Enumeration on Trees under Relabelings

Antoine Amarilli
LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France

Pierre Bourhis
CRIStAL, CNRS UMR 9189 & Inria Lille, Lille, France

Stefan Mengel
CNRS, CRIL UMR 8188, Lens, France

Abstract

We study how to evaluate MSO queries with free variables on trees, within the framework of enumeration algorithms. Previous work has shown how to enumerate answers with linear-time preprocessing and delay linear in the size of each output, i.e., constant-delay for free first-order variables. We extend this result to support *relabelings*, a restricted kind of update operations on trees which allows us to change the node labels. Our main result shows that we can enumerate the answers of MSO queries on trees with linear-time preprocessing and delay linear in each answer, while supporting node relabelings in logarithmic time. To prove this, we reuse the circuit-based enumeration structure from our earlier work, and develop techniques to maintain its index under node relabelings. We also show how enumeration under relabelings can be applied to evaluate practical query languages, such as aggregate, group-by, and parameterized queries.

2012 ACM Subject Classification Theory of computation → Logic and databases

Keywords and phrases enumeration, trees, updates, MSO, circuits, knowledge compilation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.5

Funding For the full version with proofs, see [5], `https://arxiv.org/abs/1709.06185`.

1 Introduction

Enumeration algorithms are a common way to compute large query results on databases, see, e.g., [28]. Instead of computing all results, these algorithms compute results one after the other, while ensuring that the time between two successive results (the *delay*) remains small. Ideally, the delay should be *linear* in the size of each produced solution, and independent of the size of the input database. To make this possible, enumeration algorithms can build an index structure on the database during a *preprocessing phase* that ideally runs in linear time.

Most enumeration algorithms assume that the input database will not change. If we *update* the database, we must re-run the preprocessing phase from scratch, which is unreasonable in practice. Losemann and Martens [24] proposed the first enumeration algorithm that addresses this issue: they study monadic second-order (MSO) query evaluation on trees, and show that the index structure for enumeration can be maintained under updates. More precisely, they can update the index in time polylogarithmic in the input tree T (much better than re-running the linear preprocessing). The tradeoff is that their delay is also polylogarithmic in T, whereas the delay can be independent of T when there are no updates [8].

This result of [24] leads to a natural question: does the support for updates inherently increase the delay of enumeration algorithms? This is not always the case: e.g., when evaluating first-order queries (plus modulo-counting quantifiers) on bounded-degree databases, updates can be applied in constant time [12] and the delay is constant, as in the case without updates [18, 22]. However, when evaluating conjunctive queries (CQs) on arbitrary databases, supporting updates has a cost: under complexity-theoretic assumptions, the class of CQs with efficient enumeration under updates [11] is a strict subclass of the class of CQs for the case without updates [9]. Could the same be true of MSO on trees, as [24] would suggest?

In this work, we answer this question in the negative, for a restricted update language. Specifically, we show an enumeration algorithm for MSO on trees with the same delay as in the case without updates [8], while supporting updates with a better complexity than [24] (see detailed comparison of results in Section 3). The tradeoff is that we only allow updates that change the labels of nodes, called *relabelings*, unlike [24] where updates can also insert and delete leaves. We still show how these relabelings are useful to evaluate practical query languages, such as *parameterized* queries and *group-by queries with aggregates*. A *parameterized* query allows the user to specify some parameters for the evaluation (e.g., select some positions on the tree). Our results support such queries: we can model the parameters as labels and apply relabeling updates when the user changes the parameters. A *group-by query with aggregates* partitions the set of results into groups based on an attribute, and computes some aggregate quantity on each group (e.g., a sum). We show how to enumerate the results of such queries. For groups, our techniques can handle them with one single enumeration structure using relabelings to switch groups. For aggregates, we can efficiently compute and maintain them in arbitrary semirings; this problem was left open by [24] even for counting, and is practically relevant in its own right [26]. Of course, by Courcelle's theorem [15], our results generalize to MSO queries on bounded-treewidth data (see [3]), where relabelings mean adding or removing unary facts (i.e., the tree decomposition is unchanged).

The proof of our main result follows the approach of [4] and is inspired by knowledge compilation in artificial intelligence and by factorized representations in database theory. Specifically, we encode knowledge (in our case, the query result) as a circuit in a restricted class, and we then use the circuit for efficient reasoning and for aggregates as in [17]. In [4], we have used this circuit-based approach to recapture existing enumeration results for MSO on trees [8, 23]. In this work, we refine the approach and show that it can support updates. Our key new ingredient are *hybrid circuits*: they have both *set-valued* gates that represent the values to enumerate, and *Boolean* gates that encode the tree labels which can be updated. We first show that we can efficiently compute such circuits to capture the possible results of an MSO query under all possible labelings of a tree. Second, we show how to efficiently enumerate the set of assignments captured by these circuits, also supporting updates that toggle the Boolean gates affected by a relabeling. We also introduce some standalone tools, e.g., a lemma to *balance* the input trees to MSO queries (Lemma 4.3), ensuring that hybrid circuits have logarithmic depth so that changes can be propagated quickly; and a constant-delay enumeration algorithm for reachability in forests under updates (Section 7).

Paper structure. We start with preliminaries in Section 2, and define our problem and give our main result in Section 3. In Section 4, we review the set-valued provenance circuits of [4], and show our balancing lemma. We introduce hybrid circuits in Section 5, and show in Section 6 how to use them for enumeration under updates, using a standalone reachability indexing scheme on forests given in Section 7. Having shown our main result, we outline its consequences for application-oriented query languages in Section 8 and conclude in Section 9.

2 Preliminaries

Trees, queries, answers, assignments. In this work, unless otherwise specified, a *tree* is always binary, rooted, ordered, and full. Let Γ be a finite set called a *tree alphabet*. A Γ-*tree* (T, λ) is a pair of a tree T and of a *labeling function* λ that maps each node n of T to a *set* *of labels* $\lambda(n) \subseteq \Gamma$. We often abuse notation and identify T to its node set, e.g., write λ as a function from T to the powerset 2^Γ of Γ; we may also omit λ and write the Γ-tree as just T.

We consider queries in *monadic second-order logic* (MSO) on the *signature* of Γ-trees: it features two binary relations E_1 and E_2 denoting the first and second child of each internal node, and a unary relation P_l for each $l \in \Gamma$ denoting the nodes that carry label l (i.e., nodes n for which $l \in \lambda(n)$). MSO extends *first-order logic*, which builds formulas from atoms of this signature and from equality atoms, using the Boolean connectives and existential and universal quantification over nodes. Formulas in MSO can also use second-order quantification over sets of nodes, written as second-order variables. For instance, on $\Gamma = \{l_1, l_2, l_3\}$, we can express in MSO that every node carrying labels l_1 and l_2 has a descendant carrying label l_3.

In this work, we study MSO *queries*, i.e., MSO formulas with free variables. The free variables can be first-order or second-order, but we can rewrite any MSO query $Q(\mathbf{x}, \mathbf{Y})$ to ensure that all free variables are second-order: for instance as $Q'(\mathbf{X}, \mathbf{Y}) : \exists \mathbf{x} \ \bigwedge_i \mathrm{Sing}(X_i, x_i) \wedge Q(\mathbf{x}, \mathbf{Y})$, where $\mathrm{Sing}(X, x)$ asserts that X is exactly the singleton set $\{x\}$. Hence, we usually assume without loss of generality that MSO queries only have second-order free variables.

Given a Γ-tree T and an MSO query $Q(X_1, \ldots, X_m)$, an m-tuple $\mathbf{B} = B_1, \ldots, B_m$ of subsets of T is an *answer* of Q on T, written $T \models Q(\mathbf{B})$, if T satisfies $Q(\mathbf{B})$ in the usual logical sense. It will be more convenient to represent each answer as an *assignment*, which is a set of pairs called *singletons* that indicate that an element is in the interpretation of a variable. Formally, given an m-tuple $\mathbf{B}$ of subsets of T, the corresponding assignment is $\{\langle X_i : n \rangle \mid 1 \leq i \leq m \text{ and } n \in B_i\}$. We can convert each assignment in linear time to the corresponding answer and vice-versa, so we will use the assignment representation throughout this work. Our goal is to compute the set of assignments of Q on T, which we call the *output* of Q on T; we abuse notation and write it $Q(T)$. We measure the complexity of this task in *data complexity*, i.e., as a function of the input tree T, with the query Q being fixed.

