Query Automata*

Frank Neven† Thomas Schwentick‡

Abstract

A main task in document transformation and information retrieval is locating subtrees satisfying some pattern. Therefore, unary queries, i.e., queries that map a tree to a set of its nodes, play an important role in the context of structured document databases. We want to understand how the natural and well-studied computation model of tree automata can be used to compute such queries. We define a query automaton (QA) as a deterministic two-way finite automaton over trees that has the ability to select nodes depending on the state and the label at those nodes. We study QAs over ranked as well as over unranked trees. Unranked trees differ from ranked ones in that there is no bound on the number of children of nodes. We characterize the expressiveness of the different formalisms as the unary queries definable in monadic second-order logic (MSO). Surprisingly, in contrast to the ranked case, special stay transitions had to be added to QAs over unranked trees to capture MSO. We establish the complexity of their non-emptiness, containment, and equivalence problem to be complete for EXPTIME.

1 Introduction

The popularity of the document specification language XML [13] immensely increased the amount of research concerning structured document databases

*A preliminary version of this work was presented at the 18th ACM Symposium on Principles of Database Systems, Philadelphia, PA, 1999.

†Research Assistant of the Fund for Scientific Research, Flanders. Limburgs Universitair Centrum. E-mail: frank.neven@luc.ac.be.

‡Johannes Gutenberg-Universität Mainz, Institut für Informatik. E-mail: tick@informatik.uni-mainz.de

[1]. Such documents are usually abstracted by labeled ordered trees which in turn are modeled by context-free (CFG) or extended context-free grammars (ECFG) [2, 7, 15, 22, 23, 26, 37]. ECFGs, in particular, are context-free grammars that allow arbitrary regular expressions over grammar symbols on the right-hand side of productions. Such grammars form adequate abstractions of XML Document Type Definitions (DTDs). A crucial difference between CFGs and ECFGs, is that derivation trees of the former are ranked, in the sense that the number of children of a node is bounded by some constant, while those of the latter are not. In Figure 1, an example is depicted of an XML document corresponding to the DTD of Figure 2. Figure 3 and 4 show the corresponding abstractions. A main task in document transformation and information retrieval is locating subtrees satisfying some pattern [5, 6, 12, 27, 28, 33, 34]. Therefore, unary queries, i.e., queries that map a tree to a set of its nodes, play an important role in the context of structured document databases.

Our goal is to understand how the natural and well-studied computation model of tree automata [20, 45] on both ranked and unranked trees, can be used to compute such unary queries. We abstract away from the grammar by considering documents simply as ranked or unranked trees over some alphabet. This is no loss of generality, as tree automata can easily determine whether the input tree is a derivation tree of a given (E)CFG [20, 32, 36, 35]. We define a *query automaton* (QA) as a two-way deterministic finite automaton over trees that can select nodes depending on the state and the label at those nodes. A QA can compute queries in a natural way: the result of a QA on a tree consists of all those nodes that are selected during the computation of the QA on that tree.

First, we stress that the query automata we consider are quite different from the tree acceptors studied in formal language theory [20]. For one thing, two-way tree automata are equivalent to one-way ones [31], but it is not so difficult to see that query automata are not equivalent to bottom-up ones. Indeed, a bottom-up QA, for example, cannot compute the query "select all leaves if the root is labeled with σ", simply because it cannot know the label of the root when it starts at the leaves. A second difference is that we consider both ranked and unranked trees. Unranked trees only recently received new attention in the context of SGML and XML. Based on work of Pair and Quere [38] and Takahashi [42], Murata defined a bottom-up automaton model for unranked trees [32]. This required describing transition functions for an arbitrary number of children. Murata's approach is the following: a node is

```
<bibliography>
    <book>
        <author>
            S. Abiteboul
        </author>
        <author>
            R. Hull
        </author>
        <author>
            V. Vianu
        </author>
        <title>
            Foundations of Databases
        </title>
        <publisher>
            Addison-Wesley
        </publisher>
        <year>
            1995
        </year>
    </book>
    <article>
        <author>
            E. Codd
        </author>
        <title>
            A Relational Model of Data for Large Shared Data Banks
        </title>
        <journal>
            Communications of the ACM
        </journal>
        <year>
            1970
        </year>
    </article>
</bibliography>
```

Figure 1: Example of an XML document describing bibliographic information.

```
<!ELEMENT bibliography (book | article)+>

<!ELEMENT article      (author+, title, journal, year)>

<!ELEMENT book         (author+, title, publisher, year)>

<!ELEMENT author       PCDATA>

<!ELEMENT title        PCDATA>

<!ELEMENT journal      PCDATA>

<!ELEMENT year         PCDATA>

<!ELEMENT publisher    PCDATA>
```

Figure 2: A DTD for the XML document in Figure 1

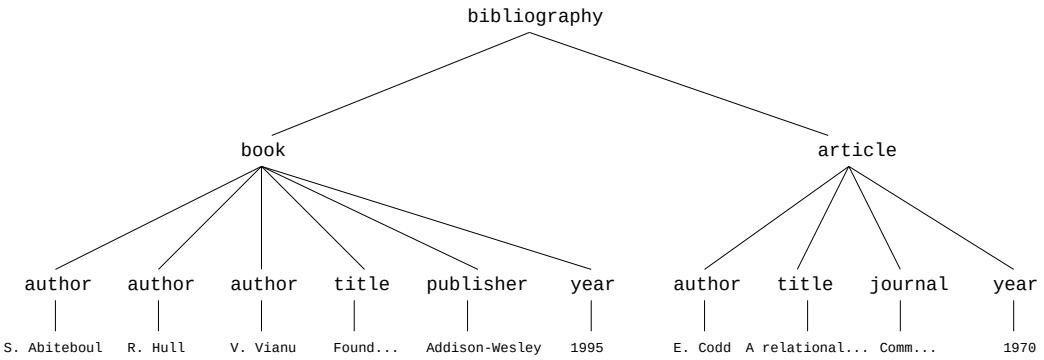


Figure 3: Tree representation of the XML document in Figure 1.

$$
\begin{array}{lcl}
\text{bibliography} & \longrightarrow & (\text{book} + \text{article})^+ \\
\text{article} & \longrightarrow & \text{author}^+ \cdot \text{title} \cdot \text{journal} \cdot \text{year} \\
\text{book} & \longrightarrow & \text{author}^+ \cdot \text{title} \cdot \text{publisher} \cdot \text{year} \\
\text{author} & \longrightarrow & (a + \cdots + z)^+ \\
\text{title} & \longrightarrow & (a + \cdots + z)^+ \\
\text{journal} & \longrightarrow & (a + \cdots + z)^+ \\
\text{year} & \longrightarrow & (a + \cdots + z)^+ \\
\text{publisher} & \longrightarrow & (a + \cdots + z)^+
\end{array}
$$

Figure 4: An extended CFG representing the DTD of Figure 2.

assigned a state by checking the sequence of states assigned to its children for membership in a regular language. In this way, the "infinite" transition function is represented in a finite way. Brüggemann-Klein, Murata and Wood initiated an extensive study of tree automata over unranked trees [9]. They showed that many results carry over to the unranked case. Surprisingly, in the context of query automata there is a discrepancy between the ranked and the unranked case. Indeed, we show that in the unranked case various QA formalisms accept the same class of tree languages,[1] while not computing the same class of queries. This indicates a substantial difference between (*i*) looking at automata from a formal language point of view (i.e., for defining tree languages) and looking at automata from a database point of view (i.e., for computing queries) (*ii*) automata on ranked and unranked trees.

In Section 2, after introducing the necessary definitions, we first recall the proof of Büchi's Theorem [10] stating that a string language is regular if and only if it is definable in MSO, and then generalize this to query automata on strings in Section 3. Here, a query automaton on strings is a two-way deterministic automaton extended with a selection function. This approach allows us to recall some important proof techniques in an easy setting, which then later will be generalized to obtain our main results. These techniques can be summarized as follows: (*i*) the definition of an automaton in MSO by, essentially, guessing states and verifying their consistency with the transition function; (*ii*) capturing the behavior of two-way automata by means of behavior functions; and (*iii*) computing MSO types by automata. We conclude this section by recalling how bottom-up tree automata can compute MSO

[1] A *tree language* is a set of trees. We say that a QA *accepts a tree* if the underlying tree automaton accepts it.

types.

