$ of occurrences of q in some Q_i. The collection of these numbers can be computed in time $O(|Q_1| + \ldots + |Q_n|)$ by enumerating all elements of all sets $Q_1, \ldots, Q_n$. Now, notice that $q \in Q_1^c \cup \ldots \cup Q_n^c$ if and only if $C(q) = n$. The set of all states with this property can be computed in time $O(|Q|)$. □

For implementing step (3), we apply the above technique to all states $p \in sta(A)$, in order to compute $\{q \in sta(B) \mid \mathsf{frb}(p, q) \in lfp(D_1(A, B))\}$. By Lemma 9, this requires time:

$$O\left(\sum_{p \in sta(A)} |sta(B)| + \sum_{\mathsf{frb}^c(p,Q) \in lfp(D_2(A,B))} |Q|\right)$$

which is smaller than $O(|sta(A)|*|sta(B)|+|lfp(D_2(A, B))|)$ and thus in $O(|A|*|B|)$ by Lemma 6.

In the concrete implementation, we will use for all states $p \in sta(A)$ a counter $C(p)$ for the current number of literals $\mathsf{frb}^c(p, Q)$ computed so far, and for all

states $q \in sta(B)$ a counter $C(p,q)$ for the number of occurrences of $q \in Q$ in literals $\mathsf{frb}^c(p,Q)$ seen so far. We can thus check $\mathsf{frb}(p,q) \in \mathit{lfp}(D_1(A,B)$ by testing $C(p,q) \neq C(p)$.

3.6 Incrementality

We can make our inclusion test incremental with respect to adding automata rules. Incrementality may be determining for efficiency. In our prime application to schema-guided query induction [9], for instance, we use incremental addition of ϵ-rules to the automaton A on the left. These model state merging operations $p_1 = p_2$ during automata induction as $p_1 \xrightarrow{\epsilon} p_2$ and $p_2 \xrightarrow{\epsilon} p_1$.

It is obvious how to perform fixed point computations of Datalog programs incrementally with respect to adding new clauses. As a consequence, step (1) of our inclusion test can be implemented incrementally, since adding automata rules to A or B amounts to adding new Datalog clauses to $D_2(A,B)$. Step (2) can be done concurrently with step (1), in such a way that the fixed point computation is stopped, once a literal $\mathtt{fail}_0$ or $\mathtt{fail}_2$ is inferred. This reduces the run-time even in a non-incremental setting, since only parts of the fixed point may be needed.

On the fly checking for $\mathtt{fail}_1$ is less obvious, i.e., performing step (3) concurrently with step (1), too. Once a literal $\mathsf{acc}(p,q)$ is inferred during the computation of the fixed point of $D_2(A,B)$, we can check in constant time whether $\mathsf{frb}(p,q)$ is implied by some literal $\mathsf{frb}^c(p,Q)$ inferred before. The converse, however, is not feasible without extra costs. When deriving $\mathsf{frb}^c(p,Q)$, we cannot check for all $q \in sta(B) \setminus Q$ whether $\mathsf{acc}(p,q)$ has been inferred before, without enumerating the complement of Q.

As we will show, this kind of tests can be safely ignored by imposing a *priority* discipline on the least fixed point computation. We will assume that literals of the form $\mathsf{acc}(p,q)$ are always inferred with lowest priority, i.e., whenever other literals can be inferred at the same time, these will be inferred before.

We have to prove that some failure condition is detected whenever some literal $\mathsf{acc}(p_1,q_1)$ is inferred before some literal $\mathsf{frb}^c(p_1,Q_1)$ with $q \notin Q_1$. This situation is depicted in Figure 8. Literal $\mathsf{frb}^c(p_1,Q_1)$ originates from a clause produced by rule (frb^c) and some literal $\mathsf{acc}(p_2,q_2)$ added earlier:

$$\frac{p_1 @ p_2 \rightarrow p \in \mathit{rul}(A) \qquad Q_1 = Q_1^B(q_2)}{\mathsf{frb}^c(p_1,Q_1) \text{:-}\ \mathsf{acc}(p_2,q_2).}$$

We show by contradiction that $\mathsf{acc}(p_1,q_1)$ got added before $\mathsf{acc}(p_2,q_2)$. Oth-

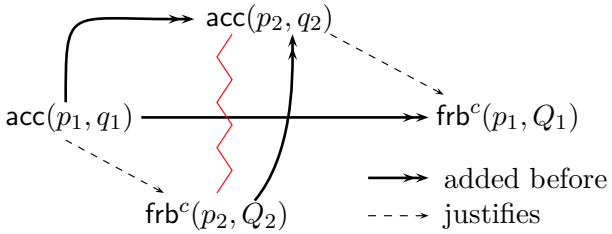

Figure 8. Early failure detection: $\mathsf{frb}^c(p_2, q_2)$ in $lfp(D_2(A, B))$ before $\mathsf{acc}(p_2, q_2)$.

erwise, $\mathsf{acc}(p_2, q_2)$ was added before $\mathsf{acc}(p_1, q_1)$, so that due to our priority assumption, $\mathsf{frb}^c(p_1, Q_1)$ was added before $\mathsf{acc}(p_1, q_1)$ which contradicts the hypothesis. Having $\mathsf{acc}(p_1, q_1)$ in the fixed point permits to apply the following clause of (frb^c):

$$\frac{p_1@p_2 \to p \in rul(A) \qquad Q_2 = Q_2^B(q_1)}{\mathsf{frb}^c(p_1, Q_2) :\!-\ \mathsf{acc}(p_1, q_1).}$$

Note that $q_1 \in Q_1^B(q_2)$ iff $q_2 \in Q_2^B(q_1)$. Thus $q_2 \notin Q_2$ since $q_1 \notin Q_1$. Consequently, $\mathsf{acc}(p_2, q_2), \mathsf{frb}^c(p_2, Q_2) \in lfp(D_2(A, B))$ raises $\mathtt{fail}_1$, and this is correctly detected by the modified algorithm, since $\mathsf{frb}^c(p_2, Q_2)$ is inferred before $\mathsf{acc}(p_2, q_2)$.

4 Factorized Tree Automata

We next relax the determinism assumption on B in a controlled manner, that will be crucial to deal with DTDs. We replace B by deterministic factorized automata, that we introduce. These are stepwise tree automata with ϵ-rules for ranked trees, that represent deterministic stepwise tree automata in a compacter manner.

Definition 1 *A* factorized tree automaton F *over a stepwise signature* Σ *is a stepwise tree automaton with* ϵ*-rules and a partition* $sta(F) = sta_1(F) \uplus sta_2(F)$ *such that if* $q_1@q_2 \to q$ *in* $rul(F)$ *then* $q_1 \in sta_1(F)$ *and* $q_2 \in sta_2(F)$.

We say that q is of *sort* i in F if $q \in sta_i(F)$. The sort determines which states may be used in the i-th position of the binary symbol @ in rules of F.

Every factorized automaton F defines a tree automaton $ta(F)$ without ϵ-rules that recognizes the same language. Both automata have the same signature and states; the rules of $ta(F)$ are inferred as follows from those of F:

$$(\mathrm{E}_1)\ \frac{a \to q \in rul(F)}{a \to q \in rul(ta(F))} \qquad (\mathrm{E}_2)\ \frac{q_1 \xrightarrow{\epsilon}{}^*_F r_1 \quad q_2 \xrightarrow{\epsilon}{}^*_F r_2 \quad r_1@r_2 \to q \in rul(F)}{q_1@q_2 \to q \in rul(ta(F))}$$

We set $\mathit{fin}(ta(F)) = \{q \mid q \xrightarrow{\epsilon}{}^*_F r,\ r \in \mathit{fin}(F)\}$. Note that the size of $ta(F)$ may be $O(|\mathit{rul}(F)| * |\mathit{sta}(F)|^2)$ which is cubic in that of F in the worst case. Besides their succinctness, the truly interesting bit about factorized tree automata is their notion of determinism. An example is given in Figure 14.

Definition 2 *A factorized tree automaton F is (bottom-up)* deterministic *if:*

$\mathtt{d}_0$*: the ϵ-free part of F is (bottom-up) deterministic;*
$\mathtt{d}_1$*: for all $q \in \mathit{sta}(F)$ and sorts $i \in \{1,2\}$, there is at most one state r of sort i such that $q \xrightarrow{\epsilon}{}^*_F r$.*

Non-redundant ϵ-rules must change the sort: if $q \xrightarrow{\epsilon}_F r$ for two states of the same sort then $r = q$ by $\mathtt{d}_1$ and $q \xrightarrow{\epsilon}{}^*_F q$. A similar argument shows that all proper chains of ϵ-rules are redundant so that $\xrightarrow{\epsilon}{}^*_F$ is equal to $\xrightarrow{\epsilon}{}^{\leq 1}_F$.

Proposition 10 *The tree automaton $ta(F)$ represented by a deterministic factorized tree automaton F is deterministic.*

Proof. Let $B = ta(F)$ which by construction is free of ϵ-rules. For every constant $a \in \Sigma$, the uniqueness of q such that $a \to q \in \mathit{rul}(B)$ follows from $\mathtt{d}_0$. For every $q_1@q_2 \to q$ in $\mathit{rul}(B)$ we have to show that q is uniquely determined by q_1 and q_2. By $\mathtt{d}_1$ there is at most one state r_1 of sort 1 such that $q_1 \xrightarrow{\epsilon}{}^*_F r_1$ at most one r_2 of sort 2 such that $q_2 \xrightarrow{\epsilon}{}^*_F r_2$. Condition $\mathtt{d}_0$ implies that there exists at most one state q such that $r_1@r_2 \to q \in \mathit{rul}(F)$. □

Conversely, every deterministic stepwise tree automaton B can be converted in time $O(|B|)$ into a deterministic factorized tree automaton F, such that $ta(F)$ is equal to B modulo state renaming. The states of F are $\mathit{sta}_i(F) = \mathit{sta}(B) \times \{i\}$ for $i \in \{1,2\}$. Rules are transformed as follows:

$$\frac{q_1@q_2 \to q \in \mathit{rul}(B)}{\begin{array}{c}(q_1,1)@(q_2,2) \to (q,1) \in \mathit{rul}(F)\\ (q,1) \xrightarrow{\epsilon} (q,2) \in \mathit{rul}(F)\end{array}} \qquad \frac{a \to q \in \mathit{rul}(B)}{\begin{array}{c}a \to (q,1) \in \mathit{rul}(F)\\ (q,1) \xrightarrow{\epsilon} (q,2) \in \mathit{rul}(F)\end{array}}$$

For inclusion testing, we fix a stepwise tree automaton A and a deterministic factorized tree automaton F, and set $B = ta(F)$. We now show how to test language inclusion $L(A) \subseteq L(F)$ without computing B (because the size of B may be cubic in the size of F). We define a ground Datalog program $D_3(A,F)$ in Figure 9, in order to simulated $D_2(A,B)$.

