Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates

Matthias Niewerth
University of Bayreuth

Luc Segoufin
INRIA and ENS Ulm

ABSTRACT

We consider the enumeration of MSO queries over strings under updates. For each MSO query we build an index structure enjoying the following properties: The index structure can be constructed in linear time, it can be updated in logarithmic time and it allows for constant delay time enumeration.

This improves from the previous known index structures allowing for constant delay enumeration that would need to be reconstructed from scratch, hence in linear time, in the presence of updates.

We allow relabeling updates, insertion of individual labels and removal of individual labels.

ACM Reference Format:
Matthias Niewerth and Luc Segoufin. 2018. Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates. In *PODS'18: 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3196959.3196961

1 INTRODUCTION

Query evaluation is a central task in databases and a vast literature is devoted to the study of its complexity. In particular computing the whole set of answers may be too demanding in terms of resources of the system as the set of answers may be much larger than the database itself.

There are many possibilities to overcome this problem: computing the "best" answers, returning the number of answers, sampling, enumerating. . . .

In this paper we view the query evaluation problem as an enumeration problem consisting in generating the answers one by one with some regularity. We try to achieve two goals. The first goal is to output the first solution as quickly as possible. The second goal is to have a small delay between any two consecutive solutions. An enumeration algorithm is then often divided into two parts: the *preprocessing phase*, before the first output, and the *enumerating phase*, when the solutions are generated. The time needed before outputting the first solution is called the *preprocessing time*. During the preprocessing phase the system produces an index structure that can be used by the second phase. The second phase consists in navigating within the index structure as fast as possible in order to compute the next solution from the current one.

Another interesting point of view is to think of the preprocessing phase as a builder of a compact representation of the output while the enumerating phase is a streaming decompression algorithm.

When viewing the query evaluation problem as an enumeration problem, the best we can hope for is a linear preprocessing time and a constant time delay. This of course cannot be always achieved but has been obtained in many interesting scenarios, MSO queries evaluated over structures of bounded treewidth [4, 16], FO queries over structures of bounded degree [11, 14], or over structures of low degree [12], or over structures of bounded expansion [15], XPath queries over XML documents [8]. . .

However all these results suffer from one limitation: if an update is made on the database the index structure built during the preprocessing phase has to be reconstructed completely from scratch. This issue has been addressed in two cases: FO queries and structures of bounded degree [7] and hierarchical conjunctive queries and arbitrary finite relational structures [6]. In these cases the index structure can be updated in constant time upon insertion and deletion of a tuple.

In this paper we also tackle this update issue and exhibit a new index that can be computed in linear time during the preprocessing phase, allows for constant-delay enumeration and can be updated in logarithmic time.

We consider MSO queries over strings (equivalently, MSO queries over structures of bounded pathwidth). Our index structure can be updated in logarithmic time upon the following updates: relabeling, insertion and removal of individual nodes.

As mentioned above, we already know index structures, computable in linear time, that permit constant delay enumeration for MSO queries over trees (or equivalently structures of bounded treewidth) [4, 16]. But it seems unlikely that those index structures can be updated in logarithmic time upon very simple updates such as relabeling a node.

This update issue has been considered in the *Boolean* case for MSO queries over words by [5]. Given an MSO Boolean query and an input word they construct in linear time an index structure such that, given the index structure, one can tell in constant time whether the input words satisfies or not the MSO query and moreover, the index structure can be modified in logarithmic time upon updates that can be either relabeling of a position of the word or adding/deleting a node at the beginning or end of the word. The same paper obtains similar results over trees but with a logarithmic square time for updating the index structure.

This result has been extended to non Boolean queries in [18] but with a non constant delay. In that paper, given an MSO query and a word they compute in linear time an index structure that allows for enumeration with logarithmic time delay and logarithmic time

PODS'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4706-8/18/06…$15.00
https://doi.org/10.1145/3196959.3196961

updates. For trees the complexity becomes logarithmic square delay and update time.

Our result improves on that by providing an index structure that can be computed in linear time, allows for constant delay enumeration and logarithmic time updates.

For this we use a new index structure that is completely different from those of [5, 16, 18]. Ours is based on Krohn-Rhodes theory. This theory says that any regular language (hence any Boolean MSO query) can be decomposed into a *cascade* of simple languages that are either "reset automaton" or "permutation automaton". In a reset automaton each input letter either induces the state of the automaton or does not change the state at all. In a permutation automaton each letter induces a permutation of the state of the automaton. The composition of these basic blocks is the wreath product [19].

Our index structure is constructed by induction on the cascade of reset and permutation automata used in the definition of the MSO query. For each block we show how to compose it with the previous one while maintaining the constant delay enumeration and logarithmic time update property. To demonstrate our index structure, we will use the following query.

Example 1.1 (Running Example). *Let $\Phi(x)$ be an MSO query that selects all positions x of a string v, such that in the prefix of v up to x the number of b's between the first position of label a and x is even.*

It is unclear whether our proof method can be extended to trees, as there are only preliminary results on Krohn-Rhodes like decompositions for trees [9]. We give a small discussion of these results in the conclusion.

Structure of the Paper

After giving the basic definitions in Section 2, we reduce the general problem to the special case of unary MSO formulas in Section 3. In Section 4, we prove a restricted variant of our main theorem that only deals with relabeling updates. This proof uses a data structure for string manipulation, whose implementation details are provided in Section 5. We explain how our algorithm and data structure can be extended to handle insertion and removal updates in Section 6 and conclude in Section 7.

2 PRELIMINARIES

Strings

We consider strings over a finite alphabet Σ. Let $v \in \Sigma^*$ be some string over Σ, then v_i denotes the label of v at position i and $v_{[i,j]}$ denotes the substring starting at position i up to position j.

For two strings v and w of the same length n, the join $v \bowtie w$ is defined as $(v_1, w_1) \dots (v_n, w_n)$. The joined string uses the alphabet $\Sigma_v \times \Sigma_w$, where Σ_v and Σ_w are the alphabets of v and w, respectively.

Automata

A *(deterministic, finite) automaton* (*DFA*) $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $\delta\colon Q \times \Sigma \to Q$ is a transition function, q_0 is the initial state, and F is the set of accepting states.

By δ^* we denote the extension of δ to strings, i.e., $\delta^*(q, w)$ is the state that can be reached from q by reading w. We define the size of an automaton to be the number of its states.

The language $L(\mathcal{A})$ accepted by $\mathcal{A}$ is the set $\{w \mid \delta^*(q_0, w) \in F\}$. A *regular language* is a language that can be recognized by a finite automaton.

Let δ be the transition function of an automaton with Q as set of states. For each symbol σ, $\delta^\sigma\colon Q \to Q$ denotes the induced transition function $q \mapsto \delta(q, \sigma)$.

We call an automaton a *reset automaton*, if for each symbol σ, δ^σ is either the identity on Q or a constant function.

We call an automaton a *permutation automaton*, if for each symbol σ, δ^σ is a permutation of Q. Each permutation automaton has an associated group $\mathcal{G} = (G, \odot, \mathbb{1})$ and an associated input homomorphism $\mathrm{h} : \Sigma \to G$, such that $G = Q$ and $\delta(q, \sigma) = q \odot \mathrm{h}(\sigma)$ for each $q \in Q$.

The terms reset automaton and permutation automaton were introduced by [19].

Transducer

A *transducer* T is a tuple $(Q, \Sigma, \Lambda, \delta, \lambda, q_0)$, where Q, Σ, δ, and q_0 are defined as in a DFA, Λ is a finite output alphabet and $\lambda\colon Q \times \Sigma \to \Lambda$ is an output function.

For a transducer T, we let $[T]$ be its input-output function, inductively defined by $[T](\varepsilon) = \varepsilon$ and $[T](va) = [T](v)\lambda(\delta^*(q_0, v), a)$. If the transducer T is clear from the context we allow to write $v^{\uparrow}$ to denote $[T](v)$.

Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, we denote by $\mathcal{A}^T$ the transducer that uses the same transitions as the automaton and outputs at every position the read symbol and the current state, i.e., $\mathcal{A}^T = (Q, \Sigma, \Sigma \times Q, \delta, \lambda, q_0)$ with $\lambda(q, \sigma) = (\sigma, q)$.

We denote a transducer as a reset transducer, if it results from a reset automaton. Likewise we use the term permutation transducer.

Cascade Product

Let $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma \times Q_1, \delta_2, q_2, F_2)$ be automata. Then the cascade product $\mathcal{A}_1 \circ \mathcal{A}_2$ is defined as the automaton

$$(Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F_1 \times F_2) \ ,$$

where $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, (a, p)))$.

The cascade product of n automata is defined as multiplication from the left, i.e.,

$$\mathcal{A}_1 \circ \dots \circ \mathcal{A}_n \stackrel{\text{def}}{=} \Big(\big(\cdots(\mathcal{A}_1 \circ \mathcal{A}_2)\cdots\big) \circ \mathcal{A}_{n-1}\Big) \circ \mathcal{A}_n \ .$$

It will be convenient to view the cascade product $\mathcal{A}_1 \circ \mathcal{A}_2$ as a composition of the associated transducers. Indeed it follows from the definitions that we can test whether a string $v \in \Sigma^*$ is accepted by $\mathcal{A}_1 \circ \mathcal{A}_2$ by reading the last letter of $[\mathcal{A}_2^T]([\mathcal{A}_1^T](v))$: it has to be of the form (a, q_1, q_2) with $\delta_1(q_1, a) \in F_1$ and $\delta_2(q_2, (a, q_1)) \in F_2$. We therefore denote by $\mathcal{A}_1^T \circ \mathcal{A}_2^T$ the corresponding composition of transducers.