Enumeration. The output of an MSO query can be huge, so we work in the setting of *enumeration algorithms* [31, 28] which we present following [4]. As usual for enumeration algorithms [28], we work in the RAM model with uniform cost measure (see, e.g., [1]), where pointers, numbers, labels for elements and facts, etc., have constant size.

An *enumeration algorithm with linear-time preprocessing* for a fixed MSO query $Q(\mathbf{X})$ on Γ-trees takes as input a Γ-tree T and computes the output $Q(T)$ of Q on T. It consists of two phases. First, the *preprocessing phase* takes T as input and produces in *linear time* a data structure J called the *index*, and an initial *state* s. Second, the *enumeration phase* repeatedly calls an algorithm $\mathcal{A}$. Each call to $\mathcal{A}$ takes as input the index J and the current state s, and returns one assignment and a new state s': a special state value indicates that the enumeration is over so $\mathcal{A}$ should not be called again. The assignments produced by the successive calls to $\mathcal{A}$ must be exactly the elements of $Q(T)$, with no duplicates.

We say that the enumeration algorithm has *linear delay* if the time to produce each new assignment A is linear in its cardinality $|A|$, and is independent of T. In particular, if all answers to Q are tuples of singleton sets (for instance, if Q is the translation of a MSO query where all free variables are first-order), then the cardinality of each assignment is constant (it is the arity of Q). In this case, the enumeration algorithm must produce each assignment with constant delay: this is called *constant-delay enumeration*. The *memory usage* of an enumeration algorithm is the maximum number of memory cells used during the enumeration phase (not counting the index J, which resides in read-only memory), expressed as a function of the size of the largest assignment (as in [8]): we say that the enumeration algorithm has *linear memory* if its memory usage is linear in the size of the largest assignment.

Previous works have studied enumeration for MSO on trees. Bagan [8] showed that for any fixed MSO query $Q(\mathbf{X})$, given a Γ-tree T, we can enumerate the output of Q on T with linear delay and memory, i.e., constant delay and memory when all free variables are first-order. This result was re-proven by Kazana and Segoufin [23] via a result of Colcombet [14], and a third proof via provenance circuits was recently proposed by the present authors [4].

3 Problem Statement and Main Result

Our goal is to address a limitation of these existing results, namely, the assumption that the input Γ-tree T will never change. Indeed, if T is updated, these results must discard the index J and re-run the preprocessing phase on the new tree. To improve on this, we want our enumeration algorithm to support *update operations* on T, and to update J accordingly instead of recomputing it from scratch. Specifically, an algorithm for *enumeration under updates* on a tree T has a preprocessing phase that produces the index J as usual, but has two algorithms during the enumeration phase: (i.) an enumeration algorithm $\mathcal{A}$ as presented before, and (ii.) an *update algorithm* $\mathcal{U}$. When we want to change the tree T, we call $\mathcal{U}$ with a description of the changes: $\mathcal{U}$ modifies T accordingly, updates the index J, and resets the enumeration state (so enumeration starts over on the new tree, and all working memory of the enumeration phase is freed). The *update time* of the enumeration algorithm is the complexity of $\mathcal{U}$: like preprocessing, but unlike delay, it is a function of the size of the (current) tree T.

To our knowledge, the only published result on enumeration for MSO queries under updates is the work of Losemann and Martens [24], which applies to words and to trees, for MSO queries with only free first-order variables. They show an enumeration algorithm with linear-time preprocessing: on words, the update complexity and delay is $O(\log |T|)$; on trees, these complexities become $O(\log^2 |T|)$. Thus the delay is worse than in the case without updates [8], and in particular it is no longer independent from T.

Main result. In this work, we show that enumeration under updates for MSO queries on trees can be performed with a better complexity that matches the case without updates: linear-time preprocessing, linear delay and memory (in the assignments), and update time in $O(\log |T|)$. This improves on the bounds of [24] (and uses entirely different techniques). However, in exchange for the better complexity, we only support a weaker update language: we can change the labels of tree nodes, called a *relabeling*, but we cannot insert or delete leaf nodes as in [24], which we leave for future work (see the conclusion in Section 9). We show in Section 8 that relabelings are still useful to derive results for some practical query languages.

Formally, a relabeling on a Γ-tree T is a pair of a node $n \in T$ and a label $l \in \Gamma$. To apply it, we change the label $\lambda(n)$ of n by adding l if $l \notin \lambda(n)$, and removing it if $l \in \lambda(n)$. In other words, the tree T never changes, and updates only modify λ. Our main result is then:

▶ **Theorem 3.1.** *For any fixed tree alphabet Γ and MSO query $Q(\mathbf{X})$ on Γ-trees, given a Γ-tree T, we can enumerate the output $Q(T)$ of Q on T with linear-time preprocessing, linear delay and memory, and logarithmic update time for relabelings.*

In other words, after preprocessing T in time $O(|T|)$ to compute the index J, we can:

- Enumerate the assignments of Q on T, using J, with delay linear in the size of each assignment, so constant if the assignments to Q have constant size.
- Toggle a label of a node of T, update J, and reset the enumeration, in time $O(\log |T|)$.

We show this result in Sections 4–7, and then give consequences of this result in Section 8.

4 Provenance Circuits

Our general technique for enumeration follows our earlier work [4]: from the query and input tree, we compute in linear time a structure called a *provenance circuit* to represent the results to enumerate, we observe that it falls in a restricted circuit class, and we conclude by showing a general enumeration result for circuits of this class. In this section, we review our construction of provenance circuits in [4], with some additional observations that will be useful for updates. In particular, we show an independent *balancing lemma* on input trees, which allows us to bound a parameter of the circuit called *dependency size*. We will extend the formalism of this section to so-called *hybrid circuits* in the next section; and we will show our enumeration result for such circuits in Sections 6 and 7.

Set circuits. We start with some preliminaries about circuits. A *circuit* $C = (G, W, g_0, \mu)$ is a directed acyclic graph (G, W) whose vertices G are called *gates*, whose edges W are called *wires*, where $g_0 \in G$ is the *output gate*, and where μ is a function giving a *type* to each gate of G (the possible types depend on the kind of circuit). The *inputs* to a gate $g \in G$ are $\mathrm{inp}(g) := \{g' \in G \mid (g', g) \in W\}$ and the *fan-in* of g is its number of inputs $|\mathrm{inp}(g)|$.

We define *set-valued circuits*, which are an equivalent rephrasing of the *circuits in zero-suppressed semantics* used in [4]. They can also be seen to be isomorphic to arithmetic circuits, and generalize factorized representations used in database theory [27]. The type function μ of a set-valued circuit maps each gate to one of $\cup$, $\times$, var. We require that $\times$-gates have fan-in 0 or 2, and that var-gates have fan-in 0: the latter are called the *variables* of C, with C_{var} denoting the set of variables. Each gate g of C *captures* a set $\mathrm{S}(g)$ of *assignments*, where each *assignment* is a subset of C_{var}. These sets are defined bottom-up as follows:

- For a variable gate g, we have $\mathrm{S}(g) := \{\{g\}\}$.
- For a $\cup$-gate g, we have $\mathrm{S}(g) := \bigcup_{g' \in \mathrm{inp}(g)} \mathrm{S}(g')$. In particular, if $\mathrm{inp}(g) = \emptyset$ then $\mathrm{S}(g) = \emptyset$.
- For a $\times$-gate g with no inputs, we have $\mathrm{S}(g) := \{\{\}\}$.
- For a $\times$-gate g with two inputs g_1 and g_2, we have $\mathrm{S}(g) := \{A_1 \cup A_2 \mid (A_1, A_2) \in \mathrm{S}(g_1) \times \mathrm{S}(g_2)\}$, which we write $\mathrm{S}(g) := \mathrm{S}(g_1) \times_{\mathrm{rel}} \mathrm{S}(g_2)$ (this is the relational product).