In Section 4, we consider QAs over ranked trees, i.e., trees with a fixed bound on the number of children that a vertex might have. A QA^r (r stands for *ranked*) is a two-way deterministic tree automaton[2] as defined by Moriya [31] extended with a selection function. We show that these automata can compute exactly the unary queries definable in monadic second-order logic (MSO).

Next, in Section 5, we consider automata on unranked trees. A first approach to define query automata for unranked trees, is to add a selection function to the two-way deterministic tree automata over unranked trees defined by Brüggemann-Klein, Murata and Wood [9]. We denote these automata with QA^u (u stands for *unranked*). Although these automata can accept all recognizable tree languages, they cannot even compute all unary queries definable in first-order logic. Intuitively, when the automaton makes a down transition at some node $\mathbf{v}$, it assigns a state to every child of $\mathbf{v}$; although every child knows its own state, it cannot know in general which states are assigned to its siblings. This means that in the unranked case very little information can be passed from one sibling to another. To resolve this, we introduce stay transitions where a two-way string-automaton reads the string formed by the states at the children of a certain node, and then outputs for each child a new state. An automaton making at most one stay transition (or, equivalently, a constant number of stay transitions) for the children of each node is a *strong* QA^u (SQA^u). We show that these automata compute exactly all MSO-definable queries. Thus, while QA^u and SQA^u recognize the same tree languages, they do not compute the same queries. The restriction on the number of stay transitions is necessary. Without any such restriction, SQA^us could simulate linear space Turing Machines.

Testing non-emptiness, containment, and equivalence of queries are fundamental operations in the field of query optimization [3]. While these problems are in general usually undecidable, their language-theoretic counter parts are well-known to be decidable. Therefore, we investigate in Section 6 the complexity of the following three problems: (i) *Given a QA, is there a tree for which there is a node that is selected?* (non-emptiness); (ii) *Given two QAs, is the query computed by one contained in the query computed by the other?* (containment); and (iii) *Given two QAs, do they compute the*

[2]These automata are very different from the (alternating) tree-*walking* automata used in, e.g., [46].

same query? (equivalence). One cannot hope to do better than EXPTIME for these decision problems, as non-emptiness of two-way deterministic tree automata over ranked trees, i.e., even without selecting nodes, is already complete for EXPTIME.

We show that the non-emptiness, the containment, and the equivalence problem of all query automata studied in this paper are in EXPTIME.

We present some concluding remarks in Section 7.

2 Basics of logic and automata on strings and trees

In this section we recall some basic facts on MSO and use them to reprove Büchi's Theorem [10] stating that a string language is regular if and only if it is definable in MSO. This approach allows us to introduce various techniques related to MSO and automata in an easy setting which we will generalize to obtain expressiveness results for query automata. Specifically, we recall how Ehrenfeucht games facilitate reasoning on MSO-equivalence types. These types constitute the building blocks of the simulation of MSO formulas in later sections. Finally, we define bottom-up tree automata and reprove the generalization of Büchi's Theorem to trees obtained by Doner, Thatcher and Wright [16, 43].

Before we start, we make the following conventions. We denote by $\mathbb{N}$ the set of positive natural numbers. Further, if S is a set then we denote by $|S|$ its cardinality.

2.1 Monadic second-order logic

A *vocabulary* τ is a finite nonempty set of constant symbols and relation names with associated arities. As usual, a *τ-structure* $\mathcal{A}$ consists of a finite set $\mathrm{dom}(\mathcal{A})$, the domain of $\mathcal{A}$, together with

- an interpretation $R^{\mathcal{A}} \subseteq \mathrm{dom}(\mathcal{A})^r$ for each relation name R in τ; here, r is the arity of R; and
- an interpretation $c^{\mathcal{A}} \in \mathrm{dom}(\mathcal{A})$ for each constant symbol in τ.

When τ is clear from the context or is not important, we just say structure rather than τ-structure. Sometimes, when the structure $\mathcal{A}$ is understood, we abuse notation and write R for the relation $R^{\mathcal{A}}$.

Example 2.1 Let τ be the vocabulary consisting of a binary relation symbol E and two constants min and max. Let w be the τ-structure with domain $\mathrm{dom}(w) := \{1, \ldots, n\}$, $E^w := \{(i, i+1) \mid i \in \{1, \ldots, n-1\}\}$, $\min^w := 1$, and $\max^w := n$. Then w represents a chain of length n, where min and max are interpreted by the first and the last element, respectively. ■

Monadic second-order logic (MSO) allows the use of *set variables* ranging over sets of domain elements, in addition to the individual variables ranging over the domain elements themselves as provided by first-order logic. We will assume some familiarity with this logic and refer the unfamiliar reader to the book of Ebbinghaus and Flum [17] or the chapter by Thomas [45].

Example 2.2 We give an example of an MSO formula. As usual we denote set variables by capital letters and first-order variables by small letters. Let φ be the following MSO sentence over the vocabulary of Example 2.1:

$$\begin{aligned}(\exists X)\big(X(\min) \wedge \\ &(\forall x)(\forall y)((X(x) \wedge E(x,y)) \to \neg X(y)) \\ &(\forall x)(\forall y)((\neg X(x) \wedge E(x,y)) \to X(y)) \\ &\qquad\qquad\qquad\qquad \wedge \neg X(\max)\big).\end{aligned}$$

This formula defines the chains of even length. Indeed, for each chain, the set variable X can only be interpreted by the set of elements occurring on odd positions and the formula becomes true only if the last element does not belong to X. ■

In the following we will make use of some basic facts about MSO. For a tuple $\bar{a} = a_1, \ldots, a_n$ of elements in $\mathcal{A}$, we write $(\mathcal{A}, \bar{a})$ to denote the finite structure that consists of $\mathcal{A}$ with $a_1, \ldots, a_n$ as distinguished constants. Let $\mathcal{A}$ and $\mathcal{B}$ be two structures, let $\bar{a}$ and $\bar{b}$ be tuples of elements in $\mathcal{A}$ and $\mathcal{B}$, respectively, and let k be a natural number. Then we write $(\mathcal{A}, \bar{a}) \equiv_k^{\mathrm{MSO}} (\mathcal{B}, \bar{b})$ and say that $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ are $\equiv_k^{\mathrm{MSO}}$-*equivalent*, if for each MSO sentence φ of quantifier depth at most k it holds

$$(\mathcal{A}, \bar{a}) \models \varphi \quad \Leftrightarrow \quad (\mathcal{B}, \bar{b}) \models \varphi.$$

That is, $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ cannot be distinguished by MSO sentences of quantifier depth (at most) k. It readily follows from the definition that $\equiv_k^{\mathrm{MSO}}$ is

an equivalence relation. Moreover, $\equiv_k^{\mathrm{MSO}}$-equivalence can be nicely characterized by Ehrenfeucht games.

The *k-round MSO game* on two structures $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$, denoted by $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, is played by two players, the *spoiler* and the *duplicator*, in the following way. In each of the k rounds the spoiler decides to make a *point move* or a *set move*. If the i-th move is a point move, then the spoiler selects an element $c_i \in \mathrm{dom}(A)$ or $d_i \in \mathrm{dom}(B)$ and the duplicator answers by selecting one element of the other structure. When the i-th move is a set move, the spoiler chooses a set $P_i \subseteq \mathrm{dom}(A)$ or $Q_i \subseteq \mathrm{dom}(B)$ and the duplicator chooses a set in the other structure. After k rounds there are elements $c_1, \ldots, c_\ell$ and $d_1, \ldots, d_\ell$ that were chosen in the point moves in $\mathrm{dom}(A)$ and $\mathrm{dom}(B)$ respectively and there are sets $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ that were chosen in the set moves in $\mathrm{dom}(A)$ and $\mathrm{dom}(B)$, respectively. The duplicator now wins this play if the mapping which maps c_i to d_i is a partial isomorphism from $(\mathcal{A}, \bar{a}, P_1, \ldots, P_n)$ to $(\mathcal{B}, \bar{b}, Q_1, \ldots, Q_n)$. That is, for all i and j, $c_i \in P_j$ iff $d_i \in Q_j$, and for every atomic formula $\varphi(\bar{x})$ containing no set variable, $\mathcal{A} \models \varphi[\bar{c}, \bar{a}]$ iff $\mathcal{B} \models \varphi[\bar{d}, \bar{b}]$.