We use new predicates for F in order to infer corresponding properties of B on the level of F. The accessibility predicate $\mathsf{f.acc}$ for F subsumes the accessibility predicate acc for B. Subsumption may be proper as stated by the rule (f.acc)

$$(\mathrm{acc}_{/3a})\ \frac{p_1@p_2 \to p \in rul(A) \quad q_1@q_2 \to q \in rul(F)}{\mathsf{acc}(p,q) :\!-\ \mathsf{f.acc}(p_1,q_1), \mathsf{f.acc}(p_2,q_2).}$$

$$(\mathrm{f.acc})\ \frac{p \in sta(A) \quad q \xrightarrow{\epsilon}{}^{\leq 1}_{F} r}{\mathsf{f.acc}(p,r) :\!-\ \mathsf{acc}(p,q).}$$

$$(\mathrm{f.frb}^c_2)\ \frac{p_1@p_2 \to p \in rul(A) \quad q_1 \in sta_1(F)}{\mathsf{f.frb}^c_2(p_2, Q^F_2(q_1)) :\!-\ \mathsf{f.acc}(p_1,q_1).}$$

$$(\mathrm{f.frb}^c_1)\ \frac{p_1@p_2 \to p \in rul(A) \quad q_2 \in sta_2(F)}{\mathsf{f.frb}^c_1(p_1, Q^F_1(q_2)) :\!-\ \mathsf{f.acc}(p_2,q_2).}$$

$$(\mathrm{frb}^c_2)\ \frac{p_1@p_2 \to p \in rul(A) \quad q_1 \in sta(F)}{\mathsf{frb}^c_2(p_2, R^F_2) :\!-\ \mathsf{acc}(p_1,q_1).}$$

$$(\mathrm{frb}^c_1)\ \frac{p_1@p_2 \to p \in rul(A) \quad q_2 \in sta(F)}{\mathsf{frb}^c_1(p_1, R^F_1) :\!-\ \mathsf{acc}(p_2,q_2).}$$

$$(\mathrm{frb}^c_{/a})\ \frac{p_1@p_2 \to p \in rul(A) \quad q_2 \notin R^F_2}{\mathsf{frb}^c(p_1,\emptyset) :\!-\ \mathsf{acc}(p_2,q_2).} \qquad \frac{p_1@p_2 \to p \in rul(A) \quad q_1 \notin R^F_1}{\mathsf{frb}^c(p_2,\emptyset) :\!-\ \mathsf{acc}(p_1,q_1).}$$

$$(\mathrm{fail}_{2a})\ \frac{p \in fin(A) \quad \forall r.\ q \xrightarrow{\epsilon}{}^{\leq 1}_{F} r \Rightarrow r \notin fin(F)}{\mathtt{fail}_2 :\!-\ \mathsf{acc}(p,q).}$$

We use sets of states $Q^F_2(q_1) = \{q' \mid q_1@q' \to q'' \in rul(F)\}$, $Q^F_1(q_2)$ symmetrically, and sets of states reaching a sort $R^F_i = \{q \mid \exists r \in sta_i(F).\ q \xrightarrow{\epsilon}{}^{\leq 1}_{F} r\}$. The clauses from $(\mathrm{acc}_{/1})$, $(\mathrm{acc}_{/2})$ and (fail_0) in $D_2(A,F)$ belong to $D_3(A,F)$, too.

Figure 9. Inferring clauses of Datalog program $D_3(A,F)$ simulating $D_2(A,B)$.

of $D_3(A,F)$. *Vice verca*, we infer accessibility in F from accessibility in B according to the rule $(\mathrm{acc}_{/3a})$. Rules $(\mathrm{acc}_{/1})$ and $(\mathrm{acc}_{/2})$ of $D_2(A,F)$ remain valid for accessibility in B, too.

Lemma 11 $\mathsf{acc}(p,q) \in lfp(D_3(A,F))$ *iff* $\mathsf{acc}(p,q) \in lfp(D_2(A,B))$

Proof. Technical but straightforward. The details are given in Appendix A.

We need to refine predicate frb into predicates frb_1 and frb_2 that take sorts into account, and corresponding predicates $\mathsf{f.frb}_1$ and $\mathsf{f.frb}_2$ in the factorized case. Their semantics can be defined as follows, where A, B are tree automata and F is a factorized tree automaton.

$$
\begin{aligned}
(A, B \models \mathsf{frb}_2(p_2, q_2)) \Leftrightarrow\ & (\exists p, p_1, q_1.\ A, B \models \mathsf{acc}(p_1, q_1),\\
& \quad p_1@p_2 \to p \in \mathit{rul}(A),\ q_2 \notin Q_2^B(q_1))\\
(A, F \models \mathsf{f.frb}_2(p_2, r_2)) \Leftrightarrow\ & (\exists p, p_1, r_1.\ A, F \models \mathsf{f.acc}(p_1, r_1),\\
& \quad p_1@p_2 \to p \in \mathit{rul}(A),\ r_2 \notin Q_2^F(r_1))
\end{aligned}
$$

The semantics of frb_1 and $\mathsf{f.frb}_1$ are symmetric. The relation to the previous predicate frb is that $A, B \models \mathsf{frb}(p, q)$ if and only if $A, B \models \mathsf{frb}_1(p, q) \vee \mathsf{frb}_2(p, q)$.

The Datalog program $D_3(A, F)$ infers for sorts $i \in \{1, 2\}$ literals with predicates $\mathsf{f.frb}_i^c$ that are to be understood by grouping of $\mathsf{f.frb}_i$ literals, and similarly frb_i^c by grouping of frb_i. These grouping mechanisms account for sorts, since complementation is with respect to sorts.

$$
\begin{aligned}
A, F \models \mathsf{f.frb}_i^c(p, Q) &\Leftrightarrow \forall q \in \mathit{sta}_i(F) \setminus Q.\ A, F \models \mathsf{f.frb}_i(p, q)\\
A, F \models \mathsf{frb}_i^c(p, Q) &\Leftrightarrow \forall q \in \mathit{sta}_i(F) \setminus Q.\ A, F \models \mathsf{frb}_i(p, q)
\end{aligned}
$$

The clauses produced by (f.frb$_i^c$) and (frb$_i^c$) are sound with respect to this semantics for deterministic F's. This is easier to see for (f.frb$_i^c$) than for (frb$_i^c$). We prove it by the next lemma. For states p, q, r and sorts $i \in \{1, 2\}$ we define:

$$
\begin{aligned}
A, B \vdash \mathsf{frb}_i(p, q) \quad &\text{iff}\ \ \exists Q \subseteq \mathit{sta}(B) \setminus \{q\}.\ \mathsf{frb}^c(p, Q) \in \mathit{lfp}(D_2(A, B))\\
&\quad \textit{via} \text{ the i'th conclusion of rule (frb}^c\text{)}\\
A, F \vdash \mathsf{f.frb}_i(p, r) \quad &\text{iff}\ \ \exists R \subseteq \mathit{sta}(F) \setminus \{r\}.\ \mathsf{f.frb}_i^c(p, R) \in \mathit{lfp}(D_3(A, F))\\
A, F \vdash \mathsf{frb}_i(p, r) \quad &\text{iff}\ \ \exists R \subseteq \mathit{sta}(F) \setminus \{r\}.\ \mathsf{frb}_i^c(p, R) \in \mathit{lfp}(D_3(A, F))\\
&\quad \text{or } \mathsf{frb}^c(p, \emptyset) \in \mathit{lfp}(D_3(A, F)) \textit{ via} \text{ the i'th rule of (frb}^c{}_{/a}\text{)}
\end{aligned}
$$

Lemma 12 (Core) *$A, B \vdash \mathsf{frb}_i(p, q)$ if and only if $A, F \vdash \mathsf{frb}_i(p, q)$ or the unique state r of sort i with $q \xrightarrow[F]{\epsilon}{}^{\leq 1} r$ exists and satisfies $A, F \vdash \mathsf{f.frb}_i(p, r)$.*

Proof. By symmetry it is sufficient to consider the case $i = 1$. We show that $A, B \vdash \mathsf{frb}_1(p_1, q_1)$ iff $A, F \vdash \mathsf{frb}_1(p_1, q_1)$ or the unique state r_1 of sort 1 with $q_1 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_1$ exists and satisfies $A, F \vdash \mathsf{f.frb}_1(p_1, r_1)$. There are two implications to be proved.

"$\Rightarrow$". Assume $A, B \vdash \mathsf{frb}_1(p_1, q_1)$ so that there exists $Q \subseteq \mathit{sta}(B)$ with $q_1 \notin Q$ such that $\mathsf{frb}^c(p_1, Q) \in \mathit{lfp}(D_2(A, B))$ has been deduced by the first rule of (frb^c). Thus, there exist a rule $p_1@p_2 \to p$ of A and $q_2 \in \mathit{sta}(B)$ such that $\mathsf{acc}(p_2, q_2) \in \mathit{lfp}(D_2(A, B))$ and $Q = \{q' \mid q'@q_2 \to q'' \in \mathit{rul}(B)\}$. Membership of $\mathsf{acc}(p_2, q_2)$ in $\mathit{lfp}(D_3(A, F))$ is induced by Lemma 11. We have to distinguish whether or not there exists $r_1 \in \mathit{sta}_1(F)$ such that $q_1 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_1$.

(1) Case where no r_1 of sort 1 exists such that $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_1$. By rule (frb^c_1), it follows that $\mathsf{frb}^c_1(p_1, Q') \in \mathit{lfp}(D_3(A,F))$ with $Q' = \{q' \mid q' \xrightarrow{\epsilon}{}^{\leq 1}_{F} r,\ r \text{ of sort } 1\}$ by (E_2). Since $q_1 \notin Q'$ this yields $A, F \vdash \mathsf{frb}_1(p_1, q_1)$.

(2) Case where there exists r_1 of sort 1 such that $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_1$. We consider two exhaustive subcases.

   (a) Subcase where exists r_2 of sort 2 such that $q_2 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_2$. A clause by rule (f.acc) shows that $\mathsf{f.acc}(p_2, r_2) \in \mathit{lfp}(D_3(A,F))$. Let $R = Q^F_1(r_2)$. If $r_1 \in R$ then exists $r_1@r_2 \to r \in \mathit{rul}(F)$; so that $q_1@q_2 \to r \in \mathit{rul}(B)$ and thus $q_1 \in Q$ in contradiction to our hypothesis $q_1 \notin Q$. Consequently, $r_1 \notin R$. Furthermore, $(\mathrm{f.frb}^c_1)$ yields $\mathsf{f.frb}^c_1(p_1, R)$ in $\mathit{lfp}(D_3(A,F))$ and thus $A, F \vdash \mathsf{f.frb}_1(p_1, r_1)$ where $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_1$ and r_1 is of sort 1.

   (b) Subcase where no r_2 of sort 2 exists such that $q_2 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_2$. In this case, the sort of q_2 is 1. By $(\mathrm{frb}^c_{/1a})$, it follows that $\mathsf{frb}^c(p_1, \emptyset)$ belongs to $\mathit{lfp}(D_3(A,F))$, and thus $A, F \vdash \mathsf{frb}_1(p_1, q_1)$.

"$\Leftarrow$". We distinguish the two cases of the disjunction.

(1) Case $A, F \vdash \mathsf{frb}_1(p_1, q_1)$. There are two subcases to consider:

   (a) Subcase where there exists a literal $\mathsf{frb}^c_1(p_1, Q)$ in $\mathit{lfp}(D_3(A,F))$ such that $q_1 \notin Q$. This literal is derived by a clause from rule (frb^c_1). Hence, there exists a rule $p_1@p_2 \to p$ of A and a state q_2 of F such that $\mathsf{acc}(p_2, q_2) \in \mathit{lfp}(D_3(A,F))$, and $Q = \{q \mid \exists r \in \mathit{sta}_1(F).\ q \xrightarrow{\epsilon}{}^{\leq 1}_{F} r\}$. Membership of $\mathsf{acc}(p_2, q_2)$ in $\mathit{lfp}(D_2(A,B))$ is proved by Lemma 11. Since $q_1 \notin Q$, there is no $r_1 \in \mathit{sta}_1(F)$ such that $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r$. Consequently, q_1 can only be of sort 2. By (E_2), one can not derive any rule in B where q_1 is in first position of the left-hand side, and in particular rules of B where q_2 is in second position of the left-hand side, i.e., $q_1 \notin Q'$ such that $Q' = \{q' \mid q'@q_2 \to q'' \in \mathit{rul}(B)\}$. Finally, we use a clause from rule (frb^c) in order to prove membership of $\mathsf{frb}^c(p_1, Q')$ in $\mathit{lfp}(D_2(A,B))$, and thus $A, B \vdash \mathsf{frb}_1(p_1, q_1)$.

