MSO QUERIES ON TREES: ENUMERATING ANSWERS UNDER UPDATES USING FOREST ALGEBRAS

SARAH KLEEST-MEISSNER [a], JONAS MARASUS [a], AND MATTHIAS NIEWERTH [b]

[a] Humboldt-Universität zu Berlin, Germany

[b] Universität Bayreuth, Germany

Abstract. We describe a framework for maintaining forest algebra representations that are of logarithmic height for unranked trees. Such a representations can be computed in $\mathcal{O}(n)$ time and updated in $\mathcal{O}(\log(n))$ time. The framework is of potential interest for data structures and algorithms for trees whose complexity depend on the depth of the tree (representation). We provide an exemplary application of the framework to the problem of efficiently enumerating answers to MSO-definable queries over trees which are subject to local updates. We exhibit an algorithm that uses an $\mathcal{O}(n)$ preprocessing phase and enumerates answers with $\mathcal{O}(\log(n))$ delay between them. When the tree is updated, the algorithm can avoid repeating expensive preprocessing and restart the enumeration phase within $\mathcal{O}(\log(n))$ time. Our algorithms and complexity results in the paper are presented in terms of node-selecting tree automata representing the MSO queries.

1. Introduction

Efficient query evaluation is one of the most central problems in databases. Given a query Q and a database D, we are asked to compute the set $Q(D)$ of tuples of Q on D. In general, the number of tuples in $Q(D)$ can be extremely large: when Q has arity k and D has size n, then $Q(D)$ can contain up to n^k tuples. Since databases are typically very large, it may be unfeasible to compute $Q(D)$ in its entirety.

A straightforward solution to this problem is top-k query answering, where one is interested in the k most relevant answers according to some metric. Another way to deal with this problem is to produce the answers one by one without repetition. This is known as *query enumeration* (see, e.g., [Bag06, Cou09, DS11, KS13a, KS13b, Seg13]). More precisely, query enumeration aims at producing a small number of answers first and then, on demand, producing further small batches of answers as long as the user desires or until all answers are depleted. Most existing algorithms for query enumeration consist of two phases: the *preprocessing phase*, which lasts until the first answer is produced, and the *enumeration phase* in which next answers are produced without repetition. It is natural to try to optimize two kinds of time intervals in this procedure: the time of the preprocessing phase and the *delay* between answers, which is the time required between two answers in the enumeration

Key words and phrases: MSO, query enumeration, trees, forest algebras.

Preprint submitted to
Logical Methods in Computer Science

phase. Thus, when one can answer $Q(D)$ with preprocessing time p and delay d, one can compute $Q(D)$ in time $p + d \cdot |Q(D)|$, where $|Q(D)|$ is the number of answers.

Much attention has been given to finding algorithms that answer queries with a linear-time preprocessing phase and constant-time delay between answers [Seg13]. The preprocessing phase is usually used to build an index that allows for efficient enumeration. Since databases can be subjected to frequent updates and preprocessing typically costs linear time, it is usually not an option to recompute the index after every update. We want to address this concern and investigate what can be done if one wants to deal with such updates more efficiently than simply re-starting the preprocessing phase.

We cannot expect efficient algorithm for query enumeration in the general case, as even deciding whether a (Boolean) query has an answer is NP-hard. Therefore, we have to restrict the problem. In the literature, there are two general directions towards tractable query enumeration under updates: restricting the class of queries or restricting the structure of the data.

There has been research on enumerating certain classes of conjunctive queries with constant delay and sublinear update time [BKS17, BKS18b]. Similarly, there are algorithms that can enumerate FO+MOD queries with constant delay and constant update time on bounded degree databases [BKS18a].

In this article, we follow the second approach and study the enumeration problem for MSO queries with free node variables over *trees*. Furthermore, the trees can be subjected to local updates. We consider updates that relabel a node or insert/delete a leaf. Our aim is to provide an index structure that can be efficiently updated, when the underlying tree changes. This makes the enumeration phase *insensitive to such updates*: when our algorithm is producing answers with a small delay in the enumeration phase and the underlying data D is updated, we can update the index and re-start enumerating on the new data within the same delay.

The complexity results in this article are presented in terms of the size of the tree; the arity k of the query; and the number of states of a non-deterministic node-selecting finite tree automaton for the query. The connection between run-based node-selecting automata and MSO-queries is well-known, see, e.g. [NPTT05, TW68].

When measuring complexity in terms of query size, we have to keep in mind that MSO queries can be non-elementary smaller than their equivalent non-deterministic node-selecting tree automata. Therefore, our enumeration algorithm is non-elementary in terms of the MSO formula, which cannot be avoided unless P = NP [FG04]. For this reason, MSO is usually not used as a query language in practice; although it is widely regarded as a good yardstick for expressiveness.

Our complexities are exponential in the arity k of the queries. However for practical scenarios, k is usually very small. We note that $k = 2$ suffices for modeling XPath queries, which are central in XML querying.

Although we do not obtain constant-delay algorithms as in previous work on static trees, we can prove that, in the dynamic setting $\mathcal{O}(\log(n))$ delay is possible. This means that, after receiving an update, we do not need to restart the $\mathcal{O}(n)$ preprocessing phase but only require $\mathcal{O}(\log(n))$ time to produce the first answer on the updated tree and continue enumerating from there. We allow updates to arrive at any time: If an update arrives during the enumeration phase, we immediately start the enumeration phase for the new structure.

Update	Delay	Remarks	Reference
$\mathcal{O}(\log^2(n))$	—	only Boolean queries; $\mathcal{O}(\log(n))$ on strings	[BPV04]
$\mathcal{O}(\log^2(n))$	—	Boolean XPath queries	[BGM10]
—	$\mathcal{O}(1)$	updates in $\mathcal{O}(n)$ by recomputation	[Bag06]
—	$\mathcal{O}(1)$	different proof using decomposition forests	[KS13b]
—	$\mathcal{O}(1)$	different proof using circuits	[ABJM17]
—	$\mathcal{O}(1)$	streaming algorithm	[MR22]
$\mathcal{O}(\log^2(n))$	$\mathcal{O}(\log^2(n))$	complexities drop to $\mathcal{O}(\log(n))$ on strings	[LM14]
$\mathcal{O}(\log(n))$	$\mathcal{O}(1)$	only works on strings; huge constants	[NS18]
$\mathcal{O}(\log(n))$	$\mathcal{O}(1)$	only relabel updates; uses circuits	[ABM18]
$\mathcal{O}(\log(n))$	$\mathcal{O}(1)$	uses the techniques developed in this article	[ABMN19]
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	uses forest algebras	this work

Table 1. Data complexity of existing solutions. Preprocessing time is always in $\mathcal{O}(n)$.

In [ABMN19] it has been shown that the constant delay enumeration approach for MSO queries over static trees from [ABJM17, ABM18] can be extended to allow updates, if the trees are represented by balanced forest algebra formulas that are described in this article and where already depicted in a preliminary version [Nie18].

Previous Results on MSO Queries on Trees. We have collected previous results on evaluation and enumeration of MSO queries on strings and trees in Table 1.

For MSO sentences, this problem has been studied by Balmin, Papakonstantinou, and Vianu [BPV04]. Balmin et al. show how one can efficiently maintain satisfaction of a finite tree automaton (and therefore, an MSO property) on a tree t which is subjected to updates. More precisely, when an update transforms t to t', they want to be able to decide very quickly after the update whether t' is accepted by the automaton. Taking n as the size of t, they show that, using a one-time preprocessing phase of time $\mathcal{O}(n)$ to construct an auxiliary data structure, one can always decide within time $\mathcal{O}(\log^2(n))$ after the update whether t' is accepted. The delay between answers is irrelevant in the setting of Balmin et al. since their queries always have a Boolean answer. Björklund et al. [BGM10] show a similar result for XPath queries, which are less expressive than MSO but can be exponentially more succinct than tree automata, which leads to better constants. Losemann and Martens [LM14] extended the work of Balmin et al. to enumeration of k-ary queries under updates with $\mathcal{O}(\log^2(n))$ delay and update time. Our goal is to improve the delay and update time to $\mathcal{O}(\log(n))$.

The enumeration problem of static trees was studied by Bagan [Bag06], who showed that (fixed) monadic second-order (MSO) queries can be evaluated with linear time preprocessing and constant delay over structures of bounded tree-width. Independently, another constant delay algorithm (but with $\mathcal{O}(n\log(n))$ preprocessing time) was obtained by Courcelle et al. [Cou09]. Kazana and Segoufin [KS13b] provided an alternative proof of Bagan's result based on a deterministic factorization forest theorem by Colcombet [Col07], which is itself based on a result of Simon [Sim90]. Such (deterministic) factorization forests provide a good divide-and-conquer strategy for words and trees, but it is unclear how they can be maintained under updates. It seems that they would have to be recomputed entirely after an update which is too expensive for our purposes.

With exception of [Bag06], which presents an algorithm that is cubic in terms of the tree automaton, these papers present complexities in terms of the size of the trees only, that is, they consider the MSO formula to be constant. To the best of our knowledge, the data structures in these approaches cannot be updated efficiently if the underlying tree is updated. An overview of enumeration algorithms with constant delay was given in [Seg13].

Muñoz and Riveros [MR22] present a streaming algorithm for MSO queries on trees that enumerates answers with constant delay.

Heavy Path Decomposition vs. Forest Algebras. A main idea in [BPV04] is a decomposition of trees into heavy paths which allows one to decompose the problem for trees into $\mathcal{O}(\log(n))$ similar problems on words, for which a solution was given by Patnaik and Immerman in [PI97]. This allows to solve the incremental evaluation problem with $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(\log^2(n))$ update time, where one log factor stems from the heavy path decomposition and the other from solving the problem over strings using monoids of finite string automata.

The approach of [BPV04] was later extended to enumeration of MSO queries by Losemann and Martens [LM14]. They tweaked the monoid to contain additional information needed to find the symbols that appear in query results, which allows logarithmic delay and update time. Just as Balmin et al., Losemann and Martens use heavy path decomposition to lift the algorithm from words to trees resulting again in an additional logarithmic factor in the delay and update time.

We adapt the algorithm of Losemann and Martens from monoids to forest algebras in Section 6. This saves us a logarithmic factor compared to their results. We believe that the framework we introduce in Section 3 to compute and maintain forest algebra formulas with logarithmic height can be applied in other areas to lift results from strings to trees.

Tree Decomposition vs. Forest Algebras. In [ABM18], Amarilli et al. use tree decompositions [BH98] to convert arbitrary trees to trees of logarithmic height. Having a tree of logarithmic height is central in their algorithm to allow enumeration of MSO queries over trees under relabeling updates. According to Amarilli et al., the biggest obstacle in generalizing their work to allow structural updates (insertion/deletion of nodes) of the tree was the inability to update the tree decomposition when the input tree changes.

Following up on the preliminary version of this article [Nie18], it has been shown in [ABMN19] that the enumeration algorithm from [ABM18] can be generalized to allow structural updates on the tree using the methods we develop in sections 3 and 4.

Korhonen et al. [KMN+23] present an algorithm that maintains a tree decomposition of a given graph under updates. If G is guaranteed to never exceed treewidth k, they maintain a tree decomposition of treewidth at most $6k+5$ with an amortized update time of $\mathcal{O}_{\|}2^{\sqrt{\log(n)}\log(\log(n))}$, where $\mathcal{O}_k$ hides factors that depend on k. They can also maintain satisfaction of $CMSO_2$-sentences in the same time bounds.

Further Related Work. There are implementations for and experimental results on incremental evaluation of XML documents wrt. DTDs [BML$^+$04] and regular expressions with counters on strings [BMT15]. The query evaluation problem has also been studied from a descriptive complexity point of view, e.g., for conjunctive queries [ZS14] and the reachability query on graphs [DKM$^+$15].

Contributions. We depict a mechanism for representing trees by the means of forest algebra formulas in Section 3. Furthermore, we show how these representations can be maintained after updates that can relabel, insert, or delete individual nodes of the tree. We believe that these techniques are interesting on their own. Building on top of a preliminary version of this article, [ABMN19] already uses the techniques provided here to achieve further improvements in the enumeration of MSO queries on trees.

Section 4 shows how forest algebras are related to tree automata through their transition algebra. This relationship is exactly the same as the relationship between monoids and string automata.

We provide a first application of our framework in Section 5, where we show how to solve the incremental evaluation problem for Boolean MSO formulas on trees by means of forest algebra formulas. This approach allows for logarithmic update time after linear time preprocessing.

Finally, in Section 6, we show how forest algebras can be used to enumerate MSO queries over trees with logarithmic delay and logarithmic update time after a linear preprocessing step. Even though there is an improved algorithm available in [ABMN19], we believe that the presented algorithm is useful. First, it gives a quite simple demonstration how the general techniques presented in Section 3 can be applied, and second it is a much simpler algorithm that might actually be more efficient for medium sized trees than the more complex algorithm presented in [ABMN19].

Earlier Version. An earlier version of this article was presented at the 2018 Symposium on Logic in Computer Science (LICS 2018) [Nie18]. The earlier version does not contain formal proofs. Unfortunately, the unpublished proofs turned out to have a flaw. Therefore we modified the algorithms slightly in order to show the claimed complexity bounds. We believe that also the algorithms as presented in [Nie18] meet the same complexity bounds, but we were not able to prove this.

2. Preliminaries

Trees, Forests, Contexts. In this article, trees and forests are labeled by some finite alphabet Σ, rooted, and ordered. Contexts are forests which contain a single hole, denoted by $\square$, which may not be part of any forest. In detail: A *forest* is a tuple $F = (V, E, \leq, \mathsf{lab})$, where V is a finite set of nodes with $\square \notin V$, $E \subseteq V \times V$ is the set of edges, $\leq \subseteq V \times V$ is the sibling order, and $\mathsf{lab}\colon V \to \Sigma$ is the labeling function.

If $(v, w) \in E$, we say w is a *child* of v and v is the *parent* of w. If (v, w) is within the transitive closure of E, we say w is a *descendant* of v and v is an *ancestor* of w. A node without parents is called a *root* and we denote the set of all roots of F by $\mathsf{roots}(F)$. A node without children is called a *leaf*. The transitive closure of E must be acyclic and no node may have more than one parent. If $v_1 \leq v_2$, we say, v_1 lies *left* of v_2. For every node v, $\leq$

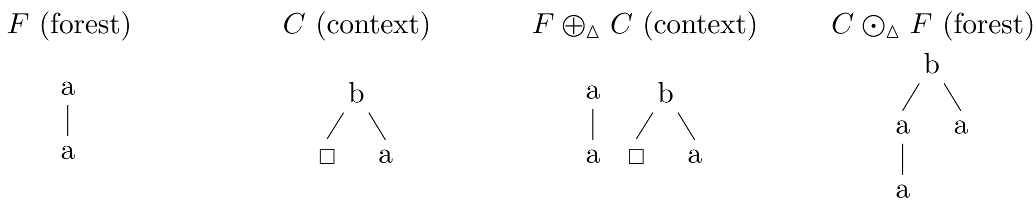


FIGURE 1. Examples for concatenation and context application

must be a linear order on the set of all children of v. Likewise, $\leq$ needs to be a linear order on $\mathsf{roots}(F)$. Other than that, $\leq$ must have no additional elements. A forest with a single root is called a *tree*.

A *context* $C = (V, E, \leq, \mathsf{lab})$ is a forest with the exception that V contains the hole $\square$. The hole must be a leaf and does not carry a label, i.e., lab is a function on $V \setminus \{\square\}$.

We overload notation and use a symbol $a \in \Sigma$ to denote the forest that contains a single node with label a. Similarly we use $a_\square$ with $a \in \Sigma$ to denote the context with an a-labeled root that has the hole as its only child. We use ε and $\square$ to denote the empty forest and the context that only consists of the hole, respectively.

We usually will denote forests by F and contexts by C possibly with subscripts. We use D (with subscripts) to denote that the object can be either a forest or a context.

We will usually assume w.l.o.g. that for different forests and contexts the sets of nodes are disjoint (except $\square$). This can be achieved by renaming the nodes of one of the operands.

We define two operations on contexts and forests: The (horizontal) *concatenation* $\oplus_\vartriangle$ places two forests or a forest and a context next to each other and the (vertical) *context application* $\odot_\vartriangle$ which inserts a forest or context into a context by replacing the hole with the roots of the inserted forest or context. Not all combinations of operands are allowed in order for the result to be well defined:

- It is not possible to concatenate two contexts; and
- the left operand of a context application has to be a context.

We depict examples of the two operations in Figure 1. In detail the operations are defined as follows:

Let $D_1 = (V_1, E_1, \leq_1, \mathsf{lab}_1)$ and $D_2 = (V_2, E_2, \leq_2, \mathsf{lab}_2)$ be two forests or one forest and a context in either order. We compute $D = D_1 \oplus_\vartriangle D_2$ by taking the disjoint componentwise union and specifying that all roots of D_2 have to be right of all roots of D_1. Formally we define $D_1 \oplus_\vartriangle D_2 = (V, E, \leq, \mathsf{lab})$, where

$$\begin{aligned}
V &= V_1 \cup V_2 \\
E &= E_1 \cup E_2 \\
\leq &= \leq_1 \cup \leq_2 \cup \{(v_1, v_2) \mid v_1 \in \mathsf{roots}(D_1), v_2 \in \mathsf{roots}(D_2)\} \\
\mathsf{lab} &= \mathsf{lab}_1 \cup \mathsf{lab}_2
\end{aligned}$$

We note that $D_1 \oplus_\vartriangle D_2$ is a forest if D_1 and D_2 are both forests and it is a context if one of D_1, D_2 is a context. If D_1 and D_2 are both contexts, then $D_1 \oplus_\vartriangle D_2$ is undefined.

Let $C_1 = (V_1, E_1, \leq_1, \mathsf{lab}_1)$ be a context and $D_2 = (V_2, E_2, \leq_2, \mathsf{lab}_2)$ be a context or a forest. We compute $D = C_1 \odot_\vartriangle D_2$ by first removing the hole from C_1, then taking the componentwise disjoint union and afterwards tweaking the set of edges and sibling order of D such that the ordered list of roots of D_2 appears where the hole in C_1 was before. Let

	Axiom	Name
(A1)	$\varepsilon \oplus_{HH} F_1 \;=\; F_1 \;=\; F_1 \oplus_{HH} \varepsilon$	neutral element
(A2)	$\varepsilon \oplus_{HV} C_1 \;=\; C_1 \;=\; C_1 \oplus_{VH} \varepsilon$	
(A3)	$\Box \odot_{VV} C_1 \;=\; C_1 \;=\; C_1 \odot_{VV} \Box$	
(A4)	$\Box \odot_{VH} F_1 \;=\; F_1$	
(A5)	$(F_1 \oplus_{HH} F_2) \oplus_{HH} F_3 \;=\; F_1 \oplus_{HH} (F_2 \oplus_{HH} F_3)$	associativity
(A6)	$(F_1 \oplus_{HH} F_2) \oplus_{HV} C_1 \;=\; F_1 \oplus_{HV} (F_2 \oplus_{HV} C_1)$	
(A7)	$(C_1 \oplus_{VH} F_1) \oplus_{VH} F_2 \;=\; C_1 \oplus_{VH} (F_1 \oplus_{HH} F_2)$	
(A8)	$(F_1 \oplus_{HV} C_1) \oplus_{VH} F_2 \;=\; F_1 \oplus_{HV} (C_1 \oplus_{VH} F_2)$	
(A9)	$(C_1 \odot_{VV} C_2) \odot_{VV} C_3 \;=\; C_1 \odot_{VV} (C_2 \odot_{VV} C_3)$	
(A10)	$(C_1 \odot_{VV} C_2) \odot_{VH} F_1 \;=\; C_1 \odot_{VH} (C_2 \odot_{VH} F_1)$	
(A11)	$(F_1 \oplus_{HV} C_1) \odot_{VH} F_2 \;=\; F_1 \oplus_{HH} (C_1 \odot_{VH} F_2)$	interaction
(A12)	$(F_1 \oplus_{HV} C_1) \odot_{VV} C_2 \;=\; F_1 \oplus_{HV} (C_1 \odot_{VV} C_2)$	
(A13)	$(C_1 \oplus_{VH} F_1) \odot_{VH} F_2 \;=\; (C_1 \odot_{VH} F_2) \oplus_{HH} F_1$	
(A14)	$(C_1 \oplus_{VH} F_1) \odot_{VV} C_2 \;=\; (C_1 \odot_{VV} C_2) \oplus_{VH} F_1$	

TABLE 2. Axioms of Forest Algebras

$C_1^{\boxtimes} = (V_1^{\boxtimes}, E_1^{\boxtimes}, \leq_1^{\boxtimes}, \mathsf{lab}_1)$ be the forest derived from C_1 by removing the hole. Formally we define $C_1 \odot_{\vartriangle} D_2 = (V, E, \leq, \mathsf{lab})$, where

$$\begin{aligned} V &= V_1^{\boxtimes} \cup V_2 \\ E &= E_1^{\boxtimes} \cup E_2 \cup \{(v_1, v_2) \mid (v_1, \Box) \in E_1, v_2 \in \mathsf{roots}(C_2)\} \\ \leq &= \leq_1^{\boxtimes} \cup \leq_2 \cup \{(v_1, v_2) \mid (v_1, \Box) \in \leq_1, v_2 \in \mathsf{roots}(C_2) \text{ or } v_1 \in \mathsf{roots}(C_2), (\Box, v_2) \in \leq_1\} \\ \mathsf{lab} &= \mathsf{lab}_1 \cup \mathsf{lab}_2 \end{aligned}$$

We note that $C_1 \odot_{\vartriangle} D_2$ is a context if D_2 is a context and it is a forest if D_2 is a forest.