We note that the operation $()^T$ distributes over the cascade product, i.e.,

$$(\mathcal{A}_1 \circ \dots \circ \mathcal{A}_n)^T = \mathcal{A}_1^T \circ \dots \circ \mathcal{A}_n^T \ .$$

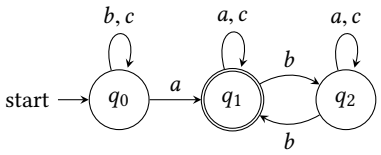
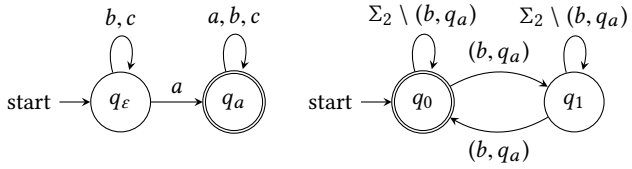


Figure 1: Automaton $\mathcal{A}$ for $\Phi(x)$ from Example 1.1



Automaton $\mathcal{A}_1$ Automaton $\mathcal{A}_2$

Figure 2: Krohn-Rhodes decomposition of $\mathcal{A}$

Krohn-Rhodes Theorem

Theorem 2.1 ([17, 19]). *Let $\mathcal{A}$ be a DFA. Then there exists a cascade $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$ of automata, such that*

- *$L(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n) = L(\mathcal{A})$; and*
- *each $\mathcal{A}_i$ is either a reset automaton or a permutation automaton.*

The original theorem by Krohn and Rhodes [17] was formulated over monoids and uses the wreath product instead of automata cascades. The version depicted above using cascades is by Maler [19] who also shows that the construction of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is effective.

In terms of transducers Krohn-Rhodes Theorem implies that recognition by an automaton $\mathcal{A}$ can be decided by checking the last letter of the output of a composition of transducers computed from $\mathcal{A}$, each of the transducers being either a reset or permutation automaton.

Example 2.1. *The query Φ from Example 1.1 can be answered using the automaton $\mathcal{A}$ depicted in Figure 1 by running $\mathcal{A}$ on the input string and returning all nodes visited by $\mathcal{A}$ in state q_1.*

In Figure 2 is depicted the Krohn-Rhodes decomposition of $\mathcal{A}$. The automaton $\mathcal{A}_1 = (\Sigma, Q_1, \delta_1, q_\varepsilon, \{q_a\})$ checks whether the prefix has an a, while the automaton $\mathcal{A}_2 = (\Sigma \times Q_1, \delta_2, q_0, \{q_0\})$ counts the number of b's modulo 2. Note, that $\mathcal{A}_2$ only counts b's after $\mathcal{A}_1$ has read the first a, as it only changes state on input symbol (b, q_a) and not on (b, q_ε). It is easy to verify that $\mathcal{A} = \mathcal{A}_1 \circ \mathcal{A}_2$. The automaton $\mathcal{A}_1$ is a reset automaton, as the labels b and c are self loops on all states while the labels a is a reset label that changes the state to q_a (regardless of the previous state). The automaton $\mathcal{A}_2$ is a permutation automaton, as the label (b, q_a) permutes both states, while all other labels perform the identity permutation.

Queries and enumeration

In this paper a query is a formula of monadic second order logic (MSO) with only first-order free variables. The *signature* of MSO queries on strings uses as universe all positions in a string, a linear order $<$ over the positions and for each symbol a, a unary relation R_a that contains exactly those positions labeled a. In the following, we use the relations $\leq$, $\geq$, $>$ and $=$ that can be derived from $<$ in the usual manner and we identify R_a with a, i.e., we write $a(x)$ to denote that position x is labeled a.

It is well known that over words, sentences of MSO correspond to regular languages. In the non Boolean case the answers of the query are the tuples of positions making the formula true.

We say that a query q can be enumerated using linear preprocessing time and constant delay over a class of databases (in this paper strings) if there exists an algorithm working in two consecutive phases given a database D in the class:

- a preprocessing phase, computing in linear time[1] an index structure;
- an enumeration phase, outputting one by one and without repetitions the set of answers to q over D with a constant time delay between any two consecutive outputs.

If moreover the intermediate index structure can be updated in logarithmic time upon an update of the databases we say that the enumeration algorithm has logarithmic time for updates. We consider the following updates: relabeling of a position, insertion and removal of individual positions.

3 REDUCTION TO THE UNARY CASE

In this section we reduce the problem to the unary case. Given a procedure that can enumerate unary MSO queries in lexicographic order with constant delay and logarithmic updates after linear preprocessing time, we show how to extend it to arbitrary MSO queries. This is based on classical composition lemmas for MSO over words and a normal form for MSO due to Simon's factorization forests.

We assume that from any unary MSO query ψ and any string v we can have, after a pre-processing time linear in $|v|$, a function $\text{next}(\psi, x)$ that computes from any position x of v the smallest position $y > x$ such that $v \models \psi(y)$ in constant time. Moreover this function can be maintained in logarithmic time upon updates. We will see in Section 4 how this can be achieved. Our goal in this section is to extend the unary case to arbitrary MSO queries.

3.1 Monomials: simple binary MSO queries

We first lift the unary case to simple binary queries that are called *monomials*.

A monomial over a finite alphabet A is a regular expression of the form $a_0 A_1^* a_1 A_2^* \cdots A_m^* a_m$ where the a_i are in A and the A_i are subset of A. They define binary queries over words v in A^*: the set of pairs of positions (x, y) such that the factor between x and y is in the language defined by the regular expression.

We will use the following properties of monomials

Proposition 3.1. *Let $\Psi = a_0 A_1^* \cdots A_m^* a_m$ be a monomial query and (x_1, y_1), (x_2, y_2), and (x_3, y_3) be answers to Ψ on a string v. Then the following statements are true:*

(a) Monotonicity 1: $y_i = \min\{z \mid v \models \Psi(x_i, z)\}$ and $x_1 \leq x_2$ imply $y_1 \leq y_2$

(b) Monotonicity 2: $y_i = \max\{z \mid v \models \Psi(x_i, z)\}$ and $x_1 \leq x_2$ imply $y_1 \leq y_2$

(c) Continuity: $x_1 = x_3$ and $y_1 \leq y_2 \leq y_3$ imply $v \models \Psi(x_1, y_2)$

[1] We use the RAM model of computation, which is standard when dealing with linear time.

Proof. We start with (a). Towards a contradiction, assume $y_2 < y_1$. Let $z_1^1, \ldots, z_m^1$ be positions attesting that $v \models \Psi(x_1, y_1)$, i.e., $v_{z_i^1} = a_i$ for $i \in [0, m]$ and $v_{[z_i^1+1, z_{i+1}^1-1]} \in A_{i+1}^*$ for $i \in [0, m-1]$. Let analogously $z_1^2, \ldots, z_m^2$ be positions attesting $v \models \Psi(x_2, y_2)$. Let k be the smallest number such that $z_k^1 \geq z_k^2$ (exists because $y_2 < y_1$). As $v_{[z_{k-1}^1+1, z_k^1-1]} \in A_k^*$, we can conclude that $v_{[z_{k-1}^1+1, z_k^2-1]} \in A_k^*$ and therefore the positions

$$z_1^1, \ldots, z_{k-1}^1, z_k^2, \ldots, z_m^2$$

are attesting $v \models \Psi(x_1, y_2)$. Hence $y_1 \leq y_2$, a contradiction.

We now prove (b). Towards a contradiction, assume $y_2 < y_1$.

Let $z_1^1, \ldots, z_m^1$ be positions attesting that $v \models \Psi(x_1, y_1)$, i.e., $v_{z_i^1} = a_i$ for $i \in [0, m]$ and $v_{[z_i^1+1, z_{i+1}^1-1]} \in A_{i+1}^*$ for $i \in [0, m-1]$. Let analogously $z_1^2, \ldots, z_m^2$ be positions attesting $v \models \Psi(x_2, y_2)$. Let k be the smallest number such that $z_k^1 \geq z_k^2$ (exists because $y_2 < y_1$). As $v_{[z_{k-1}^1+1, z_k^1-1]} \in A_k^*$, we can conclude that $v_{[z_{k-1}^1+1, z_k^2-1]} \in A_k^*$ and therefore the positions

$$z_1^2, \ldots, z_{k-1}^2, z_k^1, \ldots, z_m^1$$

are attesting $v \models \Psi(x_2, y_1)$. Hence $y_1 \leq y_2$, a contradiction.

It remains to prove (c). We consider two cases depending on the relative position of x_1 and x_2.

Assume $x_1 \leq x_2$. If $y_2 = y_3$ we are done. So assume $y_2 < y_3$. The construction of the proof of (a) then gives a witness for $v \models \Psi(x_1, y_2)$ as desired.

Assume now $x_2 \leq x_1$. If $y_2 = y_1$ we are done. So assume $y_1 < y_2$. The construction of the proof of (b) then gives a witness for $v \models \Psi(x_1, y_2)$ as desired. □

Given a monomial query Ψ and a string v, call "blue" a position of the set of the first projection of the answers to Ψ on v and "red" a position of the set of the second projection. Hence Ψ returns a subset of the pairs of blue-red nodes. Positions are naturally ordered by the order on v. The proposition says that given a blue node, the set of red nodes that forms a solution with it forms a segment among the red nodes (continuity). Moreover this segment shifts to the right when we move from one blue node to the next one to its right (monotonicity).

This justifies the following unary MSO queries associated to any monomial Ψ:

$$\begin{aligned}
\Psi_1(x) &= \exists y\ \Psi(x, y) \\
\Psi_2(y) &= \exists x\ \Psi(x, y) \\
\Psi_2^{\min}(y) &= \exists x\ \left[\Psi(x, y) \wedge \forall y' < y\ \neg\Psi(x, y')\right] \\
\Psi_2^{\max}(y) &= \exists x\ \left[\Psi(x, y) \wedge \forall y' > y\ \neg\Psi(x, y')\right] \\
\Psi_1^*(x) &= \exists y\ \Big[\Psi(x, y) \wedge (\forall y' < y\ \neg\Psi(x, y')) \wedge \\
&\qquad \forall x' < x\ [\Psi_1(x') \rightarrow \exists y'' < y\ \Psi(x', y'')]\Big] \\
\Psi_1^{**}(x) &= \exists y\ \Big[\Psi(x, y) \wedge (\forall y' > y\ \neg\Psi(x, y')) \wedge \\
&\qquad \forall x' < x\ [\Psi_1(x') \rightarrow \neg\Psi(x', y)]\Big]
\end{aligned}$$

The first two queries compute the blue and red nodes. The third and forth queries compute the set of red nodes that are minimal (resp. maximal) for some blue node. The fifth and sixth query return those blue nodes that induce a strict shift of their associated red segment.