The set $\mathrm{S}(C)$ *captured* by C is $\mathrm{S}(g_0)$ for g_0 the output gate of C. Note that each assignment of $\mathrm{S}(C)$ is a satisfying assignment of C when seen in the usual semantics of monotone circuits.

Structural requirements. Before defining our provenance circuits, we introduce some structural restrictions that they will respect, and that will be useful for enumeration.

The first requirement is that the circuit is a *d-DNNF*. Our definition of d-DNNF is inspired by [16] but applies to set-valued circuits, as in [4] (see also the z-st-d-DNNFs of [30]). For each gate g of a set-valued circuit C, we define the *domain* $\mathsf{dom}(g)$ of g as the variable gates having a directed path to g. In particular, for $g \in C_{\mathrm{var}}$, we have $\mathsf{dom}(g) = \{g\}$, and if $\mathrm{inp}(g) = \emptyset$ then $\mathsf{dom}(g) = \emptyset$. We now call a $\times$-gate g *decomposable* if it has no inputs or if, letting $g'_1 \neq g'_2$ be its two inputs, the domains $\mathsf{dom}(g'_1)$ and $\mathsf{dom}(g'_2)$ are disjoint. This ensures that no variable of C occurs both in an assignment of $\mathrm{S}(g'_1)$ and in an assignment of $\mathrm{S}(g'_2)$. We call a $\cup$-gate g *deterministic* if, for any two inputs $g'_1 \neq g'_2$ of g, the sets $\mathrm{S}(g'_1)$ and $\mathrm{S}(g'_2)$ are disjoint, i.e., there is no assignment that occurs in both sets. We call C a *d-DNNF* if every $\times$-gate g is decomposable and every $\cup$-gate g is deterministic. Under this assumption, we can tractably compute the cardinality of the set $\mathrm{S}(C)$ captured by C.

The second requirement on circuits is called *upwards-determinism* and was introduced in [3]. In that paper, it was used to show an improved memory bound; in the present paper, we will always be able to enforce it. A wire (g, g') in a set-valued circuit C is called *pure* if:

- g' is a $\cup$-gate; or
- g' is a $\times$-gate and, letting g'' be the other input of g', we have $\{\} \in \mathrm{S}(g'')$, i.e., g'' captures the empty assignment.

We say that a gate g is *upwards-deterministic* if there is at most one gate g' such that (g, g') is pure. We call C *upwards-deterministic* if every gate of C is.

The third requirement concerns the *maximal fan-in* of circuits, which is simply defined for a set-valued circuit C as the maximal fan-in of a gate of C. We will require that the maximal fan-in is bounded by a constant.

The fourth and last requirement concerns a new parameter called *dependency size*. To introduce this, we define the *dependent gates* $\Delta(g')$ of a gate g' in a set-valued circuit C as the gates g such that there is a directed path from g' to g. Intuitively, the set $\mathrm{S}(g)$ captured by g may then depend on the set $\mathrm{S}(g')$ captured by g'. The *dependency size* of C is $\Delta(C) := \max_{g \in C} |\Delta(g)|$, i.e., the maximal number of gates that are dependent on any given gate g. We will require this parameter to be connected to the height of the input tree.

Set-valued provenance circuits. We can now define provenance circuits like in [4]. A set-valued circuit C is a *provenance circuit* of a MSO query $Q(X_1, \ldots, X_m)$ on a Γ-tree T if:

- The variables of C correspond to the possible singletons, formally: $C_{\mathrm{var}} = \{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in T\}$; and
- The set of assignments captured by C is the output of Q on T, formally: $\mathrm{S}(C) = Q(T)$. Equivalently, for any tuple $\mathbf{B} = (B_1, \ldots, B_m)$ of subsets of T, we have $T \models Q(\mathbf{B})$ iff the assignment $\{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in B_i\}$ is in $\mathrm{S}(C)$.

▶ **Example 4.1.** Consider the unlabeled tree T of Figure 1, the alphabet $\Gamma = \{B\}$, and the MSO query $Q(x)$ with one free first-order variable asking for the leaf nodes whose B-annotation is different from that of its parent (i.e., the node carries label B and the parent does not, or vice-versa). Consider the labeling λ mapping 1 to $\{B\}$ and 2 and 3 to $\emptyset$. A set-valued circuit capturing the provenance of Q on (T, λ) is given in Figure 2.

We then know from [3] that provenance circuits can be computed efficiently, and they can be made to respect our structural requirements:

▶ **Theorem 4.2** (from [4], Theorem 7.3). *For any fixed MSO query $Q(\mathbf{X})$ on Γ-trees, given a Γ-tree T, we can compute in time $O(|T|)$ a set-valued provenance circuit C of Q on T. Further, C is a d-DNNF, it is upwards-deterministic, its maximal fan-in is constant, and its dependency size is in $O(\mathrm{h}(T))$, where h denotes the height of T.*

Proof sketch. We recall the main proof technique: we convert Q to a bottom-up deterministic tree automaton A on Γ-trees, and we add nodes to T to describe the possible valuations of variables. The provenance circuit C then captures the possible ways that A can read T depending on the valuation: we compute it with the construction of [6], and is a d-DNNF thanks to automaton determinism (see [2]). Upwards-determinism is shown like in [3].

The bounds on fan-in and dependency size are not stated in [4, 3] but already hold there. Specifically, the maximal fan-in is a function of the transition function of A, i.e., it does not depend on T. The bound on dependency size holds because C is constructed following the structure of T: we create for each tree node a gadget whose size depends only on A, and we connect these gadgets precisely following the structure of T, so that $\Delta(g)$ for any gate g of C can only contain gates from the node n of g or from ancestors of n in the tree. ◀

In the context of updates, the bound of dependency size will be crucial: intuitively, it describes how many gates need to be updated when an update operation modifies a gate of the circuit. As this bound depends on the height of the input tree, we will conclude this section by a *balancing lemma* that ensures that this height can always be made logarithmic (which matches our desired update complexity). We will then add support for updates in the next section by extending circuits to *hybrid circuits*.

Balancing lemma. Our balancing lemma is a general observation on MSO query evaluation on trees, and is in fact completely independent from provenance circuits. It essentially says that the input tree can be assumed to be balanced. Formally, we will show that we can rewrite any MSO query Q on Γ-trees to an MSO query Q' on a larger tree alphabet Γ' so that any input tree T for Q can be rewritten in linear time to a balanced tree T' on which Q' returns exactly the same output. Because we intend to support update operations, the input tree T will be unlabeled, and the rewritten tree T' will work for any labeling of T. Formally:

▶ **Lemma 4.3.** *For any tree alphabet Γ and MSO query $Q(\mathbf{X})$ on Γ-trees, we can compute a tree alphabet $\Gamma' \supseteq \Gamma$ and MSO query $Q'(\mathbf{X})$ on Γ'-trees such that the following holds. Given any unlabeled tree T with node set N, we can compute in linear time a Γ'-tree (T', λ') with node set $N' \supseteq N$, such that $\mathrm{h}(T') = O(\log |T|)$ and such that, for any labeling function $\lambda : T \to 2^\Gamma$, we have $Q(\lambda(T)) = Q'(\lambda''(T'))$, where $\lambda''(n)$ maps $n \in T'$ to $\lambda(n)$ if $n \in T$ and $\lambda'(n)$ otherwise.*

Proof sketch. We prove Lemma 4.3 by seeing the input tree T as a relational structure I of treewidth 1, and invoking the result by Bodlaender [13] to compute in linear time a constant-width tree decomposition of I which is of logarithmic height. We then translate the query Q to a MSO query Q' on tree encodings of this width, and compute from T the tree encoding T' corresponding to the tree decomposition (we rename some nodes of T' to ensure that the nodes of T are reflected in T'). Note that the balanced tree decompositions of [13] were already used for similar purposes elsewhere, e.g., in [19], end of Section 2.3. ◀

5 Hybrid Circuits for Updates

In this section, we extend set-valued circuits to support updates, defining *hybrid circuits*. We then extend Theorem 4.2 for these circuits. Last, we introduce a new structural notion of *homogenization* of hybrid circuits and show how to enforce it. We close the section by stating our main enumeration result on hybrid circuits, which implies our main theorem (Theorem 3.1), and is proved in the two next sections.