Forest Algebras. Here, we introduce forest algebras that were first described by Bojańczyk and Walukiewicz [BW07]. We prefer the syntax used in the Handbook of Automata Theory [Boj21] that also provides a nice introduction.

A *forest algebra*

$$\mathcal{F} \;=\; (H, V, \oplus_{HH}, \oplus_{HV}, \oplus_{VH}, \odot_{VV}, \odot_{VH}, \varepsilon, \Box)$$

consists of two monoids, $(H, \oplus_{HH}, \varepsilon)$ and $(V, \odot_{VV}, \Box)$ along with three monoidal actions:

$\oplus_{HV} : H \times V \to V$ (left monoidal action of H on V),
$\oplus_{VH} : V \times H \to V$ (right monoidal action of H on V), and
$\odot_{VH} : V \times H \to H$ (right monoidal action of V on H).

Intuitively, each element of H represents a forest and each element in V represents a context. The monoid operations correspond to concatenation of forests and context application (on a context), respectively. The neutral elements of H and V correspond to the empty forest and the forest consisting only of the hole. The monoidal actions correspond to concatenation of a forest and a context (or the other way round) and context application of a context on a forest.

In Table 2, we depict the axioms that forest algebras need to satisfy. The two upper groups of axioms are the usual axioms for monoids and monoidal actions stating the effect of the neutral elements and the laws of associativity. The axioms in the bottom group describe the interactions of both monoids and are special for forest algebras. The intuitive explanation of these axioms is that if one concatenates a forest F_1 and a context C_1 (in either order), it is solely the context that is responsible for what happens in a subsequent context application. Therefore a subsequent context application should have the same effect as first doing the context application on C_1 and afterwards doing the concatenation with F_1.

We note that the Axioms A11–A14 are not depicted in [Boj21]. This seems to be an oversight. Without these axioms it is possible to construct a forest algebra $\mathcal{F}$, such that there does not exist a morphism from the free forest algebra (see below) to $\mathcal{F}$. However, [Boj21] states that such a morphism exists for every forest algebra.

We will often drop the indices of the monoid operations and monoidal actions, i.e., we will just use $\oplus$ and $\odot$. Which operation is needed is clear from the operands. In some cases, we do not even specify, whether we refer to concatenation or context application. In this case, we use $\bigcirc$. Given a formula Ψ and an inner node v of the parse tree, we denote by $\bigcirc_v$ the operation at node v.

Free Forest Algebra. The *free forest algebra* over an alphabet Σ is defined as

$$\mathcal{F}_\Sigma = (H, V, \oplus_\vartriangle, \oplus_\vartriangle, \oplus_\vartriangle, \odot_\vartriangle, \odot_\vartriangle, \varepsilon, \square)\ ,$$

where $(H, \oplus_\vartriangle, \varepsilon)$ is the monoid of all forests over Σ with the concatenation operation and $(V, \odot_\vartriangle, \square)$ is the monoid of all contexts over Σ with the context application operation. We note that all five operations of the algebra are provided by the two operations $\oplus_\vartriangle$ and $\odot_\vartriangle$ by restricting them to the respective domains.

Lemma 2.1. *The free forest algebra over any alphabet σ satisfies all axioms of forest algebras.*

Proof sketch. All axioms can be shown by applying the definition of $\oplus_\vartriangle$ and $\odot_\vartriangle$. The axioms for the neutral elements ε and $\square$ (axioms A1–A4) hold, as the operations $\oplus_\vartriangle$ and $\odot_\vartriangle$ are defined by the means of unions where one side of each union is empty.

For the associativity axioms of the horizontal concatenation (axioms A5–A8) it suffices to observe that $\leq$ is constructed in a way that is compatible with associativity. All other components of the resulting forest or context are constructed by taking the union, which is associative.

The associativity of the context application (Axioms A9 and A10) can be shown by distinguishing the holes in C_1 and C_2. The context application involving C_3 or F_1 is always applied to the hole that originates from C_2. Analogously, in the axioms A11–A14, the context application involving C_2 or F_2 is always applied to the hole originating from C_1. $\square$

Parse Trees (of Formulas). The *parse tree* of a forest algebra formula (from some algebra $\mathcal{F} = (V, H)$) is a binary tree with inner nodes labeled by $\{\oplus_{HH}, \oplus_{HV}, \oplus_{VH}, \odot_{VV}, \odot_{VH}\}$ and leaf nodes marked by some element of $V \cup H$.In most cases we will omit the indices from $\oplus$ and $\odot$, as the concrete operation is implied by the types of the operands. However, we will always assume that the concrete operation is stored in the node. This way it is always possible to tell which operand of a concatenation contains the hole (if any).

In our application, leaf nodes of parse trees will always correspond to exactly one node of the tree. That is, we do not use the empty forest ε or the context consisting solely of the hole $\square$. Nor do we have leaves in a parse tree that correspond to more than one node of the represented tree. In the case of formulas from the free forest algebra, leaves will be labeled by elements from $\Sigma \cup \{a_\square \mid a \in \Sigma\}$, where elements from Σ indicate individual nodes with the corresponding label and elements of the form $a_\square$ indicate a node with the given label that has as its only child the hole.

Given a formula Ψ, we denote by $||\Psi||$ the number of leaves in (the parse tree of), which is always equal to the number of nodes in the represented forest or context. As parse trees are always binary trees, they will have one inner node less than leaves. Therefore, a parse tree is always of roughly twice the size than the represented forest or context.

The *balance* $\mathsf{balance}(v)$ of an inner node v of the parse tree is the height difference of the two trees, rooted at the children of v. Positive values for $\mathsf{balance}(v)$ denote that the right subtree is higher than the left subtree. By $\mathcal{B}(\Psi)$ we denote the sum of all absolute balances, i.e.,

$$\mathcal{B}(\Psi) \;=\; \sum_{v \in \Psi} |\mathsf{balance}(v)|\;.$$

To avoid ambiguities in the language—the balancedness of a formula increases when the (absolute) balances decrease—we say that the balance of a formula or node *improves* (by some amount) to denote that the absolute value of balance decreases (by some amount). Similarly we use the verb *worsen* in the opposite case.

For each node v in a parse tree, we denote by $\mathsf{child_L}(v)$ and $\mathsf{child_R}(v)$, the left and right child of v, respectively. If $\mathsf{balance}(v) \neq 0$, we denote with $\mathsf{child_D}(v)$ the child of v that belongs to the deeper subtree. For every node v of a formula Ψ, the *long path* of v denoted by $\mathsf{lp}(v)$ is defined recursively as follows:

$$\mathsf{lp}(v) \;=\; \begin{cases} v & \text{if } \mathsf{balance}(v) = 0 \\ v \cdot \mathsf{lp}(\mathsf{child_L}(v)) & \text{if } \mathsf{balance}(v) < 0 \\ v \cdot \mathsf{lp}(\mathsf{child_R}(v)) & \text{if } \mathsf{balance}(v) > 0 \end{cases}$$

We use $\mathsf{lp}^{-1}(v)$ to denote the upwards path starting at v that contains all nodes u with v being on $\mathsf{lp}(u)$.

We use forest-algebra formulas synonymously with their parse trees. In this sense the height of a forest-algebra formula $\mathsf{height}(\Psi)$ is defined as the height of its parse tree, i.e., the maximal number of nodes on path from the root to a leaf.

We use v and w to denote nodes of either a given tree T, or a parse tree of a forest algebra formula Ψ, respectively. To keep the notation clean, we identify leaves of Ψ with nodes of T, whenever we have a formula Ψ describing a tree T. Given a node v of the parse tree of Ψ, we use Ψ_v to denote the subformula of Ψ rooted at v and T_v to denote the forest or context described by Ψ_v.

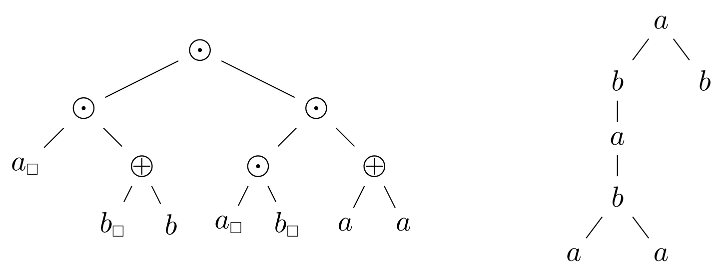


FIGURE 2. Parse tree of a forest-algebra formula and its generated forest.

Morphisms of Forest Algebras. A homomorphism $h = (h_H, h_V)$ from a forest algebra $\mathcal{F}_1 = (H_1, V_1)$ to a forest algebra $\mathcal{F}_2 = (H_2, V_2)$ is given by two monoid morphisms $h_H : H_1 \to H_2$, $h_V : V_1 \to V_2$ that additionally satisfy

$$h_H(C \odot F) \;=\; h_V(C) \odot h_H(F)$$

for all $C \in V_1$ and $F \in H_1$. To simplify notation, we will omit the indices H and V and use h for both morphisms.

For every forest algebra $\mathcal{F} = (V, H)$ one can define a morphism from the free forest algebra $\mathcal{F}_\Sigma$ by providing a function $h : \{a_\square \mid a \in \Sigma\} \to V$, i.e., by providing the mappings for all contexts that have a single root with the hole as its only child. The function h can be extended to a morphism from the free forest algebra by applying the algebra operations. For example, the mapping for a single a-labeled node can be computed as $h(a_\square) \odot_{VH} \varepsilon$.

Updates of Forests and Formulas. We consider the following *updates* on trees:

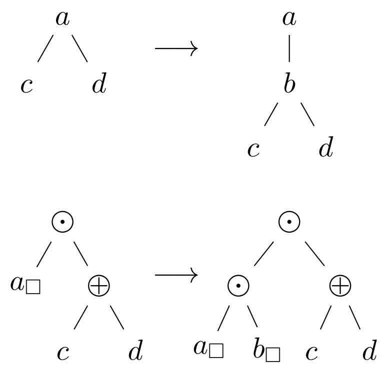
(i) $\mathsf{relab}(v, a)$: Replace the current label of node v by a.
(ii) $\mathsf{subdiv}(v, a)$: Insert a new a-node as the only child of v making all existing children of v children of the new node.
(iii) $\mathsf{insert_L}(v, a)$, $\mathsf{insert_R}(v, a)$: Insert a new a-node as left or right sibling of node v.
(iv) $\mathsf{delete}(v)$: Delete node v, where v must have at most one child. If v has a child, v is replaced by its child.

We refer to the updates as *relabeling*, *subdivision*, *leaf insertion*, and *deletion*. Usually we subsume leaf insertions and subdivions under the broader term *insertions*. We note that the insertion of a node below a leaf is a special case of a subdivision. The update of deleting a node v that has exactly one child is also called edge contraction in graph theory.

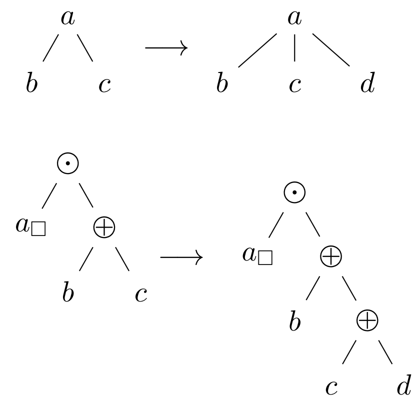
We depict examples of every update (except relabeling) in Figure 3. The figure always depicts an update in a tree and the corresponding update in the formula (see below for an description). We use unique node labels in each tree, such that corresponding nodes in the tree and the formula can be easily identified.

We now describe how each update can be applied to a formula Ψ that represents the tree T. We assume that given a node v in T we can locate the corresponding node v' in the formula in constant time, e.g., by adding pointers to each node in T. In the following, v' is the node in Ψ that corresponds to v.

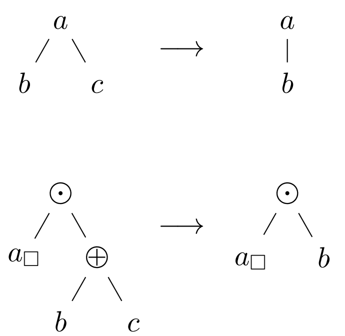
(i) $\mathsf{relab}(v, a)$: It suffices to locate v' in the formula and change its label to a or $a_\square$, depending on whether v is a leaf or an inner node of T.
(ii) $\mathsf{subdiv}(v, a)$: We locate v' in the formula and add a new node w that is labeled $\odot_{VH}$ (if v is a leaf) or $\odot_{VV}$ (if v is an inner node) at the place of v'. We make v' the left



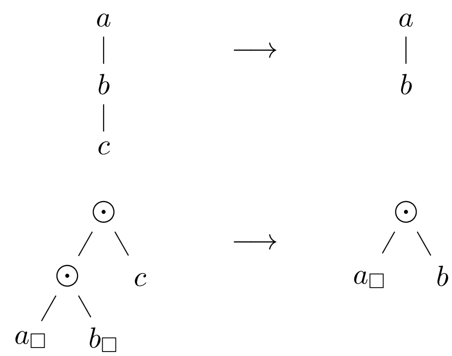
(A) Subdividing the a-node with a new b-node



(B) Inserting a d-node right of the c-node



(C) Deleting the c-node



(D) Deleting the c-node (without siblings)

FIGURE 3. Examples depicting the tree updates considered in this paper and their corresponding rewrites in the forest algebra formula

child of w. The right child of w is a new node that has label a (if v was a leaf) or $a_\square$ (if v was an inner node). If v was a leaf, the label of v' needs to be changed from some label b to the label $b_\square$. An example is depicted in Figure 3a.

(iii) $\mathsf{insert}_\mathsf{L}(v, a)$, $\mathsf{insert}_\mathsf{R}(v, a)$: We replace v' with $(a \oplus v')$ or $(v' \oplus a)$, respectively. An example is depicted in Figure 3b.

(iv) $\mathsf{delete}(v)$: We distinguish two cases.
- Case 1: The node v has a left sibling, a right sibling, or exactly one child. That is, the smallest subformula that strictly contains v' is of the form $(v' \oplus \Psi')$, of the form$(\Psi' \oplus v')$, or of the form $(v' \oplus \Psi')$, respectively, where Ψ' is some subformula. We replace the whole subformula with Ψ', i.e., the $\oplus$-node and v' are removed and Ψ' is placed at the previous position of the $\oplus$-node. An example is depicted in Figure 3c.
- Case 2: The node v has no sibling and is a leaf. That is, the smallest subformula that strictly contains v' is of the form $(\Psi' \odot v')$. In this case, v does not have a sibling in T. We note, that Ψ' is necessarily a context, while $(\Psi' \odot v')$ is a forest. Thus, we cannot simply replace $(\Psi' \odot v')$ by Ψ'. Instead we first have to locate

the node w' corresponding to the parent w of v and change its label from some label $a_\square$ to a (i.e., remove the hole below the node). Furthermore, we have to adapt the labels of ancestors of w' in Ψ', as all of these ancestors change their type from context to forest. Therefore the labels $\oplus_{VH}$ and $\oplus_{HV}$ have to be changed to $\oplus_{HH}$ and the label $\odot_{VV}$ has to be changed to $\odot_{VH}$. At the end, v' has to be removed and the parent of v' has to be replaced by its left subformula. An example is depicted in Figure 3d.

We note that the above cases for $\mathsf{delete}(v)$ are indeed complete. The only case where v is not a leaf and may thus occur on the left side of a context application is when it has at most one child.

The updates relabeling, subdivision and leaf insertion are local to the node v. This does not hold for deletion, as the parent node w of the deleted node will be affected if its last child is deleted. In this case we have to change all nodes on the path between v' and w' in Ψ. We note that it is easy to find the node w' within the parse tree in this case. We are in the case 2 described above, i.e., there is a subformula $(\Psi' \odot v')$ in Ψ. We can perform a top-down search in Ψ' always going to the left subformula at each $\oplus_{VH}$-node and the right subformula at each $\oplus_{HV}$- and $\odot_{VV}$-node. There can be no $\oplus_{HH}$- and $\odot_{VH}$-nodes on this path.

3. Maintaining Forest Algebra Formulas under Updates

This section depicts how (parse trees for) forest algebra formulas can be maintained when the represented tree is updated. That is, the main result of this section is:

Theorem 3.1. *Given a tree T, it is possible to compute in time $\mathcal{O}(|T|)$ a forest algebra formula Ψ in the free forest algebra, such that*

- *Ψ represents T;*
- *the parse tree of Ψ is of height at most $10\log(|T|)$; and*
- *each update to T can be translated to an update of Ψ, such that the new formula can be computed in time $\mathcal{O}(\log(|T|))$ and has height at most $10\log(|T|)$.*

3.1. High Level Description of the Proof.

To keep the parse tree of Ψ shallow, we use rotations similar to those used in AVL trees [AVL62]. Unfortunately, the well-known rotations used to balance AVL trees only work if the underlying algebra is fully associative, which does not hold for forest algebras, as, e.g., $c \odot (f_1 \oplus f_2) \neq (c \odot f_1) \oplus f_2$. Therefore, we provide additional rotations that can be used where the traditional rotations fail.

Towards our invariant we assign colors to nodes of a formula Ψ as follows:

$$\mathsf{color}(v) = \begin{cases} \text{green} & \text{if} \quad \mathsf{height}(\Psi_v) \leq 10\log(||\Psi_v||) \\ \text{yellow} & \text{if} \quad \mathsf{height}(\Psi_v) - 1 \leq 10\log(||\Psi_v||) < \mathsf{height}(\Psi_v) \\ \text{red} & \text{otherwise} \end{cases}$$

The equality can be rearranged to give bounds on the number of leaves of a subformula depending on the height. A green node v satisfies $||\Psi_v|| \geq 2^{\frac{\mathsf{height}(\Psi_v)}{10}}$ and a yellow node v satisfies $||\Psi_v|| \geq 2^{\frac{\mathsf{height}(\Psi_v)-1}{10}}$.

Strong Invariant Every node is green.

Weak Invariant There are no red nodes and there is a node v, such that all yellow nodes are on $\mathsf{lp}^{-1}(v)$.

Observation 3.2. A formula Ψ that satisfies either invariant has logarithmic height.

The strong invariant will hold up between runs of the insertion algorithm, but not necessarily during the runtime of the insertion algorithm. The weak invariant is always satisfied.