Algorithm 1 Enumerate Monomial

```
1: function NEXT(x, y, y_min, y_max)
2:     if y < y_max then
3:         y ← next(Ψ_2, y)
4:     else if x < x_max then
5:         x ← next(Ψ_1, x)
6:         if Ψ_1^*(x) then y_min ← next(Ψ_2^min, y_min)
7:         if Ψ_1^**(x) then y_max ← next(Ψ_2^max, y_max)
8:         y ← y_min
9:     output(x, y)
```

We now show how to enumerate monomial queries assuming we can enumerate their associated unary MSO queries. The lexicographically smallest solution can simply be computed by taking the smallest blue node with its associated minimal red node. These nodes can be obtained in constant time, as we assume that unary queries can be enumerated lexicographically. The enumeration of the remaining solutions is based on the function described in Algorithm 1. Given a solution (x, y) this function computes in constant time the next solution. The enumeration is then trivial. The algorithm assumes that we have precomputed a global constant called $x_{\max}$ denoting the maximal blue node. It also maintains two variables $y_{\min}$ and $y_{\max}$ containing the minimal and maximal y associated to the current solution x.

We note that the calls to next in the Algorithm are enumerating the unary MSO formulas Ψ_1, Ψ_2, $\Psi_2^{\min}$, and $\Psi_2^{\max}$. They are not recursive calls.

We first show that the algorithm is correct. If the test of line 2 is true then it follows from Proposition 3.1(c), that the next solution is indeed computed by line 3. Otherwise the next solution is the next blue node and the next minimal associated red node. Line 5 computes the next blue node and line 6 computes the associate minimal red nodes. It is either the current one or the next one as shown by Proposition 3.1(a). It remains to updates value of $y_{\max}$ and this is done by line 7. It's correct because of Proposition 3.1(b).

We now argue that it can be implemented to work in constant time. The only difficulty are the tests Ψ_1^* and Ψ_1^{**} in lines 6 and 7. These tests can be done in constant time by making use of the fact that we go through the blue nodes in increasing order, as we assume that enumeration of unary queries can be done lexicographically. We argue for $\Psi_1^*(x)$, the tests for Ψ_1^{**} can be eliminated in the same way.

In order to perform the test $\Psi_1^*(x)$ we add an extra variable x_* that we initialize with the second value[2] satisfying Ψ_1^*. We then replace the test $\Psi_1^*(x)$ with $x = x_*$. If the test is true then we replace x_* with $\text{next}(\Psi_1^*, x_*)$ before proceeding. If the test is not true, then $x < x_*$, and we are sure that Ψ_1^* is not true.

Finally note that there is no extra preprocessing. The only preprocessing is the one of the unary queries associated to Ψ. And those can be updated in logarithmic time. This solves the monomial case. Notice that the enumeration is done in lexicographical order.

[2]The first value satisfying Ψ_1^* is the first blue node.

3.2 From monomials to arbitrary MSO queries.

Of course monomials are definable in MSO. It turns out that, modulo recoloring, MSO queries are essentially Boolean combination of monomials. This observation is a simple consequence of a result by Colcombet and has been used in several places, for instance [16]. For the sake of completeness we sketch the proof below. All details are present in [16].

Adding unary MSO interpretation to monomials. We note that the enumeration procedure for monomials presented in Section 3.1 immediately extends to monomials where each letter is interpreted as a unary MSO query. That is, $a(x)$ should be understood as $\phi_a(x)$ for some unary MSO query ϕ_a, or in more detail, a monomial $a_0A_1^*a_1A_2^*\cdots A_m^*a_m$ should be understood as the formula that says that there are positions $x_0 < x_1 < \cdots < x_m$, such that $\phi_{a_i}(x_i)$ holds for each i and every position y between x_{i-1} and x_i satisfies $\phi_a(y)$ for some $a \in A_i$. Indeed none of the proofs in Section 3.1 relies on the fact that a is a letter but just on the fact that it is a property of the current position.

From monomials to binary queries. This is a key step that follows from Colcombet's normal form of MSO over words [10]. This result says that any binary MSO query is equivalent, over words, to a query of the form $\exists\bar{y}\forall\bar{z}\theta$ where θ is a Boolean combination of atoms of the form $a(x)$ or $x < y$ and unary MSO formulas. It is then easy to see that such formulas are union of monomials with unary MSO interpretations for letters [3, 21]. See [16] for details.

Each monomial of the union can be enumerated lexicographically with constant delay and logarithmic updates from Section 3.1. Because the enumeration is lexicographic, the whole union can also be enumerated with constant delay (folklore, see for instance [13]). The enumeration stores the last seen result for each monomial of the union and always outputs the smallest of these results.

From binary queries to arbitrary queries. This is a simple consequence of the Composition Lemma: any MSO query over words is a union of conjunctions of binary queries. Each conjunct induces a linear order constraint among all variables. Hence the union is strict, i.e. no answer can be an answer of two disjuncts, and each disjunct can be treated separately. Furthermore we can assume that each disjunct involves consecutive variables in the linear order. We therefore have an induced linear order among the formulas of the conjunct that can be treated lexicographically one by one using the procedure described above.

Altogether this shows that once the unary case is solved, any MSO query can be enumerated with constant delay and logarithmic updates after a linear pre-processing phase.

4 ENUMERATION OF UNARY QUERIES

In this section we consider only relabeling updates and give an enumeration algorithm for unary MSO queries. We use a data structure allowing for constant delay enumeration and logarithmic time updates. We only give in this section a high level description of the data structure with its main features. In Section 5 we provide the technical details of its implementation. In Section 6 we will see how to handle the remaining updates such as insertions and removals. It turns out that relabeling is the hardest case.

Theorem 4.1. *On strings, unary MSO queries can be enumerated using linear preprocessing time, constant delay and logarithmic time for relabeling updates.*

As usual when dealing with MSO formula, the constants are non-elementary in the size of the formula. In order to get better constants one needs to use equivalent but more verbose query languages.

The rest of this section is devoted to the proof of Theorem 4.1. The first step is to decompose a unary MSO query into regular languages. This is based on the following folklore result[3]:

Lemma 4.2. *To any unary MSO query $\Psi(x)$ there exists a finite set I and Boolean MSO formulas $\Phi_{i,\text{left}}$ and $\Phi_{i,\text{right}}$ for $i \in I$ such that for every word v of length n and any position a of v we have*

$$v \models \Psi(a) \quad \text{iff} \quad \bigvee_{i\in I} \left(v_{[1,a]} \models \Phi_{i,\text{left}} \wedge v_{[a,n]} \models \Phi_{i,\text{right}}\right) .$$

Because Boolean MSO formulas define regular languages, we view the $\Phi_{i,\text{left}}$ and $\Phi_{i,\text{right}}$ as regular languages. Because regular languages are closed under Boolean operations, we can further assume in Lemma 4.2 that the union is disjoint. We can therefore treat each component of the disjunct separately as this will not give any duplicate during the enumeration phase. In the following we assume that we work with a unary MSO query $\Psi(x)$, given as a pair $(\Phi_{\text{left}}, \Phi_{\text{right}})$ of regular languages as in Lemma 4.2.

In Subsection 4.1, we provide a data structure allowing for constant delay enumeration and logarithmic time updates for the unary query returning those positions x such that initial segment up to x belongs to the regular language Φ_{left}. The case of Φ_{right} is mostly symmetric to the previous one and is briefly described in Section 4.2. The data structure representing $\Psi(x)$ is essentially a cross-product of the previous two structures. In Subsection 4.4 we sum up and give the proof of Theorem 4.1.

For our example query $\Phi(x)$, Lemma 4.2 yields exactly one pair of languages, where the left language is the language defined by the automaton $\mathcal{A}$ of Figure 1 and the right language accepts every string, as $\Phi(x)$ does not care which symbols occur to the right of x.

4.1 Data structure for Φ_{left}

Let $\mathcal{A}_1^T \circ \cdots \circ \mathcal{A}_m^T$ be the transducers decomposing the regular language Φ_{left} as given by Theorem 2.1. By construction, each transducer is either a reset transducer or a permutation transducer. We assume that $\mathcal{A}_i$ has Q_i as set of states, δ_i as transition function, Σ_i as alphabet and, in the case of permutation transducer, G_i as associated group and h_i as associated homomorphism function $\Sigma_i \to G_i$.

The general idea is as follows. Given an input string v our data structure maintains its cascade images by the various transducers, i.e. $\mathcal{A}_1^T(v)$, $\mathcal{A}_2^T(\mathcal{A}_1^T(v))$ and so on. We view each string as a *layer*, the bottom layer being v, the bottom but one layer being $\mathcal{A}_1^T(v)$ and the top layer being $\mathcal{A}_m^T(\cdots(\mathcal{A}_1^T(v))\cdots)$. For each string in the data structure, we make sure we can enumerate with constant delay all positions having a label from a fixed set of labels. Applying this to the top string and to the accepting states yields the constant delay enumeration. Updates are propagated bottom-up. The relabeling

[3] It can be proved using a simple Ehrenfeucht-Fraïssé game argument.