Hybrid circuits. A hybrid circuit is intuitively similar to a set-valued circuit, but it additionally has *Boolean variables* (which can be toggled when updating), Boolean gates ($\wedge$, $\vee$, $\neg$), and gates labeled $\boxtimes$ which keep or discard a set of assignments depending on a Boolean value. Formally, a *hybrid circuit* $C = (G, W, g_0, \mu)$ is a circuit where the possible gate types are svar (*set-valued variables*), bvar (*Boolean variables*), $\cup$, $\times$, $\boxtimes$, $\wedge$, $\vee$, and $\neg$. We call a gate *Boolean* if its type is bvar, $\wedge$, $\vee$, or $\neg$; and *set-valued* otherwise. We require that the output gate g_0 is set-valued and that the following conditions hold:

- svar-gates and bvar-gates have fan-in exactly 0;
- All inputs to $\wedge$-gates, $\vee$-gates, and $\neg$-gates are Boolean, and $\neg$-gates have fan-in exactly 1;
- All inputs to $\cup$ and $\times$-gates are set-valued, and $\times$-gates have fan-in either 0 or 2;
- $\boxtimes$-gates have one set-valued input and one Boolean input (so they have fan-in exactly 2).

We write C_{bvar} to denote the gates of C of type bvar, called the *Boolean variables*, and define likewise the *set-valued variables* C_{svar}. An example hybrid circuit is illustrated in Figure 3.

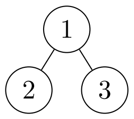

Figure 1 Example unlabeled tree.

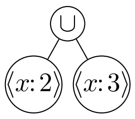

Figure 2 Example set circuit.

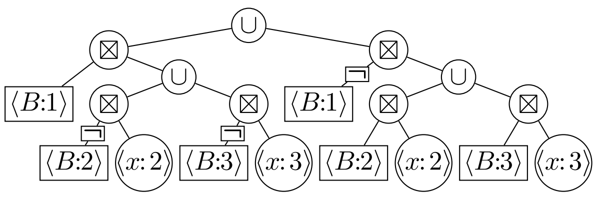

Figure 3 Example hybrid circuit. Boolean gates are squared, set-valued gates are circled, and variables are repeated.

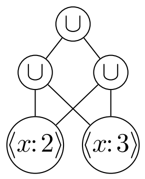

Figure 4 Example switchboard.

Unlike set-valued circuits, which capture only one set of assignments, hybrid circuits capture several different sets of assignments, depending on the value of the Boolean variables (intuitively corresponding to the tree labels). This value is given by a *valuation* of C, i.e., a function $\nu : C_{\text{bvar}} \to \{0, 1\}$. Given such a valuation ν, each Boolean gate g *captures* a Boolean value $\mathrm{V}_\nu(g) \in \{0, 1\}$, computed bottom-up in the usual way: we set $\mathrm{V}_\nu(g) := \nu(g)$ for $g \in C_{\text{bvar}}$, and otherwise $\mathrm{V}_\nu(g)$ is the result of the Boolean operation given by the type $\mu(g)$ of g, applied to the Boolean values $\mathrm{V}_\nu(g')$ captured by the inputs g' of g (in particular, a $\wedge$-gate with no inputs always has value 1, and a $\vee$-gate with no inputs always has value 0).

We then define the *evaluation* of C under ν as the set-valued circuit $\nu(C)$ obtained as follows. First, replace each Boolean gate g of C by a $\times$-gate with no inputs (capturing $\{\{\}\}$) if $\mathrm{V}_\nu(g) = 1$, and by a $\cup$-gate with no inputs (capturing $\emptyset$) if $\mathrm{V}_\nu(g) = 0$. Second, relabel each $\boxtimes$-gate g of C to be a $\times$-gate. Using $\nu(C)$, for each set-valued gate g of C, we define the set *captured by g under ν*: it is the set of assignments (subsets of C_{svar}) that g captures in $\nu(C)$. The set $\mathrm{S}_\nu(C)$ *captured* by C under ν is then $\mathrm{S}_\nu(g_0)$, for g_0 the output gate of C.

We last lift the structural definitions from set-valued circuits to hybrid circuits. The *maximal fan-in* and *dependency size* of a hybrid circuit are defined like before (these definitions do not depend on the kind of circuit). A hybrid circuit C is a *d-DNNF*, resp. is *upwards-deterministic*, if for every valuation ν of C, the set-valued circuit $\nu(C)$ has the same property. For instance, the hybrid circuit in Figure 3 is upwards-deterministic and is a d-DNNF.

Hybrid provenance circuits. We can now use hybrid circuits to define provenance with support for updates. The set-valued variables of the circuit will correspond to singletons as before, describing the interpretation of the *free variables* of the query; and the Boolean variables stand for a different kind of singletons, describing which *labels* are carried by each node. To describe this formally, we will consider an *unlabeled* tree T, and define a *labeling assignment* of T for a tree alphabet Γ as a set of singletons of the form $\langle l : n\rangle$ where $l \in \Gamma$ and $n \in T$. Given a labeling assignment α, we can define a labeling function λ_α for T, which maps each node $n \in T$ to $\lambda(n) := \{l \in \Gamma \mid \langle l : n\rangle \in \alpha\}$. Now, we say that a hybrid circuit C is a *provenance circuit* of a MSO query $Q(X_1, \ldots, X_m)$ on an *unlabeled* tree T if:

- The set-valued variables of C correspond to the possible singletons in an assignment, formally $C_{\text{svar}} = \{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in T\}$;
- The Boolean variables of C correspond to the possible singletons in an update assignment, formally $C_{\text{bvar}} = \{\langle l : n\rangle \mid l \in \Gamma \text{ and } n \in T\}$;

- For any labeling assignment α, let ν_α be the Boolean valuation of C_{bvar} mapping each $\langle l : n\rangle$ to 0 or 1 depending on whether $\langle l : n\rangle \in \alpha$ or not, and let λ_α be the labeling function on T defined as above. Then we require that the set of assignments $\mathrm{S}_{\nu_\alpha}(C)$ captured by C under ν_α is exactly the output of Q on $\lambda_\alpha(T)$, formally, $\mathrm{S}_{\nu_\alpha}(C) = Q(\lambda_\alpha(T))$.

In other words, for each labeling λ of the tree T, considering the valuation ν that sets the Boolean variables of C accordingly, then $\nu(C)$ is a provenance circuit for Q on $\lambda(T)$.

▶ **Example 5.1.** Recall the query $Q(x)$ and alphabet $\Gamma = \{B\}$ of Example 4.1, and the tree T of Figure 1. A hybrid circuit C capturing the provenance of Q on T is given in Figure 3 (with variable gates being drawn at multiple places for legibility): square leaves correspond to Boolean variables testing node labels, and circle leaves correspond to set-valued variables capturing a singleton of the form $\langle x{:}n\rangle$ for some $n \in T$. In particular, for the labeling λ of Example 4.1, the corresponding valuation ν maps $\langle B{:}1\rangle$ to 1 and $\langle B{:}2\rangle$ and $\langle B{:}3\rangle$ to 0, and the evaluation $\nu(C)$ of C under ν captures the same set as the circuit of Figure 2.