We note that if the height of a subformula is increased by one (e.g., due to an insertion), then green nodes can become yellow and yellow nodes can become red. Similarly, if we reduce the height of a subformula by a rotation (introduced in the next subsection), then yellow nodes will turn green.

In Section 3.2 we introduce the rotations used by our algorithm and proof some technical results. The main intuition behind our proof is given by the lemmas 3.10 and 3.11. Together they imply that whenever there is a yellow node v, then we can apply some rotation. We will see that this rotations can lead to another node (strictly below v) becoming yellow. Thus it might be necessary to apply several rotations. In the sections 3.3, 3.4, and 3.5, we show how we can construct a formula for a given tree in linear time, how we can handle insertions, and how we can handle deletions, respectively.

3.2. **Rotations.** Rotations of formulas follow a similar spirit as rotations in AVL trees. They rewrite the formula (preserving equivalence) in such a way that one subformula (i.e., a subtree of the parse tree) is moved one level up and another subformula is moved one level down. Formally, rotations are defined on the algebraic level as follows:

Definition 3.3. A *rotation* α is a rewriting of a forest algebra formula Ψ into an equivalent forest algebra formula $\alpha(\Psi)$ using one of the equations in Table 3, where x_1 to x_3 are variables that can be replaced by arbitrary formulas of the correct type.

A rotation α is applicable at a subformula Ψ_v if one side of the defining equation can be matched to Ψ_v by replacing x_1, x_2, and x_3 with subformulas Ψ_1, Ψ_2, and Ψ_3 from Ψ. Applying the rotation α at Ψ_v is the process of replacing Ψ_v with $\alpha(\Psi_v)$, which results from the other side of the equation by also replacing x_1, x_2, and x_3. We note that all rotations are applicable in both directions.

In Table 3, next to the equations defining the rotations, we depict graphical representations of the rotations. We have annotated the nodes labeled x_1, x_2, and x_3 with arrows $\uparrow$, $\downarrow$ to denote that the subformula is moved upwards or downwards in the rotation. For the rotations 1a and 1b, all operations depicted by $\bigcirc$-nodes need either to be concatenations or context applications, but no mixture. E.g., the rotations 1a in the figure depicts 6 different rotations, as can be seen in Table 3. All $\bigcirc$-nodes could be $\oplus_{HH}$ in case that x_1, x_2, and x_3 are forests, there are three other possibilities for concatenations, where exactly one of x_1, x_2, and x_3 depicts a context. Additionally there are two possibilities involving context applications. The rotations 2a, 2b, 3a, and 3b change the order in which a concatenation and a context application are performed. This also changes the relative order of the operands x_2 and x_3 in the rotations 3a and 3b.

Example 3.4. Figure 4 depicts a tree T, a formula Ψ representing T and the formula that results from applying rotation 3b at the node marked with v.

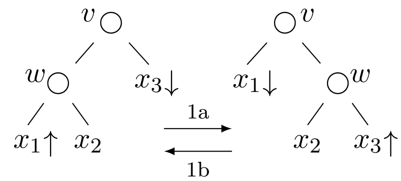
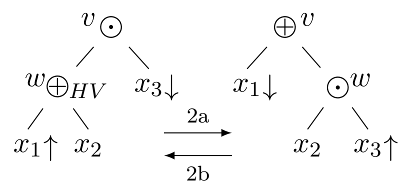
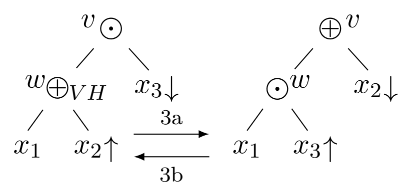
Rotation	Sketch of Parse Tree
$(x_1 \oplus_{HH} x_2) \oplus_{HH} x_3 = x_1 \oplus_{HH} (x_2 \oplus_{HH} x_3)$	
$(x_1 \oplus_{VH} x_2) \oplus_{VH} x_3 = x_1 \oplus_{VH} (x_2 \oplus_{HH} x_3)$	
$(x_1 \oplus_{HV} x_2) \oplus_{VH} x_3 = x_1 \oplus_{HV} (x_2 \oplus_{VH} x_3)$	
$(x_1 \oplus_{HH} x_2) \oplus_{HV} x_3 = x_1 \oplus_{HV} (x_2 \oplus_{HV} x_3)$	
$(x_1 \odot_{VV} x_2) \odot_{VV} x_3 = x_1 \odot_{VV} (x_2 \odot_{VV} x_3)$	
$(x_1 \odot_{VV} x_2) \odot_{VH} x_3 = x_1 \odot_{VH} (x_2 \odot_{VH} x_3)$	
$(x_1 \oplus_{HV} x_2) \odot_{VH} x_3 = x_1 \oplus_{HH} (x_2 \odot_{VH} x_3)$	
$(x_1 \oplus_{HV} x_2) \odot_{VV} x_3 = x_1 \oplus_{HV} (x_2 \odot_{VV} x_3)$	
$(x_1 \oplus_{VH} x_2) \odot_{VH} x_3 = (x_1 \odot_{VH} x_3) \oplus_{HH} x_2$	
$(x_1 \oplus_{VH} x_2) \odot_{VV} x_3 = (x_1 \odot_{VV} x_3) \oplus_{VH} x_2$	

Table 3. Rotations as equations (left) and parse trees (right). Vertical arrows indicate that the respective subformula is moved one level up or down in the rotation. We will refer to individual rotations using the numbers on the horizontal arrows.

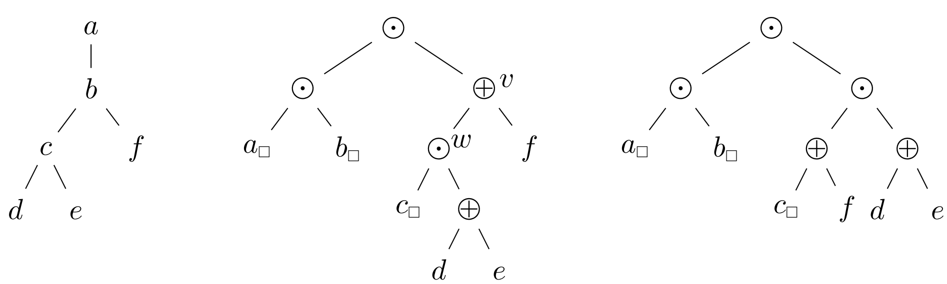


Figure 4. Example of rotation 3b applied at node v

Lemma 3.5. *The rotations depicted in Table 3 are valid operations, i.e., the equations hold for all possible substitutions of x_1 to x_3 with formulas of the correct type.*

Proof. The equations in Table 3 follow immediately from the axioms of associativity and interaction in Table 2. □

Technically, a rotation is applicable at a node v of a formula Ψ whenever the structure of Ψ_v corresponds to the structure required by the rotation. E.g., rotation 1a is applicable whenever the node v and its left child are both labeled with $\oplus$ or both are labeled with $\odot$. Obviously, applying a rotation does not always improve the "balancedness" of the formula. In order to improve the balancedness of a formula, we only apply a rotation, if the balances

rotation	height reducing		height preserving	
	$\mathsf{balance}(v)$	$\mathsf{balance}(w)$	$\mathsf{balance}(v)$	$\mathsf{balance}(w)$
1a	≤ -2	< 0	≤ -2	> 0
1b	$\geq +2$	> 0	$\geq +2$	< 0
2a	≤ -2	< 0	≤ -2	> 0
2b	$\geq +2$	> 0		
3a	≤ -2	> 0	≤ -2	< 0
3b	≤ -2	> 0		

TABLE 4. Balance conditions for height reducing and height preserving rotations

of v and w are as indicated in Table 4. As depicted in the table, depending on the balances, we classify the rotations into height reducing rotations and height preserving rotations. We do not specify height preserving conditions for rotations 2b and 3b, as we will only use the rotation 2b and 3b if they strictly reduce the height.

We now have two lemmas further characterizing the effects of height reducing and height preserving rotations. To ease the presentation of the lemmas and their proofs we introduce some more notation. We depict the subformulas of the rotations by their movement, i.e., we depict by $\Psi_\uparrow$ the subformula (among x_1, x_2, and x_3, see Table 3) that is moved one level up. Similarly we depict by $\Psi_\downarrow$ the subformula that is moved one level down and by $\Psi_\rightarrow$ the subformula that stays on the same level. E.g. for rotation 1a, $\Psi_\uparrow = x_1$, $\Psi_\downarrow = x_3$, and $\Psi_\rightarrow = x_2$ and for rotation 1b, the roles of $\Psi_\uparrow$ and $\Psi_\downarrow$ are reversed, i.e., $\Psi_\uparrow = x_3$ and $\Psi_\downarrow = x_1$. Furthermore, we use v' and w' to denote the nodes v and w after the rotation.

Using this notation, we have that for all rotations it holds that prior to the rotation, $\Psi_\uparrow$ and $\Psi_\rightarrow$ are the two children of w and $\Psi_\downarrow$ is the other child of v (i.e., the sibling of w). Similarly, after the rotation, $\Psi_\uparrow$ becomes a child of v' and $\Psi_\downarrow$ and $\Psi_\rightarrow$ become the children of w'.

Lemma 3.6. *Let α be a height reducing rotation that is applicable at v, then*

(a) *α reduces the height of Ψ_v by one, i.e., $\mathsf{height}(\alpha(\Psi_v)) = \mathsf{height}(\Psi_v) - 1$;*
(b) *α improves the balance of Ψ_v by at least three, i.e., $\mathcal{B}(\alpha(\Psi_v)) \leq \mathcal{B}(\Psi_v) - 3$;*
(c) *all yellow nodes on $\mathsf{lp}^{-1}(v)$ change their color to green;*
(d) *all nodes on $\mathsf{lp}^{-1}(v)$ except v improve their balance by one; and*
(e) *the color of w' is yellow or green if all nodes in Ψ_v are green.*

We remind that $\mathcal{B}(\Psi)$ denotes the sum of the absolute balances of all nodes in Ψ.

Proof. We first observe that by the condition on the balances, we have that

$$\mathsf{height}(\Psi_\uparrow) \quad > \quad \max\big(\mathsf{height}(\Psi_\downarrow), \mathsf{height}(\Psi_\rightarrow)\big)\ . \tag{$\dagger$}$$

(a) The statement follows from (†) and the fact that $\Psi_\uparrow$ is moved one level up.
(b) We now compute the change of $\mathcal{B}(\Psi_v)$. As the balances of the subformulas $\Psi_\uparrow$, $\Psi_\downarrow$, and $\Psi_\rightarrow$ do not change, it is enough to look at the balances of v and w. As we are only interested in changes of the absolute value of the balances, it does not matter which

child is the left/right child of v and w.

$$\begin{aligned}|\mathsf{balance}(v)| &= \mathsf{height}(\Psi_\uparrow) + 1 - \mathsf{height}(\Psi_\downarrow)\\ |\mathsf{balance}(w)| &= \mathsf{height}(\Psi_\uparrow) - \mathsf{height}(\Psi_\rightarrow)\\ |\mathsf{balance}(v')| &= \mathsf{height}(\Psi_\uparrow) - \max\big(\mathsf{height}(\Psi_\downarrow), \mathsf{height}(\Psi_\rightarrow)\big) - 1\\ |\mathsf{balance}(w')| &= |\mathsf{height}(\Psi_\rightarrow) - \mathsf{height}(\Psi_\downarrow)|\end{aligned}$$

We note that the right sides of the first three equations cannot be negative because of (†), which is why we can avoid taking the absolute value. Using the fact that $a + b + |a - b| = 2\max(a, b)$ for arbitrary numbers a and b, we can now compute the change of overall balance as

$$\begin{aligned}\mathcal{B}(\Psi_v) - \mathcal{B}(\Psi_{v'}) &= |\mathsf{balance}(v)| + |\mathsf{balance}(w)| - |\mathsf{balance}(v')| - |\mathsf{balance}(w')|\\ &= \mathsf{height}(\Psi_\uparrow) - \max\big(\mathsf{height}(\Psi_\downarrow), \mathsf{height}(\Psi_\rightarrow)\big) + 2\ .\end{aligned}$$

By (†), we get that $\mathcal{B}(\Psi_v)$ is improved by at least 3.

(c) As by (a), the height of Ψ_v is reduced by one, the height of each Ψ_u with $u \in \mathsf{lp}^{-1}(v)$ is decreased by one. As the number of leaves does not change, the definition of colors implies that yellow nodes on $\mathsf{lp}^{-1}(v)$ turn green.

(d) It is a direct consequence of (a), that all nodes on $\mathsf{lp}^{-1}(v)$ except v improve their balance by one.

(e) It is a trivial observation from the definition of colors that the parent node w of two green nodes has to be green or yellow. The height of Ψ_w is exactly one more than the height of the deeper subtree (which is colored green) and Ψ_w contains more leaves than this subtree. □

The overall idea of our algorithm is to use height reducing rotations whenever the height of some (sub)formula was increased by the insertion of a new node or some (sub)formula needs to be reduced in height after some deletions. However, there are some situations in which no height reducing rotation can be applied, even if the formula is severely imbalanced. In these cases we use height preserving rotations in order to restructure the formula in such a way that afterwards height reducing rotations can be applied.

Lemma 3.7. *Let α be a height preserving rotation that is applicable at v, then*

(a) $\mathsf{height}(\alpha(\Psi_v)) = \mathsf{height}(\Psi_v)$*;*
(b) $|\mathsf{balance}(v')| = |\mathsf{balance}(w)| + 1$*;*
(c) $|\mathsf{balance}(w')| = |\mathsf{balance}(v)| - 1$*;*
(d) $\mathcal{B}(\alpha(\Psi_v)) = \mathcal{B}(\Psi_v)$*; and*
(e) *the color of w' is green if there is a node $u \in \mathsf{lp}(v)$ with $|\mathsf{balance}(u)| \leq 1$ at most six levels strictly below v and all nodes in Ψ_u are green or yellow.*

Proof. Again, we denote by $\Psi_\uparrow$, $\Psi_\downarrow$, and $\Psi_\rightarrow$ the subformulas that moves one level up, moves one level down, and stays on the same level, respectively. The condition on balances for height preserving rotations state that $\mathsf{height}(\Psi_\downarrow) < \max(\mathsf{height}(\Psi_\uparrow), \mathsf{height}(\Psi_\rightarrow))$ and $\mathsf{height}(\Psi_\rightarrow) > \mathsf{height}(\Psi_\uparrow)$. We can conclude

$$\mathsf{height}(\Psi_\rightarrow) \quad > \quad \max\big(\mathsf{height}(\Psi_\uparrow), \mathsf{height}(\Psi_\downarrow)\big)\ . \tag{‡}$$

Now we show the individual statements:

(a) The statement follows from (‡) and the fact that $\Psi_\rightarrow$ stays on the lower level.

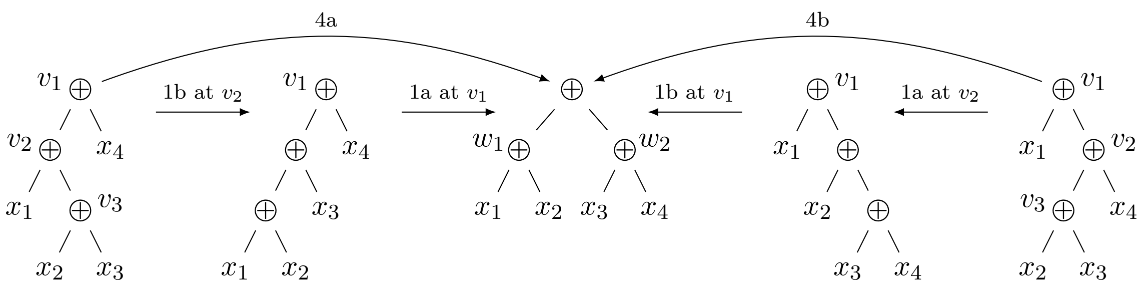


FIGURE 5. Double Rotations. The first rotation is always a rotation 1a or 1b at w followed by a rotation in the opposite direction at v. We denote double rotations as rotations 4a and 4b, respectively.

(b)
$$\begin{aligned} |\mathsf{balance}(v')| &= \left|\mathsf{height}(\Psi_{\uparrow}) - \max\big(1+\mathsf{height}(\Psi_{\rightarrow}), 1+\mathsf{height}(\Psi_{\downarrow})\big)\right| \\ &= \mathsf{height}(\Psi_{\rightarrow}) - \mathsf{height}(\Psi_{\uparrow}) + 1 \\ &= |\mathsf{balance}(w)| + 1 \end{aligned}$$

(c)
$$\begin{aligned} |\mathsf{balance}(w')| &= \mathsf{height}(\Psi_{\rightarrow}) - \mathsf{height}(\Psi_{\downarrow}) \\ &= \left|\max\big(\mathsf{height}(\Psi_{\rightarrow}), \mathsf{height}(\Psi_{\uparrow})\big) + 1 - \mathsf{height}(\Psi_{\downarrow})\right| - 1 \\ &= |\mathsf{balance}(v)| - 1 \end{aligned}$$

(d) This follows from (b) and (c), as no other nodes in Ψ_v change their balance.

(e) We observe that $w' \in \mathsf{lp}(v)$ and thus $w' \in \mathsf{lp}^{-1}(u)$, as the lowest level of Ψ_v is in $\Psi_{\rightarrow}$ for all height preserving rotations by the balances given in Table 4. We can conclude that Ψ_u is a (not necessarily strict) subformula of $\Psi_{w'}$. We furthermore use the fact that any node u' that is yellow or green has at least $2^{\frac{\mathsf{height}(\Psi_{u'})-1}{10}}$ many leaves to bound the number of leaves in Ψ_v as follows:

$$\begin{aligned} ||\Psi_{w'}|| &\geq ||\Psi_{\mathsf{child}_{\mathsf{L}}(u)}|| + ||\Psi_{\mathsf{child}_{\mathsf{R}}(u)}|| && \Psi_u \text{ is a subformula of } \Psi_{w'} \\ &\geq 2^{\frac{\mathsf{height}(\Psi_{\mathsf{child}_{\mathsf{L}}(u)})-1}{10}} + 2^{\frac{\mathsf{height}(\Psi_{\mathsf{child}_{\mathsf{R}}(u)})-1}{10}} && \text{definition of colors} \\ &\geq 2^{\frac{\mathsf{height}(\Psi_{w'})-7}{10}} + 2^{\frac{\mathsf{height}(\Psi_{w'})-8}{10}} && \text{relative height of nodes}^{1} \\ &> 2 \cdot 2^{\frac{\mathsf{height}(\Psi_v)-8}{10}} = 2^{\frac{\mathsf{height}(\Psi_v)+2}{10}} \end{aligned}$$

We note that u cannot be a leaf, as the height of u is larger than the height of its sibling. The statement follows from the definition of colors that says, that a node w' is green if and only if $||\Psi_{w'}|| \geq 2^{\frac{\mathsf{height}(\Psi_{w'})}{10}}$. □

In some cases, we use double rotations, which are exactly the double rotations used in AVL trees. We only apply double rotations if there are three consecutive $\oplus$-nodes v_1, v_2, v_3 on $\mathsf{lp}(v_1)$ where the direction of the balances alternates. A double rotation then consists of a height preserving rotation at v_2 followed by a height reducing rotation at v_1. We depict the two possibilities for double rotations in Figure 5 and will denote them as rotations 4a and 4b respectively. We may use the term *simple rotation* to emphasize that a rotation is not a double rotation. From Lemma 3.7(e) we get the following corollary.

[1]The node u is at most 5 levels below w' and both children of u have a height difference of exactly 1.

Corollary 3.8. *If all nodes are green or yellow and a double rotation is applied at some node v_1 that has a node u with $|\mathsf{balance}(u)| \leq 1$ at most 7 levels below on $\mathsf{lp}(v_1)$, then the node w_1 as indicated in Figure 5 (in case of rotation 4a) or the node w_2 (in case of rotation 4b) is green after the rotation.*

We note that the node w_1 or w_2 is the node w' of the first of the two rotations in the double rotation. Therefore, we can directly apply Lemma 3.7(e).