We say that the duplicator has a *winning strategy* in $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, or shortly that she wins $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, if she can win each play no matter which choices the spoiler makes.

The following fundamental proposition is well known (see, e.g., [17] for a proof).

Proposition 2.3 *The duplicator wins* $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$ *if and only if*

$$(\mathcal{A}, \bar{a}) \equiv_k^{\mathrm{MSO}} (\mathcal{B}, \bar{b}).$$

It is well known that, for each k, the relation $\equiv_k^{\mathrm{MSO}}$ has only a finite number of equivalence classes. We denote the set of these classes by Φ_k and refer to the elements of Φ_k by $\equiv_k^{\mathrm{MSO}}$*-types*. We denote by $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ the $\equiv_k^{\mathrm{MSO}}$-type of a structure $\mathcal{A}$ with the elements in $\bar{a}$ as distinguished constants; thus, $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ is the equivalence class of $(\mathcal{A}, \bar{a})$ w.r.t. $\equiv_k^{\mathrm{MSO}}$. By $\tau_k^{\mathrm{MSO}}(\mathcal{A})$ we denote the $\equiv_k^{\mathrm{MSO}}$-type of the structure $\mathcal{A}$ without distinguished elements. It is often useful to think of $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ as the set of MSO-sentences of quantifier depth k that hold in $(\mathcal{A}, \bar{a})$. That is, we also view $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ as the set $\{\varphi \mid (\mathcal{A}, \bar{a}) \models \varphi\}$ of MSO sentences of quantifier depth k. As $\equiv_k^{\mathrm{MSO}}$ is finite, upon logical equivalence, there are only a finite number of MSO sentences of quantifier depth k.

Equivalence types will be the main tool to simulate MSO formulas by automata. To illustrate their usage, we will recall how they can be employed to prove Büchi's Theorem [10].

2.2 Regular string languages

In the following Σ denotes a finite alphabet. A *string* $w = \sigma_1 \cdots \sigma_n$ over Σ is a sequence of Σ-symbols. We denote the length of w by $|w|$ and for each $i \in \{1, \ldots, |w|\}$, we denote σ_i by w_i. We refer to $\{1, \ldots, |w|\}$ as the set of *positions* of w.

To define sets of strings by MSO formulas, we associate to each string w over Σ, a finite structure with domain $\{1, \ldots, |w|\}$, denoted by $\mathrm{dom}(w)$, over the binary relation symbol $<$, and the unary relation symbols $(O_\sigma)_{\sigma \in \Sigma}$. The interpretation of $<$ is the obvious one, and for each $\sigma \in \Sigma$, O_σ is the set of positions labeled with a σ, i.e., $O_\sigma = \{i \mid w_i = \sigma\}$. In the following, we will make no distinction between the string w and the relational structure that corresponds to it.

For each k and each k-type $\theta \in \Phi_k$, we fix some string $w(\theta)$ with

$$\tau_k^{\mathrm{MSO}}(w(\theta)) = \theta.$$

A *nondeterministic finite automaton* M (NFA) over Σ is a tuple $(S, \Sigma, \delta, I, F)$ where S is finite set of states, $\delta : S \times \Sigma \to 2^S$ is the transition function, $I \subseteq S$ is the set of initial states, and $F \subseteq S$ is the set of final states. We denote the canonical extension of the transition function to strings by δ^*. A string $w \in \Sigma^*$ is accepted by M if $\delta^*(s_0, w) \in F$ for an $s_0 \in I$. The language accepted by M, denoted by $L(M)$, is defined as the set of all strings accepted by M. The size of M is defined as $|S| + |\Sigma|$. As usual, a string language is *regular* if it is accepted by an NFA. If $|I| = 1$ and $|\delta(s, \sigma)| \leq 1$ for all $s \in S$ and $\sigma \in \Sigma$, then M is a *deterministic* finite automaton (DFA) and we treat δ as a function $S \times \Sigma \to S$. Additionally, we write s_0 in the definition of M when $I = \{s_0\}$.

We will reprove Büchi's Theorem [10] stating that a string language is regular if and only if it can be defined by an MSO sentence. Here, an MSO sentence φ defines the language $\{w \in \Sigma^* \mid w \models \varphi\}$. The simulation of an MSO sentence by a DFA will be based on the computation of equivalence types. To this end we will use the following proposition which in particular says that $\tau_k^{\mathrm{MSO}}(\sigma_1 \cdots \sigma_{n-1}\sigma_n)$ only depends on $\tau_k^{\mathrm{MSO}}(\sigma_1 \cdots \sigma_{n-1})$ and

$\tau_k^{\mathrm{MSO}}(\sigma_n)$. This observation will be used in the proof of Theorem 2.5 to define the transition function of the DFA computing the $\equiv_k^{\mathrm{MSO}}$-type of input strings.

By using the above Ehrenfeucht games and Proposition 2.3, one easily show the following.

Proposition 2.4 *Let $k \geq 0$ and let w, v, w' and v' be strings. If $w \equiv_k^{\mathrm{MSO}} w'$ and $v \equiv_k^{\mathrm{MSO}} v'$, then $w \cdot v \equiv_k^{\mathrm{MSO}} w' \cdot v'$.*

Proof. By Proposition 2.3 it suffices to show that the duplicator wins $G_k^{\mathrm{MSO}}(w \cdot v; w' \cdot v')$. We already know that she wins the subgames $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$. The duplicator, therefore, plays in $G_k^{\mathrm{MSO}}(w \cdot v; w' \cdot v')$ according to his winning strategies in $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$. We make this strategy precise, but only consider moves of the spoiler on the string $w \cdot v$. Responses to moves where the spoiler picks elements in $w' \cdot v'$ can be treated analogously. If the spoiler chooses an element in w (v) then the duplicator answers according to his winning strategy in $G_k^{\mathrm{MSO}}(w; w')$ ($G_k^{\mathrm{MSO}}(v; v')$). If the spoiler makes a set move and chooses $P_1 \cup P_2$ in $w \cdot v$, where P_1 and P_2 contain the elements in w and v, respectively, then the duplicator chooses sets Q_1 and Q_2 in w' and v' according to his winning strategy in $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$, respectively.

This is indeed a winning strategy. Let $c_1, \ldots, c_\ell$ and $d_1, \ldots, d_\ell$ be the elements chosen in point moves in $w \cdot v$ and $w' \cdot v'$, respectively, and let $P_1, \ldots, P_r$ and $Q_1, \ldots, Q_r$ be the sets of elements chosen in set moves in $w \cdot v$ and $w' \cdot v'$, respectively. By construction, the mapping $\bar{c} \mapsto \bar{d}$ restricted to the different components (w and w', and v and v') is a partial isomorphism between these corresponding components extended with the sets $\overline{P}$ and $\overline{Q}$. Hence, it only remains to check that the relation $<$ is preserved between elements coming from different components. This is always the case, as all elements of w (w') precede those of v (v'), the duplicator chooses elements in w' (v') whenever the spoiler chooses elements in w (v), and the duplicator chooses elements in w (v) whenever the spoiler chooses elements in w' (v'). ■

We are ready to prove Büchi's Theorem [10]:

Theorem 2.5 *A language $L \subseteq \Sigma^*$ is regular if and only if it is definable in MSO.*