We can now extend Theorem 4.2 to compute a hybrid provenance circuit as follows:

▶ **Theorem 5.2.** *For any fixed MSO query $Q(\mathbf{X})$ on Γ-trees, given an unlabeled tree T, we can compute in time $O(|T|)$ a hybrid provenance circuit C which is a d-DNNF, is upwards-deterministic, has constant maximal fan-in, and has dependency size in $O(\mathrm{h}(T))$.*

Proof sketch. The proof is analogous to that of Theorem 4.2. The only difference is that the automaton now reads the label of each node as if it were a variable, so that the provenance circuit C also reflects these label choices as Boolean variables. ◀

Homogenization. We will make enumeration simpler by imposing one last requirement on hybrid circuits. A hybrid circuit C is *homogenized* if there is no valuation ν of C and set-valued gate g of C such that $\{\} \in \mathrm{S}_\nu(g)$. Note that the requirement does not apply to the Boolean gates of C, nor to the gates that replace them in evaluations $\nu(C)$ of C, so it equivalently means that C does not contain $\times$-gates with no inputs. Intuitively, set-valued gates in C that capture the empty assignment would waste time in the enumeration. We will show that we can rewrite circuits in linear time to make them homogenized, while preserving our requirements; but we need to change our definitions slightly to ensure that the circuit can still capture the empty assignment overall. To do so, we add the possibility of distinguishing a Boolean gate g_1 of a hybrid circuit C as its *secondary output*; in this case, given a valuation ν of C, the set $\mathrm{S}_\nu(C)$ captured by C under ν is $\mathrm{S}_\nu(C)$ plus the empty assignment $\{\}$ if the secondary output g_1 evaluates to 1, i.e., if $\mathrm{V}_\nu(g_1) = 1$. We say that two hybrid circuits C and C' (with or without secondary outputs) are *equivalent* if $C_{\mathrm{bvar}} = C'_{\mathrm{bvar}}$, $C_{\mathrm{svar}} = C'_{\mathrm{svar}}$, and for any valuation ν of C, we have $\mathrm{S}_\nu(C) = \mathrm{S}_\nu(C')$. We then have:

▶ **Lemma 5.3.** *For any hybrid circuit C, we can build in linear time a hybrid circuit C' with a secondary output g_1, such that C' is homogenized and it is equivalent to C. Further, if C is a d-DNNF and is upwards-deterministic, then so is C'; if C has bounded fan-in then the same holds of C'; and we have $\Delta(C') = O(\Delta(C))$.*

Proof sketch. This is shown analogously to homogenization in [4], which follows the technique of Strassen [29] (only done for two “layers”, namely, empty and non-empty assignments). ◀

Hence, up to linear-time processing, we can additionally assume that the circuits of Theorem 5.2 are homogenized. We can now use this theorem, the lemma above, and Lemma 4.3, to reduce enumeration for MSO on trees (as in our main theorem, Theorem 3.1) to the task of enumerating the set captured by a hybrid circuit satisfying some structural properties. The result that we need is the following (we prove it in the next two sections):

▶ **Theorem 5.4.** *Given an upwards-deterministic, d-DNNF, homogenized hybrid circuit C with constant fan-in, given an initial Boolean valuation ν of C_{bvar}, there is an enumeration algorithm with linear-time preprocessing to enumerate the set $\mathrm{S}_\nu(C)$ captured by C under ν, with linear delay and memory in each produced assignment, and with update time in $O(\Delta(C))$: an* update *consists here of toggling one value in ν.*

6 Enumerating Assignments of Hybrid Circuits

In this section and the next, we prove Theorem 5.4 by giving an algorithm for enumeration under updates. We start by describing the preprocessing phase, computing two simple structures: a *shortcut function* and a *partial evaluation*; we also explain how this index can be efficiently updated. We then describe an algorithm for the enumeration phase, which needs an additional index structure to achieve the required delay. We close the section by presenting the missing index, called a *switchboard*. The switchboard must support a kind of reachability queries with a specific algorithm for enumeration under updates: we give a self-contained presentation of this scheme in the next section.

Preprocessing phase: shortcuts and partial evaluation. The first index structure that we precompute on our hybrid circuit C consists of a *shortcut function* to avoid wasting time in chains of $\boxtimes$-gates. For each $\boxtimes$-gate g, we precompute the one set-valued gate, called $\delta(g)$ which is not a $\boxtimes$-gate and which has a directed path to g going only through $\boxtimes$-gates. The function δ can clearly be computed in a linear-time bottom-up pass during the preprocessing, and it will never need to be updated (it does not depend on ν). For notational convenience, we extend δ by setting $\delta(g) := g$ for any set-valued gate g which is not a $\boxtimes$-gate.

The second index structure that we precompute is a *partial evaluation*, which depends on the valuation ν: it is a function ω_ν from the gates of C to $\{0, 1\}$ satisfying the following:

- For every Boolean gate g, we have $\omega_\nu(g) = \mathrm{V}_\nu(g)$.
- For every set-valued gate g, we have $\omega_\nu(g) = 1$ iff $\mathrm{S}_\nu(g)$ is non-empty.

The function ω_ν is intuitively an evaluation of the Boolean gates in the circuit, extended to the set-valued gates to determine whether their set is empty or not. We can easily compute ω_ν bottom-up from ν. Further, whenever ν is changed on a Boolean variable gate g, we can update ω_ν by recomputing it bottom-up on $\Delta(g)$. Formally:

▶ **Lemma 6.1.** *Given a hybrid circuit C of constant fan-in, given a valuation ν of C, we can compute ω_ν in linear time from ν and C. Further, for any $g \in C_{\mathrm{bvar}}$, letting ν' be the result of toggling the value of ν on g, we can update ω_ν to $\omega_{\nu'}$ in time $O(\Delta(g))$.*

Hence, we can compute ω_ν and δ in the preprocessing and maintain them under updates.

Enumeration phase. We can use the shortcut function and partial evaluation to enumerate the assignments in the set $\mathrm{S}_\nu(C)$ of our hybrid circuit C. Of course, if $\omega_\nu(g_0) = 0$ then we detect in constant time that there is nothing to enumerate. Otherwise, the enumeration scheme proceeds essentially like in [4]; to achieve the right delay bounds, it will need an additional index that we will present later. We start by enumerating $\mathrm{S}_\nu(g_0)$, and describe what happens when we try to enumerate $\mathrm{S}_\nu(g)$ for a set-valued gate g; we will always ensure that $\omega_\nu(g) = 1$. The base case is when g is a set-valued variable, in which case the only assignment to enumerate is $\{g\}$. There are three induction cases: $\times$-gates, $\boxtimes$-gates, and $\cup$-gates.

First, assume that g is a $\times$-gate. As C is homogenized, g has two inputs g_1 and g_2. Then we have $\mathrm{S}_\nu(g) = \mathrm{S}_\nu(g_1) \times_{\mathrm{rel}} \mathrm{S}_\nu(g_2)$. Hence, we can simply enumerate $\mathrm{S}_\nu(g)$ as the lexicographic product of $\mathrm{S}_\nu(g_1)$ and $\mathrm{S}_\nu(g_2)$. In particular, as $\omega_\nu(g) = 1$, we have $\omega_\nu(g_1) = \omega_\nu(g_2) = 1$, so neither set is empty. Formally, we have the following lemma:

▶ **Lemma 6.2.** *For any $\times$-gate g with inputs g_1 and g_2, if we can enumerate $\mathrm{S}_\nu(g_1)$ and $\mathrm{S}_\nu(g_2)$ with delay and memory respectively θ_1 and θ_2, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta_1 + \theta_2 + c$ for some constant c.*

Note that the constant c paid at the $\times$-gate is not a problem to achieve linear delay and memory, because it is paid at most $n - 1$ times when enumerating an assignment A of size n. Indeed, C is homogenized, so A is always split non-trivially at each $\times$-gate, and g is decomposable in $\nu(C)$, so the two sub-assignments never share any variable.

Second, assume that g is a $\boxtimes$-gate. As $\omega_\nu(g) = 1$, we clearly have $\mathrm{S}_\nu(g) = \mathrm{S}_\nu(\delta(g))$. Hence, we can simply follow the pointer to $\delta(g)$ and enumerate $\mathrm{S}_\nu(\delta(g))$. Intuitively, the cost of this operation can be covered by that of g, because $\delta(g)$ can no longer be a $\boxtimes$-gate.