Observation 3.9. Given a formula Ψ and a node v in Ψ, there is at most one rotation α that can be applied at v while respecting the balance conditions in Table 4.

In our algorithms, we will use a function DoRotation(v) that applies the unique possible rotation that erspects the balance conditions in Table 4 at node v and returns the node that is labeled with w in Table 3. This function is convenient, as neither need case distinctions on the possible rotation nor on whether we have to continue with the left or the right child of v.

The following two lemmas are heavily used in all our algorithms. Together, they establish that whenever the formula is heavily imbalanced, then the balance can be improved as some rotation is possible. Intuitively, Lemma 3.10 says that whenever there is a yellow node v, then the long path starting at v is imbalanced, and Lemma 3.11 says that whenever there is an imbalanced path, then we can apply some rotation. Usually both lemmas are applied together, but sometimes Lemma 3.11 is used on its own and we establish the precondition by other means.

Lemma 3.10. *Let Ψ be a formula that only has green and yellow nodes and v_0 be a yellow node. Let $\mathsf{lp}(v_0) = v_0 \cdots v_k$ be the long path starting at v_0. Then $k \geq 8$ and for $0 \leq i \leq 8$ it holds that $|\mathsf{balance}(v_i)| \geq 2$.*

Proof. Let $h = \mathsf{height}(\Psi_v)$ be the height of Ψ_v and $j \in \{1, \dots, k\}$ be the smallest number such that $|\mathsf{balance}(v_j)| < 2$. As every long path ends with a node of balance 0, such a j exists. By definition of balances and paths in trees, we have that the height of Ψ_{v_j} is $h - j$ and the heights of the two subformulas of Ψ_{v_j} are $h - j - 1$ and $h - j - 1 - |\mathsf{balance}(v_j)|$. As both subformulas of Ψ_{v_j} only have green and yellow nodes, we can conclude that Ψ_{v_j} has at least

$$2^{\frac{h-j-1}{10}} + 2^{\frac{h-j-1-|\mathsf{balance}(v_j)|}{10}} \quad \geq \quad 2 \cdot 2^{\frac{h-j-2}{10}} \quad = \quad 2^{\frac{h-j+8}{10}}$$

many leaves, where $h - j - 1$ and $h - j - 1 - |\mathsf{balance}(v_j)|$ are the heights of the children of v_j. The inequality follows by $|\mathsf{balance}(v_j)| \leq 1$. As v_0 is yellow, i.e., $||\Psi_v|| < 2^{\frac{h}{10}}$, and $||\Psi_{v_j}|| < ||\Psi_{v_0}||$, we can conclude that j has to be at least 9. This concludes the proof. □

Lemma 3.11. *Let Ψ be a formula and v be a node in Ψ such that the first 8 nodes on $\mathsf{lp}(v)$ have an absolute balance of at least two. Then there is a node $u \in \mathsf{lp}(v)$, such that either*

- *a height reducing rotation is applicable at u;*
- *a height reducing double rotation is applicable at u; or*
- *u is a $\odot$-node, $\mathsf{balance}(u) \leq -2$ and a height preserving rotation is applicable at u.*

Proof. We denote v by v_0. Let $\mathsf{lp}(v_0) = v_0 \cdots v_k$. By precondition, we have $|\mathsf{balance}(v_i)| \geq 2$ for $0 \leq i \leq 7$. We show the statement by a series of case distinctions on the balances and labels of the nodes $v_0, \dots, v_7$.

The first case is that there is a $\odot$-node v_i with $i < 7$ and $\mathsf{balance}(v_i) \leq -2$. In this case we can apply one of the rotations $1a$ (if v_{i+1} is a $\odot$-node), $2a$ (if v_{i+1} is a $\oplus_{HV}$-node), or $3a$

(if v_{i+1} is a $\oplus_{VH}$-node). We note that v_{i+1} cannot be a $\oplus_{HH}$-node as $\Psi_{v_{i+1}}$ must represent a context. This rotation is either height reducing or height preserving, depending on the sign of $\mathsf{balance}(v_{i+1})$.

From now on, we assume that all $\odot$-nodes from $v_0, \ldots, v_6$ have a positive balance, as otherwise the first case would apply. We note that by assumption none of these nodes can have a balance of -1, 0, or $+1$.

The second case is that there are two consecutive $\odot$-nodes v_i and v_{i+1} with $i \leq 5$. In this case, we can apply the height reducing rotation 1b at v_i.

The third case is that there are three consecutive $\oplus$-nodes v_i, v_{i+1}, and v_{i+2} with $i \leq 4$. In this case, we can apply one of the rotations 1a, 1b, 4a, or 4b, depending on the signs of $\mathsf{balance}(v_i)$ and $\mathsf{balance}(v_{i+1})$.

Case 4: If none of the first three cases apply, then there are neither two consecutive $\odot$-nodes nor three consecutive $\oplus$-nodes on $v_0, \ldots, v_6$. But then there have to be two $\oplus$-nodes v_i and v_j both immediately followed by $\odot$-nodes v_{i+1} and v_{j+1} with $0 \leq i < j \leq 5$.

Case 4a: Either v_i is a $\oplus_{HH}$-node, v_i is a $\oplus_{VH}$-node with $\mathsf{balance}(v_i) < 0$, or v_i is a $\oplus_{HV}$-node with $\mathsf{balance}(v_i) > 0$. In this case we can either apply height reducing rotation 2b or height reducing rotation 3b. We remind that the $\odot$-node v_{i+1} has a positive balance, as otherwise case 1 would apply.

Case 4b: The remaining cases are that v_i is a $\oplus_{VH}$-node with $\mathsf{balance}(v_i) > 0$ or v_i is a $\oplus_{HV}$-node with $\mathsf{balance}(v_i) < 0$. In both cases, $\Psi_{v_{i+1}}$ represents a forest. If $\Psi_{v_{i+1}}$ represents a forest, then all nodes v_ℓ with $i < \ell \leq 6$ are either $\oplus_{HH}$ or $\odot_{VH}$ nodes. A formula that represents a forest must have either a $\oplus_{HH}$-node or a $\odot_{VH}$-node at the top. For a $\oplus_{HH}$-node, both subformulas have to represent forests, and for a $\odot_{VH}$-node, the right subformula has to represent a forest. And on $\mathsf{lp}(v)$ we are always going to the right subformula at each $\odot$-node, as we assume that these nodes have positive balance. Therefore, v_j has to be a $\oplus_{HH}$-node and we can either apply 2b or 3b. This concludes the proof. □

3.3. **Preprocessing in Linear Time.** This subsection is devoted to prove the following theorem:

Theorem 3.12. *Given a tree T, one can construct in time $O(|T|)$ a formula Ψ of height $O(\log(|T|))$ that represents T.*

We depict the algorithm as Algorithm 1. The main function CONSTRUCT takes T as argument, constructs a formula Ψ corresponding to the root of the tree, and calls CONSTRUCTRECURSIVE to insert all descendants of the root into Ψ. At the end it performs some optimizations (by rotations) before returning the formula Ψ. The function CONSTRUCTRECURSIVE does a preorder traversal of T and inserts each node into Ψ, using either subdivision (to insert the first child of some node) or leaf insertion (to insert all other children). As the insertion increases the height of the subformula Ψ_v, the algorithm calls the function OPTIMIZEUPWARDS that goes upwards and searches for a place to apply a height reducing rotation. A height reducing rotation at some node u neutralizes the height increase due to the insertions for the ancestors of u. If OPTIMIZEUPWARDS does not find a possibility for a height reducing rotation, it does not modify the formula. It stops once it reaches a node whose height did not change, because the insertion was in the shallower subtree. For a more detailed description of how OPTIMIZEUPWARDS works, we refer to the following proofs, especially the proof of Lemma 3.15.

Algorithm 1 Construct formula Ψ from tree T

```
function construct(non-empty tree T)
    Ψ ← lab(root(T))
    constructRecursive(root(T), root(Ψ))
    optimizeAll(root(Ψ))
    return Ψ

function constructRecursive(v_t: a node of T, v: leaf of Ψ corresponding to v_t)
    if v_t has a child then
        subdiv(v, lab(first-child(v_t)))                          ▷ see Section 2
        optimizeUpwards(v)
        constructRecursive(first-child(v_t), child_R(v))
    if v_t has a right sibling then
        insert_R(v, lab(next-sibling(v_t)))                        ▷ see Section 2
        optimizeUpwards(v)
        constructRecursive(next-sibling(v_t), child_R(v))

function optimizeUpwards(v: node in some formula Ψ)
    while v ≠ ⊥ and no height reducing rotation was done do
        if hr-rotationPossible(v) then
            DoRotation(v)
        else if v is a ⊙-node with balance(v) ≤ −7 then
            while height preserving rotation possible at v do
                w ← DoRotation(v)
                v ← w                         ▷ Go to the node w as depicted in Table 3
        else
                 ⎧ Parent(v)   if v = child_D(Parent(v))   ▷ stop if height of Ψ_v is not larger
            v ←  ⎨                                             than height of its sibling
                 ⎩ ⊥           otherwise

function optimizeAll(v: node in some formula Ψ)
    if v is an inner node then
        optimizeAll(child_L(v))
        optimizeAll(child_R(v))
        doAllRotations(v)

function doAllRotations(v: node in some formula Ψ)
    while rotationPossible(v) do
        DoRotation(v)
        doAllRotations(child_L(v))
        doAllRotations(child_R(v))
```

After a complete formula is constructed, the algorithm optimizes the complete formula by doing a postorder traversal of the formula and applying all possible rotations. This optimizing step is necessary, as the construction does not apply all possible rotations and might create subformulas that are of height linear in the number of nodes in the subformula. Especially if we allow deletions later, this might lead to formulas that are heavily unbalanced.

Lemma 3.13. *Algorithm 1 is correct, i.e., after the algorithm the formula Ψ represents the tree T.*

Proof. It is straightforward to verify that the function CONSTRUCT does indeed construct a correct formula if one ignores all calls to OPTIMIZEUPWARDS and the call to OPTIMIZEALL. The function does a preorder traversal of T and inserts each node it finds into the formula.

The only changes applied to the formula by the functions OPTIMIZEUPWARDS and OPTIMIZEALL are rotations. By Lemma 3.5, which says that rotations do not change the represented tree, we get that Algorithm 1 does indeed construct a formula that represents T. □

Furthermore, it is easy to see that after the algorithm is finished, no rotation can be applied. Therefore, lemmas 3.10 and 3.11 give us the following:

Corollary 3.14. *Algorithm 1 produces a formula that satisfies the strong invariant.*

It remains to show that the algorithm runs in linear time in the size of T. Even if phrased differently, the main idea of the proof follows the line of an amortized analysis that Mehlhorn and Tsakilidis [MT86] did for AVL trees. We first analyze the behavior of OPTIMIZEUPWARDS at $\odot$-nodes with a big negative balance.

Lemma 3.15. *If the function OPTIMIZEUPWARDS arrives at a $\odot$-node u with $\mathsf{balance}(u) \leq -7$, then it applies a series of height preserving rotations, followed by a height reducing rotation on $\mathsf{lp}(u)$.*

Proof. We use v_0 to denote the node v of the initial call to OPTIMIZEUPWARDS. We establish the following invariant: As long as the while-condition is true, the node v_0 is always on $\mathsf{lp}(v)$. Indeed this trivially holds true if v is reassigned in Line 22 and it also holds true if v is reassigned in Line 24, as the node w after a height preserving rotation is always on $\mathsf{lp}(v)$.

Let now v be a $\odot$-node with $\mathsf{balance}(v) \leq -7$ that is encountered by the algorithm. We first observe that at a $\odot$-node with a balance less than minus one, the only scenario where no rotation is applicable is that the left child has balance 0. Otherwise we could apply one of the rotations 1a, 1b, 2a, or 3a. We now argue that the left child of v cannot have $\mathsf{balance}(v) = 0$. Indeed, v_0 that is always on $\mathsf{lp}(v)$ is the only node on $\mathsf{lp}(v)$ that can have balance 0. However $\mathsf{height}(v_0) = 2$, as v_0 is the parent of the newly inserted node. This would imply $|\mathsf{balance}(v)| \leq 1$.

So we have established that a rotation at v is possible. If the possible rotation is height reducing, we are done. Otherwise, the rotation is height preserving. The node w returned by DOROTATION(v) in Line 21 is the deep child of v. The balance of w is one more (i.e., it is less negative) than the balance of v was before the rotation (Lemma 3.7(c)), and this node w has to be a $\odot$-node. The algorithm continues until either a height reducing rotation is possible at the current node or it has done at least six height preserving rotations in a row, as the balance of the next node it looks at is always exactly one more than the previous one (Lemma 3.7(c)). In the former case, we apply the height reducing rotation and are done. In the latter case, the nodes where the algorithm applied height preserving rotations are consecutive on $\mathsf{lp}^{-1}(u)$, where u is the lowest node where a height preserving rotation was performed. By Lemma 3.7(b) all those nodes have an absolute balance of at least 2. Therefore, by Lemma 3.11, a rotation is possible at one of these nodes. This rotation cannot be height preserving, as the algorithm just applied height preserving rotations at all of these nodes. Thus either the balance is positive (rotations 1a and 2a), or the node is a $\oplus$-node (rotation 3a). The algorithm will go upwards again once no height preserving rotation is possible any more, apply this height reducing rotation and stop. It will therefore not go above the highest node on which a height reducing rotation was applied. This concludes the proof. □

We introduce a bit of notation that helps with the following proof. For the amortized analysis, we use the following potential function:

$$\Phi(\Psi) \quad = \quad |\{v \mid \mathsf{balance}(v) = 0\}| \;+ \sum_{\substack{v \,\in\, \{w \in \Psi \mid \mathsf{lab}(w) = \odot \text{ and} \\ -6 \leq \mathsf{balance}(w) \leq 0\}}} 7 + \mathsf{balance}(v)$$

The first term counts the number of nodes in Ψ with balance 0. These nodes can get an absolute balance of 1 by the insertion. They have to be skipped when going upwards as no rotation is possible. The second term, i.e., the sum, accounts for $\odot$-nodes with a balance between -6 and 0. These nodes also might have to be skipped when going upwards. How often the same node can be skipped during consecutive insertions depends on its balance. Once the imbalance is big enough, we know that we can perform a series of height preserving rotations followed by one height reducing rotation. We note that for these nodes the balance is always negative and thus $7 + \mathsf{balance}(w)$ produces numbers between 1 and 6 inclusive. The number corresponds to how often a node might need to be skipped. Obviously, $\Phi(\Psi)$ cannot be negative.

Lemma 3.16. *The summarized runtime of all calls to* OPTIMIZEUPWARDS *in Algorithm 1 is in* $O(|T|)$.

Proof. We analyze the change of the potential $\Phi(\Psi)$ during the insertion of a new node v. We need to show two things, that the potential is increased by at most a constant value during the insertion operation, and that the runtime is linear in the overall decrease of potential. The proof does not calculate the net change of potential directly. Instead we account for increments and decrements separately.

First of all, we calculate the increments. The insertion adds a new inner node $\mathsf{Parent}(v)$ that has a balance of 0 and could be a $\odot$-node. This increases the potential by at most 8. The only other node u that can increase its potential is the parent of the topmost node on $\mathsf{lp}^{-1}(\mathsf{Parent}(v))$. This node u can improve its balance by 1 and thus possibly increase its potential by 2. All other changes of balance that directly result from the insertion (and not from a rotation) can only worsen the balance of some node. Therefore, they cannot increase the potential.

We already know that the algorithm performs at most one height reducing rotation (as it will directly stop afterwards) and at most one series of height preserving rotations, i.e., it enters the while-loop in Line 20 at most once. After one series of height preserving operations, either a height reducing rotation is possible at the current node, or at most six nodes above (see discussion of the algorithm above). All nodes where we did a height preserving rotation will have an absolute balance of at least two and if they are a $\odot$-node they have a positive balance. Therefore, none of the rotations did increase the potential of any of these nodes. We only need to account for an increased potential due to the height reducing rotation and for the w-node of the last height preserving rotation. This can be at most constant. We note that the balances of the ancestors of the highest node reached in the whole insertion operation cannot change if we do perform a height reducing rotation, as this rotation cancels out the height increase due to the insertion.

We still need to look at the nodes on the path from u to $\mathsf{Parent}(v)$, i.e., from the topmost node reached by the algorithm to the parent of the inserted node. The balances of these nodes worsen by one. This will decrease the potential $\Phi(\Psi)$ by one for every node that changes the balance from zero to some non-zero value, i.e., for every node on the path that prior to the insertion had a balance of zero. Furthermore it decreases the potential by one for every $\odot$-node that has a balance between -6 and 0 (inclusive). We now show that the decrease in potential is linear in the runtime of OPTIMIZEUPWARDS.

In fact, by Lemma 3.11, at least every seventh node on the path from u to $\mathsf{Parent}(v)$ had a balance of 0, or was a $\odot$-node with a balance in the range -6 to 0. We remind that all nodes on this path that had a non-zero balance before the insertion will have an absolute balance of at least 2 after the insertion. Therefore, if there are more than six consecutive nodes that did not have a balance of zero prior to the insertion, nor are $\odot$-nodes with a balance in the given range, either we would stop as described in Lemma 3.15, or at one of these nodes we can apply a height reducing rotation. This is a contradiction to the assumption that u is the highest node reached by the function.

As the potential cannot become negative, is increased by at most a constant for each insertion, and is decreased by at least a number linear in the runtime of OPTIMIZEUPWARDS, the summarized runtime of all calls to OPTIMIZEUPWARDS is in $O(|T|)$. □

It remains to show the runtime of the OPTIMIZEALL function. Towards this goal, we want to understand how applying a rotation at a node v affects $\mathcal{B}(\Psi)$.

Lemma 3.17. *Applying a height preserving rotation at some node v in Ψ does not change $\mathcal{B}(\Psi)$. Applying a height reducing rotation at some node v in Ψ strictly improves $\mathcal{B}(v)$.*

Proof. By Lemma 3.7(d), a height preserving rotation at v does not change $\mathcal{B}(\Psi_v)$ and it is easy to see that a height preserving rotation cannot change the balance of any node outside of Ψ_v. By Lemma 3.6(b), a height reducing rotation at v improves $\mathcal{B}(\Psi_v)$ by at least three. There is at most one node u in Ψ (the parent of the topmost node on $\mathsf{lp}^{-1}(v)$), such that $|\mathsf{balance}(u)|$ worsens. Furthermore, $|\mathsf{balance}(u)|$ can only worsen by one. Altogether we can cocnlude that $\mathcal{B}(\Psi)$ strictly improves. □

Lemma 3.18. *The runtime of* OPTIMIZEALL *is linear in $O(|T|)$ if called from the construction algorithm.*

Proof. We first observe that the total imbalance $\mathcal{B}(\Psi)$ of the formula Ψ is linear in $O(|T|)$ just before the call to OPTIMIZEALL. In fact, for each insertion, $\mathcal{B}(\Psi)$ is only worsened by one for each node on the path from u to $\mathsf{Parent}(v)$, where v is the newly inserted node and u is the topmost node reached by the OPTIMIZEUPWARDS function. As the runtime of OPTIMIZEUPWARDS is in $\Theta(\mathsf{height}(\Psi_u))$, and the summarized runtime of all those calls is linear in $|T|$ by Lemma 3.16, $\mathcal{B}(\Psi)$ is linear in $|T|$. From Lemma 3.17 we know that rotations performed by OPTIMIZEUPWARDS can only improve $\mathcal{B}(\Psi)$.

Towards the lemma statement, we show that the runtime of OPTIMIZEALL is linear in $|T| + \mathcal{B}(\Psi)$. It is obvious that OPTIMIZEALL is called exactly once for each inner node in the formula, as this is a classic post-order traversal. And the number of calls to DOALLROTATIONS is one for each inner node of the formula and additionally two calls for every performed rotation. It thus suffices to show that the number of performed rotation is linear in $\mathcal{B}(\Psi)$.