   (b) Subcase where $\mathsf{frb}^c(p_1, \emptyset)$ belongs to $\mathit{lfp}(D_3(A,F))$ *via* a clause of the first inference rule of $(\mathrm{frb}^c_{/a})$. Hence, there exists an automaton rule $p_1@p_2 \to p$ of A and a state q_2 of F such that $\mathsf{acc}(p_2, q_2) \in \mathit{lfp}(D_3(A,F))$, and there is no $r_2 \in \mathit{sta}_2(F)$ such that $q_2 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_2$. As a consequence, q_2 is of sort 1 so that $Q^B_1(q_2) = \emptyset$, and obviously $q_1 \notin Q^B_1(q_2)$. Lemma 11 yields $\mathsf{acc}(p_2, q_2) \in \mathit{lfp}(D_2(A,B))$. A clause from the first (frb^c) rule of $D_2(A,B)$ applies so that $\mathsf{frb}^c(p_1, Q^B_1(q_2))$ belongs to $\mathit{lfp}(D_2(A,B))$. Thus $A, B \vdash \mathsf{frb}_1(p_1, q_1)$.

(2) Case where the unique state r_1 of sort 1 with $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_1$ exists and satisfies $A, F \vdash \mathsf{f.frb}_1(p_1, r_1)$. There exists R such that $\mathsf{f.frb}^c_1(p_1, R)$ belongs to $\mathit{lfp}(D_3(A,F))$ and $r_1 \notin R$. According to rule $(\mathrm{f.frb}^c_1)$ there exist a rule $p_1@p_2 \to p$ of A and a state r_2 of F such that $R = \{r' \mid r'@r_2 \to r'' \in$

$rul(F)\}$ and $\mathsf{f.acc}(p_2, r_2)$ belongs to $lfp(D_3(A,F))$. By rule (f.acc) there exists q_2 such that $q_2 \xrightarrow{\epsilon}{}^{\leq 1}_{F} r_2$ and $\mathsf{acc}(p_2, q_2)$ belongs to $lfp(D_3(A,F))$. By Lemma 11, $\mathsf{acc}(p_2, q_2)$ also belongs to $lfp(D_2(A,B))$. Let $Q = \{q' \mid q'@q_2 \to q'' \in rul(B)\}$. If q_1 belongs to Q then exists $q_1@q_2 \to q \in rul(B)$; by (E_2), $r_1@r_2 \to q \in rul(F)$ and thus $r_1 \in R$, which contradicts the hypothesis $r_1 \notin R$. Then it follows that $q_1 \notin Q$. Furthermore, we have that $\mathsf{frb}^c(p_1, Q)$ in $lfp(D_2(A,B))$ *via* the first (frb^c) rule. This proves $A, B \vdash \mathsf{frb}_1(p_1, q_1)$. □

Lemma 13 *The size of $D_3(A,F)$ is in $O(|A| * |F|)$.*

Proof. The proof works as for $D_2(A,B)$ in Lemma 6. The grouping clauses produced by ($\mathrm{f.frb}^c_i$) can be rewritten in analogy to those for (frb^c) before. The sets R^F_i are of size $O(|B|)$ and occur at most $O(|A|)$ times in rules (frb^c_i), thus the overall size of the clauses produced by this rule is in $O(|A| * |B|)$, too. The analysis for the remaining rules is straightforward. □

The overall algorithm for testing $L(A) \subseteq L(F)$ is to be adapted in step (3) according to Lemma 12.

(1') compute $lfp(D_3(A,F))$ in time $O(|A| * |F|)$ (by Lemma 13);
(2') return false if $\mathtt{fail}_0$ or $\mathtt{fail}_2$ belong to $lfp(D_3(A,F))$;
(3') return false if there exists a literal $\mathsf{acc}(p,q) \in lfp(D_3(A,F))$ so that one of the conditions for $A, ta(F) \vdash \mathsf{frb}_i(p,q)$ of Lemma 12 is satisfied:
  (a) exists $\mathsf{frb}^c_i(p,R) \in lfp(D_3(A,F))$ with $q \notin R$,
  (b) or $\mathsf{frb}^c(p,\emptyset) \in lfp(D_3(A,F))$,
  (c) or $\mathsf{f.frb}^c_i(p,R) \in lfp(D_3(A,F))$ such that the unique r of sort i with $q \xrightarrow{\epsilon}{}^{\leq 1}_{F} r$ satisfies $r \notin R$.
(4') otherwise return true.

Theorem 14 *For stepwise tree automata with ϵ-rules A and deterministic factorized tree automata F over the same signature, inclusion $L(A) \subseteq L(F)$ can be decided in time $O(|A| * |F|)$.*

Proof. It remains to check condition (3') in time $O(|A| * |F|)$. Let *pred* range over predicates frb^c_i, $\mathsf{f.frb}^c_i$, and frb^c, and p over states in $sta(A)$. Then we can compute the following collection of sets:

$$\{r \in sta(F) \mid pred(p,R) \in lfp(D_3(A,F)),\ r \notin R\}$$

in time $O(|A| * |F|)$ by Lemma 9. Membership to these sets can be tested in $O(1)$, so that we can test (3') for a fixed literal $\mathsf{acc}(p,q)$ in time $O(1)$. We need at most $|lfp(D_3(A,F))|$ such tests, thus the overall time is in $O(|A| * |F|)$ by Lemma 13 and Theorem 3. □

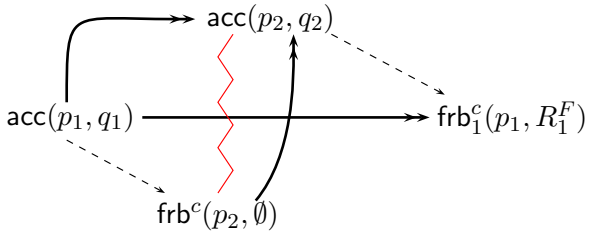

Figure 10. Early failure detection, case (a) with $i = 1$.

4.1 Incrementality

We show that we can check condition (3') on the fly. The adaptation of the algorithm is analogous to the non-factorized case, but proving its completeness becomes more tedious.

As before, we assume that literals with predicates acc are inferred with lowest priority (also lower than $\mathsf{f.acc}$). Whenever literals $\mathsf{acc}(p, q)$ are inferred during the least fixed point computation of $D_3(A, F)$, the incremental algorithm checks (3')(a-c) with respect to the current set of literals. We use counters for all predicates $\mathit{pred} \in \{\mathsf{frb}_i^c,\ \mathsf{f.frb}_i^c,\ \mathsf{frb}^c \mid i \in \{1, 2\}\}$ and states $p \in \mathit{sta}(A)$ and $q \in \mathit{sta}(F)$. The counter $C(\mathit{pred}, p)$ counts the number of literals $\mathit{pred}(p, Q)$ inferred so far, and the counter $C(\mathit{pred}, p, q)$ the number of occurrences of $q \in Q$ in literals $\mathit{pred}(p, Q)$ seen so far. Condition (3')(a) is satisfied if $C(\mathsf{frb}_i^c, p) \neq C(\mathsf{frb}_i^c, p, q)$ for $i = 1$ and $i = 2$, condition (3')(b) if $C(\mathsf{frb}^c, p) \neq C(\mathsf{frb}^c, p, q)$, and condition (3')(c) if for all $i \in \{1, 2\}$ it holds that $C(\mathsf{f.frb}_i^c, p) \neq C(\mathsf{f.frb}_i^c, p, r)$ for the unique state r of sort i such that $q \xrightarrow{\epsilon}_F^{\leq 1} r$.

The incremental algorithm checks condition (3') for all $\mathsf{acc}(p, q)$ literals only at the time point of inference. We have to prove that this is sufficient, i.e., that all cases of $\mathtt{fail}_1$ are detected even if the failure partner of $\mathsf{acc}(p, q)$ is inferred afterword's.

(a) Let $\mathsf{frb}_i^c(p_i, R_i^F)$ be inferred after $\mathsf{acc}(p_1, q_1)$. We assume $i = 1$ w.l.o.g since the case $i = 2$ is symmetric. This situation is illustrated in Figure 10. The addition of literal $\mathsf{frb}_1^c(p_1, R_1^F)$ is justified by some literal $\mathsf{acc}(p_2, q_2)$ inferred before and the following clause:

$$\frac{p_1 @ p_2 \to p \in \mathit{rul}(A) \qquad q_1 \notin R_1^F}{\mathsf{frb}_1^c(p_1, R_1^F) \coloneq \mathsf{acc}(p_2, q_2).}$$

By priority, $\mathsf{acc}(p_1, q_1)$ must have been added before $\mathsf{acc}(p_2, q_2)$; this is same argument as in the non-factorized case. Furthermore, we can apply

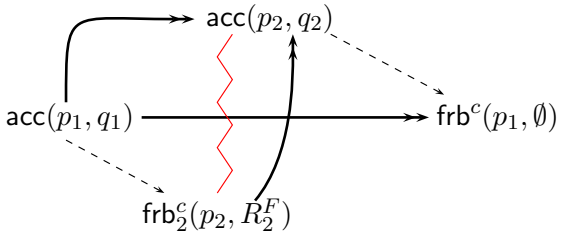

Figure 11. Early failure detection, case (b).

the following inference rule:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \qquad q_1 \notin R_1^F}{\mathsf{frb}^c(p_2, \emptyset) :\!-\ \mathsf{acc}(p_1, q_1).}$$

By priority again, the addition of $\mathsf{frb}^c(p_2, \emptyset)$ to the least fixed point of $\mathit{lfp}(D_3(A, F))$ must happen before $\mathsf{acc}(p_2, q_2)$. Thus, $\texttt{fail}_1$ is properly detected.

(b) Let $\mathsf{frb}^c(p_1, \emptyset)$ be inferred after $\mathsf{acc}(p_1, q_1)$, as illustrated in Figures 11. Literal $\mathsf{frb}^c(p_1, \emptyset)$ is justified by some literal $\mathsf{acc}(p_2, q_2)$ inferred before and the following clause:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \qquad q_2 \notin R_2^F}{\mathsf{frb}^c(p_1, \emptyset) :\!-\ \mathsf{acc}(p_2, q_2).}$$

Again, there is a symmetric case, that works the same way. Due to priority, $\mathsf{acc}(p_1, q_1)$ was added prior to $\mathsf{acc}(p_2, q_2)$. Furthermore, we can apply the following inference rule:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \qquad q_1 \in \mathit{sta}(F)}{\mathsf{frb}^c_2(p_2, R_2^F) :\!-\ \mathsf{acc}(p_1, q_1).}$$

This show that $\mathsf{frb}^c_2(p_2, R_2^F)$ is added to the fixed point. Since $q_2 \notin R_2^F$, this raises $\texttt{fail}_1$. By priority, $\mathsf{acc}(p_2, q_2)$ is added before $\mathsf{frb}^c_2(p_2, R_2^F)$, so this failure condition is properly detected by the incremental algorithm.