update on $S_{\ell-1}$:	$v_i \leftarrow \sigma$	$v_{[i,j]} \leftarrow w_{[i,j]}$
for every $q \in Q_\ell$:	$v_i^q \leftarrow (\sigma, q)$ $v_i^\uparrow \leftarrow (\sigma, \text{project}_{Q_\ell}(v_i^\uparrow))$ $v^\uparrow_{[i+1,k_i^v]} \leftarrow v^{p^v_{i+1}}_{[i+1,k_i^v]}$	$v^q_{[i,j]} \leftarrow w^q_{[i,j]}$ $v^\uparrow_{[i,j]} \leftarrow w^\uparrow_{[i,j]}$ $v^\uparrow_{[i,k_i^v]} \leftarrow v^{p_i^v}_{[i,k_i^v]}$ $v^\uparrow_{[j+1,k^v_{j+1}]} \leftarrow v^{p^w_{j+1}}_{[j+1,k^v_{j+1}]}$
for every $g \in G_\ell$:	$v_i^g \leftarrow (\sigma, \text{project}_{G_\ell}(v^g))$ $v^g_{[i+1,n]} \leftarrow v^{g \odot g_1}_{[i+1,n]}$	$v^g_{[i,j]} \leftarrow w^{g \odot g_2}_{[i,j]}$ $v^g_{[j+1,n]} \leftarrow v^{g \odot g_3}_{[j+1,n]}$

where

- $k_i^v \stackrel{\text{def}}{=}$ first position of a reset label in v after i
- $p_i^v \stackrel{\text{def}}{=} \delta(\text{project}_{Q_\ell}(v^\uparrow_{i-1}), v_{i-1})$
- $g_1 \stackrel{\text{def}}{=} \text{h}_\ell(v_{[1,i-1]}) \odot \text{h}_\ell(\sigma) \odot \text{h}_\ell(v_{[1,i]})^{-1}$
- $g_2 \stackrel{\text{def}}{=} \text{h}_\ell(v_{[1,i-1]}) \odot \text{h}_\ell(w_{[1,i-1]})^{-1}$
- $g_3 \stackrel{\text{def}}{=} \text{h}_\ell(v_{[1,i-1]}) \odot \text{h}_\ell(w_{[i,j]}) \odot \text{h}_\ell(v_{[1,j]})^{-1}$

Table 1: How updates are propagated from layer $S_{\ell-1}$ to layer S_ℓ. Top: Update in in a string v of $S_{\ell-1}$. Middle: Update if S_ℓ is a reset layer. Bottom: Update if S_ℓ is a permutation layer.

of the bottom layer may require various updates to the next layer which in turns may require significant changes to the next layer and so on. We cope with this problem as follows: we enrich the data structure with several string at each layer, intuitively each added string contains a precomputation of one of the expected changes, and we allow for internal updates slightly more complicated than just relabeling.

We now turn to the details. To each $\ell \le m$ we associate the layer set S_ℓ defined by induction assuming S_0 contains exactly one string v_{input}, which is the current input string:

$$S_\ell = \{v^\uparrow \mid v \in S_{\ell-1}\} \cup \{v^q \mid v \in S_{\ell-1}, q \in Q_\ell\}$$
if $\mathcal{A}_\ell^T$ is a reset transducer

$$S_\ell = \{v^g \mid v \in S_{\ell-1}, g \in G_\ell\}$$
if $\mathcal{A}_\ell^T$ is a permutation transducer

Here, for $v = \sigma_1 \dots \sigma_n$, v^q is the string $(\sigma_1, q) \dots (\sigma_n, q)$ for each $q \in Q_\ell$ and v^g is the string $v^g = (\sigma_1, g_1) \dots (\sigma_n, g_n)$ with $g_1 = g$ and $g_{i+1} = \delta_\ell(g_i, \alpha_i)$.

We note that for permutation layers, $v^\uparrow = v^{\mathbb{1}}$, hence each layer ℓ contains the image by $\mathcal{A}_\ell^T$ of all strings of the previous layer. Notice that all S_ℓ can be easily computed in linear preprocessing time.

In our running example, we thus have three layers, S_0, the lowest one containing the input string, S_1, the middle one associated to the reset automaton $\mathcal{A}_1$ from Figure 2, and S_2, the upper one, associated with the permutation automaton $\mathcal{A}_2$ from Figure 2.
We will illustrate our algorithm with the string $u = cbabcb$. In this case, initially S_0 contains the string u, S_1 initially contains the three strings

$$u^\uparrow = (c, q_\varepsilon)(b, q_\varepsilon)(a, q_\varepsilon)(b, q_a)(c, q_a)(b, q_a)\ ,$$

and

$$u^q = (c, q)(b, q)(a, q)(b, q)(c, q)(b, q) \quad \text{for} \quad q \in \{q_\varepsilon, q_a\}\ .$$

We allow the following updates on the strings in S_ℓ:

Definition 4.3 (Update Operations). *Here, v and w are strings from S_ℓ, and i and j are positions.*

- relabel: $v_i \leftarrow \sigma$
- replace: $v_{[i,j]} \leftarrow w_{[i,j]}$

The relabel operations changes the label at position i to σ, while the replace operation replaces the substring of v from position i to position j (inclusive) with the substring that resides in w at the same position.

In our running example we will consider an update $u_1 \leftarrow a$ that changes the first position of u to label a.

Furthermore, the data structure allows the following queries:

Definition 4.4 (String query operations).

- $\text{searchlabel}^{\rightarrow}(v, i, \Sigma')$, $\text{searchlabel}^{\leftarrow}(v, i, \Sigma')$
- $\text{enumerate}(v, \Sigma')$

The searchlabel *operation returns the biggest (smallest) position in v with some label $\sigma \in \Sigma'$ that is equal or larger (equal or smaller) than i. The* enumerate *operation enumerates all positions in v with some label $\sigma \in \Sigma'$.*

We show in Section 5, how these operations (except enumerate) can be implemented to take only logarithmic time. Furthermore, we will show that enumerate works with constant delay. Obviously, each layer has to react to any of the updates listed in Definition 4.3 from the previous layer. To ensure overall logarithmic time updates, each layer may do only constantly many updates for each update of the previous level. We show how this is done in the remaining part of this section. The propagation of the updates are summarized in Table 1. In the table we use project_{Q_ℓ} to denote the projection of some symbol to its component from Q_ℓ, i.e., keeping only the state from automaton $\mathcal{A}_\ell$ and disregarding all other information. The projection is extended from symbols to strings in the obvious way.

Reset layer. Assume $\mathcal{A}_\ell^T$ is a reset transducer and that a relabel $v_i \leftarrow \sigma$ is performed on a string v of $S_{\ell-1}$. In order to update S_ℓ we need to do the following: First we need to change the label at position i of all strings of S_ℓ that are derived from v. Second we need to make sure that the states of $v^\uparrow$ are modified appropriately. To do this let p be the state reached by $\mathcal{A}_\ell^T$ on v at position i (which can be recovered using $\text{project}_{Q_\ell}(v_i^\uparrow)$) as in Table 1). Let q be the state $\delta_\ell(p, \sigma)$ (denoted p^v_{i+1} in Table 1). Notice that q is the new state of the transducer $\mathcal{A}_\ell^T$ at position $i+1$ after reading σ in state p. Because $\mathcal{A}_\ell^T$ is a reset transducer this state remains unchanged until a reset letter is read. Let j be this position (denoted k_i^v in

Table 1). Notice that j can be computed from our data structure using the query $\text{searchlabel}^{\rightarrow}(v, i, \Sigma')$ where Σ' contains all reset letters of $\mathcal{A}_\ell^T$. Notice that because $\mathcal{A}_\ell^T$ is a reset transducer, all positions of $v^\uparrow$ before i and after j are unaffected by the update on v. Hence S_ℓ is correctly updated by the sequence of updates depicted in Table 1: $v_i^u \leftarrow (\sigma, u)$ for all $u \in Q_\ell$, $v_i^\uparrow \leftarrow (\sigma, q)$, and $v_{[i+1,j]}^\uparrow \leftarrow v_{[i+1,j]}^q$. This illustrates the need for the replace update as the interval $[i+1, j]$ may be large.

The update $u_1 \leftarrow a$ *from our running example, yields the* relabel *updates* $u_1^{q_\varepsilon} \leftarrow (a, q_\varepsilon)$, $u_1^{q_a} \leftarrow (a, q_a)$, *and* $u_1^\uparrow \leftarrow (a, q_\varepsilon)$, *and the* replace *update* $u_{[2,3]}^\uparrow \leftarrow u_{[2,3]}^{q_a}$, *as position 3 is the next position after 1 with the reset label a.*

Permutation layer. Assume now that $\mathcal{A}_\ell^T$ is a permutation transducer and that a relabel $v_i \leftarrow \sigma$ is performed on a string v of $S_{\ell-1}$. In order to update S_ℓ we need to do the following: First we need to change the label at position i of all strings in S_ℓ that are derived from v. Second we need to make sure that the states of each such strings are updated starting from position $i+1$. This second operation can be done using the fact that G_ℓ is a group and that we have already computed v^g for all $g \in G_\ell$. Let $g_\sigma = \mathrm{h}_\ell(\sigma)$ be the group element associated to σ. Let g_i be the group element reached by $\mathcal{A}_\ell^T$ at position i when running on v starting in the identity state. The new next state of $\mathcal{A}_\ell^T$ at position $i+1$ is now gg_ig_σ and the remaining states can be computed by resuming the computation from there. Notice now that gg_ig_σ is also the state reached at position $i+1$ by $\mathcal{A}_\ell^T$ on v before the update when starting in state $h = gg_ig_\sigma(g_ig_a)^{-1}$. Hence all the remaining computation is already present in v^h. The updates of S_ℓ can therefore be done using replace operations: $v_{[i+1,n]}^g \leftarrow v^{gg_ig_\sigma(g_ig_a)^{-1}}$ for each $g \in G_\ell$. We note that all these replacements should be done in parallel, effectively permuting the suffixes of the strings. The operations can be serialized using an additional string $v^{\text{temp}} \in S_\ell$ that that allows for cyclic exchanges.

The relabel $u_1 \leftarrow a$ *from our running example has triggered (among others) the* relabel *update* $u_1^\uparrow \leftarrow (a, q_\varepsilon)$ *on* S_1. *This update triggers in particular the* relabel *updates* $u_1^{\uparrow,q_0} \leftarrow (a, q_\varepsilon, q_0)$. *As this shift the count of a from the second position, we obtain the new computation by reading the string* $u^{\uparrow,q_1}$ *from the second position:* $u_{[2,n]}^{\uparrow,q_0} \leftarrow u_{[2,n]}^{\uparrow,q_1}$. *In parallel we perform a similar change on* $u^{\uparrow,q_1}$

The remaining cases are treated similarly and are depicted in Table 1 showing:

Proposition 4.5. *The updates given in Table 1 are correct.*

Notice that that one update of $S_{\ell-1}$ triggers a constant number of updates in S_ℓ. The total number of updates is thus independent from the length of the strings but exponential in the number of layers.