Proof. Suppose L is defined by the DFA $M = (S, \Sigma, \delta, s_0, F)$ with $S = \{0, \ldots, n\}$ and $s_0 = 0$. We have to find an MSO sentence expressing for every string $w \in L(M)$ that M accepts w. On w, this sentence defines the run of M on w. Such a run is encoded by pairwise disjoint subsets $Z_0, \ldots, Z_n$ of $\{1, \ldots, |w|\}$ with the following intended meaning: $i \in Z_j$ iff $\delta^*(0, w_1 \cdots w_i) = j$. We say that Z_j labels position i with state j. Clearly, the run is accepting if $|w|$ is labeled with a final state. The sentence φ is then of the form

$$(\exists Z_0) \ldots (\exists Z_n) \left(\psi(Z_0, \ldots, Z_n) \wedge (\forall x) \left(\neg(\exists y)(x < y) \rightarrow \bigvee_{i \in F} Z_i(x) \right) \right).$$

Here, ψ is the FO formula that defines $Z_0, \ldots, Z_n$ as the encoding of the run of M on the string under consideration. That is, it says that the first position should be labeled with the state $\delta(0, w_1)$ and that the other labelings should be consistent with the transition function. These are all local conditions and can, hence, readily be expressed in FO. The second part of φ expresses that the last element of the input string is labeled with a final state. A more computational view of φ is that, on input w, φ first *guesses* a state assignment and then *verifies*, by means of an FO formula, whether it has guessed correctly. That is, whether its guesses encode an accepting run of the automaton.

For the other direction we make use of types. A similar presentation was given by Ladner [29]. The method presented here is also referred to as the composition method [44]. Let φ be an MSO sentence of quantifier depth k. Clearly, it suffices to know $\tau_k^{\mathrm{MSO}}(w)$ to determine whether $w \models \varphi$. We will now show that an automaton on input w can in fact compute $\tau_k^{\mathrm{MSO}}(w)$. The set of states is Φ_k, which is finite for every k. Proposition 2.4 says that for a string v and a Σ-symbol σ, $\tau_k^{\mathrm{MSO}}(v\sigma)$ only depends on $\tau_k^{\mathrm{MSO}}(v)$ and $\tau_k^{\mathrm{MSO}}(\sigma)$. Note that $\tau_k^{\mathrm{MSO}}(\sigma)$ only depends on σ. Hence, $\tau_k^{\mathrm{MSO}}(w)$ can be computed from left to right: the initial state is $\tau_k^{\mathrm{MSO}}(\varepsilon)$, that is, the $\equiv_k^{\mathrm{MSO}}$-type of the empty string; and, if the $\equiv_k^{\mathrm{MSO}}$-type of the string seen so far is θ and the next symbol is σ, then the automaton moves to state $\tau_k^{\mathrm{MSO}}(w(\theta)\sigma)$. By Proposition 2.4, it does not matter which representative of the $\equiv_k^{\mathrm{MSO}}$-equivalence class θ we take. Finally, the automaton accepts if $\varphi \in \theta$ with θ the state obtained after reading the last input symbol.

Formally, the automaton M accepting the language defined by φ is defined as $M = (\Phi_k, \Sigma, \delta, s_0, F)$, where $s_0 = \tau_k^{\mathrm{MSO}}(\varepsilon)$, $F = \{\theta \in \Phi_k \mid \varphi \in \theta\}$, and for all $\theta \in \Phi_k$ and $\sigma \in \Sigma$, $\delta(\theta, \sigma) = \tau_k^{\mathrm{MSO}}(w(\theta)\sigma)$. ■

2.3 Regular tree languages

Trees will be denoted by the boldface characters $\mathbf{t}$, $\mathbf{s}$, $\mathbf{s}_1$, ... , while nodes of trees are denoted by $\mathbf{v}$, $\mathbf{w}$, $\mathbf{v}_1$, ... Edges in trees are always directed from the root to the leaves. We use the following convention: if $\mathbf{v}$ is a node of a tree $\mathbf{t}$, then $\mathbf{v}i$ denotes the i-th child of $\mathbf{v}$. We denote the set of nodes of $\mathbf{t}$ by $\mathrm{Nodes}(\mathbf{t})$ and the root of $\mathbf{t}$ by $\mathrm{root}(\mathbf{t})$. Further, the *arity* of a node $\mathbf{v}$ in a tree, denoted by $\mathit{arity}(\mathbf{v})$, is the number of children of $\mathbf{v}$. We say that a tree $\mathbf{t}$ has *rank* m, for $m \in \mathbb{N}$, if $\mathit{arity}(\mathbf{v}) \leq m$ for every $\mathbf{v} \in \mathrm{Nodes}(\mathbf{t})$. For a node $\mathbf{v}$ in $\mathbf{t}$, the set of its children is denoted by $\mathrm{children}(\mathbf{v})$. The subtree of $\mathbf{t}$ rooted at $\mathbf{v}$ is denoted by $\mathbf{t}_\mathbf{v}$; the *envelope* of $\mathbf{t}$ at $\mathbf{v}$, that is, the tree obtained from $\mathbf{t}$ by deleting the subtrees rooted at the children of $\mathbf{v}$ is denoted by $\overline{\mathbf{t}_\mathbf{v}}$;[3] and, for each $\sigma \in \Sigma$, the tree consisting of just one node that is labeled with σ is denoted by $\mathbf{t}(\sigma)$. The *depth* of a node $\mathbf{v}$ is the number of edges on the path from the root to $\mathbf{v}$. The *height* of $\mathbf{v}$ is the number of edges on the longest path from $\mathbf{v}$ to a leaf. Hence, the depth of the root and the height of a leaf are zero. We denote the label of $\mathbf{v}$ in $\mathbf{t}$ by $\mathrm{lab}_\mathbf{t}(\mathbf{v})$.

We end by introducing the following notation. When σ is a symbol in Σ and $\mathbf{t}_1$, ... , $\mathbf{t}_n$ are Σ-trees, then $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ is the Σ-tree graphically represented by

$$\begin{array}{ccc} & \sigma & \\ \swarrow & \cdots & \searrow \\ \mathbf{t}_1 & & \mathbf{t}_n. \end{array}$$

Note that in the above definitions there is no a priori bound on the number of children that a node may have. In Section 4, we restrict attention to trees of bounded rank (hereafter simply referred to as *ranked* trees). In Section 5, we consider trees without any bound on their rank. To make a clear distinction, we refer to the latter as *unranked* trees.

It remains to specify the logical structures corresponding to Σ-trees. A Σ-tree $\mathbf{t}$ can be naturally viewed as a finite structure over the binary relation symbols E and $<$, and the unary relation symbols $(O_\sigma)_{\sigma \in \Sigma}$. The edge relation E is the obvious one. The relation $<$ specifies the ordering on the children for every node $\mathbf{v}$. Finally, for each $\sigma \in \Sigma$, O_σ is the set of nodes that are labeled with a σ.

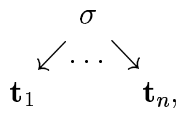
As we did for strings, we fix one particular tree $\mathbf{t}(\theta)$ for each k-type $\theta \in \tau_k^{\mathrm{MSO}}$.

[3] Note that $\mathbf{t}_\mathbf{v}$ and $\overline{\mathbf{t}_\mathbf{v}}$ have $\mathbf{v}$ in common.

We first define bottom-up deterministic tree automata and indicate how they can compute the $\equiv_k^{\mathrm{MSO}}$-types of trees. This will allow us to reprove the generalization of Büchi's Theorem for trees. In the definition below we use the superscript r to stress that we define automata for ranked trees. All definitions in the remaining of this section are for Σ-trees of rank at most m, for some fixed m.

Definition 2.6 A *deterministic bottom-up ranked tree automaton* (DBTA^r) is a triple $B = (Q, \Sigma, \delta, F)$, consisting of a finite set Q of states, a finite alphabet Σ, a set $F \subseteq Q$ of final states, and a transition

function $\delta : \bigcup_{i=0}^{m} Q^i \times \Sigma \to Q$. The semantics of B on a tree $\mathbf{t}$, denoted by $\delta^*(\mathbf{t})$, is inductively defined as follows: if $\mathbf{t}$ consists of only one node labeled with σ then $\delta^*(\mathbf{t}) = \delta(\sigma)$; if $\mathbf{t}$ is of the form



then $\delta^*(\mathbf{t}) = \delta(\delta^*(\mathbf{t}_1), \ldots, \delta^*(\mathbf{t}_n), \sigma)$. A Σ-tree $\mathbf{t}$ is

accepted by B if $\delta^*(\mathbf{t}) \in F$.