(c) Let $\mathsf{f}.\mathsf{frb}^c_i(p_i, R_i)$ be inferred after $\mathsf{acc}(p_1, q_1)$, where $q_1 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_1$ of sort i and $r_1 \notin R_i$. Again, we can assume $i = 1$ for reasons of symmetry. See Figure 12 for illustration. Literal $\mathsf{f}.\mathsf{frb}^c_1(p_1, Q_1)$ is justified by some literal $\mathsf{f}.\mathsf{acc}(p_2, r_2)$ added before, and the clause below where $R_1 = Q_1^F(r_2)$. Furthermore, $\mathsf{f}.\mathsf{acc}(p_2, r_2)$ stems from some literal $\mathsf{acc}(p_2, q_2)$ and the second clause:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \qquad r_2 \in \mathit{sta}_2(F)}{\mathsf{f}.\mathsf{frb}^c_1(p_1, R_1) :\!-\ \mathsf{f}.\mathsf{acc}(p_2, r_2).} \qquad \frac{p_2 \in \mathit{sta}(A) \qquad q_2 \xrightarrow{\epsilon}{}^{\leq 1} r_2}{\mathsf{f}.\mathsf{acc}(p_2, r_2) :\!-\ \mathsf{acc}(p_2, q_2).}$$

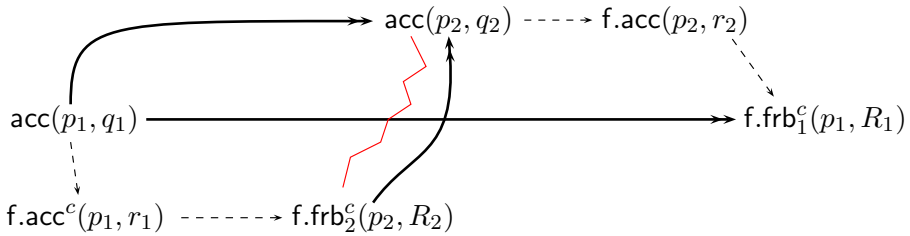

Figure 12. Early failure detection, case (c) with $i = 1$.

Due to lowest priority of acc literals again, $\mathsf{acc}(p_1, q_1)$ must be inferred before $\mathsf{acc}(p_2, q_2)$. The following clauses can be applied where $R_2 = Q_2^F(r_1)$:

$$\frac{p_1 \in \mathit{sta}(A) \qquad q_1 \xrightarrow{\epsilon}{}^{\leq 1} r_1}{\mathsf{f.acc}(p_1, r_1) \text{:-} \ \mathsf{acc}(p_1, q_1).} \qquad \frac{p_1 @ p_2 \to p \in \mathit{rul}(A) \qquad r_1 \in \mathit{sta}_1(F)}{\mathsf{f.frb}_2^c(p_2, R_2) \text{:-} \ \mathsf{f.acc}(p_1, r_1).}$$

Since $r_1 \in Q_1^F(r_2)$ if and only $r_2 \in Q_2^F(r_1)$, it follows from $r_1 \notin R_1$ that $r_2 \notin R_2$. Thus, $\mathsf{acc}(p_2, q_2)$ and $\mathsf{f.frb}_2^c(p_2, R_2)$ in $\mathit{lfp}(D_3(A, F))$ raise inclusion failure $\mathtt{fail}_1$. By priority, $\mathsf{f.frb}_2^c(p_2, R_2)$ is added before $\mathsf{acc}(p_2, q_2)$, so this failure is properly detected by the incremental algorithm. □

5 Stepwise Automata for Unranked Trees and DTDs

We lift the deterministic inclusion test to stepwise tree automata for unranked trees, so that they become applicable to deterministic DTDs. Factorization will turn out essential here.

An unranked signature Σ is a finite set of symbols (without arity restrictions). The set T_Σ^u of unranked trees over Σ is the least set that contains all pairs $a(t_1, \ldots, t_n)$ where $a \in \Sigma$ and $(t_1, \ldots, t_n)$ is a possibly empty sequence of unranked trees in T_Σ^u. Currying carries over literally from ranked to unranked trees. This yields bijective function $\mathit{curry} : T_\Sigma^u \to T_{\Sigma_@}$ which satisfies $\mathit{curry}(a(t_1, \ldots, t_n)) = a@\mathit{curry}(t_1)@\ldots@\mathit{curry}(t_n)$ for all unranked trees $a(t_1, \ldots, t_n) \in T_\Sigma^u$. Thus, we can reuse stepwise tree automata to recognize languages of unranked trees $L^u(A) = \{t \in T^u(\Sigma) \mid \mathit{curry}(t) \in L(A)\}$. Factorized automata carry over to unranked trees in the same manner, without any change. Thus, we obtain the following corollary from Theorem 14.

Corollary 15 *Let A be a stepwise tree automaton and F a deterministic factorized tree automata over the same signature Σ. Language inclusion with respect to unranked trees $L^u(A) \subseteq L^u(F)$ can be decided in time $O(|A| * |F|)$ independently of $|\Sigma|$.*

Note that hedge automata [1,14,15], whose horizontal languages are defined

```
<!ELEMENT doc    (block+)>
<!ELEMENT block  (text,(link,text?)?
                 |link,text?)>
<!ELEMENT text   (#PCDATA)>
<!ELEMENT link   (#PCDATA)>
```

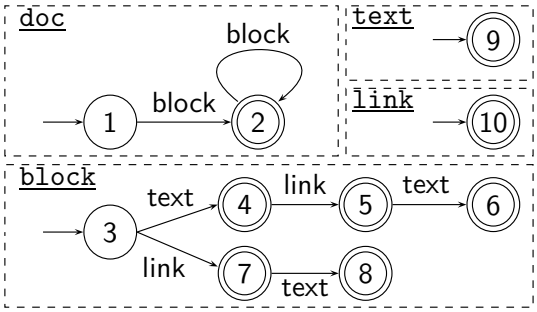

Figure 13. An example DTD and the corresponding Glushkov automata.

by finite word automata (called DFHF(DFA) in [1] and dUTA in [12]), can be translated in linear time to stepwise tree automata with ϵ-rules for unranked trees [12]. Note however, that the notion of bottom-up determinism for hedge automata is closer to unambiguity, so we cannot choose F to be a deterministic hedge automaton even if horizontal languages are represented by deterministic finite word automata.

We finally show how to convert deterministic DTDs D to deterministic factorized tree automata for unranked trees in time $O(|\Sigma|*|D|)$, so that we can reuse our algorithm for testing inclusion of stepwise tree automata in deterministic DTDs. Here, factorization avoids the quadratic blowup. When translating into stepwise tree automata [12], the number of rules may be quadratic, while the number of states is preserved. The problem is the implicit elimination of ϵ-rules.

A DTD D with elements in a set Σ is a function mapping letters $a \in \Sigma$ to regular expressions e over Σ, in which case we write $a \to_D e$. One of these elements is the distinguished start symbol. Let $L(e) \subseteq \Sigma^*$ be the word language defined by e. The language $L_a(D) \subseteq T^u_\Sigma$ of elements a of a DTD D is the smallest set of unranked trees such that:

$$L_a(D) = \{a(t_1, \ldots, t_n) \mid a \to_D e,\ a_1 \ldots a_n \in L(e), t_i \in L_{a_i}(D) \text{ for } 1 \leq i \leq n\}$$

The language of a DTD D is $L(D) = L_a(D)$ where a is the start symbol of D. The size of D is the total number of symbols in the regular expressions of D. An example in XML syntax is given in Figure 13. The set of elements of D is $\Sigma = \{\texttt{doc}, \texttt{block}, \texttt{text}, \texttt{link}\}$, of which the element `doc` is the start symbol. The regular expression for `#PCDATA` recognizes only the empty word.

A DTD is deterministic if all its regular expressions are one-unambiguous, as

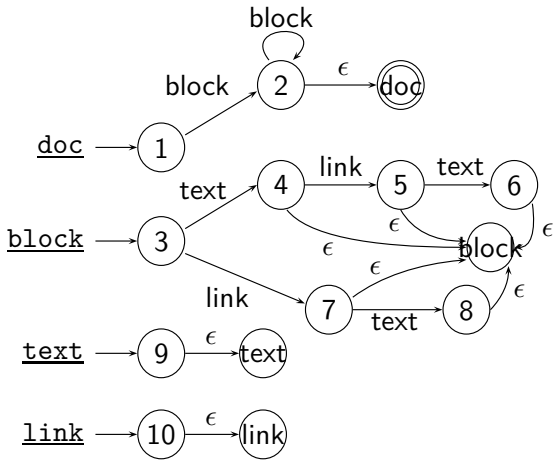

Figure 14. A representation of the deterministic factorized tree automaton for the DTD in Figure 13. Alphabet Σ is {doc,block,text,link}, states of sort 1 are $\{1, \ldots, 10\}$ and states of sort 2 are {doc,block,text,link}. A constant rule is for instance doc $\to 1$, a binary rule 2@block $\to 2$ and an ϵ-rule $2 \xrightarrow{\epsilon}$ doc.

required by the W3C. This is equivalent to say that all corresponding Glushkov automata are deterministic [11]. See Figure 13 for an example.

Theorem 16 (Brüggemann-Klein [21]) *The collection of Glushkov automata for a deterministic DTD D over Σ can be computed in time $O(|\Sigma| * |D|)$.*

Note that the construction of a Glushkov automaton of a regular expression e over alphabet Σ may take time $O(\Sigma * |e|^2)$ in the general case. Intuitively, the square factor is raised by eliminating occurring ϵ-rules on the fly. In the case of an unambiguous regular expression, the resulting Glushkov automaton will be deterministic. Its construction time is bounded by its size and thus in $O(|\Sigma| * |e|)$ because of determinism.

We transform a collection of Glushkov automata for a deterministic DTD D into a single factorized tree automaton F as follows. The set of states of sort 1 of F is the disjoint union of the states of the Glushkov automata. The states of sort 2 of F are the elements of D. For every element a, we connect all final states q of its Glushkov automaton to the state a, i.e., $q \xrightarrow{\epsilon} a \in rul(F)$. The only final state of F is the start symbol of the DTD D. The result is a finite automaton, that represents a factorized tree automaton, as for instance in Figure 14. This needs time of at most $O(|\Sigma| * |D|)$. Note that the size of the example automaton would grow quadratically, when eliminating ϵ-edges. For every $a \in \Sigma$, there is $a \to q \in rul(F)$ for the unique initial state q of the Glushkov automaton of a. For every transition $q \xrightarrow{a} q'$ of one of the Glushkov automata, we add a rule $q@a \to q' \in rul(F)$.

Note that F is deterministic as a factorized automaton. The ϵ-free part of F is deterministic since all Glushkov automata are: $\mathtt{d}_0$. Let q be a state of the

Glushkov automaton for some letter a. The only state of sort 1 q can reach by ϵ-edges is a and the only state of sort 2 is q itself. All other states of F are elements of $a \in \Sigma$, which have no outgoing ϵ-edges: $\mathtt{d}_1$.

Theorem 17 *Deterministic DTDs D over Σ can be translated to deterministic factorized tree automata recognizing the same language in time $O(|\Sigma| * |D|)$.*

Proof. The translation of a collection of Glushkov automata of a DTD to a factorized automata is in linear time. It is easy to check that it preserves the languages of unranked trees. The theorem thus follows from Theorem 16 by Brüggemann-Klein. □

Corollary 18 *Language inclusion of hedge automata A over Σ with horizontal languages defined by finite word automata in deterministic DTDs D with elements in Σ can be decided in time $O(|A| * |\Sigma| * |D|)$.*

Proof. From Corollary 15 and Theorem 17. □

6 Experiments

We have implemented the inclusion algorithm in Objective CAML, and have integrated it into a system for schema-guided learning of queries in XML trees in Champavère & al. [9]. In a first set of experiments, we consider inclusion tests for synthetic automata. Then we consider inclusion tests between automata and DTDs coming from realistic tasks in query learning.