This concludes the description of our data structure for Φ_{left}. Assuming the data structure can implement the operations of Definition 4.4 in logarithmic time, we have seen that the structure can be updated in logarithmic time. Moreover the output string of the last layer associates to each position i of the current input string the state of the automata equivalent to Φ_{left} that is reached when running from the initial position of the string.

4.2 Data structure for Φ_{right}

For the Boolean query Φ_{right}, and its decomposition into transducers, we compute a similar structure. Our goal is here to have at any position information that allows us to determine whether the suffix is in the language defined by Φ_{right}.

For this we mirror what we have done in the previous subsection. We consider the mirror language consisting of all strings w such that when read from right to left the sequence belongs to Φ_{right}.

We then proceed as in the previous subsection except that all transducers given by Krohn-Rhodes theorem now read the string from right to left. In particular, when updating reset layers, we have to search the next reset state to the left using $\text{searchlabel}^{\leftarrow}$ instead of $\text{searchlabel}^{\rightarrow}$.

4.3 Join Layer

From Sections 4.1 and 4.2 we obtain two data structures S_{left} and S_{right} that can be updated in logarithmic time. Here, S_{left} and S_{right} refer to the topmost layer of each structure. We add in this section on top of them a new layer $S_{\bowtie}$ that combines both of them.

If v and w are two strings of length n we define $v \bowtie w$ as the string of length n over the product alphabet where each position contains the labels of v and w at this position.

We define $S_{\bowtie} = \{v \bowtie w \mid v \in S_{\text{left}}, w \in S_{\text{right}}\}$. We note that $S_{\bowtie}$ can be computed in time linear in n. We now describe how we react to an update in S_{left} or S_{right}. By symmetry we only consider the case where the update occurs in S_{left}.

$v_i \leftarrow \sigma$ $\qquad$ $(v \bowtie w)_i \leftarrow (\sigma, w_i)$

$v_{[i,j]} \leftarrow w_{[i,j]}$ $\qquad$ $(v \bowtie w)_{[i,j]} \leftarrow (w \bowtie u)_{[i,j]}$

It is straightforward to verify that the updates are correct wrt. the definition of $S_{\bowtie}$.

4.4 Putting Everything Together

It is now straightforward to prove Theorem 4.1.

Proof. Let $\Psi(x)$ be a MSO unary query. By Lemma 4.2 we can assume without loss of generality that $\Psi(x)$ consists in a pair $(\Phi_{\text{left}}, \Phi_{\text{right}})$ where Φ_{left} and Φ_{right} are regular languages and $\Psi(x)$ returns all the positions i of a word such that the prefix is in Φ_{left} and the suffix is in Φ_{right}.

Given a string v we build from it the data structure of Sections 4.1 to 4.3. It follows from these sections that this structure can be built in time linear in the size of v and can be updated in time logarithmic in the size of v. It remains to show that it allows for constant delay enumeration.

For this, by construction of the structure, each position i of the output string of the data structure contains the state of the automata for Φ_{left} when executed from the beginning of the string and the state of mirror of Φ_{right} when executed from the end of the string. Let Σ_X be the set of all labels that indicate that the position should be in the output of $\Phi(x)$. Using $\text{enumerate}(v', \Sigma_X)$ we can enumerate all positions satisfying $\Phi(x)$, where v' is the output string of the data structure. □

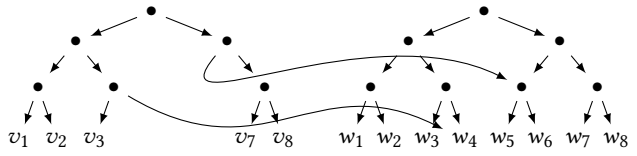


Figure 3: Datastructure representing two strings v and w after the operation $v_{[4,6]} \leftarrow w_{[4,6]}$.

5 STRING MANIPULATION DATA STRUCTURE

In this section we describe a data structure that can perform the operations used in the high-level algorithm we presented in the previous section. We split this description into two parts: First, we specify a basic data structure that supports all operations except enumeration.[4] Afterwards, we describe an advanced data structure that is an extension of the basic data structure and additionally supports constant delay enumeration.

5.1 Basic Data Structure

In order to simplify the presentation we assume that the length of each string is $n = 2^k$ for some natural number k. We represent a string v by a full binary tree t_v of height k. The string is represented at the leaves. Each leaf node x has

- an alphabet label $\sigma(x)$;
- an integer $\mathrm{pos}(x)$ from $[1, n]$ identifying its position in the string.

Each inner node x has

- a label $\mathrm{lab}(x) \in 2^\Sigma$ denoting the set of all labels used by the leaves below x,
- two pointers $\swarrow(x)$ and $\searrow(x)$ for the left and right child, respectively,
- the interval $[i, j]$ of the positions of the leaves below x.

A set S of strings is then represented by an acyclic graph containing a tree t_v for each string $v \in S$. We note that nodes of the graph can be part of several strings. However, if a leaf node is part of two strings t_v and t_w, then it will always be in the same position $\mathrm{pos}(x)$ in v and w and $v_{\mathrm{pos}(x)} = w_{\mathrm{pos}(x)} = \sigma(x)$. If some inner node x is part of two strings v and w, then it represents the same substring in both strings.

In Figure 3, we depict the data structure S for two string v and w that share some leaf nodes and one inner node. This is exactly the data structure, our algorithm would produce after the operation $v_{[4,6]} \leftarrow w_{[4,6]}$ is applied to two strings that did not share any nodes before.

Note that given a node x we can easily compute the path from the root to x. Moreover it is possible to test whether x is part of some tree t_v in logarithmic time looking at its associated interval $[i, j]$ and then following the corresponding path starting from the root of t_v.

[4] Actually, the basic data structure allows for enumeration, but with logarithmic delay, while we need constant delay.

Given all strings in S, the data structure is initialized in linear time by computing a separate tree t_v for each string v.

We now explain how the structure evolves after each update operation and how each operation can be performed within the desired time constraints.

In order to prove formal correctness, we make sure the data structure satisfies the following invariants that should hold for each string $v \in S$ before and after each operation. In the following t_v is the binary tree representing v and $x_1, \ldots, x_n$ are the leaves of t_v from left to right.

(inv1) $v = \sigma(x_1) \ldots \sigma(x_n)$;

(inv2) for each inner node x of t_v, $\mathrm{lab}(x)$ contains exactly those symbols of Σ that are used by the leaves below x;

We make sure that the initialization enforces the invariants (inv1) and (inv2). We now shortly describe how we can handle the "read" operations, before we describe how to handle the update operations.

5.2 Read Operations

The operation $\mathrm{searchlabel}^{\rightarrow}(v, i, \Sigma')$ is implemented as follows: Compute the path from the root of t_v to x_i, the leaf representing v_i. If $\sigma(x_i) \in \Sigma'$, return x_i. Otherwise go up from x_i until visiting a node x, where x_i is in the left subtree and the right subtree contains some $\sigma' \in \Sigma'$, i.e. $\Sigma' \cap \mathrm{lab}(\searrow(x)) \neq \emptyset$. Find the leftmost leaf y in the subtree rooted at $\searrow(x)$ such that $\sigma(y) \in \Sigma'$ using the labels of the inner nodes.

The operation $\mathrm{searchlabel}^{\leftarrow}(v, i, \Sigma')$ is fully symmetric. The operation $\mathrm{enumerate}(v, \Sigma')$ can be implemented with logarithmic delay using $\mathrm{searchlabel}^{\rightarrow}$. A constant delay implementation is described below in Section 5.4.

5.3 Updates

The operations changing the data structure will all use the following three steps. Only the details of these steps (especially the second step) are different.

1. Ensure that inner nodes whose child pointers or labels change are only used by one tree.
2. Do the actual update by changing the child pointers of some inner nodes and possibly adding a new leaf node.
3. Update the labels of inner nodes[5].

The first step ensures that for all unchanged strings, their tree structures will not change. Therefore the invariants (inv1) and (inv2) are maintained for them.

For this first step we introduce an operation $\mathrm{isolate}(v, x)$ that takes as argument a string v and a node x of t_v. For each y in the path from x to the root of t_v, if y also belongs to t_w for $w \neq v$, then it creates a fresh copy of y and changes the pointers such that t_v uses this new copy (all other trees are unchanged.) For a set X of nodes, $\mathrm{isolate}(v, X)$ performs the operation $\mathrm{isolate}(v, x)$ for each $x \in X$.

The second step reestablishes the invariant (inv1) for changed strings. The implementation details differ for the different update operations.

The third step reestablishes (inv2) for changed strings. It essentially consist in label changes.

[5] We never change labels of leaf nodes.

We now describe the individual update operations and argue that each operation maintains all invariants.

• We start with the relabel operation $v_i \leftarrow \sigma$.

Let x_i be the leaf representing v_i in t_v. We do the following operations:

1. isolate$(v, \text{parent}(x_i))$
2. Change the child pointer in parent(x_i) to point to a new node y_i labeled σ.
3. Update the set of used symbols in the ancestors of parent(x_i).

It is easy to see that invariants (inv1) and (inv2) are maintained. The runtime is logarithmic, as each step can be implemented in logarithmic time.

• We continue with the replace operation $v_{[i,j]} \leftarrow w_{[i,j]}$.

Let $X^v_{[i,j]}$ be the set of all inner nodes x of t_v such that the interval associated to x is included in $[i,j]$ and is maximal with respect to this property: the interval associated to the parent of x is not included in $[i,j]$. It is straightforward to check that all nodes in $X^v_{[i,j]}$ are either on the path from the root to v_i or on the path from the root to v_j. Hence $X^v_{[i,j]}$ contains at most logarithmically many nodes of t_v ($X^v_{[i,j]}$ contains at most two nodes per depth level) and can be computed in logarithmic time.

Let $\mathrm{f} : X^v_{[i,j]} \to X^w_{[i,j]}$ be the bijection between $X^v_{[i,j]}$ and $X^w_{[i,j]}$ that connects nodes whose subtrees represent the same interval.