The set of Σ-trees accepted by B is denoted by $L(B)$. A set $\mathcal{T}$ of Σ-trees is *recognizable* if there exists a tree automaton B, such that $\mathcal{T} = L(B)$.

To prove Theorem 2.8, we need a suitable generalization of Proposition 2.4 to trees. The proof of the next proposition is similar to the proof of the latter, but is a bit more subtle due to the presence of the edge relation E.

Proposition 2.7 *Let k be a natural number, $\sigma \in \Sigma$, and let $\mathbf{t}_1, \ldots, \mathbf{t}_n$, $\mathbf{s}_1, \ldots, \mathbf{s}_n$ be Σ-trees. If $\mathbf{t}_i \equiv_k^{\mathrm{MSO}} \mathbf{s}_i$, for $i = 1, \ldots, n$, then*

$$\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n).$$

Proof. We just combine the winning strategies in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_i; \mathbf{s}_i)$ to obtain a winning strategy in $G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n))$ as explained in the proof of Proposition 2.4. As will be shown below we can assume that the duplicator picks the root in $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ whenever the spoiler picks the root of $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$, and vice versa. We show that this strategy is winning. Suppose that in a play in

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n))$$

the elements $\bar{c}$ and $\bar{d}$ are chosen in the point moves in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, respectively, and the sets $\overline{P}$ and $\overline{Q}$, are chosen in set moves in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, respectively. Clearly, the mapping $\bar{c} \mapsto \bar{d}$ restricted to the different components is a partial isomorphism between these corresponding components extended with the sets $\overline{P}$ and $\overline{Q}$. Hence, it only remains to check that the relations $<$ and E are preserved for elements coming from different components. We restrict attention to elements in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$. Denote the roots of $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ by $\mathbf{v}$ and $\mathbf{w}$, respectively.

Let c_i and c_j be elements coming from different components $\mathbf{t}_a$ and $\mathbf{t}_b$, with $c_i < c_j$ and $a, b \in \{1, \ldots, n\}$. Consequently, c_i and c_j are children of $\mathbf{v}$ and $a < b$. It, hence, suffices to show that d_i and d_j are the roots of $\mathbf{s}_a$ and $\mathbf{s}_b$, respectively.

First note the following. If the spoiler picks the root of $\mathbf{t}_a$ in his l-th move, with $l < k$, in $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$, then the duplicator is forced to answer with the root of $\mathbf{s}_a$. Indeed, if she does not do so and picks another node, say $\mathbf{e}$, then in the next round the spoiler just picks the parent of $\mathbf{e}$ to which the duplicator has no answer.

Since c_i and c_j come from different components, the duplicator and the spoiler never play k rounds in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$ and $G_k^{\mathrm{MSO}}(\mathbf{t}_b; \mathbf{s}_b)$. That is, in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$ and $G_k^{\mathrm{MSO}}(\mathbf{t}_b; \mathbf{s}_b)$, the elements c_i and c_j are chosen before the k-th round. By the above argument, this means that d_i and d_j have to be the roots of $\mathbf{s}_a$ and $\mathbf{s}_b$, respectively. Therefore, $d_i < d_j$ as required.

Concerning E, we only have to consider the case where $c_i = \mathbf{v}$ and c_j is a child of $\mathbf{v}$. By a similar argument as before it follows that d_j has to be a child of $\mathbf{w}$. ■

By the previous proposition, the $\equiv_k^{\mathrm{MSO}}$-type of a tree only depends on the $\equiv_k^{\mathrm{MSO}}$-types of the subtrees rooted at the children of the root. This suggests a mechanism to compute $\equiv_k^{\mathrm{MSO}}$-types of trees in a bottom-up way. Indeed, we get a bottom-up automaton for this purpose if we choose the set of states Φ_k, and the transition function as follows: for every $\sigma \in \Sigma$, $\delta(\sigma) = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma))$ and for $\theta_1, \ldots, \theta_n \in \Phi_k$, $\delta(\theta_1, \ldots, \theta_n, \sigma) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}(\theta_1), \ldots, \mathbf{t}(\theta_n)))$. Proposition 2.7 guarantess that, as it is the case for strings, it does not matter which representatives of the $\equiv_k^{\mathrm{MSO}}$-equivalence classes $\theta_1, \ldots, \theta_n$ we take. A tree automaton, in turn, can again be defined in MSO by guessing states and then verifying in FO the consistency with the transition function. This leads to the following theorem obtained by Doner, Thatcher and Wright [16, 43].

Theorem 2.8 *A tree language is recognizable if and only if it is definable in MSO.*

For later use, we show that a bottom-up tree automaton also can compute the type $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ of each input tree $\mathbf{t}$. Therefore, we need the following proposition. Actually, we only need the second item of Proposition 2.9, the other items will be used in Section 4.3.

Proposition 2.9 *Let k be a natural number, $\mathbf{t}$ and $\mathbf{s}$ be two trees, $\mathbf{v}$ be a node of $\mathbf{t}$ and $\mathbf{w}$ be a node of $\mathbf{s}$ both of arity n.*

1. *If $(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{w}}, \mathbf{w})$ and $(\mathbf{t}_\mathbf{v}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{w}, \mathbf{w})$ then $(\mathbf{t}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}, \mathbf{w})$.*

2. *If $\mathrm{lab}_\mathbf{t}(\mathbf{v}) = \mathrm{lab}_\mathbf{s}(\mathbf{w})$ and $(\mathbf{t}_{\mathbf{v}i}, \mathbf{v}i) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{w}i}, \mathbf{w}i)$ for $i = 1, \ldots, n$, then $(\mathbf{t}_\mathbf{v}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{w}, \mathbf{w})$.*

3. *Let $i \in \{1, \ldots, n\}$. If*

   - *$(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{w}}, \mathbf{w})$,*
   - *$\mathrm{lab}_\mathbf{t}(\mathbf{v}i) = \mathrm{lab}_\mathbf{t}(\mathbf{w}i)$, and*
   - *$(\mathbf{t}_{\mathbf{v}j}, \mathbf{v}j) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{w}j}, \mathbf{w}j)$, for $j \in \{1, \ldots, n\} - \{i\}$,*

   then $(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_{\mathbf{w}i}}, \mathbf{w}i)$.

Proof. The proofs of all three statements are very similar. The basic idea is to combine the winning strategies of the duplicator on the respective subtrees into a winning strategy on the whole structures like in the case of strings in Proposition 2.4. We focus on the third case where there are altogether $n+1$ subgames including the trivial game in which one structure consists only of $\mathbf{v}i$ and the other of $\mathbf{w}i$. The winning strategy in the game on $(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i)$ and $(\overline{\mathbf{s}_{\mathbf{w}i}}, \mathbf{w}i)$ just combines the winning strategies in those $n+1$ subgames. At the end of the game, the selected vertices define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the relations $<$ and E between the chosen elements, and $\mathbf{v}i$ and $\mathbf{w}i$. The preservation of $<$ and E between chosen elements can be verified as in the proof of Proposition 2.7. Additionally, we have to check that for every corresponding pair of chosen nodes c and d: $c < \mathbf{v}i$ iff $d < \mathbf{w}i$, $\mathbf{v}i < c$ iff $\mathbf{w}i < d$, and $E(c, \mathbf{v}i)$ iff $E(d, \mathbf{w}i)$. This follows immediately, as all siblings and the parents of

$\mathbf{v}i$ and $\mathbf{w}i$ are distinguished constants, and only elements of corresponding substructures are chosen. ■

We will use the following lemma in Section 4.3.