Experiment 1. We modify the sizes of automata A and F when testing inclusion of $L(A)$ in $L(F)$. For this, we define $Mult_n$ as the minimal deterministic automaton for the language of trees of the form $f(a, \ldots, a)$ where the number of a leaves is a multiple of n. The first problem is to test inclusion of $L(Mult_n)$, n varying from 100 to 10000 with a 100-increment, into the minimal deterministic factorized automaton recognizing $L(Mult_{200})$. It should be noted that inclusion holds when $n/100$ is even. The second problem is to test inclusion of $L(Mult_{400})$ into $L(Mult_n)$ with n varying from 10 to 500 with a 10-increment. It should be noted that inclusion holds when $400/n$ is an integer.

We estimate the computation time for inclusion tests with and without early detection of inclusion failure (ED). We distinguish whether inclusion holds or not. Results are shown in Figure 15 for the first problem and in Figure 16 for the second problem.

It can be verified that the computation time of testing $L(A) \subseteq L(F)$ is linear in the size of the automaton A and in the size of automaton F. This confirms the theoretical results on complexity. It can also be seen that the computation

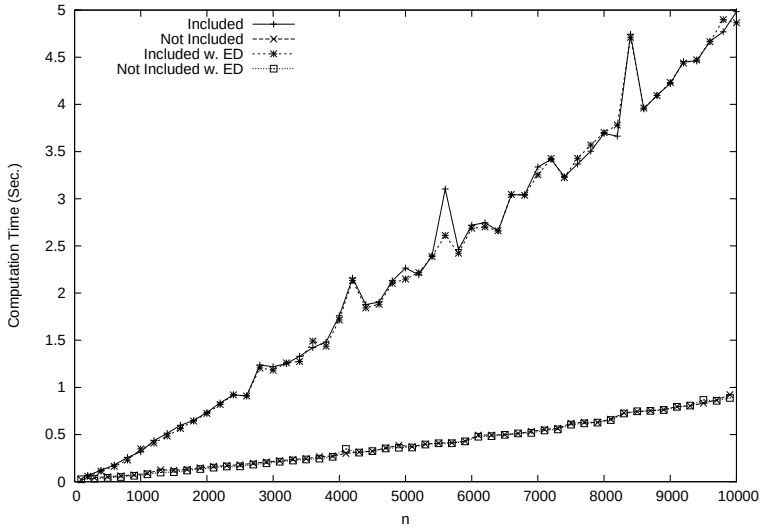

Figure 15. Computation time of testing $L(Mult_n) \subseteq L(Mult_{200})$.

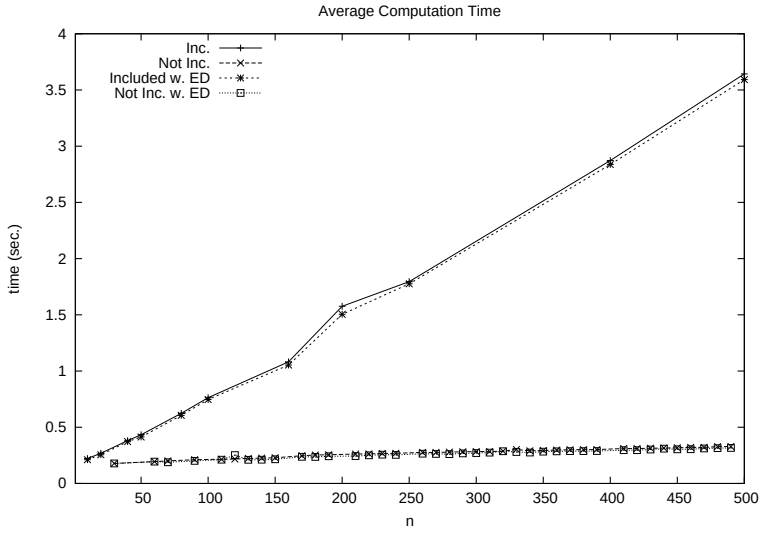

Figure 16. Computation time of testing $L(Mult_{400}) \subseteq L(Mult_n)$.

time is greater when inclusion holds. Otherwise, the computation time is lower since concurrent failure detection applies. In this experiment, there are no failures of type $\texttt{fail}_1$, so we do not use early failure detection. The gain is obtained by checking $\texttt{fail}_2$ concurrently, so that product automaton does not need to be computed entirely.

Experiment 2. In order to verify the usefulness of early detection of failures of type $\texttt{fail}_1$ (ED), we consider another example. We define $Mult2_n$ to be the minimal deterministic automaton for the language of trees of the form $g(f(a, \ldots, a))$, where the number of a leaves is a multiple of n. The problem is to test the inclusion of $L(Mult2_n)$, where n varies from 100 to 10000 with a 100-increment, into the minimal deterministic factorized automaton recognizing $L(Mult2_{200})$. The computation times are shown in Figure 17.

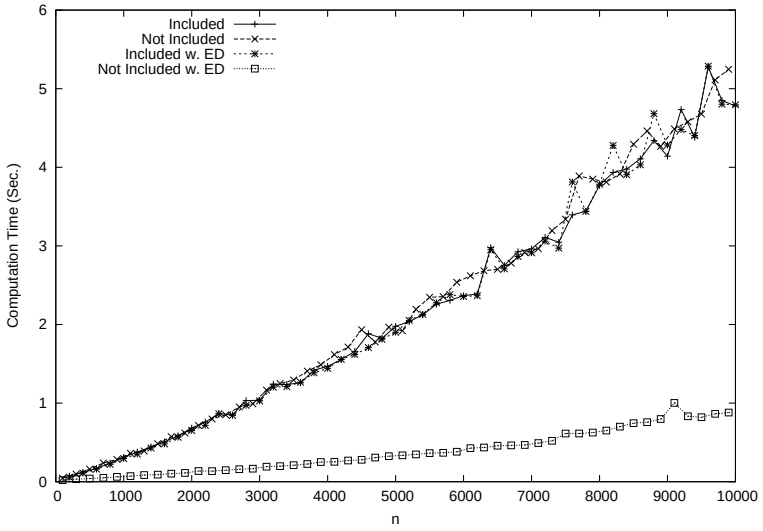

Figure 17. Average computation time of testing $L(Mult2_n) \subseteq L(Mult2_{200})$.

It can be noted that when inclusion does not hold, computation time is five times faster than for other cases. This is because, for these inclusion tests, inclusion failure comes from $\texttt{fail}_1$. Thus early failure detection allows to decrease dramatically the computation time. It can also be noted that the computation time is similar for cases where inclusion is verified (with or without early detection), and non inclusion cases without early detection. This is because, without early failure detection, computing a failure $\texttt{fail}_1$ implies to compute the whole product automaton.

Experiment 3. We now consider real-world data sets from the query induction problem. In the learning algorithm defined in Champavère & al. [9], an initial automaton is computed and is iteratively refined by merging states. A merge is accepted only if the language recognized by the new automaton still satisfies a given schema or DTD. Consequently, inclusion tests are done frequently. We compare the overall computation time of learning sessions where inclusion tests are done with or without early failure detection. We use the the transitional DTD of XHTML1 and the query learning benchmarks: Okra, Bigbook, Google and Yahoo, each of them with an increasing size of inputs. Results are shown in Figure 18.

It appears that the learning algorithm operates about twice faster with early detection of failure $\texttt{fail}_1$ than without in all benchmarks. This indicates that $\texttt{fail}_1$ occurs frequently in practice and that early detection improves efficiency a lot. More generally, it shows that the inclusion algorithm presented here can be used for real-world problems. In Champavère & al. [9], we have shown that introducing inclusion tests does not increase computation time while avoiding useless state merges, thus improving the query induction algorithm.

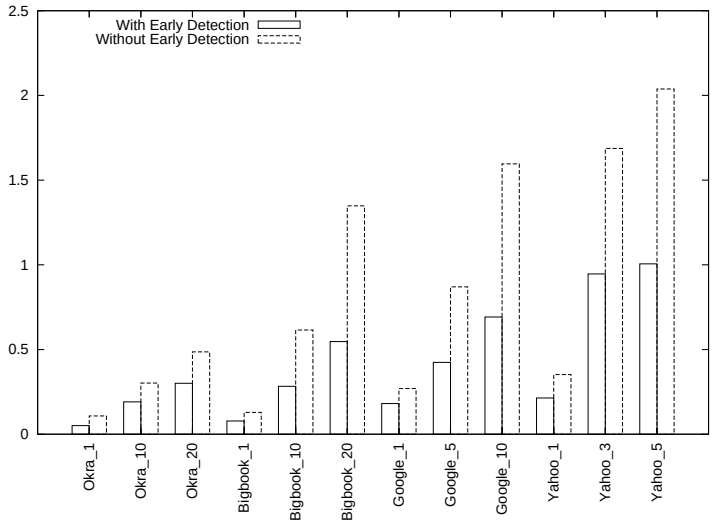

Figure 18. Average computation time of learning sessions with and without early detection of failure $\texttt{fail}_1$.

7 Top-Down Determinism

We show how to test inclusion for top-down deterministic tree automata by reduction to standard results from the literature, and by using Theorem 8 for words, seen as trees over monadic signatures. This has implications for testing inclusion in extended DTDs with restrained competition, but for tree automata recognizing first-child next-sibling encoding of unranked trees.

7.1 Tree Automata for Ranked Trees

A tree automaton A over a ranked signature Σ is *top-down deterministic* if for all symbols $f \in \Sigma$ of arity n and states $p \in \Sigma$ there are no two different rules $f(p_1, \ldots, p_n) \to p$ and $f(p'_1, \ldots, p'_n) \to p$ in $rul(A)$.

Proposition 19 *Let Σ be a ranked signature, and A and B be tree automata over Σ. If B is top-down deterministic, then we can decide language inclusion $L(A) \subseteq L(B)$ in time $O(|A| * |B|)$.*

We base the algorithm on the well-known fact that tree languages recognized by top-down deterministic tree automata are path closed [1]. The standard example for a non path closed regular language is $L_0 = \{f(a, a), f(b, b)\}$ where $a \neq b$. For sake of completeness, let us recall the definitions. The set of paths of a tree $t \in T_\Sigma$ is the subset of words $paths(t) \subseteq (\Sigma \cup \mathbb{N})^*$ defined as follows:

$$paths(a) = a, \quad \text{and} \quad paths(f(t_1, \ldots, t_n)) = \{fiw \mid 1 \leq i \leq n,\ w \in paths(t_i)\}.$$

For instance $paths(L_0) = \{f1a, f2a, f1b, f2b\}$. The path closure of a tree language $L \subseteq T_\Sigma$ is the set of all trees that contain only paths of trees in L:

$$path{-}clos(L) = \{t \mid paths(t) \subseteq paths(L)\}$$

We call L path closed if $L = path{-}clos(L)$. For instance $path{-}clos(L_0) = L_0 \cup \{f(a,b), f(b,a)\}$ so L_0 is indeed not path closed.

Lemma 20 *If L_2 is path closed then $L_1 \subseteq L_2$ iff $paths(L_1) \subseteq paths(L_2)$.*

Proof. Note that we do not assume L_1 to be path closed. The implication from left to right is trivial. For the inverse, assume $paths(L_1) \subseteq paths(L_2)$. If $t_1 \in L_1$ then $paths(t_1) \subseteq paths(L_1) \subseteq paths(L_2)$. Thus $t_1 \in path{-}clos(L_2)$, and this set is equal to L_2 by assumption of path closedness. □

Proof of Proposition 19. For every tree automaton A over Σ, we can construct a finite word automaton $P(A)$ over a finite subset of $\Sigma \uplus \mathbb{N}$ such that $L(P(A)) = paths(L(A))$. The rules of $P(A)$ are defined as follows:

$$\begin{aligned} rul(P(A)) = &\{p \xrightarrow{fi} p_i \mid f(p_1, \ldots, p_n) \to p \in rul(A),\ 1 \leq i \leq n\} \\ &\cup \{a \to p \mid a \to p \in rul(A)\} \\ &\cup \{p \xrightarrow{\epsilon} p' \mid p \xrightarrow{\epsilon} p' \in rul(A)\} \end{aligned}$$

The first kind of rules reads two letters at the same time, but can easily be rewritten into two rules reading each a single letter. Clearly, the construction of $P(A)$ is in time $O(|A|)$. Furthermore, $P(A)$ is deterministic iff A is top-down deterministic.