▶ **Lemma 6.3.** *For any $\boxtimes$-gate g, if we can enumerate $\mathrm{S}_\nu(\delta(g))$ with delay and memory θ, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta + c$ for some constant c.*

Third, assume that g is a $\cup$-gate g. Naively, we can enumerate $\mathrm{S}_\nu(g)$ as the union of the $\mathrm{S}_\nu(g')$ for the inputs g' of g for which $\omega_\nu(g') = 1$ (this union is disjoint thanks to determinism). This is correct, but does not satisfy the delay bounds, because g' may be another $\cup$-gate. A more clever scheme is to to "jump" to the $\times$-gate or set-valued variable gates on which $\mathrm{S}_\nu(g)$ depends. Let us accordingly call *exits* the gates of these two types. The set $\mathrm{S}_\nu(g)$ can then be expressed as a union of $\mathrm{S}_\nu(g')$ for the exits g' that have a directed path of $\cup$-gates and $\boxtimes$-gates to g. We introduce definitions to "collapse" these paths.

The first definition collapses paths of $\boxtimes$-gates. There is a $\boxtimes$-*path* from a set-valued gate g' to a set-valued gate $g \neq g'$, written $g' \to_{\boxtimes}^{*} g$, if there is a directed path $g' = g_1 \to \cdots \to g_n = g$ in C such that $g_2, \ldots, g_{n-1}$ are all $\boxtimes$-gates. In particular, a wire (g', g) between set-valued gates implies $g' \to_{\boxtimes}^{*} g$ (take $n = 2$), and $\delta(g) \to_{\boxtimes}^{*} g$ whenever $\delta(g) \neq g$. When g is a $\cup$-gate, there are two cases, depending on ν. First, we may have $\omega_\nu(g_{n-1}) = 1$, and then $\omega_\nu(g') = 1$ and $\mathrm{S}_\nu(g')$ contributes to $\mathrm{S}_\nu(g)$: we call the path *live under ν*. Second, we may have $\omega_\nu(g_{n-1}) = 0$, and then $\mathrm{S}_\nu(g')$ does not contribute to $\mathrm{S}_\nu(g)$ via this path.

The second definition collapses paths of $\cup$-gates. An $\cup$-*path* from a set-valued gate g' to a set-valued gate $g \neq g'$ is a sequence $g' = g_1 \to_{\boxtimes}^{*} \cdots \to_{\boxtimes}^{*} g_n = g$ in C, where $g_2, \ldots, g_{n-1}$ are all $\cup$-gates and there is a $\boxtimes$-path between any two consecutive gates. The path is *live under ν* if there is a *live* $\boxtimes$-path under ν between any two consecutive gates.

We now use these definitions to express $\mathrm{S}_\nu(g)$ as a function of the set of *exits under ν* of g in C, written D_g^ν, which is the set of exits g' having a live $\cup$-path to g under ν in C:

▶ **Lemma 6.4.** *For any valuation ν and $\cup$-gate g, we have $\mathrm{S}_\nu(g) = \bigcup_{g' \in D_g^\nu} \mathrm{S}_\nu(g')$. Further, this union is disjoint and all its terms are nonempty.*

Hence, we can enumerate $\mathrm{S}_\mu(g)$ for a $\cup$-gate g by enumerating D_g^ν and the set $\mathrm{S}_\nu(g')$ for each g' in D_g^ν. Note that g' is an exit, i.e., a variable or a $\times$-gate; so we make progress.

▶ **Lemma 6.5.** *For any $\cup$-gate g, if we can enumerate D_g^ν with delay and memory c, and can enumerate $\mathrm{S}_\nu(g')$ for every $g' \in D_g^\nu$ with delay and memory θ, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta + c + c'$ for some constant c'.*

We have described our enumeration scheme in Lemmas 6.2, 6.3, and 6.5. The only missing piece is to enumerate, for each $\cup$-gate g, the set D_g^ν of exits under ν of g, with *constant* delay and memory. To do so, we will need additional preprocessing. We will rely on upwards-determinism, and extend the tree-based index of [3] to support updates. We first present an additional structure, called the *switchboard*, that we compute in the preprocessing; and we explain in the next section an indexing scheme that we perform on this structure.

Switchboard. Our third index component in the preprocessing is called the *switchboard*. It consists of a directed graph $B = (V, E)$ called the *panel*, which does not depend on ν (so it does not need to be updated), and a valuation $\beta_\nu : E \to \{0, 1\}$ called the *wiring*. The *panel* $B = (V, E)$ is defined as follows: V consists of all $\cup$-gates, $\times$-gates, and svar-gates, and $E \subseteq V \times V$ contains the edge $(\delta(g'), g)$ for each wire (g', g) of C such that g is a $\cup$-gate. This implies that the maximal fan-in of B is no greater than that of C, and it implies that B is a DAG. The *wiring* β_ν maps every edge (g', g) of B to 1 if there is a $\boxtimes$-path from g' to g in C which is live under ν, and 0 otherwise. We can use ω_ν to compute the switchboard, and to update it in time $O(\Delta(C))$ whenever ν is updated by toggling a gate of C_{bvar}. Formally:

▶ **Lemma 6.6.** *The switchboard can be computed in linear time given C and ν, and we can update it in time $O(\Delta(C))$ when toggling any gate in ν.*

We now explain how we use the switchboard to enumerate, given a $\cup$-gate g, the set D_g^ν of the exits g' having a *live* $\cup$-path to g under ν. In terms of the switchboard, we must enumerate the exits g' that have a path to g in B whose edges are all mapped to 1 by β_ν. Hence, we must solve the following enumeration task on the switchboard: letting $\beta_\nu(B)$ be the DAG of edges of B mapped to 1 by β_ν, we are given a gate g of B, and we must enumerate all exit gates g' of B (i.e., the $\times$-gates or svar-gates) that have a directed path to g in $\beta_\nu(B)$. Further, we must be able to handle updates on $\beta_\nu(B)$, as given by updates on ν. Fortunately, thanks to upwards-determinism, this problem is easier than it looks:

▶ **Claim 6.7.** *For any valuation ν of the hybrid circuit C, the DAG $\beta_\nu(B)$ is a forest.*

▶ **Example 6.8.** Figure 4 describes the switchboard for the hybrid circuit C of Figure 3. The edges of the switchboard correspond to $\boxtimes$-paths. The switchboard itself is not a forest; however, for every valuation of C, the $\boxtimes$-paths that are live must always form a forest.

Thus, what we need is a constant-delay reachability index on forests that can be updated efficiently when adding and removing edges to the forest. This is the focus of the next section.

7 Reachability Indexing under Updates

In this section, we present our indexing scheme for reachability on forests under updates. The construction in this section is independent from what precedes. For convenience, we will orient the edges of the forest downwards, i.e., the reverse of the previous section (so g is the parent of g' in the forest if there is an edge from g' to g in the switchboard). We first define the problem and state the enumeration result, and then sketch the proof.

Definitions and main result. A *reachability forest* $F = (V, E, X)$ is a directed graph (V, E) where V is the *vertex set*, $E \subseteq V \times V$ are the *edges*, and $X \subseteq V$ is a subset of vertices called *exits*. When $(v, v') \in E$, we call v a *parent* of v', and v' a *child* of v. We impose three requirements on F: (i.) the graph (V, E) is a forest, i.e., each vertex of V has at most one parent; (ii.) there is a constant *degree bound* $c \in \mathbb{N}$ such that every vertex has at most c

children; (iii.) every exit $v \in X$ is a *leaf*, i.e., a vertex with no children. We will call *trees* the connected components of F. For convenience, we assume that F is *ordered*, i.e., there is some total order $<$ on the children of every node.