Lemma 2.10 *Let k be a natural number. There exists a DBTA^r $B = (Q, \Sigma, \delta, F)$ such that $\delta^*(\mathbf{t}) = \tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$, for every tree $\mathbf{t}$.*

Proof. We apply the same bottom-up technique as in the proof of Theorem 2.8. Only now we make use of Proposition 2.9(2) rather than Proposition 2.7. Define Q as Φ_k. Here we take Φ_k as the set of $\equiv_k^{\mathrm{MSO}}$-types of trees with one distinguished node. Further, define the transition function as follows: for every $\sigma \in \Sigma$, $\delta(\sigma) = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma), \mathrm{root}(\mathbf{t}(\sigma)))$ and for $\theta, \theta_1, \ldots, \theta_n \in \Phi_k$, $\delta(\theta_1, \ldots, \theta_n, \sigma) = \theta$ iff there exists a tree $\mathbf{t}$ with a node $\mathbf{v}$ of arity n such that $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v}) = \theta$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}i}, \mathbf{v}i) = \theta_i$, for $i = 1, \ldots, n$. By Proposition 2.9(2), it does not matter which members $\mathbf{t}_{\mathbf{v}1}, \ldots, \mathbf{t}_{\mathbf{v}n}$ of the $\equiv_k^{\mathrm{MSO}}$-equivalence classes $\theta_1, \ldots, \theta_n$ we take. ■

3 Query automata on strings

To warm up, we start with query automata on strings. These are simply two-way deterministic automata extended with a selection function. Again, this approach allows us to introduce some important proof techniques in an easy setting which then later will be generalized to obtain our main results. In particular, we recall the important notion of behavior functions and reprove a surprising lemma on two-way automata by Hopcroft and Ullman.

We first define queries. In this article, a *query* is a function $\mathcal{Q}$ that maps each structure $\mathcal{A}$ to a unary relation over its domain. MSO can be used to define queries in a straightforward way: if $\varphi(x)$ is an MSO-formula then $\varphi(x)$ defines a query $\mathcal{Q}$ via

$$\mathcal{Q}(\mathcal{A}) = \{a \mid \mathcal{A} \models \varphi[a]\}.$$

We next define two-way automata over strings.

To prevent such automata from falling of the input string, we will always feed them with strings of the form $\rhd\, w_1 \cdots w_n \,\lhd$ where $\rhd$ and $\lhd$ are new symbols not appearing in Σ. We require that automata never move to the left from a $\rhd$ and to the right from a $\lhd$.

Definition 3.1 A *two-way deterministic finite automaton* (2DFA) is a tuple $M = (S, \Sigma, s_0, \delta, F, L, R)$, where

- S is a finite set of states;
- s_0 is the initial state;
- F is the set of final states;
- L is a subset of $S \times (\Sigma \cup \{\lhd\})$, R is a subset of $S \times (\Sigma \cup \{\rhd\})$ and L and R are disjoint; and
- δ consists of the transition functions $\delta_{\leftarrow}$ and $\delta_{\rightarrow}$; in particular, $\delta_{\leftarrow} : L \to S$ is the transition function for left-moves and $\delta_{\rightarrow} : R \to S$ is the transition function for right-moves.

A *configuration* of M is a member of $S \times \mathbb{N}$, i.e., a pair consisting of a state and a position. For a string w, a *run of* M is a sequence $(s_1, j_1) \ldots (s_m, j_m)$ of configurations such that for all $i = 1, \ldots, m$,

- $j_i \in \{1, \ldots, |w|\}$;
- if $(s_i, w_{j_i}) \in L$ then $s_{i+1} = \delta_{\leftarrow}(s_i, w_{j_i})$ and $j_{i+1} = j_i - 1$; and
- if $(s_i, w_{j_i}) \in R$ then $s_{i+1} = \delta_{\rightarrow}(s_i, w_{j_i})$ and $j_{i+1} = j_i + 1$.

A run $(s_1, j_1) \ldots (s_m, j_m)$ is *the run of M on input w*, if s_1 is the initial state of M, $s_m \in F$ and there is no transition possible from (s_m, w_{j_m}).

We will only consider 2DFAs that always halt. This is a decidable property. Indeed, as we will show in the proof of Theorem 3.9, the behavior of a 2DFA M can be defined in MSO. It is then not difficult to write an MSO sentence that is satisfiable iff M does not terminate on at least one input string. It is well known that satisfiability of MSO on strings is decidable [45]. Clearly, any 2DFA can be modified such that it always halts at the end-marker. For convenience, we will assume each 2DFA is as such. A query automaton is now just a 2DFA extended with a selection function:

Definition 3.2 A *query automaton M on strings* (QA^{string}) is a tuple $(S, \Sigma, s_0, \delta, F, \lambda)$, where $(S, \Sigma, s_0, \delta, F)$ is a 2DFA, and λ is a mapping $\lambda : S \times \Sigma \to \{\bot, 1\}$.

We say that M *selects* position $i \in \{1, \ldots, |w|\}$ if the run $(s_0, j_0), \ldots, (s_m, j_m)$ of M on w is accepting and $\lambda(s_l, w_{j_l}) = 1$ for an $l \in \{0, \ldots, m\}$ with $j_l = i$. That is, i is selected by M if M selects i at least once; M does not need to select i every time it visits this position. In particular, when the run is not accepting, no position is selected. The *query expressed* by M on w is defined as $M(w) := \{i \in \{1, \ldots, |w|\} \mid M \text{ selects } i\}$.

Remark 3.3 Although 2DFAs are equivalent to one-way DFAs (see, e.g., Shepherdson [41] or Hopcroft and Ullman [25]), not all QA^{string}s are equivalent to a QA^{string} that can move in only one direction. Consider for example queries of the following kind: *select the first and last symbol if the string contains the letter* σ. This query is not computable by a QA^{string} that only moves in one direction. Indeed, when started on the first position, the one-way query automaton already has to decide whether it should select without having seen the input. The same holds when it is started on the last position and it only can move from right to left. ■

We illustrate the previous definitions with an example.

Example 3.4 We give an example of a QA^{string} computing the query: *select every position labeled with* 1 *occurring on an odd position when counting from right to left starting at the right end of the input string.* Define $M = (S, \Sigma, s_0, \delta, F, \lambda)$ with

- $\Sigma = \{0, 1\}$;
- $S = \{s_0, s_1, s_2\}$;
- $F = \{s_1, s_2\}$;
- $R = \{s_0\} \times \{0, 1, \rhd\}$;
- $L = \{s_1, s_2\} \times \{0, 1, \lhd\}$;
- $\delta_{\rightarrow}(s_0, \rhd) = \delta_{\rightarrow}(s_0, 0) = \delta_{\rightarrow}(s_0, 1) = s_0$;
- $\delta_{\leftarrow}(s_0, \lhd) = s_1$; $\delta_{\leftarrow}(s_1, 0) = \delta_{\leftarrow}(s_1, 1) = s_2$; $\delta_{\leftarrow}(s_2, 0) = \delta_{\leftarrow}(s_2, 1) = s_1$; and
- for all $s \in S$ and $a \in \Sigma$, $\lambda(s, a) = 1$ iff $s = s_1$ and $a = 1$.

The automaton operates as follows. First it walks to the right endmarker using state s_0. Hereafter, it returns to the left endmarker alternating between the states s_1 and s_2. A position is assigned the state s_1 (s_2) if it occurs on an odd (even) position when counting from the right endmarker (endmarker not included). The run on input $w = \rhd 0110 \lhd$ is the sequence

$$(s_0,1)(s_0,2)(s_0,3)(s_0,4)(s_0,5)(s_0,6)(s_1,5)(s_2,4)(s_1,3)(s_2,2)(s_1,1).$$

Hence, only position 3 is selected. This query automaton does not end at the right endmarker. However, it can easily be modified to do so. ■

Before generalizing Büchi's Theorem to query automata, we define two-way deterministic finite automata that output at each position one symbol of a fixed alphabet rather than just 0 or 1 as is the case for query automata. Such automata will turn out useful in the proof of Theorem 3.9 and will be essential for the capturing of MSO by query automata on unranked trees in Section 5.