Given two tree automata A, B over Σ such that B is top-down deterministic, we can decide language inclusion between A and B by testing language inclusion for $P(A)$ and $P(B)$. Since $P(B)$ is deterministic this can be done in time $O(|P(A)| * |P(B)|)$ independently of the alphabet by Theorem 8, which is in time $O(|A| * |B|)$ independently of the signature. □

7.2 Inclusion in Restrained-Competition EDTDs

We consider inclusion in restrained-competition extended DTDs (EDTDs) for automata that recognize unranked trees modulo the firstchild-nextsibling encoding. With respect to this binary encoding, restrained-competition EDTDs can be mapped to top-down deterministic tree automata.

Restrained competition EDTDs are strictly more expressive than deterministic DTDs, but still permit to type all nodes of an XML document in 1-pass

streaming manner [22,16]. Consider for instance the regular language of unranked trees $L_1 = \{a(a)\}$. It cannot be recognized by any DTD, since it contains a trees with two types of a-nodes that need to be distinguished. Language L_1 can be recognized by a restrained-competition EDTD, with two states $(a,1)$ and $(a,2)$ for the two types of a nodes. The start state is $(a,1)$ and the rules are as follows:

$$(a,1) \to (a,2), \qquad (a,2) \to \epsilon$$

This small example shows already that we cannot lift our translation into deterministic stepwise tree automata from deterministic DTDs to restrained-competition EDTDs. We would need two rules $a \xrightarrow{\epsilon} (a,1)$ and $a \xrightarrow{\epsilon} (a,2)$ and thus bottom-up non-determinism. The problem is that the type of an a node is only determined once knowing the type of its parent. This information is available during top-down processing but not when working bottom-up as with stepwise tree automata.

Let us define EDTDs D over a signature Σ for unranked trees more formally. They consist of a finite set of states $\mathit{sta}(D) \subseteq \Sigma \times \mathbb{N}$, a subset of start states $\mathit{start}(D) \subseteq \mathit{sta}(D)$, and a collection of rules given by a function mapping states $q \in \mathit{sta}(D)$ to regular expressions e over $\mathit{sta}(D)$, in which case we write $q \to_D e$. The languages $L_q(D) \subseteq T^u_\Sigma$ of a states $q \in \mathit{sta}(D)$ are the smallest sets of unranked trees such that if $q = (a,i)$ for some $a \in \Sigma, i \in \mathbb{N}$ then:

$$L_q(D) = \{a(t_1,\ldots,t_n) \mid q \to_D e,\ q_1 \ldots q_n \in L(e), t_i \in L_{q_i}(D) \text{ for } 1 \leq i \leq n\}$$

The language of an EDTD is $L(D) = L_q(D)$ where q is the start state of D. The size of D is the total number of symbols in regular expressions of D.

Essentially, EDTDs are the same as hedge automata whose horizontal languages are defined by regular expressions. These can define all regular languages of unranked trees. An EDTD D is *restrained-competition* if it has a unique start state and for all regular expressions $q \to_D e$ of D there exist no two different states $(a,n_1),(a,n_2) \in \mathit{sta}(D)$ and words $u, v_1, v_2 \in \mathit{sta}(D)^*$ such that $u(a,n_1)v_1,\ u(a,n_2)v_2 \in L(e)$. Also, as for DTDs we call a restrained-competition EDTD *deterministic* if all its regular expressions are one-unambiguous.

We use the usual firstchild-nextsibling encoding of unranked trees into binary trees, in order to capture restrained-competition EDTDs by top-down deterministic tree automata. Let $\Sigma_\# = \Sigma \uplus \{\#\}$ be the ranked signature with a single constant $\#$ and a collection of binary function symbols $a \in \Sigma$. The firstchild-nextsibling encoding of a unranked tree $t \in T^u(\Sigma)$ is a binary tree in $T_{\Sigma_\#}$, for instance, $\mathit{fcns}(a(b,c(d,e),f) = a(b(c(d(\#,e(\#,\#)),\#),\#),\#)$. A tree automaton A over $\Sigma_\#$ recognizes unranked trees modulo this other binary encoding, so its unranked tree language is $L^u(A) = \{t \in T^u(\Sigma) \mid \mathit{fcns}(t) \in L(A)\}$.

Lemma 21 *For all deterministic restrained-competition EDTDs D over Σ, we can compute a top-down deterministic tree automaton B over $\Sigma_\#$ with the same unranked tree language $L^u(B) = L(D)$ in time $O(|\Sigma| * |D|)$.*

Proof. We first compute the collection of Glushkov automata G_q for regular expressions e in rules $q \to_D e$. These are finite word automata whose alphabet is a finite subset of $\Sigma \times \mathbb{N}$. Brüggemann-Klein [21] states that if e is deterministic, then G_q is a deterministic word automaton over $\Sigma \times \mathbb{N}$. Restrained-competition for e implies that if there are two rules $p \overset{(a,i)}{\to} p' \in \mathit{rul}(G_q)$ and $p \overset{(a,j)}{\to} p'' \in \mathit{rul}(G_q)$, then $i = j$ (if one considers trimmed automata). Determinism also implies that $p' = p''$. As a consequence, all G_qs are deterministic.

Furthermore, for all states $p \in \mathit{sta}(G)$ and letters $a \in \Sigma$ there is at most one transition $p \overset{(a,i)}{\to} p' \in \mathit{rul}(G_q)$, and from [21] $\mathit{sta}(G)$ is of size of D. As a consequence, the overall size of the collection of all G_qs is in $O(|\Sigma| * |D|)$ and it can be computed in this time.

From there, one can build the tree automaton B over signature $\Sigma_\#$ with $L^u(B) = L(D)$. The states of automaton B are the elements in $\cup_{q \in \mathit{sta}(D)} \mathit{sta}(G_q)$. It has a unique final state, which is the unique start state in $\mathit{init}(G_q)$, where q is the start state of D. The rules of B are defined as follows:

$$\frac{p \overset{(a,i)}{\to} p' \in \mathit{rul}(G_q) \qquad \mathit{init}(G_{(a,i)}) = \{p''\}}{a(p', p'') \to p \in \mathit{rul}(B)} \qquad \frac{p \in \mathit{fin}(G_q)}{p \to \# \in \mathit{rul}(B)}$$

Automaton B is top-down deterministic by construction and recognizes $L(D)$. The construction is in linear time in the size of the collection of Glushkov automata, so the overall construction requires time $O(|\Sigma| * |D|)$. □

The above automaton construction works in the non-deterministic case as well, but we have to determinize the Glushkov automata to get the resulting tree automaton to be top-down deterministic.

Corollary 22 *For tree automata A over $\Sigma_\#$ for unranked trees modulo firstchild-nextsibling encoding and deterministic restrained competition EDTDs D over Σ, language inclusion $L^u(A) \subseteq L(D)$ can be tested in time $O(|A| * |\Sigma| * |D|)$.*

Proof. We transform D into a top-down deterministic tree automaton B over $\Sigma_\#$ that recognizes the same unranked tree language by Lemma 21. This takes time $O(|\Sigma| * |D|)$, so the size of B is in $O(|\Sigma| * |D|)$, too. We then test $L(A) \subseteq L(B)$, which can be done in time $O(|A| * |\Sigma| * |D|)$ by Proposition 19, since B is top-down deterministic. □

8 Conclusion

We have presented a new efficient algorithm for testing language inclusion in deterministic tree automata. We have introduced a notion of determinism for factorized tree automata, and shown how to lift the inclusion test to them. We have made our algorithm fully incremental in both cases.

We have studied deterministic inclusion for various notions of tree automata for unranked trees. The base case with bottom-up determinism is inclusion in factorized stepwise tree automata. We have shown how to lift the base case to inclusion in deterministic DTDs, both for stepwise tree automata and hedge automata with horizontal languages defined by finite word automata (aka EDTDs with Glushkov automata instead of regular expressions). We have implemented our inclusion test, given experimental evidence for their efficiency, and applied them in schema-guided query induction [9]. Incrementality of our algorithm turns out to be essential for practical efficiency there.

We have studied language inclusion in top-down deterministic tree automata for ranked trees. In the case of unranked trees, this permits the inclusion in restrained competition-EDTDs, for tree automata recognizing unranked trees modulo firstchild-nextsibling encoding (so possibly EDTDs).

The previous polynomial time algorithms in Martens, Neven & Schwentick [19] can check inclusion in deterministic DTDs or deterministic restrained-competition EDTDs. It assumes however, that the devices on the left and on the right are of the same type, which is too restrictive in many cases, and in particular in the application to schema-guided query induction. In the common subset, their efficiency should be the same, even though the efficiency analysis provided there needs refinement to support this statement.

Future Work. There is one question on deterministic inclusion, that we have to leave open. This is whether one can test inclusion of stepwise tree automata A in restrained-competition EDTDs D over the same signature Σ in time $O(|A| * |\Sigma| * |D|)$.

There are two difficulties here. First, one cannot convert a stepwise tree automaton into an hedge automaton, or a standard automaton operating on unranked trees modulo the firstchild-nextsibling encoding, since such a transformation may introduce a quadratic blowup. Second, one cannot represent the collection of Glushkov automata of a deterministic restrained-competition EDTD by a deterministic stepwise tree automaton of the same size. The difference is that stepwise tree automata operate bottom-up while restrained-competition EDTDs work top-down. We hope to solve this problem in future work by studying inclusion in deterministic streaming tree automata [23,24,4], which can operate in a mixed top-down and bottom-up manner.

Acknowledgements

We thank Sławek Staworko for having pointed out to us the alternative algorithm for inclusion checking for top-down deterministic tree automata and its relevance for DTDs.

References

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications. Available online: `http://www.grappa.univ-lille3.fr/tata`, 1997, revised October 2007.

[2] H. Seidl, Deciding equivalence of finite tree automata, SIAM Journal on Computing 19 (3) (1990) 424–437.

[3] H. Seidl, Haskell overloading is DEXPTIME-complete, Information Processing Letters 52 (2) (1994) 57–60.

[4] A. Neumann, H. Seidl, Locating matches of tree patterns in forests, in: International Conference on Foundations of Software Technology and Theoretical Computer Science (FCTTCS), 1998, pp. 134–145.

[5] S. D. Zilio, D. Lugiez, Xml schema, tree logic and sheaves automata, in: International Conference on Rewriting Techniques and Applications (RTA), 2003, pp. 246–263.

[6] S. Maneth, A. Berlea, T. Perst, H. Seidl, XML type checking with macro tree transducers, in: Symposium on Principles of Database Systems (PODS), 2005, pp. 283–294.

[7] T. Schwentick, Automata for XML—a survey, Journal of Computer and System Science 73 (3) (2007) 289–315.

[8] T. Milo, D. Suciu, V. Vianu, Type checking XML transformers, Journal of Computer and System Science 1 (66) (2003) 66–97.