As height reducing rotations strictly improve $\mathcal{B}(\Psi)$ and height preserving rotations leave $\mathcal{B}(\Psi)$ unchanged (Lemma 3.17), it remains to bound the number of height preserving rotations. We already know that, if there is more than a constant number of height preserving rotations on one downwards path, there has to be a height reducing rotation that can be applied near the downwards end of the path, as all the nodes on the path will have an absolute balance of at least 2 and have positive balance if they are $\odot$-nodes. Thus the algorithm will apply a height reducing rotation which improves the value linearly in the length of the path (Lemma 3.6(d)), which corresponds to the number of performed height preserving rotations. Altogether the runtime is bounded by the size of T and the overall imbalance of Ψ. □

Theorem 3.12 now follows from Lemma 3.13 (soundness), Corollary 3.14 (height of formula) and lemmas 3.16 and 3.18 (runtime).

3.4. **Maintaining Parse Trees under Insertions.** Our proof that we can maintain the strong invariant under insertions takes several steps. In Lemma 3.19, we show that after any insertion, the formula still satisfies the weak invariant. We already established in Lemma 3.10 that on the long path of any yellow node, there are several consecutive nodes that are imbalanced. This is exploited in Lemma 3.11 to show that we can apply some rotation to reduce the height of Ψ_v and change the color of the yellow node back to green. Unfortunately, the rotation might create (at most) one other yellow node. However, as this new yellow node is strictly deeper in the parse tree as the previous one, at most logarithmically many repetitions suffice to reestablish the strong invariant. This leads to Algorithm 2 that we use to recursively reestablish the strong invariant and Algorithm 3 that performs the insertion and makes use of Algorithm 2.

Lemma 3.19. *Let Ψ be a formula that satisfies the strong invariant. Then after applying an insertion update Δ on Ψ, the resulting formula $\Delta(\Psi)$ satisfies the weak invariant and all yellow nodes of $\Delta(\Psi)$ are on $\mathsf{lp}^{-1}(v)$, where v is the parent of the newly inserted node.*

Proof. In all possibilities of a node insertion in the tree (insertion of a leaf as a left/right sibling of an existing node and subdivision), a leaf of the formula Ψ is replaced by a subformula with one inner node v and two leaves. One of these leaves is the new node. Thus for all nodes on $\mathsf{lp}^{-1}(v)$ in $\Delta(\Psi)$ it holds that the height and the number of leaves in the subtree has increased by one. For all nodes not on $\mathsf{lp}^{-1}(v)$ the height did not change and the number of leaves in the subtree might have increased by one. As observed in Section 3.1, for nodes on $\mathsf{lp}^{-1}(v)$, the color can change from yellow to red, or from green to yellow. As all nodes are green prior to the insertion, there are no red nodes after the insertion and all yellow nodes after the insertion have to be on $\mathsf{lp}^{-1}(v)$. Thus the weak invariant is satisfied. □

We now state Algorithm 2 that tries to optimize the height of a given (sub-)formula. Algorithm 2 does not check the color of any nodes. Instead it just starts at a given node v, goes downwards on the long path and performs rotations if possible. The colors are only used in the proofs. The test simpleRotationPossible(v) in Line 3 returns true if one of the following is true:

- a height reducing simple rotation is possible at node v; or
- v is a $\odot$-node with negative balance and a height preserving rotation is possible at v.

Algorithm 2 Try to optimize the height of a subformula Ψ_v

1: **function** TryReduceHeight(Ψ, v)
2:     **if** $|\mathsf{balance}(v)| \geq 2$ **then**
3:         **if** simpleRotationPossible(v) **then**
4:             $w \leftarrow$ DoRotation(v)
5:             TryReduceHeight(Ψ, w)
6:         **else**
7:             TryReduceHeight($\Psi, \mathsf{child}_{\mathsf{D}}(v)$)
8:     **else if** rotationPossible(u) for some node $u \in \mathsf{lp}^{-1}(v)$ **then**†
9:         $w \leftarrow$ DoRotation(u)
10:         TryReduceHeight(Ψ, w)

†If there are several nodes u on $\mathsf{lp}^{-1}(v)$ that satisfy the condition, we use the lowest such node.

The test rotationPossible(v) in Line 8 returns true if one of the two cases above applies or a height reducing double rotation is possible at v. The function DoRotation(v) in lines 4 and 9 applies the unique (see Observation 3.9) possible rotation at node v and returns the node that is labeled with w in Table 3. In case of a double rotation, it returns the node w_1 or w_2 (see Figure 5) that can have a non-green color. We remind that according to Corollary 3.8, only one of the two nodes can have a non-green color.

We now have two lemmas that characterize the effects of Algorithm 2 and one lemma talking about the runtime.

Lemma 3.20. *Let v be a yellow node in a formula Ψ that consists only of green and yellow nodes. Let Ψ'_v denote the subformula Ψ_v after the call* TryReduceHeight(Ψ, v). *Then* $\mathsf{height}(\Psi'_v) = \mathsf{height}(\Psi_v) - 1$. *If v is the only yellow node in Ψ_v, then all nodes in Ψ'_v are green.*

Proof. We use v_i to denote the value of v in the i-th recursion step of the algorithm, where v_0 is the value of v in the topmost call.

All recursive calls are made on $\mathsf{lp}(v_0)$ unless we perform a height reducing rotation at some v_i, reducing the height of each Ψ_{v_j} with $j \leq i$ by one. Indeed, if at some node no rotation is performed, we stay on the long path, as we are going to the deeper child. And if a height preserving rotation is performed, the node w returned by the call doRotation is also on the deeper subformula.

By Lemma 3.10, we know that the first 8 nodes on $\mathsf{lp}(v_0)$ have an absolute balance of at least 2. This does not change as long as we only do height preserving rotations on the path, as we can reapply the lemma on the modified path after the rotation. By lemmas 3.10 and 3.11, we know that as long as v stays yellow, i.e, as long as the algorithm does not apply a height reducing rotation, there will always be a rotation that is applicable on $\mathsf{lp}(v_0)$. Thus at some point, the algorithm performs a height reducing rotation on $\mathsf{lp}(v_0)$, either in Line 4 or in Line 9. This shows the first part of the lemma statement. We note, that no height preserving rotation is possible on $\mathsf{lp}^{-1}(v_i)$, where v_i is the current node, as a height preserving rotation cannot enable another height preserving rotation.

We now show the second part of the lemma. If v is the only yellow node in Ψ_v, then there are only green nodes in Ψ'_v. From the first part of the lemma, it is clear, that v will be green after the call to TryReduceHeight, as the height is reduced by one.

If after a simple rotation we have another yellow node, then this node can only be the node w as denoted in Table 3. The node v cannot turn yellow, has the height cannot increase and the subformulas $\Psi_\uparrow$, $\Psi_\downarrow$, and $\Psi_\rightarrow$ are not touched by the rotation. However, we do a recursive call on every node w of every performed rotation. A simple inductive argument yields that a yellow node that was created by a rotation turns green by the recursive call on that node.

Double rotations in Line 9 work very similar. We discuss the two rotations of the double rotations seperately, starting with the height preserving rotation, i.e., the first rotation of the double rotation. By Lemma 3.7(e), we have that w is green. We remind that u is at most 6 nodes above v and $|\mathsf{balance}(v)| \leq 1$ by lemmas 3.10 and 3.11. The argumentation for the second rotation of the double rotation is exactly as for simple rotations, as we also do a recursive call on w. This concludes the proof. □

The final property that we need for the TryReduceHeight function is its runtime. Again, the lemma statement is more detailed than needed for the insertion algorithm, as we need the additional detail later for the deletion algorithm.

Lemma 3.21. *The runtime of* TryReduceHeight(Ψ, v) *is in* $O(\Delta_\mathcal{B})$*, where* $\Delta_\mathcal{B}$ *is the improvement of the balance of* Ψ*, i.e.,* $\Delta_\mathcal{B} = \mathcal{B}(\Psi) - \mathcal{B}(\Psi')$ *with* Ψ' *denoting the formula* Ψ *after the call to* TryReduceHeight.

Proof. We show that $\Delta_\mathcal{B} \geq \max(\frac{x}{7} - 7, 0)$, where x is the recursion depth counting the initial call as 1. The proof is by induction. If the algorithm never performs a height reducing rotation, then the recursion depth is bounded by 6 by Lemma 3.11. If the algorithm at some point performs a (first) height reducing rotation in Line 4, then $\mathcal{B}(\Psi)$ is improved by at least y, where y is the (current) recursion depth. The balance of Ψ_{v_y} is improved by at least three (Lemma 3.6(b)). The balance on $\mathsf{lp}^{-1}(v_y)$ is improved by at least $y-1$ (Lemma 3.6(d)), and there is at most one node (the parent of the topmost node on $\mathsf{lp}^{-1}(y)$) that worsens its balance by one.

The argument follows by induction on the recursive call in Line 5. Let x be the total recursion depth, then the recursive call has a remaining recursion depth of $x - y$, yielding an improvement of balance of at least $\max(\frac{x-y}{7} - 7, 0)$. The total improvement of balance is thus at least $\max(\frac{x-y}{7} - 7, 0) + y \geq \max(\frac{x}{7} - 7, 0)$.

Similarly, if the algorithm performs a double rotation in Line 9 at recursion level y, then $\mathcal{B}(\Psi)$ is improved by at least $\max(1, y-6) \geq \frac{y}{7}$, as the node at which the double rotation is performed is at most 6 nodes above the current node v. This implies that the improvement is at least $\frac{y}{7}$. Again, the argument follows by induction, concluding the proof. □

A direct consequence of Lemma 3.21 is that the runtime is bounded by a constant if no height reducing rotation is performed.

The final algorithm for insertions is depicted as Algorithm 3. It inserts the new node and then walks upwards until it either finds a possibility to reduce the height, or reaches the end of the long path. Nodes above did not increase height and therefore cannot change color to the worse.

Lemma 3.22. *Algorithm 3 performs the insertion operation correctly, maintains the strong invariant, and runs in time* $O(\mathsf{height}(\Psi_v))$*, where* v *is the topmost node reached in the while-loop.*

Algorithm 3 Insertion Update on a Formula Ψ

1: **function** INSERT(update $\Delta \in \{\mathsf{insert_L}(v,a), \mathsf{insert_R}(v,a), \mathsf{subdiv}(v,a)\}$, formula Ψ)
2: apply the update Δ
3: $v \leftarrow \mathsf{Parent}(v)$ ▷ v is now the parent of the newly inserted a-node
4: **while** $v \neq \bot$ and no height reducing rotation was done **do**
5: **if** ROTATIONPOSSIBLE(v) **then**
6: $w \leftarrow$ DOROTATION(v) ▷ w is the node from Table 3 after the rotation
7: $v \leftarrow w$
8: **else**
9: $v \leftarrow \begin{cases} \mathsf{Parent}(v) & \text{if } v = \mathsf{child_D}(\mathsf{Parent}(v)) \\ \bot & \text{otherwise} \end{cases}$
10: **if** $v \neq \bot$ **then** TRYREDUCEHEIGHT(Ψ, v)

Proof. The correctness follows from the facts that we apply the insertion operation as described in Section 2 in Line 2 and afterwards the only changes to the formula are rotations that are equivalence-preserving according to Lemma 3.5. The runtime follows from Lemma 3.21 and the fact that the runtime of the while-loop is bounded by the height of the formula. At any given node u, we can have at most one height preserving rotation, as afterwards the balance of u is positive (rotation 1a or 2a), or the node is a $\oplus$-node (rotation 3a).

It remains to show that the algorithm maintains the strong invariant. We assume that all nodes are green prior to the insertion. If there is no yellow node after the insertion of a new node v, the strong invariant is maintained and will not be invalidated by height preserving rotations (Lemma 3.7(e)). We note that there is always a node v' with $|\mathsf{balance}(v')| \leq 1$ at most 6 levels below v, as otherwise we already would have applied a height reducing rotation.

If the insertion creates yellow nodes, all of them have to be on $\mathsf{lp}^{-1}(\mathsf{Parent}(v))$ by Lemma 3.19. Let u be the lowest yellow node on $\mathsf{lp}^{-1}(\mathsf{Parent}(v))$. By Lemma 3.10, there have to be at least 8 nodes v' with $|\mathsf{balance}(v')| \geq 2$ at the start of $\mathsf{lp}(u)$. As the algorithm applies height preserving rotations to every $\odot$-node with balance less than minus one on its way up, it finally has to find and apply a height reducing rotation by Lemma 3.11. This reduces the height of Ψ_u and changes the color of all yellow nodes back to green, as they have the same height as before the insertion operation. However, it is possible that the node w returned by DOROTATION is a yellow node, as we cannot say much about its balance. However, all nodes except maybe w in Ψ are green. Therefore by Lemma 3.20 we can conclude that there are only green nodes in Ψ_w (and thus in Ψ) and the strong invariant is satisfied after the call to TRYREDUCEHEIGHT on w. □

3.5. **Deletions.** In this section we provide an algorithm for deletions and a slightly modified invariant that is maintained by this algorithm. We also show that the algorithm for insertion also maintains the deletion invariant for the case that insertions and deletions are interleaved. For a non-green node v, we depict by

$$\mathsf{deficit}(v) \quad = \quad 2^{\frac{\mathsf{height}(\Psi_v)}{10}} - ||\Psi_v||$$

Algorithm 4 Deletions

function REMOVE(formula: Ψ, leaf of Ψ: w)
  $\mathcal{B} \leftarrow \mathcal{B}(\Psi)$
  $v \leftarrow \mathsf{sibling}(w)$ ▷ v will be at the position of $\mathsf{Parent}(w)$ after the deletion
  $\mathsf{delete}(w)$
  while $v \neq \mathsf{root}(\Psi)$ **do**
    $v \leftarrow \mathsf{Parent}(v)$
    if height of Ψ_v did not decrease and $\mathcal{B} - \mathcal{B}(\Psi) < 20 \cdot \mathsf{height}(\Psi)$ **then**
      TRYREDUCEHEIGHT (Ψ, v) ▷ see Algorithm 2

the number of leaves that need to be inserted into Ψ_v without increasing the height of Ψ_v in order to make v green. Or the other way around $\mathsf{deficit}(v)$ denotes the number of leaves that have been removed from Ψ_v without decreasing the height since the node v was green.

Deletion Invariant: For every non green node v it holds that

$$\mathsf{deficit}(v) \quad \leq \quad \frac{\mathsf{height}(\Psi_v) \cdot ||\Psi_v|| - \mathcal{B}(\Psi_v)}{20 \cdot \mathsf{height}(\Psi_v)} \,.$$

The intention behind this invariant is that the more leaves are missing for a green node, the smaller $\mathcal{B}(\Psi_v)$ has to become. As $\mathcal{B}(\Psi_v)$ cannot become less than zero, we get an upper bound on $\mathsf{deficit}(v)$ that implies that every non green node has to be yellow.

Lemma 3.23. *Let Ψ be a formula satisfying the deletion invariant. Then all nodes in Ψ are green or yellow.*

Proof. Assume that v is a node that is neither green nor yellow. Then $||\Psi_v|| < 2^{\frac{\mathsf{height}(\Psi_v)-1}{10}}$. By the definition of $\mathsf{deficit}(v)$, we get

$$\mathsf{deficit}(v) \quad > \quad 2^{\frac{\mathsf{height}(\Psi_v)}{10}} - 2^{\frac{\mathsf{height}(\Psi_v)-1}{10}} \quad \geq \quad 0.05 \cdot 2^{\frac{\mathsf{height}(\Psi_v)}{10}} \quad \geq \quad 0.05 \cdot ||\Psi_v|| \,.$$

Now we apply the deletion invariant, multiplying both sides by 20:

$$||\Psi_v|| \quad < \quad \frac{\mathsf{height}(\Psi_v) \cdot ||\Psi_v|| - \mathcal{B}(\Psi_v)}{\mathsf{height}(\Psi_v)} \quad = \quad ||\Psi_v|| - \frac{\mathcal{B}(\Psi_v)}{\mathsf{height}(\Psi_v)}$$

We clearly see that $\mathcal{B}(\Psi_v)$ is less than zero, which is a contradiction, as $\mathcal{B}(\Psi_v)$ is defined to be a sum of positive values. □

We use Algorithm 4 to remove a node from Ψ. The algorithm first performs the actual deletion operation as described in Section 2. Afterwards we rebalance the formula as needed. The algorithm goes upwards starting at the parent of the removed node and checks at each ancestor v of w whether the height of Ψ_v did decrease. If the height did decrease, we are fine, as then the color of v has to be green, even if it was yellow before. Otherwise v might actually be yellow (the height did not decrease but we removed a leaf). Therefore, we try to reduce the height of Ψ_v by calling TRYREDUCEHEIGHT(Ψ, v). This function will decrease the height of Ψ_v if v is yellow and thus make sure that v is green afterwards. However, we abort the rebalancing process if the balance did improve by at least 20 times the height of Ψ. This abort is to ensure a logarithmic runtime, even if we have many calls to TRYREDUCEHEIGHT that individually have a logarithmic worst-case runtime. The if-condition can be checked by maintaining a global counter for the change of $\mathcal{B}(\Psi)$. The counter can be updated whenever a local node balance value is updated.

The following is then immediate from Lemma 3.21 that states that the runtime of TryReduceHeight is linear in the improvement of balancedness:

Corollary 3.24. *The runtime of Algorithm 4 is* $O(\mathsf{height}(\Psi))$.

In order to show that Algorithm 4 maintains the deletion invariant, we first show that the call to TryReduceHeight maintains this invariant.

Lemma 3.25. *Let v be a node in a formula Ψ such that no node execpt maybe v violates the deletion invariant in Ψ_v. Then the deletion invariant is satisfied in Ψ_v after the call* TryReduceHeight(Ψ, v).

Proof. We first show that v satisfies the deletion invariant after the call to TryReduceHeight. Afterwards we apply an inductive argument to show that no other node can violate the invariant. Let Ψ'_v be the formula Ψ_v after the call to TryReduceHeight. We write v' to denote the root of Ψ'_v.

If v satisfies the deletion invariant, then so does v', as the balancedness can only improve and the height can only decrease due to TryReduceHeight.

If v does not satisfy the invariant, then v is yellow and thus $\mathsf{balance}(v) \neq 0$ by Lemma 3.10. Let w be the deeper child of v and u be the sibling of w. As v is yellow, we conclude from Lemma 3.20 that $\mathsf{height}(\Psi'_v) = \mathsf{height}(\Psi_v) - 1 = \mathsf{height}(\Psi_w)$. We bound $\mathsf{deficit}(v')$ as follows, where we use the fact that $\mathsf{height}(\Psi'_v) = \mathsf{height}(\Psi_w)$ several times.

$$\begin{aligned}
\mathsf{deficit}(v') &= 2^{\frac{\mathsf{height}(\Psi'_v)}{10}} - ||v'|| && \text{(definition of } \mathsf{deficit}\text{)} \\
&= \underbrace{2^{\frac{\mathsf{height}(\Psi'_v)}{10}} - ||w||}_{\mathsf{deficit}(w)} - ||u|| && (||v'|| = ||v|| = ||w|| + ||u||) \\
&\leq \frac{\mathsf{height}(\Psi'_v) \cdot ||\Psi_w|| - \mathcal{B}(\Psi_w)}{20 \cdot \mathsf{height}(\Psi_w)} - ||u|| && (w \text{ satisfies invariant}) \\
&\leq \frac{\mathsf{height}(\Psi'_v) \cdot ||\Psi_w|| - \mathcal{B}(\Psi_v) + \mathcal{B}(\Psi_u) + |\mathsf{balance}(v)|}{20 \cdot \mathsf{height}(\Psi'_v)} - ||u|| \\
&\leq \frac{\mathsf{height}(\Psi'_v) \cdot ||\Psi_w|| - \mathcal{B}(\Psi_v)}{20\mathsf{height}(\Psi'_v)}
\end{aligned}$$

In the last but one inequality, we exploit that $\mathcal{B}(v) = \mathcal{B}(w) + \mathcal{B}(u) + |\mathsf{balance}(v)|$ and in the last inequality, we exploit that Ψ_u has exactly $||u|| - 1$ inner nodes whose absolute balances are bounded by the height. We can conclude that v' satisfies the invariant because $||\Psi'_v|| > ||\Psi_w||$ and $\mathcal{B}(\Psi'_v) \leq \mathcal{B}(v)$.