The update consists of the following steps:

1. isolate$(v, X^v_{[i,j]})$
2. for each node $x \in X^v_{[i,j]}$, replace the pointer to x in parent(x) by a pointer to $\mathrm{f}(x)$.
3. Update the set of used symbols in the ancestors of the nodes in $X^v_{[i,j]}$.

All the operations can be performed in logarithmic time. Once we have computed the paths (and corresponding paths in all other strings of S to determine which nodes need to be isolated in Step 1), all required updates to inner nodes of t_v can be done in constant time per node. Note that all affected nodes are in $X^v_{[i,j]}$ or ancestors of those nodes, thus all affected nodes are on the paths from the root to v_i and to v_j.

It is also easy to check that the operations maintain invariants (inv1) and (inv2).

This finishes the description of the basic data structure.

5.4 Advanced Data Structure

The basic data structure is all we need for the lower layers of our data structure. However for enumeration of $S_{\bowtie}$, we need one more operation: enumerate. We note that the list of labels that should be enumerated is fixed and known in advance. Therefore, we provide an implementation that enumerates exactly those positions of a string v, that are labeled by some symbol from a fixed set Σ_X.

We write $x \to_v y$ to denote that the node y should be enumerated after node x when enumerating v, i.e., $x, y \in t_v$, $\sigma(x), \sigma(y) \in \Sigma_X$, $\text{pos}(x) < \text{pos}(y)$, and there is no $z \in t_v$ with $\sigma(z) \in \Sigma_X$ and $\text{pos}(x) < \text{pos}(z) < \text{pos}(y)$.

The basic idea of our implementation is that for each pair of leaf nodes x, y with $x \to_v y$ for some string v, we add a pointer from x to y. Using these pointers, the enumerate operation can be easily implemented.

The difficulty is to maintain those pointers in logarithmic time. There are essentially two problems. First, a leaf node can be part of several trees and therefore requires several pointers, we then need a way to choose which one we need to follow. Second, a replace operation $v_{[i,j]} \leftarrow w_{[i,j]}$ can insert linearly many leaves from a tree t_w into a tree t_v. It is not possible to update the pointers of all these leaves during an update. Therefore, somehow, the existing pointers of t_w have to be used when enumerating the string v in the interval $[i,j]$. However, when leaving this interval, the enumerating algorithm should not follow pointers from t_w any more, but has to continue in t_v.

Our idea to solve this problem is to give priorities to pointers. For instance after a $v_{[i,j]} \leftarrow w_{[i,j]}$ operation, when enumerating v, the algorithm should by default follow the pointers of v and only use pointers of w when those for v are not available, giving priority to the v-pointers. Hence during the replace operation $v_{[i,j]} \leftarrow w_{[i,j]}$, we just add a v-pointer from the last Σ_X-node of v before position i to the first Σ_X-node of w in the interval $[i,j]$, and another v-pointer to the last Σ_X-node of w in the interval $[i,j]$ pointing to the first Σ_X-node of v after position j. This way, v can be correctly enumerated after the replace update.

However, we still need to refine our solution. Consider the sequence of updates $v_{[i,j]} \leftarrow w_{[i,j]}$, $v_{[i,j]} \leftarrow u_{[i,j]}$, $v_{[i,k]} \leftarrow w_{[i,k]}$ with $k > j$. After the second replace, there is a dangling v-pointer left at the node x in position j of w that is not currently used by v. This pointer will confuse the enumeration algorithm after x is reinserted into v by the last replace operation. Unfortunately, we cannot simply remove all dangling pointers after a replace, because there can be many of them. Thus we need a way of invalidating old pointers, which leads us to the following definition, where each pointer carries some information about the pointers that should be followed during the further enumeration.

We associate to each leaf node x a set of pointers pointers(x). We denote by source(p) and target(p), the source and target of the pointer p. In particular if $p \in \text{pointers}(x)$ then its source is x. We associate to each pointer p a label which consists of a set of pointers, denoted label(p). The idea of the label is as follows. By default, label(p) contains exactly one pointer, which is the next pointer to use after outputting the target of p. After a replace operation $v_{[i,j]} \leftarrow w_{[i,j]}$, label$(p)$ may contain two pointers p_1, p_2, where p_1 is the next pointer to follow (pointing to some node of w), and p_2 is some pointer pointing back to some node in v at the end of interval $[i,j]$. Once the enumeration reaches source(p_2) it should follows p_2 and no other pointers with that source.

We add dummy nodes before and after each string that are labeled with some label from Σ_X and are never modified. We use these dummy nodes to avoid all case distinction regarding whether a node is the first or the last one in a string carrying some label from Σ_X. The first node of each string v will always contain a pointer to the first real Σ_X-position of v. The last real Σ_X-position of v always points to the dummy node at the end.

We use the notation $p = x \xrightarrow{\{p_1,\ldots,p_k\}} y$ to denote that pointer p points from node x to node y and has label$(p) = \{p_1, \ldots, p_k\}$.

Algorithm 2 Enumerate String

The internal state consists of
- x ▷ the last visited node
- $P = \{p_1, \ldots, p_k\}$ ▷ a set of pointers

1: **function** STARTENUMERATE(v)
2: $x \leftarrow x_v$
3: $P \leftarrow \text{pointers}(x_v)$
4: **function** NEXT
5: $p \leftarrow p'$ s.t. $p' \in P$ and $\text{source}(p') = x$
6: $x \leftarrow \text{target}(p)$
7: $P \leftarrow P \setminus \{p\}$
8: $P \leftarrow P \cup \{q \in \text{label}(p) \mid \forall r \in P.\ q < r\}$
9: output(pos(x))

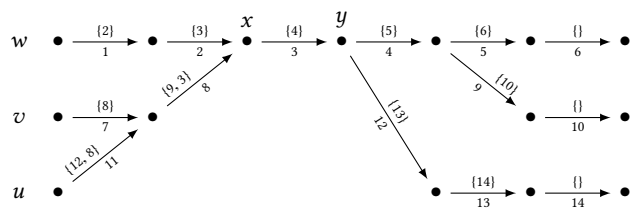


Figure 4: Example showing the use of pointers in the advanced data structure

The label is omitted if it is not relevant. We write $p < r$ to denote $\text{pos}(\text{source}(p)) < \text{pos}(\text{source}(r))$.

The advanced data structure is initialized just as the basic structure. Afterwards, a pointer $p_x = x \xrightarrow{\{p_y\}} y$ is added for each pair of leaf nodes x, y that satisfy $x \rightarrow_v y$ for some v.

Before explaining how we maintain those pointers, let's have a look at the enumeration algorithm, Algorithm 2, to see the indented meaning of the labels. During the enumeration of v, while visiting the node x, the algorithm maintain a set P, called $P(v, x)$ in the sequel, that contains a set of pointers, all with different sources, that we must follow later during the enumeration. In particular $P(x, v)$ always contains a pointer p with source x that provides the next element to output. When switching from x to y the program removes p from $P(v, x)$ (line 7) and adds all $q \in \text{label}(p)$ to P, if the source of q has a smaller position than the positions of all sources of nodes already in $P(v, x)$, enforcing a stack discipline in $P(v, x)$ (line 8). Later, we show that each $P(v, x)$ and therefore the set P in Algorithm 2 has constant size. We let $p(v, x)$ be the maximal entry of $P(v, x)$, i.e., the pointer with the rightmost source node. Intuitively, $p(v, x)$ is the pointer that ultimately returns to the string v, even after many replace updates. In the extreme case, it just points to the dummy node at the end of v.

EXAMPLE 5.1. *In Figure 4, we depicted the relevant pointers of three strings u, v, and w after the operations $v_{[3,5]} \leftarrow w_{[3,5]}$ and $u_{[2,4]} \leftarrow v_{[2,4]}$. Each pointer has a pointer identifier below and the set* label *above the arrow For the node marked x in the figure, we obtain $P(x, w) = \{3\}$, $P(x, v) = \{3, 9\}$, and $P(x, u) = \{3, 12\}$.*

It becomes clear from the example, why we filter the pointers in Line 8 of Algorithm 2. Otherwise, $P(x, u)$ would contain the pointer 9, a pointer never reached by the enumeration, as the enumeration follows pointer 12, which occurs before pointer 9.

Algorithm 3 Clean Pointers

1: **function** CLEAN(x)
2: $V \leftarrow \{v \mid x \in t_v\}$
3: **for** $v \in V$ **do**
4: $y \leftarrow \text{searchlabel}^{\rightarrow}(v, \text{pos}(x) + 1, \Sigma_X)$
5: $p_v \leftarrow x \xrightarrow{P(y,v)} y$
6: $q \leftarrow q'$ s.t. $\max(\text{label}(q')) = p(x, v)$
7: in label(q) replace $p(x, v)$ by p_v
8: $\text{pointers}(x) \leftarrow \{p_v \mid v \in V\}$

The correctness of the enumeration follows from and depends on the following invariant:

(inv3) For every two nodes x, y with $x \rightarrow_v y$ for some v there is a pointer $x \rightarrow y$ in $P(x, v)$.

Before we explain the update algorithm, we introduce an important helper function. Algorithm 3 takes a leaf node x and changes the pointers in such a way, that each string v with $x \in t_v$ uses a different pointer towards the next node. This way, subsequent changes to some pointer of x will affect only one string, allowing updates without interfering with other strings. Furthermore, Algorithm 3 ensures that the number of pointers leaving x is bounded by $|S|$.

Algorithm 3 relies on the following invariant:

(inv4) For each x with $\sigma(x) \in \Sigma_X$ and each two strings v, w with $x \in t_v$ and $x \in t_w$ it holds that $p(v, x) \neq p(w, x)$.

Algorithm 3 replaces $p(v, x)$ in the pointer that introduced $p(v, x)$ into $P(v, x)$ by p_v, which is the correct pointer to follow when at x. This replacement ensures that $P(x, v)$ contains only p_v. Algorithm 3 sets $\text{label}(p_v)$ such that $P(v, y)$ is not changed and the enumeration can safely resume from y.

Note that it is not obvious at this point how to compute $P(v, x)$ and $\max(\text{label}(q)) = p(x, v)$ in constant time. We will see how this can be achieved.