Given a reachability forest $F = (V, E, X)$ and a vertex $v \in V$, we write $\mathrm{reach}(v)$ for the set of exits reachable in F from v, i.e., the vertices of X to which v has a directed path. These are the sets that we wish to enumerate efficiently, allowing two kinds of *updates* on the edges E of F. First, a *delete operation* is written $-E'$ for a set $E' \subseteq E$, and $F = (V, E, X)$ is updated to $F - E' := (V, E \setminus E', X)$; it is still a reachability forest. Second, an *insert operation* is written $+E'$ for some $E' \subseteq V \times V$, and we require that the update result $F + E' := (V, E \cup E', X)$ still satisfies the three requirements above (with the same degree bound). In terms of the order $<$ on children, when we remove edges, we take the restriction of $<$ in the expected way, and when we insert edges, we add each new child at an arbitrary position in $<$. We then introduce *ancestry* to measure the impact of updates (analogously to dependency size): the *ancestry* $\mathcal{A}_F(v)$ of $v \in V$ is the set of vertices of F that have a directed path to v, and the *ancestry* $\mathcal{A}_F(E')$ for $E' \subseteq V \times V$ is $\bigcup_{(v,w)\in E'} \mathcal{A}_F(v)$. We then have:

▶ **Theorem 7.1.** *Given a reachability forest F, there is an enumeration algorithm with linear-time preprocessing such that: (i.) given any $v \in V$, we can enumerate* $\mathrm{reach}(v)$ *with constant delay and memory; (ii.) given an update $\pm E'$, we can apply it (replacing F by $F \pm E'$ and updating the index) with update time in $O(\mathcal{A}_F(E'))$.*

Note how we can insert (or delete) many edges at the same time, paying only once the price $\mathcal{A}_F(E')$: this point is used in the proof of Theorem 5.4 to bound the total cost of each update on the circuit. We sketch the proof of Theorem 7.1 in the rest of this section.

Construction for Theorem 7.1. Our index structure follows the one used to prove Proposition F.4 of [3]: it maps every $v \in V$ to a pointer $\mathsf{first}_F(v)$ and a pointer $\mathsf{last}_F(v)$, called the first and last *pointer*; and maps every exit $v \in X$ to a pointer $\mathsf{next}_F(v)$ called the next *pointer*. These pointers are defined using the order $<'$ given by a preorder traversal of F following $<$. Specifically, $\mathsf{first}_F(v)$ is the first exit $v' \in \mathrm{reach}_F(v)$ according to $<'$, and $\mathsf{last}_F(v)$ is the last such exit; if $\mathrm{reach}_F(v) = \emptyset$ then both pointers are null. Now, $\mathsf{next}_F(v)$ for $v \in X$ is the exit $v' \in X$ in the tree of v which is the successor of v according to $<'$; if v is the last exit of its tree, then $\mathsf{next}_F(v)$ is null. If we know these pointers, we can enumerate $\mathrm{reach}_F(v)$ for any $v \in V$ with constant delay and memory as in [3]: if $\mathsf{first}_F(v)$ is null then there is nothing to enumerate, otherwise start at $v_- := \mathsf{first}_F(v)$, memorize $v_+ := \mathsf{last}_F(v)$, and enumerate the reachable exits following the next pointers from v_- until reaching v_+. Hence, to conclude the proof of Theorem 7.1, it suffices to compute and update these pointers efficiently:

▶ **Lemma 7.2.** *Given a reachability forest F, we can compute the* first, last, *and* next *pointers of all vertices in time $O(|F|)$. Further, for any update $\pm E'$, we can apply it and update the pointers in time $O(\mathcal{A}_F(E'))$.*

Proof sketch. The first and last pointers are computed bottom-up in linear time: for a leaf v, they either point to v if $v \in X$ or to null otherwise; for an internal vertex v, we set $\mathsf{first}_F(v)$ as $\mathsf{first}_F(v')$ for the smallest child v' of v in the order $<'$ with a non-null first pointer (or null if all first pointers of children are null), and we set $\mathsf{last}_F(v)$ analogously, using the last pointer of the largest child of v in the order $<'$ for which the last pointer is non-null. Further, given an update $\pm E'$, the first and last pointers need only to be updated in $\mathcal{A}_F(E')$, and we can recompute them there with the same bottom-up scheme.

The next pointers are also computed bottom-up in linear time: at each internal vertex v, we go over its children and stitch together the sequences of next pointers of their subtrees.

Specifically, when $\mathsf{last}_F(v_1)$ is not null for a child v_1, we find the next child v_2 for which $\mathsf{first}_F(v_2)$ is not null, and set $\mathsf{next}_F(\mathsf{last}_F(v_1)) := \mathsf{first}_F(v_2)$. Again, for an update $\pm E'$, we recompute the next pointers by processing $\mathcal{A}_F(E')$ bottom-up in a similar fashion. ◀

8 Applications

We have finished the proof of our main result (Theorem 3.1), and now explain how it applies to query languages motivated by applications. Specifically, we show how to extend our techniques to support *aggregate* queries in arbitrary semirings, following the ideas of semiring provenance [21] and provenance circuits [17]. We then extend this to *group-by queries*, and last explain how updates are useful to support *parameterized queries*. Throughout this section, unlike the rest of the paper, we only study MSO queries with free first-order variables.

Aggregate queries. We will describe aggregation operators using a general structure called a *semiring* (always assumed to be commutative). It consists of a *set* K (finite or infinite), two binary operations $\oplus$ and $\otimes$, and distinguished elements $0_K, 1_K \in K$. We require that $(K, \oplus)$ and $(K, \otimes)$ are commutative monoids with neutral elements respectively 0_K and 1_K; that $\otimes$ distributes over $\oplus$, and that 0_K is absorptive for $\otimes$, i.e., $0_K \otimes a = 0_K$ for all $a \in K$. We always assume that evaluating $\oplus$ or $\otimes$ take constant time, and that elements from K take constant space. Examples of semirings include the natural numbers $\mathbb{N}$ with usual addition and product (assumed to take unit time in the RAM model); or the *security semiring* [20], the *tropical semiring* [17], etc. Note that sets of assignments with union and relational product are also a semiring, but one that does not satisfy our constant-space assumption.

To define aggregation in a semiring K on a tree T, we consider a mapping $\rho : T \to K$ giving a value in K to each node. We extend ρ to tuples $\mathbf{b}$ of T by setting $\rho(\mathbf{b}) := \bigotimes_{n \in \mathbf{b}} \rho(n)$; to assignments A on some first-order variable set $\mathbf{x}$ by setting $\rho(A) := \bigotimes_{\langle x_i : n \rangle \in A} \rho(n)$; and to sets S of assignments by setting $\rho(S) := \bigoplus_{A \in S} \rho(A)$. An *aggregate query* on Γ-trees consists of a semiring K (satisfying our assumptions) and of a MSO query $Q(\mathbf{x})$ on Γ-trees. Given a Γ-tree T and a mapping $\rho : T \to K$, the *aggregate output* $Q_\rho(T)$ of Q on T under ρ is $\rho(Q(T))$, where $Q(T)$ is the output of Q on T as we studied so far, i.e., the set of assignments A such that $T \models Q(A)$. Aggregate MSO queries on trees were already studied, e.g., by Arnborg and Lagergren [7], but our techniques allow us to handle updates:

▶ **Theorem 8.1.** *For any aggregate query $Q(\mathbf{x})$ on Γ-trees with semiring K, given a Γ-tree T and mapping $\rho : T \to K$, we can compute $Q_\rho(T)$ in time $O(|T|)$, and recompute it in time $O(\log |T|)$ after any update that relabels a node of T or that changes $\rho(n)$ for a node n of T.*

Proof sketch. We adapt hybrid circuits by replacing set-valued gates by K-valued gates. Now, the set $\mathrm{S}_\nu(g)$ captured by a gate g under a Boolean valuation ν is an element of K, so we can simplify our linear-time preprocessing by making ω_ν compute exactly $\mathrm{S}_\nu(g)$ for each gate g. We can then handle updates to ν as before, and handle updates to ρ by recomputing ω_ν bottom-up. All of this still relies on the balancing lemma (Lemma 4.3). ◀