It remains to show that no other nodes violate the invariant. This part of the proof is an induction over the recursion depth, analogous to the one in the proof of Lemma 3.20. If there is no rotation, then there can be no other node that violates the invariant by the condition in the lemma statement.

If there is a simple rotation in Line 4, the only node except v which can violate the invariant is the node w as denoted in Table 3. The induction hypotheses yields that w satisfies the invariant after the recursive call.

It remains to discuss double rotations in Line 9. We discuss the two rotations of the double rotations seperately, starting with the height preserving rotation, i.e., the first rotation of the double rotation. By Lemma 3.7(e), we have that w is green. We remind that u is at most 6 nodes above v and $|\mathsf{balance}(v)| \leq 1$. The argumentation for the second rotation of the double rotation is exactly as for simple rotations in Line 4, as we also do a recursive call on w. This concludes the proof. □

Now we show that Algorithm 4 maintains the deletion invariant:

Lemma 3.26. *The deletion invariant is maintained by Algorithm 4.*

Proof. We have to show that the deletion invariant is maintained for all nodes. It is maintained for all nodes u of Ψ that are not an ancestor of v, as the algorithm does not modify Ψ_u. It remains to show that the invariant is maintained for ancestors of w and for nodes w' that result from some rotation.

We now show the following property for every ancestor v of w which will directly yield that the invariant has been maintained for v:
Either v is green after the update, or

- the height of Ψ_v stayed the same;
- the number of leaves of Ψ_v decreased by exactly one; and
- $\mathcal{B}(\Psi_v)$ improved by at least $20 \cdot \mathsf{height}(\Psi_v)$.

The height of Ψ_v cannot increase. If the height of Ψ_v decreases, then v would be green afterwards. Thus if v is still yellow, the height of Ψ_v has stayed the same. The number of leaves of Ψ_v has decreased by one due to the deletion operation and cannot change afterwards. And from the fact that the height did not decrease despite v being yellow, we can conclude that TryReduceHeight was never called for v. As the if-condition thus needs to be false, we can conclude that $\mathcal{B}(\Psi)$ has been improved by at least $20 \cdot \mathsf{height}(\Psi)$. We note that $\mathsf{height}(\Psi_v) \leq \mathsf{height}(\Psi)$ and that the improvement of $\mathcal{B}(\Psi_v)$ equals the improvement of $\mathcal{B}(\Psi)$, as all balance changes are internal to Ψ_v in the case that Ψ_v does not change its height. We can conclude that $\mathcal{B}(\Psi_v)$ improved by at least $20 \cdot \mathsf{height}(\Psi_v)$

It remains to observe that Lemma 3.25 ensures that the deletion invariant is satisfied by all nodes w' that result from some rotation performed by TryReduceHeight. This concludes the proof. □

It remains to show that also the insertion algorithm maintains the deletion invariant. Of course, we already have shown that it even maintains a stricter invariant. However, if we arbitrarily mix insertions and deletions, we can no longer guarantee that the strong invariant is satisfied before a call to the insertion algorithm.

Lemma 3.27. *Algorithm 3 maintains the deletion invariant.*

Proof. We first observe, that an insertion into Ψ_v, where v is a yellow node, increases the number of nodes in Ψ_v by one, but cannot change the height of Ψ_v. If the insertion is not on $\mathsf{lp}(v)$, the height cannot increase. Otherwise, if the insertion is on $\mathsf{lp}(v)$, the insertion algorithm will perform a height decreasing rotation somewhere on $\mathsf{lp}(v)$ because v is yellow. We already showed this in the proof of Lemma 3.22.

As the deletion invariant did hold before the insertion, we know that before the insertion we have

$$\mathsf{deficit}(v) \quad \leq \quad \frac{\mathsf{height}(\Psi_v) \cdot ||\Psi_v|| - \mathcal{B}(\Psi_v)}{20 \cdot \mathsf{height}(\Psi_v)} .$$

After the insertion we have that $||\Psi_v||$ is increased by 1 and $\mathcal{B}(\Psi_v)$ is at most worsened by $\mathsf{height}(\Psi_v)$, as the insertion can only worsen $|\mathsf{balance}(w)|$ by at most one for each ancestor w of the inserted node. We remind that all rotations performed by the insertion algorithm can only improve $\mathcal{B}(\Psi_v)$. Altogether, the equation of the invariant still holds true, as $\mathsf{deficit}(v)$

shrinks by one while the right side of the equation in the invariant cannot shrink. The value $\mathcal{B}(\Psi_v)$ can be worsened by at most $\mathsf{height}(\Psi_v)$, which is absorbed by the increment of $||\Psi_v||$. □

Wrap-Up. Altogether we have shown that given a tree T, we can compute a formula Ψ that represents T in linear time. This formula will be of logarithmic height. Especially we have shown that this formula satisfies the deletion invariant and that after any update to T, we can update Ψ in logarithmic time to match the new tree in such a way that the deletion invariant is maintained. Theorem 3.1 follows by the combination of Theorem 3.12, Lemma 3.22, Corollary 3.24, Lemma 3.26, and Lemma 3.27.

4. Stepwise Automata and Transition Algebras

In this section we introduce the tree automata that we use to represent MSO queries. Furthermore, we also introduce their transition algebras which are forest algebras that capture the whole behavior of a tree automaton on some forest or context. Transition algebras are related to tree automata in the same way as transition monoids are related to string automata.

Stepwise Tree Automata. Stepwise tree automata were first described in [CNT04] using a curry encoding of unranked trees. For convenience, we use the definition from [MN07] that directly works on unranked trees.

A *stepwise nondeterministic tree automaton* or *NFTA* is a tuple $N = (Q, \Sigma, \delta, \mathsf{Init}, q_I, q_F)$ where Q is the finite set of states, Σ is a finite alphabet, q_I and q_F are the global initial and final state, $\mathsf{Init}\colon \Sigma \to 2^Q$ assigns a local set of initial states to every symbol of Σ, and $\delta \subseteq Q \times Q \times Q$ is a transition relation. We use $q_1 \xrightarrow[q_2]{} q_3$ to denote a transition that goes from q_1 to q_3 when reading state q_2.

Intuitively, a stepwise tree automaton computes a run bottom-up. After assigning states $q_1, \dots, q_n$ to the n children of some node v, it assigns a state to v, by starting in some initial state (determined by the label of v) and reading the string $q_1 \dots q_n$, i.e, it reads the states of all children. The resulting state is assigned to v. Whether a run is accepting is determined by the state assigned to the root.

This informal description of runs of a stepwise tree automaton actually describes two different types of runs: a vertical run that assigns states to nodes, and, for each node, a horizontal run along the children of a node. The horizontal run corresponds exactly to the run of a string automaton over the input alphabet Q. We place the state q_2 under the arrow in the notation $q_1 \xrightarrow[q_2]{} q_3$ because the states q_1 and q_3 are part of the horizontal run of some node v while the state q_2 is the state of a child of v in the vertical run. For our formal description, we combine the vertical run and all horizontal run into a single run as follows.

Formally, a *run* λ of N on a labeled tree T is an assignment of transitions to states $\lambda\colon \mathrm{Nodes}(T) \to \delta$, where we write $\lambda_{\mathsf{pre}}(v)$, $\lambda_{\mathsf{self}}(v)$, and $\lambda_{\mathsf{post}}(v)$ to denote the individual components of the transition $\lambda(v) = q_{\mathsf{pre}} \xrightarrow[q_{\mathsf{self}}]{} q_{\mathsf{post}}$. A run has to satisfy the following conditions for every node v:

$$\begin{array}{lcll} \lambda_{\mathsf{pre}}(v) & = & q_I & \text{if } v \text{ is the root} \\ \lambda_{\mathsf{pre}}(v) & \in & \mathsf{Init}(\mathsf{lab}(\mathsf{Parent}(v))) & \text{if } v \text{ has no left sibling} \end{array}$$

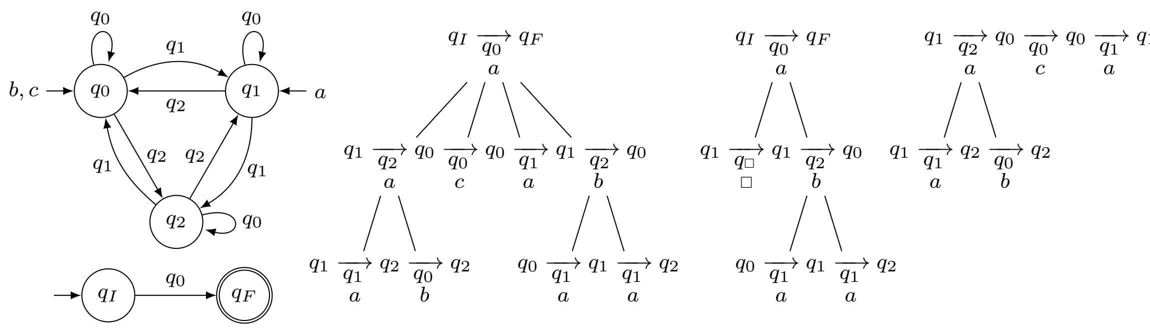


FIGURE 6. NFTA (left), tree with accepting run (middle), context and forest with runs (right)

$$
\begin{aligned}
\lambda_{\mathsf{pre}}(v) &= \lambda_{\mathsf{post}}(w) && \text{if } w \text{ is the left sibling of } v\\
\lambda_{\mathsf{self}}(v) &\in \mathsf{Init}(\mathsf{lab}(v)) && \text{if } v \text{ is a leaf}\\
\lambda_{\mathsf{self}}(v) &= \lambda_{\mathsf{post}}(w) && \text{if } w \text{ is the last child of } v
\end{aligned}
$$

We note that our depiction of a run is equivalent to the literature depiction of horizontal and vertical runs. The usual depiction of a vertical run corresponds to λ_{self}, while the list of transitions assigned to the children of a node v corresponds to the list of transitions of the horizontal run as depicted in the literature.

The rules above establish that $\lambda_{\mathsf{self}}(v)$ is computed, just as a (nondeterministic) string automaton computes the final state when starting in some initial state from $\mathsf{Init}(\mathsf{lab}(v))$ and reading the string of states $\lambda_{\mathsf{self}}(v_1)\cdots\lambda_{\mathsf{self}}(v_n)$ of the children $v_1,\ldots,v_n$ of v.

A run is *accepting* if $\lambda(r) = q_F$, where r is the root of T. A tree T is *accepted* if there exists an accepting run on T. The set of all accepted trees is denoted by $L(N)$. By states(λ) we denote the image of λ_{self}.

We note that our notation of acceptance is equivalent to the usual notation where a tree is accepted if $\lambda_{\mathsf{self}}(\mathsf{root}(T)) \in F$ for some set of final states F. One simply has to add the transitions $\{q_I \xrightarrow[q]{} q_F \mid q \in F\}$ to δ. We prefer our acceptance model, as it will simplify the definition of transition algebras a lot. Also this mode of acceptance is much closer to the model of string automata and can be easily generalized to forest languages.

Example 4.1. In Figure 6, we depict an NFTA that checks whether the number of a-nodes in the tree is equivalent to 0 modulo 3. The states q_0, q_1, and q_2 encode the number of a-nodes modulo 3. The initial state for a-nodes is q_1 to count the a-node itself even before starting the horizontal run. The initial states for all other symbols is q_0. Transitions between states encode modulo arithmetic and the transition $q_I \xrightarrow[q_0]{} q_F$ encodes the acceptance condition. We also depict a tree T together with an accepting run λ. We explain the horizontal run of the root node. The run starts in q_1 to count the a-label of the root. It then adds 2 more a-nodes from the first subtree, 0 a-nodes for the c-node, the a-node, and finally again 2 a-nodes from the last subtree. It reaches state q_0 as there are 6 a-nodes in total. On the right, we depict a decomposition $T = C \odot_{\vartriangle} F$ of T into a context C and a forest F together with a decomposition of λ into (partial) runs for C and F as defined below.

We want to be able to compose runs, just as we compose trees from forests and contexts. Therefore, we define runs on forests and contexts as follows: A run of N on a forest F is

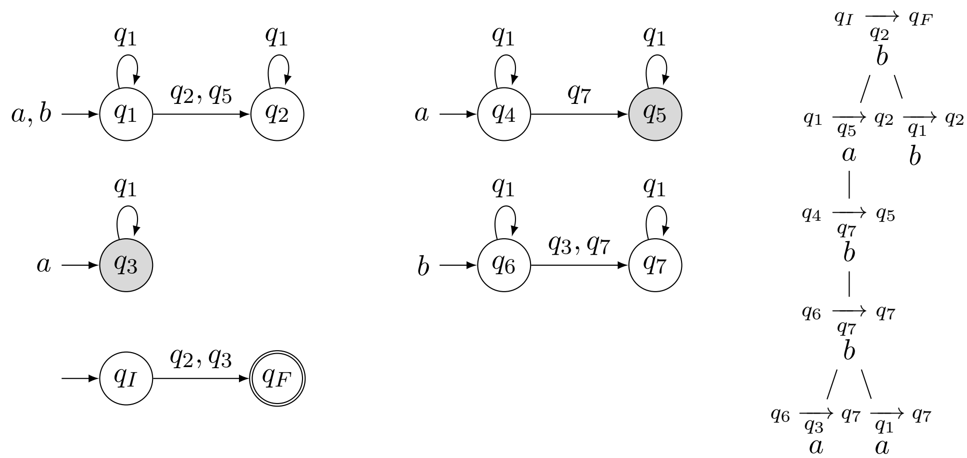


FIGURE 7. 2-NFSTA M with $S = \{(q_3, q_5)\}$ (left) and tree with accepting run λ (right). The transition assigned to some node by λ is denoted above its label.

defined just as a run of N on a tree, with the following difference: $\lambda_{\mathsf{pre}}(v)$ is allowed to be any state, where v is the leftmost root, i.e. the unique root of F that does not have a left sibling. This modification is necessary, as there is no way of knowing how the context where F will be inserted in looks like.

To define runs on contexts, we add a new state $q_\Box$, define $\mathsf{Init}(\Box) = q_\Box$ and add transitions $(Q \setminus \{q_\Box\}) \times \{q_\Box\} \times (Q \setminus \{q_\Box\})$ to δ. Now runs on contexts are defined just like runs on forests using the extended automaton. This has the effect that a run on a context does not make any assumptions on what happens inside the forest that will be inserted into the hole to form a tree or forest.[2] We note that accepting runs are only defined for trees, not for forests or contexts.

Selecting Automata. In order to evaluate non-Boolean queries, we utilize (node- and tuple-)selecting finite tree automata (see, e.g., [FGK03, Nev99]) as formalism for queries. It is well-known that these can express MSO queries with free node variables over unranked trees [NPTT05, Theorem 7].

For $k \in \mathbb{N}$, a *k-ary non-deterministic finite selecting tree automaton (k-NFSTA) M* is a pair (N, S), where N is a NFTA over Σ with states Q and $S \subseteq Q^k$ is a set of *selecting tuples*. The *size* of M is defined as $|Q| + |S|$. When M reads a tree T, it computes a set of tuples in $\text{Nodes}(T)^k$. More precisely, we define

$$M(T) = \{ \ (v_1, \ldots, v_k) \ | \ \text{there is an accepting run } \lambda \text{ of } N \text{ on } T \text{ and a tuple } (p_1, \ldots, p_k) \in S \text{ such that } \lambda_{\mathsf{self}}(v_\ell) = p_\ell \text{ for } \ell \in \{1, \ldots, k\} \ \} .$$

Notice that, if $T \notin L(N)$ then $M(T) = \emptyset$.

[2] Of course, we could restrict the additional transitions to only those that are possible by some forest, but it does not make any difference for asymptotic worst case complexity and therefore we stick with this simpler construction.

Example 4.2. Figure 7 illustrates a 2-NFSTA M over $\Sigma = \{a, b\}$ that outputs each pair of a-labeled nodes that are connected by a path of b-labeled nodes. The states have the intended meaning as follows: q_1 and q_2 are states not belonging to the path, where q_1 is for nodes that do not have the path below them and q_2 is for nodes that have the path below them. Therefore, every state (except q_I, q_F) has a q_1-loop to ignore parts of the tree that are unrelated to the path. The states q_3 and q_5 mark the lower end upper end of the path, respectively, while q_7 marks b-nodes on the path. The states q_4 and q_6 cannot appear as states of any node, as otherwise the run cannot be completed to an accepting run. They are guesses as initial states for the upper a-node of the path and b-nodes on the path. However, for an accepting run, there has to be a child with state q_3 or q_7 below. The only selecting tuple is (q_3, q_5). We accept if the root has state q_2 or q_3, thus either it is some node with the path below it, or the top a-node of the path.

Next to the automaton we depict a tree with some accepting run that returns the a-node below the root and the left a-leaf. A symmetric run returns a pair of nodes with the other a-leaf.

Signatures of Runs. The *signature* of a run over a forest or context captures the states before the first and after the last root. Furthermore, in case of a context, the signature additionally captures the states before and after the hole. Let $v_1 \leq \cdots \leq v_k$ be the list of roots. Formally, we define the signature of a run λ as follows:

$$\mathrm{SIG}(\lambda) \quad = \quad \begin{cases} \big(\lambda_{\mathsf{pre}}(v_1), \lambda_{\mathsf{post}}(v_k)\big) & \text{if } \lambda \text{ is over a forest} \\ \Big(\big(\lambda_{\mathsf{pre}}(v_1), \lambda_{\mathsf{post}}(v_k)\big), \big(\lambda_{\mathsf{pre}}(\Box), \lambda_{\mathsf{post}}(\Box)\big)\Big) & \text{if } \lambda \text{ is over a context} \end{cases}$$

We denote the set of all possible signatures over the states Q with $\mathrm{SIG}_Q = Q^2 \cup (Q^2)^2$.

Transition Algebra. The *transition algebra* of a stepwise tree automaton is the generalization of the transition monoid of a finite string automaton. Indeed the horizontal monoid is defined exactly like the transition monoid of a finite string automaton over the alphabet Q. Each element of the transition algebra captures the signatures of all possible runs over the underlying forest or context.