[9] J. Champavère, R. Gilleron, A. Lemay, J. Niehren, Schema-guided induction of monadic queries, 2008, `http://www.grappa.univ-lille3.fr/~champavere/Recherche/publications/CGLN08SGI.submitted.pdf`.

[10] J. Carme, J. Niehren, M. Tommasi, Querying unranked trees with stepwise tree automata, in: International Conference on Rewriting Techniques and Applications (RTA), 2004, pp. 105–118.

[11] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Information and Computation 142 (2) (1998) 182–206.

[12] W. Martens, J. Niehren, On the minimization of XML schemas and tree automata for unranked trees, Journal of Computer and System Science 73 (4) (2007) 550–583.

[13] S. Raeymaekers, Information extraction from web pages based on tree automata induction, Ph.D. thesis, Katholieke Universiteit Leuven (Jan. 2008).

[14] J. W. Thatcher, Characterizing derivation trees of context-free grammars through a generalization of automata theory, Journal of Computer and System Science 1 (1967) 317–322.

[15] A. Brüggemann-Klein, D. Wood, M. Murata, Regular tree and regular hedge languages over unranked alphabets: Version 1 (Apr. 07 2001), `http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz`.

[16] W. Martens, F. Neven, T. Schwentick, G. J. Bex, Expressiveness and complexity of XML schema, ACM Transactions on Database Systems 31 (3) (2006) 770–813.

[17] J. Champavère, R. Gilleron, A. Lemay, J. Niehren, Efficient inclusion checking for deterministic tree automata and DTDs, in: International Conference on Language and Automata Theory and Applications (LATA), 2008, to appear.

[18] A. Tozawa, M. Hagiya, XML schema containment checking based on semi-implicit techniques, in: International Conference on Implementation and Application of Automata (CIAA), 2003, pp. 51–61.

[19] W. Martens, F. Neven, T. Schwentick, Complexity and decision problems for XML Schemas and chain regular expressions. Journal extension of MFCS 2004 paper. Under review, `http://lrb.cs.uni-dortmund.de/~martens/data/mfcs04journal.pdf`.

[20] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM computing surveys 33 (3) (2001) 374–425.

[21] A. Brüggemann-Klein, Regular expressions to finite automata, Theoretical Computer Science 120 (2) (1993) 197–213.

[22] M. Murata, D. Lee, M. Mani, Taxonomy of XML schema languages using formal language theory, ACM Transactions on Internet Technology 5 (4) (2005) 660–704.

[23] O. Gauwin, J. Niehren, Y. Roos, Streaming tree automata, Information Processing Letters (2008). Accepted.

[24] R. Alur, Marrying words and trees, in: Symposium on Principles of Database Systems (PODS), 2007, pp. 233–242.

A Proof of Lemma 11

We prove both inclusions independently by induction over the construction of least fixed points. We first prove the inclusion from right to left, and then do the other way.

To each step of the computation of acc in $\mathit{lfp}(D_2(A,B))$ corresponds some steps for computing acc in $\mathit{lfp}(D_3(A,F))$. Let us assume that we are adding $\mathsf{acc}(p,q)$ to $\mathit{lfp}(D_2(A,B))$. One uses ($\mathrm{acc}_{/1}$), ($\mathrm{acc}_{/2}$) or ($\mathrm{acc}_{/3}$):

($\mathbf{acc}_{/1}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_2(A,B))$ is proved by a fact of $D_2(A,B)$ inferred as follows:

$$\frac{a \to p \in \mathit{rul}(A) \qquad a \to q \in \mathit{rul}(B)}{\mathsf{acc}(p,q).}$$

By (E_1) it holds that $a \to q \in \mathit{rul}(F)$, then we can apply the corresponding rule of $D_3(A,F)$ in order to show $\mathsf{acc}(p,q) \in \mathit{lfp}(D_3(A,F))$:

$$\frac{a \to p \in \mathit{rul}(A) \qquad a \to q \in \mathit{rul}(F)}{\mathsf{acc}(p,q).}$$

($\mathbf{acc}_{/2}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_2(A,B))$ is proved by a clause of $D_2(A,B)$ inferred from ϵ-rules of A:

$$\frac{p' \xrightarrow{\epsilon}_A p \qquad q \in \mathit{sta}(B)}{\mathsf{acc}(p,q) \coloneq \mathsf{acc}(p',q).}$$

Since B and F have the same states, $\mathsf{acc}(p,q) \in \mathit{lfp}(D_3(A,F))$, too.

($\mathbf{acc}_{/3}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_2(A,B))$ is contributed by applying a clause of $D_2(A,B)$ inferred as follows, under the assumption that $\mathsf{acc}(p_1,q_1)$, $\mathsf{acc}(p_2,q_2) \in \mathit{lfp}(D_2(A,B))$.

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \qquad q_1@q_2 \to q \in \mathit{rul}(B)}{\mathsf{acc}(p,q) \coloneq \mathsf{acc}(p_1,q_1), \mathsf{acc}(p_2,q_2).}$$

From (E_2) and $q_1@q_2 \to q \in \mathit{rul}(B)$ it follows that there are r_1, r_2 such that $r_1@r_2 \to q \in \mathit{rul}(F)$, $q_1 \xrightarrow{\epsilon}{}^*_F r_1$ and $q_2 \xrightarrow{\epsilon}{}^*_F r_2$. Thus:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \quad r_1@r_2 \to q \in \mathit{rul}(F) \quad q_1 \xrightarrow{\epsilon}{}^*_F r_1 \quad q_2 \xrightarrow{\epsilon}{}^*_F r_2}{\mathsf{acc}(p,q) \coloneq \mathsf{acc}(p_1,q_1), \mathsf{acc}(p_2,q_2).}$$

Since F is deterministic, $q_1 \xrightarrow{\epsilon}{}^*_F r_1$ is equivalent to $q_1 \xrightarrow{\epsilon}{}^{\leq 1}_F r_1$ and $q_2 \xrightarrow{\epsilon}{}^*_F r_2$ to $q_2 \xrightarrow{\epsilon}{}^{\leq 1}_F r_2$. Thus:

$$\frac{p_1@p_2 \to p \in \mathit{rul}(A) \quad r_1@r_2 \to q \in \mathit{rul}(F) \quad q_1 \xrightarrow{\epsilon}{}^{\leq 1}_F r_1 \quad q_2 \xrightarrow{\epsilon}{}^{\leq 1}_F r_2}{\mathsf{acc}(p,q) \coloneq \mathsf{acc}(p_1,q_1), \mathsf{acc}(p_2,q_2).}$$

Transformation rule ($\mathrm{acc}_{/3a}$) provides the following Datalog clause of $D_3(A,F)$:

$$\frac{p_1@p_2 \to p \in rul(A) \qquad r_1@r_2 \to q \in rul(F)}{\mathsf{acc}(p,q) :\!-\ \mathsf{f.acc}(p_1,r_1), \mathsf{f.acc}(p_2,r_2).}$$

The induction hypothesis yields $\mathsf{acc}(p_1,q_1), \mathsf{acc}(p_2,q_2) \in \mathit{lfp}(D_3(A,F))$. Transformation rule (f.acc) provides clauses which prove $\mathsf{f.acc}(p_1,r_1), \mathsf{f.acc}(p_2,r_2) \in \mathit{lfp}(D_3(A,F))$. In combination with the above clause of $D_3(A,F)$ this shows that $\mathsf{acc}(p,q) \in \mathit{lfp}(D_3(A,F))$.

Conversely we prove that each step of the computation of acc in $D_3(A,F)$ corresponds to some steps for computing acc in $\mathit{lfp}(D_2(A,B))$. Assume we are adding $\mathsf{acc}(p,q)$ to $\mathit{lfp}(D_3(A,F))$. One now uses ($\mathrm{acc}_{/1}$), ($\mathrm{acc}_{/2}$) or ($\mathrm{acc}_{/3a}$):

($\mathbf{acc}_{/1}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_3(A,F))$ is proved by a fact of $D_3(A,F)$ inferred as follows:

$$\frac{a \to p \in rul(A) \qquad a \to q \in rul(F)}{\mathsf{acc}(p,q).}$$

By (E_1) it holds that $a \to q \in rul(B)$, then we can apply the corresponding rule of $D_2(A,B)$ in order to show $\mathsf{acc}(p,q) \in \mathit{lfp}(D_2(A,B))$:

$$\frac{a \to p \in rul(A) \qquad a \to q \in rul(B)}{\mathsf{acc}(p,q).}$$

($\mathbf{acc}_{/2}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_3(A,F))$ is proved by a clause of $D_3(A,F)$ inferred from ϵ-rules of A:

$$\frac{p' \xrightarrow{\epsilon}_A p \qquad q \in sta(F)}{\mathsf{acc}(p,q) :\!-\ \mathsf{acc}(p',q).}$$

Since F and B have the same states, $\mathsf{acc}(p,q) \in D_2(A,B)$, too.

($\mathbf{acc}_{/3a}$) Membership of $\mathsf{acc}(p,q)$ in $\mathit{lfp}(D_3(A,F))$ is contributed by a clause of $D_3(A,F)$:

$$\frac{p_1@p_2 \to p \in rul(A) \qquad q_1@q_2 \to q \in rul(F)}{\mathsf{acc}(p,q) :\!-\ \mathsf{f.acc}(p_1,q_1), \mathsf{f.acc}(p_2,q_2).}$$

This implies that there are r_1, r_2 such that $r_1 \xrightarrow{\epsilon}{}^{\leq 1}_F q_1$ and $r_2 \xrightarrow{\epsilon}{}^{\leq 1}_F q_2$, and also that $\mathsf{acc}(p_1,r_1)$ and $\mathsf{acc}(p_2,r_2)$ belong to $\mathit{lfp}(D_3(A,F))$, because memberships of $\mathsf{f.acc}(p_1,q_1)$ and $\mathsf{f.acc}(p_2,q_2)$ in $\mathit{lfp}(D_3(A,F))$ can only be obtained using:

$$\frac{p_1 \in sta(A) \qquad r_1 \xrightarrow{\epsilon}{}^{\leq 1}_F q_1}{\mathsf{f.acc}(p_1,q_1) :\!-\ \mathsf{acc}(p_1,r_1).} \quad \text{and} \quad \frac{p_2 \in sta(A) \qquad r_2 \xrightarrow{\epsilon}{}^{\leq 1}_F q_2}{\mathsf{f.acc}(p_2,q_2) :\!-\ \mathsf{acc}(p_2,r_2).}$$

From (E_2) and all what precedes, it follows that $r_1@r_2 \to q \in rul(B)$ and thus:

$$\frac{p_1@p_2 \to p \in rul(A) \qquad r_1@r_2 \to q \in rul(B)}{\mathsf{acc}(p,q) :\!-\ \mathsf{acc}(p_1,r_1), \mathsf{acc}(p_2,r_2).}$$

By induction hypothesis $\mathsf{acc}(p_1,r_1)$ and $\mathsf{acc}(p_2,r_2)$ belong to $\mathit{lfp}(D_2(A,B))$,

Figure B.1. Algorithm in pseudo-language for testing inclusion.

```
// Inputs:
//   - A: productive stepwise tree automaton
//   - F: deterministic factorized tree automaton over the same signature
// Output: true iff L(A) ⊆ L(F)
fun inclusion(A,F)
  exception fail0 fail1 fail2

  ⟨⟨ create counters ⟩⟩

  ⟨⟨ create literal collection ⟩⟩

  ⟨⟨ create agenda with priorities ⟩⟩

  try
    // compute lfp(D3(A,F))
    // check for failures on the fly
    // raise exception once a failure condition gets valid

    ⟨⟨ saturate agenda with priorities ⟩⟩

    return true // no accessible states got forbidden!
  catch fail0 fail1 fail2 then
    return false
end
```

and then the rule ($\text{acc}_{/3}$) can be applied to get $\mathsf{acc}(p,q) \in \mathit{lfp}(D_2(A,B))$. □

B Implementation

We present more details on a concrete implementation of the inclusion test for factorized tree automata of Section 4. The same implementation can be used for tree automata without factorization, after conversion into factorized automata. We use a pseudo-functional programming language with imperative state, and have used Objective CAML for implementation in practice.