Definition 3.5 A *generalized string query automaton* (GSQA) M is a tuple

$$(S, \Sigma, s_0, \delta, F, \lambda, \Gamma),$$

where $(S, \Sigma, s_0, \delta, F)$ is a 2DFA, Γ is a finite output alphabet, and λ is a function from $S \times \Sigma$ to $\Gamma \cup \{\bot\}$. We always assume $\bot \notin \Gamma$.

We will only consider GSQA that output at each position of the input string exactly one Γ-symbol different from $\bot$ and which always halt. Therefore, for each position i of a string w, we denote by $M(w,i)$ the unique symbol output by M at position i. By $M(w)$ we denote the string $M(w,1)\cdots M(w,|w|)$. Let f be a length preserving function from Σ^* to Γ^*. We say that f is *computed* by a GSQA M if $M(w) = f(w)$ for all strings w.

The condition that a GSQA outputs exactly one Γ-symbol different from $\bot$ at each position is not essential for the results in this paper. We could also just have taken $M(w,i)$ as the last Γ-symbol different from $\bot$ output at position i. The former automata are just easier to work with.

Example 3.6 We modify the QA^{string} of Example 3.4 into a generalized query automaton. To this end, we redefine λ as the function $\lambda : S \times \Sigma \to \{0, 1, *, \bot\}$ as follows:

$$\begin{array}{ll} \lambda(s_0,0) = 0; & \lambda(s_0,1) = 0; \\ \lambda(s_1,0) = 0; & \lambda(s_1,1) = *; \\ \lambda(s_2,0) = 0; & \lambda(s_2,1) = 1. \end{array}$$

This automaton just copies the input string, but replaces every symbol 1 with $*$ when it occurs on an odd position when counting from right to left from the endmarker. Thus, $M(\triangleright\ 0110\ \triangleleft) = \triangleright\ 0 * 10\ \triangleleft$. ■

We generalize Büchi's Theorem to query automata. In this proof we will introduce the concept of behavior function[4] which will play a major role in Sections 4, 5, and 6. Additionally, we use a remarkable lemma due to Hopcroft and Ullman [24] on two way automata which will turn out to be crucial in Sections 4 and 5.

We first provide a suitable generalization of Proposition 2.4.

Proposition 3.7 *Let k be a natural number, let w and v be strings, let $i \in \{1, \ldots, |w|\}$, and let $j \in \{1, \ldots, |v|\}$. If $(w_1 \cdots w_i, i) \equiv_k^{\mathrm{MSO}} (v_1 \cdots v_j, j)$ and*

$$(w_i \cdots w_{|w|}, 1) \equiv_k^{\mathrm{MSO}} (v_j \cdots v_{|v|}, 1),$$

then $(w, i) \equiv_k^{\mathrm{MSO}} (v, j)$.

Proof. We just combine the winning strategies in the subgames

$$G_k^{\mathrm{MSO}}(w_1 \cdots w_i, i; v_1 \cdots v_j, j)$$

and $G_k^{\mathrm{MSO}}(w_i \cdots w_{|w|}, 1; v_j \cdots v_{|v|}, 1)$ to obtain a winning strategy in the game $G_k^{\mathrm{MSO}}(w, i; v, j)$, like in the proof of Proposition 2.4. We have to be a bit careful as position i and j in $w_1 \cdots w_{|w|}$ and $v_1 \cdots v_{|v|}$, respectively, occur in both subgames $G_k^{\mathrm{MSO}}(w_1 \cdots w_i, i; v_1 \cdots v_j, j)$ and

$$G_k^{\mathrm{MSO}}(w_i \cdots w_{|w|}, 1; v_j \cdots v_{|v|}, 1).$$

However, the combined strategy is well defined on these positions: the duplicator picks position i (j) when the spoiler picks position j (i) as she does so in both subgames, simply because the common positions occur as distinguished constants in the subgames. ■

Using the above proposition we obtain the following lemma:

Lemma 3.8 *Let k be a natural number. There exists a DFA $M = (S, \Sigma, s_0, \delta, F)$ such that $\delta^*(w) = \tau_k^{\mathrm{MSO}}(w, |w|)$, for every string w.*

[4] We note that Shepherdson [41] already used behavior functions to simulate two-way automata by one-way ones.

Proof. The automaton M just works like the automaton in the proof of Theorem 2.5. The only difference is that it has to take the distinguished constant into account. Therefore, M has $\Phi_k \cup \{s_0\}$ as set of states where s_0 is the start state and where Φ_k is the set of $\equiv_k^{\mathrm{MSO}}$-types with one distinguished position. By Proposition 3.7, $\tau_k^{\mathrm{MSO}}(w\sigma, |w|+1)$ only depends on $\tau_k^{\mathrm{MSO}}(w, |w|)$ and $\tau_k^{\mathrm{MSO}}(\sigma, 1)$. Note that the latter only depends on σ. So, the transition function δ is defined as follows: for each $\sigma \in \Sigma$, $\delta(s_0, \sigma) = \tau_k^{\mathrm{MSO}}(\sigma, 1)$ and for each $\theta \in \Phi_k$, $\delta(\theta, \sigma) = \tau_k^{\mathrm{MSO}}(w\sigma, |w|+1)$ where w is a string with $\tau_k^{\mathrm{MSO}}(w, |w|) = \theta$. ■

We are now ready to prove the main result of this section.

Theorem 3.9 *A query is computable by a QA^{string} if and only it is definable in MSO.*

Proof. Let $M = (S, \Sigma, s_0, \delta, F, \lambda)$ be a QA^{string}. We will construct an MSO formula $\varphi(x)$ that defines the query computed by M.

In the case of a one-way DFA M', the state assumed by M' at each position of the input string completely determined the behavior of M'. Accordingly, we simulated M' in the proof of Theorem 2.5, by simply guessing this state assignment. Now, we do not only have to describe the behavior of a two-way automaton, but we also have to know which positions it selects. Therefore, we define the following partial functions for M on a string w. If $i \in \{1, \ldots, |w|\}$ then the behavior function $f^{\leftarrow}_{w_1 \cdots w_i} : S \to S$ is defined as

$$f^{\leftarrow}_{w_1 \cdots w_i}(s) := \begin{cases} s & \text{if } (s, w_i) \in R; \\ s' & \text{if } (s, w_i) \in L \text{ and whenever } M \\ & \text{starts its computation on } w \text{ at} \\ & \text{position } i \text{ in state } s \text{ then } s' \text{ is} \\ & \text{the first state in which it re-} \\ & \text{turns at } i. \end{cases}$$

We need one more notion. For each $i = 1, \ldots, |w|$, the set of states *assumed* by M at i is defined as $\mathrm{Assumed}(w, i) := \{s_l \mid l \in \{1, \ldots, m\} \text{ and } j_l = i\}$ with $(s_1, j_1) \ldots (s_m, j_m)$ the run of M on w.

For each position i of the input string w the formula φ guesses the function $f^{\leftarrow}_{w_1 \cdots w_i}$, the set $\mathrm{Assumed}(w, i)$, and the first state in which M reaches i, denoted by $\mathrm{first}(w, i)$. Formally, the formula guesses sets $Z_{f,B,s}$ for all partial functions $f : S \to S$, sets $B \subseteq S$, and $s \in S$, with the intended meaning:

$i \in Z_{f,B,s}$ iff $f = f^{\leftarrow}_{w_1 \cdots w_i}$, $B = \text{Assumed}(w,i)$ and $s = \text{first}(w,i)$. Note that the number of sets $Z_{f,B,s}$ is bounded, independently of w. The correctness of these guesses is easily verified in FO since they are determined by local consistency checks only. To see this we introduce the following definitions. For each partial function $f : S \to S$ and state $s \in S$, let $\text{States}(f,s)$ be the smallest set containing s and if $s' \in \text{States}(f,s)$ then $f(s') \in \text{States}(f,s)$. That is, if M has reached position i in state s then $\text{States}(f^{\leftarrow}_{w_1 \cdots w_i}, s)$ is the set of states in which M visits position i before making a right move at i. There are now two possibilities. Either there exists a state $s' \in \text{States}(f^{\leftarrow}_{w_1 \cdots w_i}, s)$ with $(s', w_i) \in R$ or there does not. The second case indicates that M starts to cycle, while the first case means that M makes its next right move at position i in state s'. Hence, we define $\text{right}(f,s,\sigma) = s'$ with $s' \in \text{States}(f,s)$ and $(s',\sigma) \in R$ if such an s' exists. Otherwise, $\text{right}(f,s,\sigma)$ is undefined. We now have enough terminology to show that the consistency checks only depend on local information.