Formally, the *transition algebra* of a given stepwise tree automaton $N = (\Sigma, Q, \delta, \mathsf{Init}, F)$ is defined as

$$\mathcal{F}_N = (H, V, \oplus_{HH}, \oplus_{HV}, \oplus_{VH}, \odot_{VV}, \odot_{VH}, \mathrm{id}_Q, \mathrm{id}_{Q^2}) \ ,$$

where $H = 2^{Q^2}$ and $V = 2^{(Q^2)^2}$ are the powersets of the possible signatures of forests and contexts, respectively. The neutral elements id_Q and id_{Q^2} are the identity relations over Q and Q^2, respectively. That is $\mathrm{id}_Q = \{(q, q) \mid q \in Q\}$ and $\mathrm{id}_{Q^2} = \{((q_1, q_2), (q_1, q_2)) \mid (q_1, q_2) \in Q^2\}$. All operations are relational joins over those states that need to be identical in order for the runs to be combined followed by a projection onto the states relevant for the signature of the combined run. The intuition of the operations are depicted in Figure 8, where we depict how signatures can be combined by the five operations. While in the figure we sketch the operations for a pair of signatures, in the algebra we have to join all compatible pairs of signatures. Formally, the operations are given by the following equations, where $q_1, \ldots, q_6$ are states of the automaton:

$$F_1 \oplus_{HH} F_2 \quad = \quad \big\{(q_1, q_3) \mid (q_1, q_2) \in F_1, (q_2, q_3) \in F_2\big\}$$

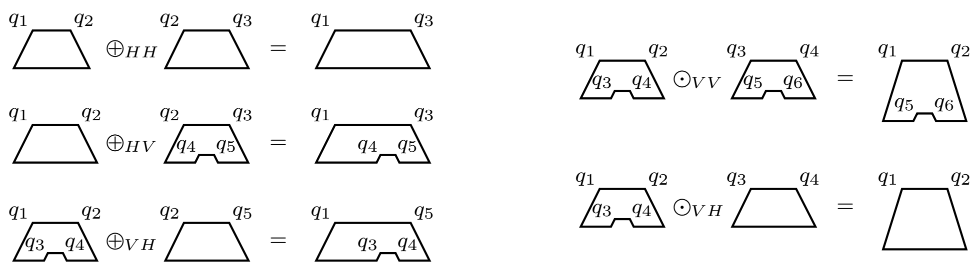


Figure 8. Graphical representation of the operations of the transition algebra. Trapezoids represent forests while trapezoids with a cutout trapezoid represent contexts.

$$
\begin{aligned}
C_1 \odot_{VV} C_2 &= \big\{\big((q_1,q_2),(q_5,q_6)\big) \mid \big((q_1,q_2),(q_3,q_4)\big) \in C_1, \big((q_3,q_4),(q_5,q_6)\big) \in C_2\big\} \\
C \odot_{VH} F &= \big\{(q_1,q_2) \mid \big((q_1,q_2),(q_3,q_4)\big) \in C, (q_3,q_4) \in F\big\} \\
F \oplus_{HV} C &= \big\{\big((q_1,q_3),(q_4,q_5)\big) \mid (q_1,q_2) \in F, ((q_2,q_3),(q_4,q_5)) \in C\big\} \\
C \oplus_{VH} F &= \big\{\big((q_1,q_5),(q_3,q_4)\big) \mid \big((q_1,q_2),(q_3,q_4)\big) \in C, (q_2,q_5) \in F\big\}
\end{aligned}
$$

Lemma 4.3. *For every NFTA N, the corresponding transition algebra $\mathcal{F}_N$ is a forest algebra.*

Proof sketch. We have to show that all axioms from Table 2 hold. The axioms A1–A4 (neutral elements) hold, as we do joins with identity relations. All other axioms hold because of the associativity of the relational join. □

As all operations are defined by the means of the relational join, we have the following upper bound on their complexity using the trivial join algorithm.

Observation 4.4. Given a transition algebra $\mathcal{F}$ over a set of states Q, all operations in $\mathcal{F}$ can be performed in at most $\mathcal{O}(|Q|^6)$ time.

The morphism from the free forest algebra $\mathcal{F}_\Sigma$ over Σ to the transition algebra $\mathcal{F}_N$ of N is defined by

$$h(a_\Box) \;=\; \big\{\mathrm{SIG}(\lambda) \mid \lambda \text{ is a valid run of } N \text{ on } a_\Box\big\}$$

for all symbols $a \in \Sigma$. While we define h by specifying the mappings for the atomic contexts, the intuition is that the homomorphism maps every forest or context to the set of signatures for all possible runs. This intuition is formalized in the following lemma.

Lemma 4.5. *For every forest or context D it holds that*

$$h(D) \;=\; \big\{\mathrm{SIG}(\lambda) \mid \lambda \textit{ is a valid run of } N \textit{ on } D\big\}\,.$$

Proof. We show both directions by induction. We start with the if-direction. The statement holds for $D = a_\Box$ by definition. For $D = a$, let λ be some run of N over D and $(q_1, q_2) = \mathrm{SIG}(\lambda)$. By definition of signatures and runs, there has to be some state $q_3 \in \mathsf{Init}(a)$, such that $q_1 \xrightarrow[q_3]{} q_2 = \lambda(v)$. As $q_1 \xrightarrow[q_3]{} q_2$ and $q_3 \xrightarrow[q_\Box]{} q_3$ are both transitions in δ, we can conclude that $\big((q_1,q_2),(q_3,q_3)\big) \in h(a_\Box)$ and thus $(q_1,q_2) \in h(a) = h(a_\Box \odot_{VH} \varepsilon)$.

We showcase the induction step for the operation $\odot_{VH}$, i.e., for $D = C \odot_{VH} F$. We have to show that for every run λ on $C \odot_\Delta F$ with signature (q_1, q_2), it holds that $(q_1, q_2) \in h(C) \odot_{VH} h(F)$. By definition of $\odot_{VH}$, this boils down to showing that there are states q_3 and q_4 such that $\big((q_1, q_2), (q_3, q_4)\big) \in h(C)$ and $(q_3, q_4) \in h(F)$. By the induction assumption, this is the case if there are runs λ_C on C and λ_F on F with signatures $\mathrm{SIG}(\lambda_C) = \big((q_1, q_2), (q_3, q_4)\big)$ and $\mathrm{SIG}(\lambda_F) = (q_3, q_4)$. We now show that there are such runs λ_F and λ_C. We get λ_F by restricting λ to the nodes in F. We define $q_3 = \lambda_{\mathsf{pre}}(v_1)$ and $q_4 = \lambda_{\mathsf{post}}(v_k)$, where $v_1, \ldots, v_k$ are the roots of F. By definitions of signatures we have that $\mathrm{SIG}(\lambda_F) = (q_3, q_4)$ as desired. Likewise, we get λ_C by restricting λ to the nodes of C and using the transition $q_3 \xrightarrow[q_\square]{} q_4$ for the hole. The signature of λ_C is indeed $\big((q_1, q_2), (q_3, q_4)\big)$ as desired. The induction step for the other four operations works analogously, where the existing run is split into two runs as indicated in Figure 8. This concludes the proof of the if-direction.

We continue with the only-if-direction. The case $D = a_\square$ is again trivial, we continue with $D = a$. From $(q_1, q_2) \in h(a)$, we can conclude that there has to be a state q_3 such that $((q_1, q_2), (q_3, q_3)) \in h(a_\square)$, as $h(a) = h(a_\square) \odot_{VH} \mathrm{id}_Q$. Therefore $q_3 \in \mathsf{Init}(a)$. We can conclude that the run λ that assigns $q_1 \xrightarrow[q_3]{} q_2$ to the a-node is a valid run with $\mathrm{SIG}(\lambda) = (q_1, q_2)$.

Again, we showcase the induction step for $D = C \odot_{VH} F$. We have to show that for every tuple $(q_1, q_2) \in h(C \odot_{VH} F)$ there is a run λ on $C \odot_{VH} F$ such that $\mathrm{SIG}(\lambda) = (q_1, q_2)$. By definition of $\odot_{VH}$, there have to exist states q_3 and q_4 such that $\big((q_1, q_2), (q_3, q_4)\big) \in h(C)$ and $(q_3, q_4) \in h(F)$. By the induction assumption, there are runs λ_C on C with signature $\big((q_1, q_2), (q_3, q_4)\big)$ and λ_F on F with signature (q_3, q_4). By the definition of signatures, the run λ_C uses the transition $q_3 \xrightarrow[q_\square]{} q_4$ for the hole. The run λ is then derived from λ_C and λ_F by taking the disjoint union $\lambda_C \cup \lambda_F$ of both runs and omitting the mapping for the hole. It is easy to verify that the result is a valid run for $C \odot_{VH} F$. Again, the induction step for the other operations works analogously, where runs can be combined as indicated in Figure 8. This concludes the proof. □

Evaluation and Enumeration Problems for NFTAs. Let M be a selecting automaton, T the input tree for M, and $M(T)$ be the answer of M on T. We are interested in efficiently maintaining $M(T)$ under updates of T. This means that we can have an update u to T, yielding another tree T', and we wish to efficiently compute $M(T')$. The latter cost should be more efficient than computing $M(T')$ from scratch.

We allow a single preprocessing phase in which we can compute an *auxiliary data structure* $\mathrm{Aux}(T)$ that we can use for efficient query answering. When T is updated to T', we therefore want to efficiently compute $M(T')$ and efficiently update $\mathrm{Aux}(T)$ to $\mathrm{Aux}(T')$.

If M is simply an NFTA (i.e., a 0-ary NFSTA), then this problem is known as *incremental evaluation* and was studied by, e.g., [BPV04]. Here, we perform *incremental enumeration*, meaning that we extend the setting of Balmin et al. from 0-ary queries to k-ary queries. We measure the complexity of our algorithms in terms of the following parameters: *(i)* size of $\mathrm{Aux}(T)$, *(ii)* time needed to compute $\mathrm{Aux}(T)$, *(iii)* time needed to update $\mathrm{Aux}(T)$ to $\mathrm{Aux}(T')$, and *(iv)* time delay we can guarantee between answers of $M(T')$. The underlying model of computation is a random access machine (RAM) with uniform cost measure.

In the remainder we use IncEval and IncEnum to refer to the incremental evaluation and enumeration problems, respectively.

5. Incremental Evaluation

In this section, we use the forest algebra framework of Section 3 to prove the following theorem:

Theorem 5.1. IncEval *for an NFTA $N = (\Sigma, Q, \delta, \mathit{Init}, F)$ and a tree T can be solved with a preprocessing phase of time $\mathcal{O}(|Q|^6 \cdot |T|)$, auxiliary structure of size $\mathcal{O}(|Q|^4 \log(|T|))$, and with update time $\mathcal{O}(|Q|^6 \log(|T|))$ after each new update.*

Proof. We use the algorithms of Section 3 to compute and maintain a balanced representation Ψ of the tree T. For each node v of the parse tree of Ψ, we store the element $h(\Psi_v)$ of the transition algebra of N that corresponds to the forest or context represented by Ψ_v. By Lemma 4.5, the evaluation problem can be solved by looking whether (q_I, q_F) is contained in the transition algebra element represented at the root of the parse tree.

As a last step in the preprocessing, we can compute the algebra elements that we store at the nodes of Ψ in a bottom-up way, computing one algebra operation at each inner node of Ψ. For the updates, we observe that after the update itself, we only need to update the algebra elements of the (logarithmically many) ancestors of the inserted/removed/relabeled node. In the case of deleting the last child of some node v, we also have to recompute the logarithmically many algebra elements of the ancestors of v in the formula.

Furthermore, for each performed rotation, we need to compute one algebra operation to compute the element represented at the node w after the rotation (see Table 3 for the definition of w). As the rotations are equivalence-preserving (Lemma 3.5), the elements for all other nodes stay the same. Especially the element represented at v does not change.

We can therefore conclude all the run times from the lemma statements using Observation 4.4 and Theorem 3.1, as the number of rotations is clearly bounded by the runtime of the update algorithm. □

6. Enumerating MSO Queries

This section extends the incremental evaluation result from the previous section to enumeration of non-Boolean MSO queries. That is, we prove the following theorem:

Theorem 6.1. IncEnum *for a k-NFSTA M and a tree T can be solved with auxiliary data of size $\mathcal{O}(|Q^4| \cdot |S| \cdot 2^k \cdot |T|)$ which can be computed in time $\mathcal{O}(|Q^6| \cdot |S| \cdot 2^k \cdot |T|)$, maintained within time $\mathcal{O}(|Q^6| \cdot |S| \cdot 2^k \log(|T|))$ per update, and which guarantees delay $\mathcal{O}(|Q^6| \cdot k \cdot |S| \cdot 2^k \cdot \log(|T|))$ between answers.*

Towards the proof, we first give a high level algorithm for the enumeration in Section 6.1. In Section 6.2, we finish the proof by providing an implementation for the core method used in the algorithm and proving its runtime.

6.1. High Level Algorithm.

The high level presentation of our enumeration algorithm is depicted as Algorithm 5. We assume a total order $\leq_T$ on the nodes of T that can depend on our auxiliary data structure. This order is independent of the sibling order of T. The algorithm then enumerates all answers in lexicographic order according to $\leq_T$. To avoid some case distinctions, we use a symbol $\perp$ such that $\perp \leq_T v$ for any node v.

To understand the algorithm, we need the notion of an incomplete answer: We call a tuple $A \in \text{Nodes}(T)^\ell$ with $\ell \leq k$ an *incomplete answer* if it is a prefix of some answer

Algorithm 5 Enumeration of $M(T)$

Input: k-NFSTA $M = ((Q, \Sigma, \delta, F), S)$, tree T, incomplete answer A
Output: Enumeration of all answers in $M(T)$ that are compatible with A

1: **function** Enum(M, T, A)
2:     **if** $|A| = k$ **then** Output(A)
3:     **else**
4:         $A' \leftarrow$ Complete($A, \bot$)
5:         **while** $A' \neq \bot$ **do**
6:             Enum(M, T, A')
7:             $v \leftarrow A'_{|A'|}$
8:             $A' \leftarrow$ Complete(A, v)

$B \in M(T)$. We assume that the empty tuple () is always an incomplete answer, i.e., even if $M(T) = \emptyset$ to avoid some corner cases. We write $A \preceq B$ for two (in-)complete answers A and B, if A is a prefix of B. By $|A| := \ell$ we denote the number of nodes of the incomplete answer A.

To enumerate all answers, Enum has to be called with the empty incomplete answer (). The sub-procedure Complete extends a given incomplete answer A with another node according to the following definition.

Definition 6.2. Let $A = (v_1, v_2, \dots, v_j)$ be an incomplete answer, then

$$\text{Complete}(A, u) := (v_1, v_2, \dots, v_j, v) ,$$

where v is the smallest node such that $u <_T v$ and $(v_1, v_2, \dots, v_j, v)$ is an incomplete answer. If no such node exists, then we define $\text{Complete}(A, u) := \bot$.

By definition of Complete, the lines 4 to 8 iterate over all incomplete answers A' that result from A by adding one additional node. Before we show how to efficiently implement Complete, we prove correctness of Algorithm 5.

Lemma 6.3. Enum($M, T, ()$) *enumerates all answers in* $M(T)$.

Proof. We show that for every incomplete answer A, the function call Enum(M, T, A) outputs exactly the answers B such that A is a prefix of B. The lemma statement follows, as the empty answer () is a prefix of every answer.

We first observe, that every output of the algorithm is clearly a valid answer in $M(T)$. It remains to show that every output is an answer and that no answer is output twice.

The proof is by induction over $|A|$. The base case is $|A| = k$. In this case, the only compatible answer is A, which is output in Line 2 of the algorithm. Let now $A = (v_1, \dots, v_\ell)$ be an incomplete answer and $B = (v_1, \dots, v_\ell, v_{\ell+1}, \dots, v_k)$ be some answer compatible with A. Eventually some call to Complete in Line 4 or 8 will return the incomplete answer $(v_1, \dots, v_\ell, v_{\ell+1})$. By the induction hypotheses, the recursive call in Line 6 will output B. Also no answer can be output twice, as the incomplete answers that are processed in the while loop are strictly increasing according to the lexicographic order induced by $\leq_T$. □

6.2. **Implementation of Complete.** We now present the implementation of COMPLETE. Our auxiliary data structure is a balanced forest algebra formula Ψ that represents the tree T. We use the *extended transition algebra* that we define below. To allow logarithmic delay, we need to know whether some node that can extend the incomplete answer is contained in some subformula. Technically, we could extend the signatures by the set of states that is visited in the run of the automaton. However, this would be quite inefficient, as we are only interested in states that occur together in the same selecting tuple.

Let $M = (N, S)$ be a k-NFSTA and for each selecting tuple $s = (q_1^s, \ldots, q_k^s)$ in S let $Q_s = \{q_1^s, \ldots, q_k^s\}$ be the set of states that occur in S. We define the *extended transition algebra* $\mathcal{F}_{M,s}^+$ for some selecting tuple $s \in S$ as follows:

$$\begin{aligned} \mathcal{F}_{M,s}^+ &= (H^+, V^+, \oplus_{HV}^+, \oplus_{VH}^+, \odot_{VH}^+) \\ H^+ &= (2^{Q^2 \times 2^{Q_s}}, \oplus_{HH}^+, \mathrm{id}_Q \times \{\emptyset\}) \\ V^+ &= (2^{(Q^2)^2 \times 2^{Q_s}}, \odot_{VV}^+, \mathrm{id}_{Q^2} \times \{\emptyset\}) \end{aligned}$$

We recall, that an element of the transition algebra is a set of possible signatures of runs. An element of the extended transition algebra is a set of *extended signatures*. An extended signature consists of a regular signature—a pair of states in the horizontal monoid or a pair of pairs of states in the vertical monoid—and a set $Q' \subseteq Q_s$ of those states from Q_s that are visited by the run. This is used in the enumeration algorithm to evaluate whether some position of the selecting tuple can be bound to a node inside the forest or context.

Formally, we call a tuple (x, r) from $\mathrm{SIG}_Q \times 2^{Q_s}$ an extended signature and use the syntax $\mathrm{SIG}^+(\lambda) = (x, r)$. The operations of the extended transition algebra are defined by:

$$d_1 \bigcirc^+ d_2 = \big\{(x, r) \in \mathrm{SIG}_Q \times 2^{Q_s} \;\mid\; \exists (y_1, r_1) \in d_1, (y_2, r_2) \in d_2. \\ x \in \{y_1\} \bigcirc \{y_2\} \text{ and } r = r_1 \cup r_2\big\}\ ,$$

where $\bigcirc^+$ is some operation of the algebra and $\bigcirc$ refers to the corresponding operation in the transition algebra $\mathcal{F}_N$ defined in Section 4. We define an homomorphism h^+ from the free forest algebra over Σ to $\mathcal{F}_{M,s}^+$ by

$$h^+(a_\square) \;=\; \big\{\mathrm{SIG}^+(\lambda) \;\mid\; \lambda \text{ is a valid run on } a_\square\big\}\ .$$

Again, the intuition is that the homomorphism maps all forests and contexts to the set of extended signatures of their possible runs. We omit a proof, as it is very similar to the proof of Lemma 4.5 and we do not need the result.

We like to stress that in the case $k = 0$, the extended transition algebra $\mathcal{F}_M^+$ is isomorphic to the transition algebra $\mathcal{F}_N$, as $2^\emptyset = \{\emptyset\}$. This reflects that a stepwise tree automaton is just the special case of a node selecting stepwise tree automaton with a single empty selecting tuple.

We note that the horizontal monoid H^+ works exactly, as illustrated by Losemann and Martens [LM14] in the word case if there is only one selecting tuple. Especially, our definition of $\oplus_{HH}^+$ is equivalent to the definition of $\bowtie$ in [LM14].

Observation 6.4. Given a k-NFSTA $M = (N, S)$, operations in $\mathcal{F}_M^+$ can be carried out in time $\mathcal{O}(|Q|^6 \cdot 2^k)$ using join operations.

We define the total order $\leq_T$ on the nodes of T, that is used by the Algorithm 5, as follows: $v \leq_T w$ if and only if v occurs before w in the parse tree of Ψ, reading the leaves from left to right. We stress that the order $\leq_T$ depends on the formula Ψ. Especially, the order can change in non-obvious ways during insertion and deletion updates, as the structure of Ψ may change, due to rotations. We sketch how to achieve enumeration in pre- or post-order in Section 6.3.

For the following presentation we like to remind that T_v denotes the forest or context that results from T by restricting T to the nodes that occur in Ψ_v, if Ψ_v denotes a context, add a hole at the appropriate position.

We already know that elements of $\mathcal{F}_M^+$ can be interpreted as sets of extended signatures of possible runs. However, not all runs of M on T_v (and thus not all signatures in $h^+(T_v)$) are actually useful for completing a given incomplete answer A. To be useful, an extended signature $(x, r) \in h^+(T_v)$ has to satisfy two conditions: It has to be the signature of a run λ of T_v that

(1) is compatible with A, i.e., for the nodes in A it visits the states indicated by s; and
(2) can be extended to some accepting run λ' over T that visits all states in Q_s.