For simplicty, we restrict ourselves to an inclusion test without dynamic addition of new automata rules. It is not difficult, however, to make the same algorithm incremental in that respect, by returning the complete data structures at the end of the computation, rather than a Boolean value only.

The algorithm applies function `inclusion(A,F)` in Figure B.1 to a productive stepwise tree automaton A with ϵ-rules and a deterministic factorized tree automaton F. It computes the least fixed point of $D_3(A,F)$ by saturation. Failure condition $\texttt{fail}_1$ is tested using early detection as argued in Section 4.1. The two other failure conditions $\texttt{fail}_0$ and $\texttt{fail}_2$ are covered by saturation rules (fail_0) and (fail_{2a}). Once a failure is detected, an exception is raised in order to exit the saturation loop.

At initialization time, we create the counters $C(\mathit{pred}, p)$ and $C(\mathit{pred}, p, q)$ for all predicates $\mathit{pred} \in \{\mathsf{frb}^c_i,\ \mathsf{f}.\mathsf{frb}^c_i,\ \mathsf{frb}^c \mid i \in \{1,2\}\}$, states $p \in \mathit{sta}(A)$ and

Figure B.2. ⟨⟨ create literal collection ⟩⟩

```
// define a heap for memorizing acc and f.acc literals
let heap = Set.new(∅) in
// test membership of a litteral in the fixed point
fun Literals.mem(lit)
  case lit
  of acc(p,q) then
    return heap.mem(acc(p,q))
  of f.acc(p,q) then
    return heap.mem(f.acc(p,q))
end
fun Literals.induced(ind-lit)
  case ind-lit
  of frb(p,q) then
    case sort(q)
    of 1 then
      if q ∉ R_2^F then
        return C(frb_2^c,p,q) ≠ C(frb_2^c,p) // case (3')(a)
    of 2 then
      ... // symmetric to 1
    if C(frb^c,p,q) ≠ C(frb^c,p) then return true // case (3')(b)
  of f.frb(p,q) then
    return C(frb^c_{sort(q)},p,q) ≠ C(frb^c_{sort(q)},p) // case (3')(c)
end
// add a litteral to the fixed point
proc Literals.add(lit)
  case lit
  of acc(p,q) then
    if not Literals.mem(acc(p,q)) then
      if exists r ∈ sta(F) such that q →^ε_F r then
        if Literals.induced(f.frb(p,q)) or Literals.induced(f.frb(p,r))
            or Literals.induced(frb(p,q)) then
          raise fail_1 // early detection of fail_1
        if p ∈ fin(A) and q ∉ fin(F) and r ∉ fin(F) then
          raise fail_2 // apply (fail_2)
      else // ∄ r ∈ sta(F) such that q →^ε_F r
         // same as above but without r
      heap.put(acc(p,q))
      Agenda.put_low(acc(p,q))
  of f.acc(p,q) then
    if not Literals.mem(f.acc(p,q)) then
      heap.put(f.acc(p,q))
      Agenda.put_high(f.acc(p,q))
  of frb^c(p,∅) then
    Agenda.put_high(frb^c(p,∅))
  of frb_i^c(p,R) then
    Agenda.put_high(frb_i^c(p,R))
  of f.frb_i^c(p,Q) then
    Agenda.put_high(f.frb_i^c(p,Q))
end
```

$q \in sta(F)$. As stated in Section 4.1, $C(pred, p)$ counts the number of literals $pred(p, Q)$ inferred so far, and counter $C(pred, p, q)$ the number of occurrences of $q \in Q$ in literals $pred(p, Q)$ seen so far. Counters are implemented as objects, each of which provides a method `increments()`. Value comparison ($=$) between two counters returns true if their value is equal, false otherwise.

Furthermore, we create an object `Literals` in Figure B.2 that collects literals of the least fixed point of $D_3(A, F)$. Testing membership of a literal

Figure B.3. ⟨⟨ create agenda with priorities ⟩⟩

```
// define a higher priority stack
let high = Stack.new(∅) in
// define a lower priority stack
let low = Stack.new(∅) in
// interface
fun Agenda.nonempty()
  return high.nonempty() and low.nonempty()
end
fun Agenda.nonempty_high()
  return high.nonempty()
end
fun Agenda.nonempty_low()
  return low.nonempty()
end
fun Agenda.get_high()
  return high.pop()
end
fun Agenda.get_low()
  return low.pop()
end
proc Agenda.put_high(literal)
  high.push(literal)
end
proc Agenda.put_low(literal)
  low.push(literal)
end
```

to this collection can be done by calling function `Literals.mem(lit)`. For acc and f.acc literals, the test simply checks whether it has been added before. `Literals.induced(ind-lit)` allows to check whether frb or f.frb literals implicitely belong to $D_3(A, F)$, that is to say that $A, ta(F) \vdash \mathsf{frb}_i(p, q)$ of Lemma 12 is satisfied. This is done efficiently (i.e., whithout inferring the implicit literals for both predicates) as stated in Section 4.1, by checking disequality of corresponding counters.

We use an object `Agenda` defined in Figure B.3 that stores all literals to which some clauses remain to be applied. Literals in the agenda are either tagged with priority `high` or `low`, as required for early detection of failure $\mathtt{fail}_1$. High priority literals may serve in clauses that produce literals of high priority only. The first addition of a literal to the agenda is always with `high`. Once all consequences of high priority are produced, the tag is changed to `low`. Here, we optimize the previous processing by noticing that low priority sufficies for treating acc literals, as well as frb^c, frb_i^c and $\mathsf{f.frb}_i^c$ literals only have high priorities. The functions of object `Agenda` are `nonempty()`, `get_high()`, `get_low()`, `put_high(lit)`, and `put_low(lit)`.

Procedure `Literals.add` is implemented in Figure B.2. For literal acc, after ensuring that it does not already belong to the fixed point, one first checks whether the conditions for failure $\mathtt{fail}_1$ are satified, in which case an exception is raised. This point of the algorithm corresponds exactly to the early dectection of failure $\mathtt{fail}_1$. Afterwards it checks for $\mathtt{fail}_2$, accordingly to rule

Figure B.4. ⟨⟨ saturate agenda with priorities ⟩⟩

```
// schedule literals for constant rules
forall a,p such that a → p ∈ rul(A) do
  if exists q ∈ sta(F) such that a → q ∈ rul(F) then
    Literals.add(acc(p,q)) // apply (acc/1)
  else
    raise fail0 // apply (fail0)
// saturate agenda with priorities
while Agenda.nonempty() do
  while Agenda.nonempty_high() do
    ⟨⟨ apply rules with higher priority ⟩⟩
  if Agenda.nonempty_low() then
    ⟨⟨ apply rules with lower priority ⟩⟩
```

(fail_2). Eventually, the literal is memorized to the heap and then scheduled for saturation with low priority. For literal $\mathsf{f.acc}$, the operation is much more simple. No further checking than the existence in the fixed point is required to put it into the heap and then to the agenda with high priority. Predicates frb^c, frb^c_i and $\mathsf{f.frb}^c_i$ are always put to the agenda with high priority without further checking.

After initializations, the algorithm enters the saturation process of Figure B.4. In a first time, it applies the rule (acc_1) for adding accessible states of $A \times F$ from constants. In parallel it tests for failure $\texttt{fail}_0$ and raises the appropriate exception if the rule (fail_0) can be applied. During this process, the agenda is filled with low priority acc literals.

In a second step, the saturation procedure loops on the agenda and applies the priority policy as follows. It first applies all the rules for literals with the highest priority untill it is empty. Then it applies the rules for a unique literal with low priority. This step can indeed involve the scheduling of tasks with a higher priority. The loop stops whenever the whole agenda is empty, or if some exception is raised during saturation.

Applying rules with high priority is described in Figure B.5. It does not concern acc literals as previously stated. For $\mathsf{f.acc}$ literals, the highest priority is to infer $\mathsf{f.frb}^c_i$ literals with respect to sorts $i \in \{1, 2\}$ by applying (f.frb^c_1) and (f.frb^c_2) rules of Figure 9. Then the literal is scheduled for low priority operations. For literals frb^c, frb^c_i and $\mathsf{f.frb}^c_i$, there is no rule to apply. However, counters have to be incremented the way it has been described in Section 3.4. For the three predicates, the required operation is similar. Arguments are composed of a state p of A and a set Q (or R, or $\emptyset$) of states of F. Counter for literal named *pred* and state p is first incremented, and set Q is iterated over state q of F for incrementing the counter for *pred*, p and q.

Applying rules with low priority is described in Figure B.6. It concerns only

Figure B.5. ⟨⟨ apply rules with higher priority ⟩⟩

```
case Agenda.get_high()
of f.acc(p,q) then
  case sort(q)
  of 1 then
    let p1 = p in
      forall p2,p' such that p1@p2 → p' ∈ rul(A) do
        Literals.add(f.frb^c_2(p2,Q^F_2)) // apply (f.frb^c_2)
  of 2 then
    ... // apply (f.frb^c_1) symmetrically to 1
  Agenda.put_low(f.acc(p,q)) // schedule for low priority operations
of frb^c(p,∅) then
  C(frb^c,p).increments()
of frb^c_i(p,R) then
  C(frb^c_i,p).increments()
  foreach r ∈ R do
    C(frb^c_i,p,r).increments()
of f.frb^c_i(p,Q) then
  C(f.frb^c_i,p).increments()
  foreach q ∈ Q do
    C(f.frb^c_i,p,q).increments()
```

Figure B.6. ⟨⟨ apply rules with lower priority ⟩⟩

```
case Agenda.get_low()
of acc(p,q) then
  Literals.add(f.acc(p,q)) // apply (f.acc)
  if exists r ∈ sta(F) such that q →^ε_F r then
    Literals.add(f.acc(p,r)) // apply (f.acc)
  forall p' such that p →^ε_A p' do
    Literals.add(acc(p',q)) // apply (acc_/2)
  // apply (frb^c_1) and (frb^c_/a) for p as right state
  let (p2,q2) = (p,q) in
    forall p1,p' such that p1@p2 → p' ∈ rul(A) do
      Literals.add(frb^c_1(p1,R^F_1)) // apply (frb^c_1)
      if q2 ∉ R^F_2 then
        Literals.add(frb^c(p1,∅)) // apply (frb^c_/a)
  // apply (frb^c_2) and (frb^c_/a) for p as left state
  ... // symmetric to the left
of f.acc(p,q) then
  case sort(q)
  of 1 then
    let (p1,q1) = (p,q) in
      forall p2,p' such that p1@p2 → p' ∈ rul(A) do
        forall q2,q' such that q1@q2 → q' ∈ rul(F) do
          if Literals.mem(f.acc(p2,q2)) then
            Literals.add(acc(p',q')) // apply (acc_3a)
  of 2 then
    ... // symmetric to 1
```

literals acc and f.acc. For both, the job consists in verifying the necessary conditions to apply all the rules where they appear in the tail of a Datalog clause, namely (f.acc), ($\mathrm{acc}_{/2}$), (frb^c_1), (frb^c_2) and ($\mathrm{frb}^c_{/a}$) for acc literals, and only ($\mathrm{acc}_{/3a}$) for f.acc since other rules are applied with high priority.