EXAMPLE 5.2. *To understand Algorithm 3, we again look at Figure 4 and study the changes introduced by the call* CLEAN(x). *The algorithm introduces a new pointer $p_v = x \xrightarrow{\{9,4\}} y$ for v and replaces the* 9 *in* label(8) *by p_v. Similarly, it introduces new pointers p_u and p_w. The effect for the enumeration of v is, that p_v is used instead of* 3. *The set $P(v, y)$ is unchanged as the* 9 *is reintroduced by pointer p_v. An update to v that requires the pointer leaving x to be changed to a different node can now be done without interfering with the strings u and w.*

We now present the update algorithms for relabel and replace updates. We use the notation $p \leftarrow q$ to denote that the target node and label set of pointer p is changed to those of pointer q.

Algorithm 4 starts just like in the basic data structure. Afterwards it adjusts the pointers as needed. In both cases (relabel and replace), it first searches the position x preceding the update and uses Algorithm 3 to ensure that each pointer is only used by one string. If the updated part of the string (interval $[i, j]$ with $j = i$ in

Algorithm 4 Update Advanced Data structure

1: **function** RELABEL(v, i, σ)
2: update t_v as in basic data structure
3: $x \leftarrow \text{searchlabel}^{\leftarrow}(v, i-1, \Sigma_X)$
4: $x' \leftarrow \text{searchlabel}^{\rightarrow}(v, i+1, \Sigma_X)$
5: CLEAN(x)
6: **if** $\sigma \in \Sigma_X$ **then**
7: $y \leftarrow$ leaf of v at position i
8: $\text{pointers}(y) \leftarrow \{y \xrightarrow{P(v,x')} x'\}$
9: $p(v, x) \leftarrow x \xrightarrow{\text{pointers}(y)} y$
10: **function** REPLACE(v, w, i, j)
11: update t_v as in basic data structure
12: $x \leftarrow \text{searchlabel}^{\leftarrow}(v, i-1, \Sigma_X)$
13: $x' \leftarrow \text{searchlabel}^{\rightarrow}(v, j+1, \Sigma_X)$
14: CLEAN(x)
15: **if** $w_{[i,j]} \cap \Sigma_X \neq \emptyset$ **then**
16: $y \leftarrow \text{searchlabel}^{\rightarrow}(w, i, \Sigma_X)$
17: $y' \leftarrow \text{searchlabel}^{\leftarrow}(w, j, \Sigma_X)$
18: $q \leftarrow y' \xrightarrow{P(v,x')} x'$
19: $p(v, x) \leftarrow x \xrightarrow{\{p \in P(w,y) \mid p < q\} \cup \{q\}} y$
20: $\text{pointers}(y') \leftarrow \text{pointers}(y') \cup \{q\}$
21: CLEAN(y')

the case of relabel) does not contain any label from Σ_X, Algorithm 3 already adjusts the pointer leaving x to the next position x' after $[i, j]$ that has some label from Σ_X. Otherwise, it changes the pointer to point to the first position in $[i, j]$ with a matching label and adds an additional pointer pointing to x' from the last position in $[i, j]$ with a matching label. Adding the pointer to x' is trivial in the case of relabel, as it leaves from a new node only used by v. The call CLEAN(y') ensures that the number of pointers leaving y' is bounded by $|S|$.

While it is easy to see that for each node x, $|\text{pointers}(x)|$ is bounded by $|S|$, this is not the case for the sets $P(x, v)$. Unfortunately there exist some bad update sequences that allow $P(x, v)$ to grow to linear size. To not add even more complexity to the data structure, our solution here is to show that such bad update sequences do not appear for $S_{\bowtie}$, as used in Section 4.

LEMMA 5.1. *In $S_{\bowtie}$, for each v and and x, $|P(v, x)|$ is bounded by $|S_{\bowtie}|$.*

PROOF SKETCH. We call a pointer p a v-pointer, if it was added on the behalf of v: as pointer p_v in Algorithm 3, or in Algorithm 4, during an update of string v.

It can be shown that $P(v, x)$ can only get two v-pointers, if there is a replace operation $v_{[i,j]} \leftarrow w_{[i,j]}$, after some node of v was already introduced into w by a sequence of replace operations, as in this case $P(w, y)$ could already contain a v-pointer, while the v-pointer q is added in Line 19 of Algorithm 4.

From the definition of the high-level algorithm in Section 4, we can conclude that for all sequences of replace operations that copy some node from v to w that are followed by a replace that copies some nodes from w to v it holds that j is always equal to n. In this case, $P(w, y)$ cannot contain a v-pointer, as this pointer would point to the dummy node at the end of v, which is never part of the string w. □

A detailed proof is given in Appendix A. It remains to show that the invariants (inv3) and (inv4) always hold and the algorithms run in the given time constraints. We start with the invariants.

PROPOSITION 5.2. *Invariants (inv3) and (inv4) are satisfied after initialization of the data structure and the Algorithms 2, 3, and 4 do not alter the invariants.*

PROOF SKETCH. It is easy to verify that the invariants hold after initialization, as each node is only used by one string. Furthermore, Algorithm 2 does not modify the data structure and thus also does not alter the invariants.

Algorithms 3 and 4 satisfy, that for each string v and node x, where $p(v, x)$ changes, it is changed to a new pointer, only used by v. Therefore, (inv4) is preserved.

It remains to show that (inv3) is preserved, which can be done by a careful analysis on the changes of the sets $P(v, x)$ implied by the algorithms. □

PROPOSITION 5.3. *Initialization can be done in linear time, Algorithm 2 can be implemented with constant delay and initialization, and Algorithms 3 and 4 can be implemented to use only logarithmic time.*

PROOF. It is easy to see that the initialization works in linear time and the enumeration works with constant delay. We remind that P is bounded by a constant.

Towards the runtime of Algorithms 3 and 4, we observe that the crucial parts are to compute q in Line 6 of Algorithm 3 and the computation of the sets $P(x, v)$.

To compute q, we just store a reference to the pointer q together with the pointer $p(x, v)$. This reference is updated whenever a pointer is inserted to our data structure, i.e., in Line 8 of Algorithm 3 and Lines 8, 9, 19, and 20 of Algorithm 4.

Towards a logarithmic time lookup of $P(v, x)$ we add an interval tree for each string v that contains for each pointer p used in t_v an entry $([i, j], p)$ describing the interval $[i, j]$, such that p is contained in each $P(v, x)$ with $\text{pos}(x) \in [i, j]$. To compute $P(v, x)$, one has to lookup all entries $([i, j], p)$ with $\text{pos}(x) \in [i, j]$. □

Altogether, we have shown that the data structure can be implemented with constant delay enumeration, logarithmic update time, and logarithmic time for searchlabel queries.

6 LABEL INSERTIONS AND REMOVALS

We have described how we can handle relabeling updates. This can be generalized to other updates such as node insertions and removals. W.l.o.g., we can assume that all our regular languages have a neutral letter e that is not used in the original alphabet. We then implement the insertion of a node by first inserting a node of label e and then relabeling that node. We do likewise for removals.

It remains to describe how an insertion of a node of label e can be implemented in our data structure. We only sketch the main ideas in this long abstract.

Recall from Section 5 that the data structure has one tree t_v for each string $v \in S$. For simplicity we assumed that the trees were full binary trees. However, for the correctness of the implementation (and the runtime requirements), we only need the following properties:

- the trees are binary trees;
- the depth is bounded logarithmically in the length of the strings;
- every tree has the same structure, i.e., for each pair $v, w \in S$ it holds that for each node in t_v there exists a node in t_w that represents the same string interval.

When inserting or deleting a node we need to make sure that the depth remains logarithmic, measured in the size of the new tree. Using the classical rebalancing operations of [1], we can ensure that at any time, the depth is bounded above by $\log_\varphi(n+2)$, where φ is the golden ratio. In order to preserve the third item, we rebalance all the trees of S in parallel.

We also need to change the information contained in all nodes: the label and the associated interval. As it is not possible to update the interval associated to each node x, we replace that information with the number of leaf nodes below x. This information can easily be updated in logarithmic time during an insertion or remove operation and is enough for our needs. In particular, using the number of leaves information, one can compute the position of each node in logarithmic time.

Consider now the label associated to each node x. When inserting a neutral letter at position i in the input string, all its images in all layers need to be updated. Fortunately, because the letter is neutral, this only modifies the position i in all layers[6] Hence, for all strings u in S we need to modify the label of the node at position i and the label of all its ancestors in t_u. This can be done in logarithmic time.

To be able to enumerate node positions, we also store together with each pointer $x \xrightarrow{m} y$ its length, i.e., the number of skipped nodes. Given a node x together with its position i, it is trivial to compute the position of y using this length information.

To update the length information after an insertion or removal at position i, the algorithm computes $x = \mathsf{searchlabel}^{\leftarrow}(v, i, \Sigma_X)$ for each v and updates the pointers leaving v as needed.

Applying the modifications described in this section to the data structure of Section 5 and extending the algorithm described in Section 4 to allow insertion and removal of neutral letters yields the following theorem.

Theorem 6.1. *On strings, unary MSO queries can be enumerated using linear preprocessing time, constant delay and logarithmic time for updates that relabel, insert, or remove a single position.*

7 CONCLUSION

We have exhibited a data structure that can be computed in linear time, can be updated in logarithmic time and allow for constant delay enumeration of MSO queries over words. By interpretation it immediately extends to structures of bounded path-width.

[6] A label indicating that the input string (in the lowest layer) has a neutral label will keep each transducer in its current state. The neighbouring positions depend only on the labels of the lower layers (at the same position) and the state (which does not change). Thus they are unmodified.

The next step would be to extend it to trees. For MSO queries over trees the best known results are those of [18] with a data structure that can be updated in logarithmic square time and allow for logarithmic square delay enumeration. Current work [20] enhances the techniques of [18] to allow for logarithmic updates and logarithmic delay. Another recent paper [2] provides a data structure that allows for constant delay enumeration. The downside is that it is only known how to update this structure for relabel updates, i.e., insertion or deletion of a leaf requires to recompute the data structure from scratch.