Let now $A = (v_1, \dots, v_\ell)$ be an incomplete answer and We write $\lambda \models_s A$, if $\lambda_{\mathsf{self}}(v_i) = q_i^s$ for $i \in \{1, \dots, \ell\}$. Towards the above conditions, we define sets of relevant tuples of each node v of Ψ. The sets $R^1_{A,s}(v)$ account for the first condition, and are defined by

$$R^1_{A,s}(v) \quad = \quad \begin{cases} h^+(T_v) \ \cap \ (\mathrm{SIG}_Q \times \{\{q_i^s\}\}) & \text{if } v = v_i \\ h^+(T_v) & \text{if } v \notin A \text{ is a leaf of } \Psi \\ R^1_{A,s}(\mathsf{child_L}(v)) \odot_v^+ R^1_{A,s}(\mathsf{child_R}(v)) & \text{if } v \text{ is not a leaf of } \Psi \end{cases}$$

For leaf nodes of the formula that occur in A, we only keep those tuples from $h^+(T_v)$ that are compatible with A, i.e., that assign the correct state to v. We note that if v occurs at two positions i and j in A, then q_i^s and q_j^s are necessarily equivalent. For leaf nodes of the formula that do not occur in A, we simply retain all possible signatures. Finally, for inner nodes of the formula, we use $\mathcal{F}_M^+$ to compute $R^1_{A,s}$, i.e., we perform the same join operation that is used to compute $h(T_v)$ in the first place. The only difference is that we apply the operation only to those signatures in the two children of v that are compatible with A.

The sets $R^2_{A,s}(v)$ additionally account for the second condition and are computed top-down, starting with the set of signatures compatible with A that belong to accepting runs in the root:

$$R^2_{A,s}(v) \quad = \quad \begin{cases} R^1_{A,s}(v) \ \cap \ (\{(q_I, q_F)\} \times \{Q_s\}) & \text{if } v = \mathsf{root}(\Psi) \\ \{x \in R^1_{A,s}(v) \ \mid \ \big(R^1_{A,s}(w) \odot_u^+ \{x\}\big) \cap R^2_{A,s}(u) \neq \emptyset\} & \text{if } v = \mathsf{child_R}(u) \\ \{x \in R^1_{A,s}(v) \ \mid \ \big(\{x\} \odot_u^+ R^1_{A,s}(w)\big) \cap R^2_{A,s}(u) \neq \emptyset\} & \text{if } v = \mathsf{child_L}(u) \end{cases}$$

Here, w is always the sibling of v. For the root, we simply drop all signatures that indicate that the run is either not accepting or does not contain all states that are needed to produce an answer. For inner nodes, we use again the operations of $\mathcal{F}_M^+$ to compute the desired signatures, but this time, we know the result of the operation (it is stored at the parent u) and need to compute all those signatures from $R^1_{A,s}(v)$ that can be combined with some signature of v's sibling w to obtain some signature from $R^2_{A,s}(u)$. There are two cases, depending on whether v is left of w or vice versa.

The computation of $R^1_{A,s}$ followed by $R^2_{A,s}$ is very similar to Yannakakis algorithm [Yan81] for computing the result of acyclic relational joins. This is not surprising at all, as essentially we are computing an acyclic join. The only difference is that for the last component, i.e., for computing the subset of Q_s that occurs in the run we compute the union.

The following two lemmas show that the definitions of $R^1_{A,s}$ and $R^2_{A,s}$ work as expected.

Lemma 6.5. *It holds that* $(x, r) \in R^1_{A,s}(v)$ *if and only if there exists a run* λ *on* T_v *such that* $\mathrm{SIG}^+(\lambda) = (x, r)$ *and* $\lambda \models_s A$.

Proof. The proof is by induction and very similar to the proof of Lemma 4.5. We start with the if-direction.

If there is a run λ on T_v that is compatible with A, then $(\mathrm{SIG}(\lambda), \mathrm{states}(\lambda) \cap Q_s) \in R^1_{A,s}(v)$. For the base cases this follows directly from the definition of $R^1_{A,s}$. For the induction case, where $\Psi_v = \Psi_l \bigcirc^+ \Psi_r$ for some operation $\bigcirc^+$ and some subformulas Ψ_l and Ψ_r this follows, as we can decompose λ into two runs λ_l and λ_r (just as in the proof of Lemma 4.5) and by the induction assumption we have $(\mathrm{SIG}(\lambda_l), \mathrm{states}(\lambda_l)) \in R^1_{A,s}(\mathsf{child_L}(v))$ and $(\mathrm{SIG}(\lambda_r), \mathrm{states}(\lambda_r)) \in R^1_{A,s}(\mathsf{child_R}(v))$.

For the only if direction assume that $(x, r) \in R^1_{A,s}(v)$. Then there is a run λ on T_v that is compatible with A. For the base cases this follows again directly from the definition of $R^1_{A,s}$. For the induction case, we get from the definition of $R^1_{A,s}$ that there are extended signatures $(x_1, r_1) \in R^1_{A,s}(\mathsf{child_L}(v))$ and $(x_2, r_2) \in R^1_{A,s}(\mathsf{child_R}(v))$, such that $x \in \{x_1\} \bigcirc_v \{x_2\}$ and $r = r_1 \cup r_1$. By the induction assumption there are runs λ_i such that $\mathrm{SIG}^+(\lambda_i) = (x_i, r_i)$ for $i \in \{1, 2\}$. Following the proof of Lemma 4.5, λ_1 and λ_2 can be combined to a run λ over T_v with $\mathrm{SIG}(\lambda) = x$. As $r = r_1 \cup r_2$ it follows that $\mathrm{SIG}^+(\lambda) = (x, r)$. □

Lemma 6.6. *It holds that* $(x, r) \in R^2_{A,s}(v)$ *if and only if there exists a run* λ *on* T *such that* λ *is accepting,* $\mathrm{SIG}^+(\lambda_v) = (x, r)$ *and* $\lambda \models_s A$, *where* λ_v *is the restriction of* λ *to* T_v.

Proof. We start with the if direction. The proof is by a top-down induction. At the root, the claim holds by the definition of $R^2_{A,s}$ and Lemma 6.5. We node that at the root $\lambda = \lambda_v$. Let now v be a non-root node of Ψ. W.l.o.g., we assume that v is the left child of its parent u. The other case is symmetric. If there is an accepting run λ on T such that $\lambda \models_s A$, then by the induction assumption there is the signature $= \mathrm{SIG}^+(\lambda_u) \in R^2_{A,s}(u)$, where λ_u is the restriction of λ to T_u. Let w be the right sibling of v. By Lemma 6.5 it holds that $\mathrm{SIG}^+(\lambda_w) \in R^1_{A,s}(w)$, where λ_w is the restriction of λ to T_w. By the definition of the extended transition algebra and the fact that λ_v and λ_w are compatible and be combined to λ_u, we have that $\{\mathrm{SIG}^+(\lambda_v)\} \bigcirc^+_u \{\mathrm{SIG}^+(\lambda_w)\} = \{\mathrm{SIG}^+(\lambda_u)\}$. By the definition of $R^2_{A,s}$ we have that $\mathrm{SIG}^+(\lambda_v) \in R^2_{A,s}(v)$.

We continue with the only if direction. Let (x, r) be a signature from $R^2_{A,s}(v)$. By definition of $R^2_{A,s}$ we have that there are signatures $(x_u, r_u) \in R^2_{A,s}(u)$ and $(x_w, r_W) \in R^1_{A,s}(w)$. By the induction assumption there is an accepting run λ such that $\lambda \models_s A$ and $\mathrm{SIG}^+(\lambda_u) = (x_u, r_u)$, where λ_u is the restriction of λ to T_u. By Lemma 6.5 and the definition of$R^1_{A,s}$, we have that there is a run λ'_u on T_u, such that $\lambda'_u \models A$, $\mathrm{SIG}^+(\lambda'_u) = \mathrm{SIG}^+(\lambda_u)$ and the restrictions λ'_v and λ'_w of λ'_u to T_v and T_w, respectively are such that $\mathrm{SIG}^+(\lambda'_v) = (x, r)$ and $\mathrm{SIG}^+(\lambda'_w) \in R^1_{A,s}$. The run λ', where re replace λ_u with λ'_u in λ is an accepting run such that $\lambda' \models_s A$ and the restriction of λ' to T_v has signature (x, r). We note that we can replace λ_u by λ'_u, as both runs have the same signature. This concludes the proof. □

The definitions of $R^1_{A,s}$ and $R^2_{A,s}$ yield straightforward algorithms to compute these sets. The computation of $R^1_{A,s}$ can be done bottom-up (just as the computation of $\mathcal{F}^+$),

Algorithm 6 Procedure complete as used in Algorithm 5

Input: incomplete answer $A = (v_1, v_2, \dots, v_j, \bot, \dots, \bot)$, node u
Output: the answer COMPLETE(A, u) from Definition 6.2

1: **function** COMPLETE(A, u)
2: **return** COMPLETE$(A, \mathrm{Root}(\Psi), u)$
3: **function** COMPLETE(A, v, u)
4: compute $R^2_{A,s}(v)$ for $s \in S$
5: **if** $\max(\mathrm{Nodes}(T_v)) \leq u$ or $q^s_{j+1} \notin \mathrm{states}(R^2_{A,s}(v))$ for every $s \in S$ **then**
6: **return** $\bot$
7: **if** isLeaf(v) **then**
8: $A' \leftarrow (v_1, v_2, \dots, v_j, v)$
9: **else**
10: $A' \leftarrow$ COMPLETE$(A, \mathsf{child}_\mathsf{L}(v), u)$
11: **if** $A' = \bot$ **then** $A' \leftarrow$ COMPLETE$(A, \mathsf{child}_\mathsf{R}(v), u)$
12: **if** $A' \neq \bot$ **then** compute $R^1_{A',s}(v)$ for $s \in S$
13: **return** A'

while the computation of $R^2_{A,s}$ can be done top-down. We note that the runtime of the naive algorithm to compute $R^1_{A,s}(u)$ is linear in $||\Psi_u||$, as the computation is bottom-up. However, after extending an incomplete answer A with an additional node v to an incomplete answer A', it is sufficient to compute $R^1_{A',s}(u)$ for those nodes u in Ψ that are an ancestor of v, as—by the definition of the sets $R^1_{A',s}$—it holds that $R^1_{A',s}(u) = R^1_{A,s}(u)$ for all nodes u such that Ψ_u does not contain v. In fact $R^1_{A,s}(u) = h^+(\Psi_u)$ if no node from A occurs in Ψ_u.

We now have all ingredients for an implementation of the procedure COMPLETE that we depict in Algorithm 6. We first prove correctness before we give an upper bound on the runtime. We use $\mathrm{states}(R^2_{A,s}(v))$ to denote the set of states that occur in some tuple of $R^2_{A,s}(v)$, i.e.,

$$\mathrm{states}(R^2_{A,s}(v)) = \bigcup_{(x,r) \in R^2_{A,s}(v)} r\ .$$

Lemma 6.7. *The procedure* COMPLETE *correctly computes the incomplete answer as required by Definition 6.2.*

Proof. The main challenge of the procedure is to find a node v_{j+1} that can be used to extend the incomplete answer A. As our order of the nodes of T is induced by the order of the leaves of Ψ, we have to find the leftmost leaf of Ψ that can be used to extend A. By the definition of $R^2_{A,s}$, this is the leftmost leaf v with $v > u$ and $q^s_{j+1} \in \mathrm{states}(R^2_{A,s}(v))$ for some $s \in S$.

The procedure returns in Line 6, only if it is sure that no such node v_{j+1} can be found among the descendants of v, either because all nodes below v are smaller or equal than u, or because $q_{j+1} \notin \mathrm{states}(R^2_{A,s}(v))$ and therefore also $q_{j+1} \notin \mathrm{states}(R^2_{A,s}(w))$ for any w below v by the definition of $R^2_{A,s}$.

If v is a leaf, the procedure either returns $\bot$ in Line 6 or computes the correct incomplete answer A'. We note that if the algorithm does not return in Line 6 and v is a leaf, then v is the desired node.

If v is not a leaf, the algorithm first descends into the left subtree and only if no appropriate node was found there descends into the right subtree. It thus ensures to find the leftmost leaf satisfying the conditions. □

Lemma 6.8. *The procedure* Complete *runs in time* $\mathcal{O}(\log(|T|) \cdot |Q|^6 \cdot |S| \cdot 2^k)$.

Proof. The time spent in each invocation of Complete (excluding time spent in recursive calls) is dominated by the computation of $R^2_{A,s}(v)$ and $R^1_{A',s}(v)$ for each $s \in S$. Both operations can be carried out in time $\mathcal{O}(|Q|^6 \cdot 2^k)$ for each $s \in S$ using ordinary join algorithms. As observed above, we can compute $R^1_{A',s}(v)$ using the existing information for $R^1_{A,s}$ for the child of v that was not used to extend A to A'.

It remains to show that the total number of recursive calls is bounded by $\mathcal{O}(\log(|T|))$. Obviously it is enough to count non-tail calls, as for every non-tail call, there can be at most two tail calls. Calls that return in Line 6 are clearly tail calls. Therefore, we only count calls that return in Line 13. There can be at most $\mathsf{height}(\Psi)$ calls that return a value that is not $\perp$, as such a value prevents further recursive calls. And returning $\perp$ in Line 13 is only possible if $u \in \Psi_v$. If u is smaller than all nodes in Ψ_v, then the definition of $R^2_{A,s}$ ensures that we can find a node to extend A in Ψ_v. And if u is larger than all nodes in Ψ_v, then the algorithm already returns in Line 6. Altogether we have shown that there can be at most $\mathcal{O}(\log(|T|))$ many recursive calls. □

We now have all ingredients to show Theorem 6.1.

Proof. The delay follows from Lemma 6.8 and the fact that we need at most k calls to Complete to compute the next answer. The preprocessing and update times follow from Theorem 3.1 and Observation 6.4. We note that we need to compute one algebra operation for each inner node of the formula during preprocessing and we need at most logarithmically many algebra operations per update, exactly as in Theorem 5.1. □

6.3. **Enumerating in Pre-order and Post-order.** There is one unaesthetic detail in our algorithm, that can be fixed: Our implementation enumerates the tree in the order in which nodes appear in the formula, i.e., the enumeration order depends on the internal state of our data structure and can change due to updates. With a small change in the forest algebra and the enumeration algorithm, it is possible to enumerate T in pre-order or post-order. Algorithm 6 needs to be changed so that it does three recursive calls at each inner node that represents a context application. One to search for v in the context among the nodes before the hole, a second that searches the tree, the context is applied to, and a third searching the context again, but now on the nodes after the hole. To allow these three calls, the vertical monoid of the forest algebra needs to be extended such that it carries the information which states of S are visited before and after the hole, respectively.

7. Concluding Remarks and Further Directions

We depicted a framework for representing trees by forest algebra formulas of logarithmic depth. The algorithms for preprocessing and maintaining the formulas unfortunately have some quirks in them, which are needed to prove our runtime bounds. For example, we do not always apply all possible rotations after an update, as we could not prove that this simpler algorithm still runs in logarithmic time in the worst-case. It would be nice to show the same bounds for simplified versions of the algorithms.

We showed that by using our framework it is possible, after a linear preprocessing phase, to enumerate MSO queries over trees with logarithmic delay and that we can restart enumeration after logarithmic time after an update to the tree. In [ABMN19] it has already been shown that our framework is usable for an entirely different enumeration approach that solves the same problem. Using a more sophisticated enumeration algorithm on top of our forest algebra framework, it has been shown that constant delay enumeration can be achieved while the updates can still be carried out in logarithmic time. This result is at least close to optimal. It is not possible that the delay and the update time are both in $o(\log(n)/\log(\log(n)))$ [ABMN19].

Possible follow up work is to look which other results for static trees can be lifted to trees with updates using the framework, just as [ABMN19] did with the enumeration algorithm from [ABJM17], or which other results for dynamic strings can be lifted to dynamic trees, just as we did with the enumeration algorithm from [LM14] in Section 6.

A natural question is whether our approach can be generalized to a bigger class of structures. As long as no updates are considered, many results for trees carry over to structures of bounded treewidth. This usually works by computing a tree decomposition for a graph of bounded treewidth and then solve the problem on the tree decomposition. A very interesting but probably also very hard question is, whether the algebraic approach of forest algebras can somehow be extended to structures of bounded treewidth. More concretely: is there an algebraic framework that can represent a graph by a formula Φ of logarithmic height and—after a local update to the underlying graph—compute an updated formula Φ' representing the updated graph in a reasonable time?

References

[ABJM17] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, pages 111:1–111:15, 2017.

[ABM18] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *International Conference Database Theory (ICDT)*, pages 5:1–5:18, 2018.

[ABMN19] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 89–103. ACM, 2019. `doi:10.1145/3294052.3319702`.

[AVL62] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[Bag06] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Computer Science Logic (CSL)*, pages 167–181, 2006.

[BGM10] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. *ACM Transactions on Database Systems (TODS)*, 35(4):29:1–29:43, 2010.

[BH98] H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998.

[BKS17] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *International Symposium on Principles of Database Systems (PODS)*, pages 303–318, 2017.

[BKS18a] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018. `doi:10.1145/3232056`.

[BKS18b] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *International Conference Database Theory (ICDT)*, pages 8:1–8:19, 2018.

[BML+04] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682, 2004.

[BMT15] H. Björklund, W. Martens, and T. Timm. Efficient incremental evaluation of succinct regular expressions. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 1541–1550, 2015.

[Boj21] M. Bojańczyk. Algebra for trees. In J.-E. Pin, editor, *Handbook of Automata Theory*, volume I, pages 801–838. European Mathematical Society, 2021.

[BPV04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems (TODS)*, 29(4):710–751, 2004.

[BW07] M. Bojańczyk and I. Walukiewicz. Forest algebras. In *Automata and Logic: History and Perspectives*, pages 107–132. Amsterdam University Press, 2007.

[CNT04] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *Rewriting Techniques and Applications (RTA)*, pages 105–118, 2004.

[Col07] T. Colcombet. A combinatorial theorem for trees. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 901–912, 2007.

[Cou09] B. Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

[DKM+15] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in dynfo. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 159–170, 2015.

[DS11] A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Computer Science Logic (CSL)*, pages 189–202, 2011.

[FG04] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.

[FGK03] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Logic in Computer Science (LICS)*, page 188, 2003.

[KMN+23] Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Dynamic treewidth. *arXiv preprint arXiv:2304.01744*, 2023.

[KS13a] W. Kazana and L. Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Symposium on Principles of Database Systems (PODS)*, pages 297–308, 2013.

[KS13b] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):25:1–25:12, 2013.

[LM14] K. Losemann and W. Martens. Mso queries on trees: Enumerating answers under updates. In *Joint Meeting of Computer Science Logic (CSL) and Logic in Computer Science (LICS)*, 2014.

[MN07] W. Martens and J. Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[MR22] Martín Muñoz and Cristian Riveros. Streaming enumeration on nested documents. In *25th International Conference on Database Theory*, page 23, 2022.

[MT86] K. Mehlhorn and A. Tsakalidis. An amortized analysis of insertions into AVL trees. *Siam Journal on Computing*, 15(1):22–33, February 1986.

[Nev99] F. Neven. *Design and Analysis of Query Languages for Structured Documents*. PhD thesis, Limburgs Universitair Centrum, 1999.

[Nie18] M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 769–778. ACM, 2018. doi:10.1145/3209108.3209144.

[NPTT05] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *International Conference on Database Programming Languages (DBPL)*, pages 217–231, 2005.

[NS18] M. Niewerth and L. Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *International Symposium on Principles of Database Systems (PODS)*, 2018. to appear.

[PI97] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences (JCSS)*, 55(2):199–209, 1997.

[Seg13] L. Segoufin. Enumerating with constant delay the answers to a query. In *International Conference on Database Theory (ICDT)*, pages 10–20, 2013.

[Sim90] I. Simon. Factorization forests of finite height. *Theoretical Computer Science (TCS)*, 72(1):65–94, 1990.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–82, 1968.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, pages 82–94, 1981.

[ZS14] T. Zeume and T. Schwentick. Dynamic conjunctive queries. In *International Conference Database Theory (ICDT)*, pages 38–49, 2014.