One important application of this result is *maintaining* the number of query answers under updates, a question left open by [24]. We answer the question for relabeling updates (and in the set semantics), using the semiring $\mathbb{N}$ and mapping each node to 1 with ρ:

▶ **Corollary 8.2.** *For any MSO query $Q(\mathbf{x})$ on Γ-trees, given a Γ-tree T, we can compute the number $|Q(T)|$ of answers of Q on T in time $O(|T|)$, and we can update it in time $O(\log |T|)$ after a relabeling of T.*

However, we can also use Theorem 8.1 for more complex aggregation semirings:

▶ **Example 8.3.** Let $\Gamma = \{A, B\}$, let $Q(x)$ be a MSO query with one variable that selects some tree nodes (e.g., select the B-labeled nodes which are descendants of some A-labeled node), let (T, λ) be a Γ-tree, and let χ be a function that maps each node of T to an element of the set $\mathbb{D}$ of floating-point numbers (with fixed precision). We can compute in linear time the *average* of $\chi(n)$ for the nodes n such that $T \models Q(n)$, and update it in logarithmic time when relabeling a node of T or changing a value of χ. This follows from Theorem 8.1: we use the semiring of pairs in $\mathbb{N} \times \mathbb{D}$ and the mapping $\rho : n \mapsto (1, \chi(n))$ to compute and maintain the number of selected nodes and the sum of their χ-images, from which we can deduce the average in constant time.

Group-by. We have adapted our techniques to show results for aggregate queries under updates. However, supporting updates is also useful for *group-by queries*. A *group-by query* consists of a MSO query $Q(\mathbf{x}, \mathbf{y})$ on Γ-trees with two tuples of first-order variables, and of a semiring K. A *group* on a Γ-tree T is a set of tuples $\mathcal{G}(\mathbf{b}) := \{(\mathbf{b}, \mathbf{c}) \mid T \models Q(\mathbf{b}, \mathbf{c})\}$ for some tuple $\mathbf{b}$ of nodes of T. The *output* $Q_\rho(T)$ of Q on T under a mapping $\rho : T \to K$ contains one pair $(\mathbf{b}, \rho(\mathcal{G}(\mathbf{b})))$ for each tuple $\mathbf{b}$ such that $\mathcal{G}(\mathbf{b})$ is non-empty.

▶ **Example 8.4.** Consider a MSO query $Q(x, y)$ and the semiring $\mathbb{N}$. The output of Q on a Γ-tree T under a mapping ρ contains one pair per $n \in T$, annotated with the sum of $\rho(n')$ for $n' \in T$ such that $T \models Q(n, n')$, where we exclude the nodes n for which the sum is empty.

▶ **Theorem 8.5.** *For any group-by query $Q(\mathbf{x}, \mathbf{y})$ and semiring K, given a Γ-tree T and $\rho : T \to K$, we can enumerate $Q_\rho(T)$ with linear-time preprocessing and delay in $O(\log |T|)$*

Proof sketch. We use two enumeration structures. First, we prepare the structure of Theorem 8.1 for $Q(\mathbf{x}, \mathbf{y})$ but writing the valuation of $\mathbf{x}$ as part of the tree label. Second, we enumerate the non-empty groups with constant delay using Theorem 3.1 on $\exists \mathbf{y}\, Q(\mathbf{x}, \mathbf{y})$. For each tuple $\mathbf{b}$ in the output of the second structure, letting $\mathcal{G}(\mathbf{b})$ be the corresponding group, we update the first structure to compute $\rho(\mathcal{G}(\mathbf{b}))$ in time $O(\log |T|)$. ◀

Parameterized queries. We conclude by presenting another kind of practical queries that we can support thanks to updates. A *parameterized* MSO query $Q(\mathbf{x}, \mathbf{y})$ on Γ-trees has two kinds of first-order variables, like group-by: we call $\mathbf{x}$ the *parameters*. The idea is that, given a Γ-tree T, the user chooses a tuple $\mathbf{b}$ to instantiate the parameters $\mathbf{x}$, and we must enumerate efficiently the results of $Q(\mathbf{b}, \mathbf{y})$; however the user can change their mind and modify $\mathbf{b}$ to change the value of the parameters. We know by Theorem 3.1 that we can support these queries efficiently: after a linear-time preprocessing of T, we can enumerate the results of $Q(\mathbf{b}, \mathbf{y})$ with constant delay; and we can react to changes to $\mathbf{b}$ in time $O(\log |T|)$ by performing an update on the enumeration structure.

9 Conclusion

We have studied MSO queries on trees under *relabeling* updates, and shown how to enumerate their answers with linear-time preprocessing, delay and memory linear in each valuation, and update time logarithmic in the input tree. We have shown this by extending our circuit-based approach [4] to hybrid circuits, and we have deduced consequences for practical query languages, in particular for efficient aggregation. Our results have another technical property that we have not presented in the main text: like those of [24], they are also tractable in the size of the query when representing it as a deterministic automaton.

The main direction for future work would be to extend our result to support insertions and deletions of leaves, like [24], hopefully preserving our improved bounds: while deletions can be emulated with relabelings, insertions are trickier. Such a result was very recently shown in [25] for the case of words rather than trees. We believe that many of our constructions on trees should adapt to insertions and deletions. The main challenge is to extend Lemma 4.3, which we believe to be an interesting question in its own right: the technique of [10] may be applicable here, although it would lead to an $O(\log^2 n)$ update time.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

2 Antoine Amarilli. *Leveraging the structure of uncertain data*. PhD thesis, Télécom ParisTech, 2016. URL: `https://tel.archives-ouvertes.fr/tel-01345836`.

3 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. *CoRR*, abs/1702.05589, 2017. `arXiv:1702.05589`.

4 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 111:1–111:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.111`.

5 Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. *CoRR*, abs/1709.06185, 2017. `arXiv:1709.06185`.

6 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances (extended version). *CoRR*, abs/1511.08723, 2015. `arXiv:1511.08723`.

7 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2), 1991.

8 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

9 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. `doi:10.1007/978-3-540-74915-8_18`.

10 Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004. `doi:10.1145/1042046.1042050`.

11 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318. ACM, 2017. `doi:10.1145/3034786.3034789`.

12 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017,*

Venice, Italy, volume 68 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ICDT.2017.8.

13 Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998. doi:10.1137/S0097539795289859.

14 Thomas Colcombet. A combinatorial theorem for trees. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 901–912. Springer, 2007. doi:10.1007/978-3-540-73420-8_77.

15 Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

16 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001. doi:10.3166/jancl.11.11-34.

17 Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 201–212. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.22.

18 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007. doi:10.1145/1276920.1276923.

19 David Eppstein and Denis Kurz. K-best solutions of MSO problems on tree-decomposable graphs. *CoRR*, abs/1703.02784, 2017. arXiv:1703.02784.

20 J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: queries and provenance. In Maurizio Lenzerini and Domenico Lembo, editors, *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 271–280. ACM, 2008. doi:10.1145/1376916.1376954.

21 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007. doi:10.1145/1265530.1265535.

22 Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2), 2011. doi:10.2168/LMCS-7(2:20)2011.

23 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):25:1–25:12, 2013. doi:10.1145/2528928.

24 Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 67:1–67:10. ACM, 2014. doi:10.1145/2603088.2603137.

25 Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, 2018. To appear.

26 Milos Nikolic and Dan Olteanu. Incremental maintenance of regression models over joins, 2017. arXiv:1703.07484.

27 Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015. doi:10.1145/2656335.

28 Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 13–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.13.

29 Volker Strassen. Vermeidung von divisionen. *Journal für die reine und angewandte Mathematik*, 264, 1973. URL: https://eudml.org/doc/151394.

30 Teruji Sugaya, Masaaki Nishino, Norihito Yasuda, and Shin-Ichi Minato. Fast compilation of s-t paths on a graph for counting and enumeration. In *AMBN*, volume 73 of *PMLR*, 2017. URL: http://proceedings.mlr.press/v73/teruji-sugaya17a.html.

31 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016. arXiv:1605.05102.