In order to extend our technique to trees, yielding a data structure that allows for constant delay enumeration and logarithmic updates (relabel, insertion/deletion of leaves), we would need an equivalent to Krohn-Rhodes Theorem, decomposing any regular tree language into a cascade of basic regular languages. Unfortunately Krohn-Rhodes theory is not ready yet over trees. Preliminary results exists. In particular it seems that the cascade decomposition obtained in [9] for CTL queries, with a building block similar to reset automata, combined with our method, would yield a structure computable in linear time, allowing for logarithmic time updates and constant delay enumeration of CTL queries over trees.

Finally we note that, unlike previous index structures used for constant delay enumeration of MSO queries [16], our index structure cannot answer in constant time whether a given tuple is a solution or not. It seems plausible that there exists a trade-off between this feature and efficient update.

REFERENCES

[1] Georgy Adelson-Velsky and Evgenii Landis. 1962. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR* 146, 2 (1962), 263–266.

[2] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. 2018. Enumeration on Trees under Relabelings. In *Intl. Conf. on Database Theory (ICDT)*. 5:1–5:18.

[3] Mustapha Arfi. 1987. Polynomial Operations on Rational Languages. In *Symp. on Theoretical Aspects of Computer Science (STACS)*. 198–206.

[4] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *Proc. 20th International Workshop/15th Annual Conference of the EACSL (CSL'06)*. 167–181.

[5] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental validation of XML documents. *ACM Trans. Database Syst.* 29, 4 (2004), 710–751.

[6] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *Symp. on Principles of Database Systems (PODS)*.

[7] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering FO+MOD Queries Under Updates on Bounded Degree Databases. In *Intl. Conf. on Database Theory (ICDT)*.

[8] Mikołaj Bojańczyk and Pawel Parys. 2011. XPath evaluation in linear time. *J. ACM* 58, 4 (2011), 17.

[9] Mikołaj Bojańczyk, Howard Straubing, and Igor Walukiewicz. 2012. Wreath Products of Forest Algebras, with Applications to Tree Logics. *Logical Methods in Computer Science* 8, 3 (2012).

[10] Thomas Colcombet. 2007. A Combinatorial Theorem for Trees. In *Intl Conf. on Automata and Logic Programming (ICALP)*. 901–912.

[11] Arnaud Durand and Etienne Grandjean. 2007. First-order queries on structures of bounded degree are computable with constant delay. *ACM Transactions on Computational Logic* 8, 4 (2007).

[12] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. 2014. Enumerating answers to first-order queries over databases of low degree. In *Principles of Database Systems (PODS'14)*.

[13] Wojciech Kazana. 2013. *Query evaluation with constant delay.* Ph.D. Dissertation. ENS Cachan.

[14] Wojciech Kazana and Luc Segoufin. 2011. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science* 7, 2 (2011).

[15] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proc. 32nd ACM Symposium on Principles of Database Systems (PODS'13)*. 297–308.

[16] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic* 14, 4 (2013).

[17] Kenneth Krohn and John Rhodes. 1965. Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines. *Trans. Amer. Math. Soc.* 116 (1965), 450–464.
[18] Katja Losemann and Wim Martens. 2014. MSO queries on trees: enumerating answers under updates. In *Computer Science Logic and Logic in Computer Science (CSL-LICS'14)*.
[19] Oded Maler. 2010. On the Krohn-Rhodes Cascaded Decomposition Theorem. In *Time for Verification, Essays in Memory of Amir Pnueli*. 260–278.
[20] Matthias Niewerth. 2018. MSO Queries on Trees: Enumerating Answers under Updates Using Forest Algebras. submitted.
[21] Wolfgang Thomas. 1982. Classifying Regular Events in Symbolic Logic. 25, 3 (1982), 360–376.

A PROOF OF LEMMA 5.1

In this appendix, we provide a detailed proof for the following Lemma:

Lemma 5.1. *In $S_{\triangleright\triangleleft}$, for each v and and x, $|P(v,x)|$ is bounded by $|S_{\triangleright\triangleleft}|$.*

Before we proof the lemma, we need some additional tools. First, we provide an inductive definition of some preorder $\lesssim_{S_{\triangleright\triangleleft}}$ over the strings in $S_{\triangleright\triangleleft}$. We write

- $v \sim w$ to denote $v \lesssim w$ and $w \lesssim v$; and
- $v \lesssim\!\!\!\!\!\!\not\;\; w$ to denote $v \lesssim w$ and not $w \lesssim v$.

The preorders for the layers of the left and the right structure are inductively defined as follows:

- $\lesssim_{S_0}$ is the trivial preorder over the singleton S_0.
- If $S_\ell \in \mathcal{R}$, then $\lesssim_{S_\ell}$ is the smallest preorder that satisfies
  - $v^\uparrow \lesssim\!\!\!\!\!\!\not\;\;_{S_\ell} v^q$ for all $q \in Q_\ell$ and all $v \in S_{\ell-1}$; and
  - $v^q \lesssim_{S_\ell} w^q$ and $v^\uparrow \lesssim_{S_\ell} w^\uparrow$ for all $q \in Q_\ell$ and all $v, w \in S_{\ell-1}$ with $v \lesssim_{S_{\ell-1}} w$.
- If $S_\ell \in \mathcal{P}$, then $\lesssim_{S_\ell}$ is the smallest preorder that satisfies
  - $v^{g_1} \sim v^{g_2}$ for all $g_1, g_2 \in G_\ell$ and all $v \in S_{\ell-1}$; and
  - $v^g \lesssim_{S_{\ell-1}} w^g$ for all $g \in G_\ell$ and all $v, w \in S_{\ell-1}$ with $v \lesssim_{S_{\ell-1}} w$.

The preorder for the join layer is defined by

$$\lesssim_{S_{\triangleright\triangleleft}} = \{((v \bowtie w), (v' \bowtie w')) \mid v \lesssim_{S_{\text{left}}} v' \text{ and } w \lesssim_{S_{\text{right}}} w'\},$$

where S_{left} and S_{right} are the topmost layers of the left and right structure, respectively.

The following lemma states a property of the high-level algorithm that we need to show Lemma 5.1.

Lemma A.1. *For each* replace *update $v_{[i,j]} \leftarrow w_{[i,j]}$ in $S_{\triangleright\triangleleft}$ it holds that*

(a) $v \lesssim_{S_{\triangleright\triangleleft}} w$; and
(b) if $v \sim_{S_{\triangleright\triangleleft}} w$ then $j = n$.

Proof. In the following, the superscript x can stand for any symbol from $\uparrow$, Q_ℓ, and G_ℓ, depending on whether $\mathcal{A}_\ell^T$ is a reset transducer or a permutation transducer.

Using Table 1 and the inductive definition of $\lesssim\!\!\!\!\!\!\not\;\; S$, an easy inductive argument yields that (a) holds. We note that the update $v_{[i,j]} \leftarrow w_{[i,j]}$ in $S_{\ell-1}$ implies that $v \lesssim_{S_{\ell-1}} w$ by induction hypothesis, and thus $v^x \lesssim_{S_\ell} w^x$ by the definition of $\lesssim_{S_\ell}$.

Similarly, it can be verified that (b) holds. If $v^x \sim_{S_\ell} w^x$, then by definition of $\lesssim_{S_\ell}$ we have that $v \sim_{S_{\ell-1}} w$. Using the induction hypothesis, we get $j = n$. □

We say a pointer is a v-pointer, if it is added on behalf of string v in the initialization, in Algorithm 3, or in Algorithm 4 when changing string v.

Now we can prove Lemma 5.1. We will show that the following invariant is always satisfied, which directly yields the statement of Lemma 5.1.

(inv5) For all strings $v, w \in S_{\triangleright\triangleleft}$ and all nodes $x \in t_v$, it holds that
- if $v \lesssim\!\!\!\!\!\!\not\;\;_{S_{\triangleright\triangleleft}} w$ and $v \neq w$, $P(v,x)$ contains at most one w-pointer;
- if not $v \lesssim\!\!\!\!\!\!\not\;\;_{S_{\triangleright\triangleleft}} w$, $P(v,x)$ contains no w-pointer;
- $P(v,x)$ contains at most two v-pointers;
- if $P(v,x)$ contains two v-pointers, then one of them points to the dummy node at the end of v; and
- $p(v,x)$ always is a v-pointer.

It is easy to verify that (inv5) holds after initialization, as each set $P(v,x)$ contains exactly one pointer, which is a v-pointer.

We now have a look at all operations that change pointers and the implied changes to the P-sets. We start with Algorithm 3.

The pointer p_v computed in Line 5 and assigned in Line 8 does not change any set $P(u,z)$, as $\text{label}(p_v)$ is chosen to maintain $P(u,x')$. Furthermore, the only change of Line 7 to the sets $P(x,v)$ is that it replaces one v-pointer by a different v-pointer. Thus the invariant is preserved by Algorithm 3.

We now look at Algorithm 4.

The pointer added in Line 9, ensures that the set $P(v,x')$ does not change. Similarly, the set $\text{label}(q)$ of the pointer q defined in Line 18 is computed such that $P(v,x')$ does not change. We note that these pointers are not used by any other strings.

The pointer change in Line 8 yields, that $P(v,y)$ contains exactly the pointer added in Line 9, which is a v-pointer. The line does not change other P-sets. Altogether, relabel updates preserve the invariant.

It remains to look at the pointers added in Line 19. As, by (inv5), $P(w,y)$ does not contain two u-pointers for any string u, it is enough to show that either

- $P(w,y)$ has no v-pointer; or
- the v-pointer p in $P(w,y)$ satisfies $q < v$, where q is the pointer defined in Line 18.

Here, we use Lemma A.1. If $v \lesssim\!\!\!\!\!\!\not\;\;_{S_{\triangleright\triangleleft}} w$, then $P(w,y)$ has no v-pointer. Otherwise, by Lemma A.1, the replace operation satisfies $j = n$. Therefore, the w-pointer of $P(w,y)$ that points to the last node of w (if $P(w,y)$ has two w-pointers) will not show up in $P(v,y)$. Furthermore, the v-pointer introduced in Line 20, points to the dummy node of v. Altogether (inv5) is preserved.