1. $\text{first}(w,1) = s_0$;

2. for $i = 1, \ldots, |w|-1$, $f^{\leftarrow}_{w_1 \cdots w_{i+1}}$ only depends on $f^{\leftarrow}_{w_1 \cdots w_i}$, w_i and w_{i+1}, and $\text{first}(w,i+1)$ only depends on $\text{first}(w,i)$, w_i and $f^{\leftarrow}_{w_1 \cdots w_i}$. Specifically, for $i = 1, \ldots, |w|-1$ and for all $s \in S$,

$$f^{\leftarrow}_{w_1 \cdots w_{i+1}}(s) = \begin{cases} s & \text{if } (s, w_{i+1}) \in R \\ \delta_{\rightarrow}\left(\text{right}\left(f^{\leftarrow}_{w_1 \cdots w_i}, \delta_{\leftarrow}(s, w_{i+1}), w_i\right), w_i\right) & \text{otherwise.} \end{cases}$$

   We use the convention that $f^{\leftarrow}_{w_1 \cdots w_{i+1}}(s)$ is undefined whenever

$$\text{right}(f^{\leftarrow}_{w_1 \cdots w_i}, \delta_{\leftarrow}(s, w_{i+1}), w_i)$$

   or $\delta_{\rightarrow}$ is undefined. Further,

$$\text{first}(w, i+1) = \delta_{\rightarrow}\left(\text{right}\left(f^{\leftarrow}_{w_1 \cdots w_i}, \text{first}(w,i), w_i\right), w_i\right);$$

3. $\text{Assumed}(w,|w|)$ only depends on $\text{first}(w,|w|)$ and $f^{\leftarrow}_{w_1 \cdots w_{|w|}}$. Specifically,

$$\text{Assumed}(w,|w|) = \text{States}\left(f^{\leftarrow}_{w_1 \cdots w_{|w|}}, \text{first}(w,|w|)\right);$$

   and

4. for $i = 1, \ldots, |w| - 1$, $\mathrm{Assumed}(w, i)$ only depends on $f^{\leftarrow}_{w_1 \cdots w_i}$, w_{i+1}, $\mathrm{first}(w, i)$, and $\mathrm{Assumed}(w, i+1)$. Specifically, for $i = 1, \ldots, |w| - 1$,

$$\begin{aligned}\mathrm{Assumed}(w, i) = \mathrm{States}(f^{\leftarrow}_{w_1 \cdots w_i}, \mathrm{first}(w, i)) \\ \cup \bigcup \{\mathrm{States}(f^{\leftarrow}_{w_1 \cdots w_i}, s) \mid \exists s' \in \mathrm{Assumed}(w, i+1) \\ \wedge\, \delta_{\leftarrow}(s', w_{i+1}) = s\}.\end{aligned}$$

The above conditions uniquely determine $\mathrm{first}(w, i)$, $f^{\leftarrow}_{w_1 \cdots w_i}$, and $\mathrm{Assumed}(w, i)$, for each i. Intuitively, the states first(w, i) and the functions $f^{\leftarrow}$ are fixed from left to right, whereafter the sets Assumed are fixed from right to left. Clearly, these conditions can be checked in FO. Finally, φ verifies whether M halts in an accepting state (this only depends on $\mathrm{Assumed}(w, |w|)$) and, if so, selects those positions i that are selected by M which now only depend on the sets $\mathrm{Assumed}(w, i)$s.

Conversely, let $\varphi(x)$ be an MSO formula of quantifier depth k. We will describe an automaton N computing the query defined by φ. In particular, N computes $\tau_k^{\mathrm{MSO}}(w, i)$ for every position i of the input string w, which, by Proposition 3.7, only depends on $\tau_k^{\mathrm{MSO}}(w_1 \cdots w_i, i)$ and $\tau_k^{\mathrm{MSO}}(w_i \cdots w_{|w|}, 1)$.

We start with the (one-way) DFA M_1 of Lemma 3.8 to compute

$$\tau_k^{\mathrm{MSO}}(w_1 \cdots w_i, i)$$

and its right-to-left variant M_2 to compute

$$\tau_k^{\mathrm{MSO}}(w_i \cdots w_{|w|}, 1).$$

A powerful and surprising lemma by Hopcroft and Ullman [24, 4] allows us to combine M_1 and M_2 into an automaton N that does exactly what we want. Adapted to our setting, the lemma says the following:

Lemma 3.10 *Let $M_1 = (P, \Sigma, \delta_1, p_0, F_1)$ be a left-to-right deterministic automaton on strings and let $M_2 = (Q, \Sigma, \delta_2, q_0, F_2)$ be a right-to-left one. There exists a generalized query automaton A that outputs, at each position of the input string, the pair (p, q) of states that M_1 and M_2 take at this position, respectively. That is, on input w, A outputs for position i the pair $(\delta_1^*(p_0, w_1 \cdots w_i), \delta_2^*(q_0, w_{|w|} \cdots w_i))$.*

The result now readily follows since N can first simulate M_1 and M_2 as stated in Lemma 3.10 and then select every position i for which it would

output a pair (θ_1, θ_2) such that there exists a string v and a position j with $\tau_k^{\mathrm{MSO}}(v_1 \cdots v_j, j) = \theta_1$, $\tau_k^{\mathrm{MSO}}(v_j \cdots v_{|v|}, 1) = \theta_2$, and $v \models \varphi[j]$.

For the sake of completeness and since Lemma 3.10 will be used again later on, we sketch a proof of it based on the survey paper of Engelfriet [18].

The automaton A first computes $\delta_1^*(p_0, w)$ by walking to the right end of w while simulating M_1. When it reaches the endmarker it goes one step to the left and outputs the pair

$$(\delta_1^*(p_0, w), \delta_2(q_0, w_{|w|})),$$

and starts to walk back to the left endmarker of w while simulating M_2. The difficulty, however, is to maintain $\delta_1^*(p_0, w_1 \cdots w_i)$, for each position i. We will describe a general method for doing this. Therefore, let $p = \delta_1^*(p_0, w_1 \cdots w_i)$ and assume that A has output the pair (p, q) at the i-th position of w. We now show how A computes $\delta_1^*(p_0, w_1 \cdots w_{i-1})$ from p. Suppose, $\{p' \mid \delta_1(p', w_i) = p\} = \{p_1, \ldots, p_k\}$. That is, $\{p_1, \ldots, p_k\}$ is the set of states from which A could have reached p by reading w_i. If $k = 1$ then there is no problem. Hence, assume $k \geq 2$. Then A simulates M_1 backwards from each state in $\{p_1, \ldots, p_k\}$ simultaneously. That is, when it arrives at position j it knows for each $p_l \in \{p_1, \ldots, p_k\}$ the set of states $\gamma(p_l, w_{j+1} \cdots w_{i-1}) = \{p' \mid \delta_1^*(p', w_{j+1} \cdots w_{i-1}) = p_l\}$. Note that these γ-sets are pairwise disjoint.

This computation continues until one of the two following conditions occurs.

1. If at position j all γ-sets become empty except one (say $\gamma(p_l, w_{j+1} \cdots w_{i-1})$), then p_l is the state we were looking for.

2. If A arrives at the beginmarker then the required state is the one whose γ-set contains the start state.

Now A "knows" the correct state p_l of M_1 at position i. The only remaining problem is that it is now at some position j and has to find its way back to position i. By the construction above, one step before A found out about p_l (at position $j + 1$), there had been at least 2 different sets of states from which M_1 reaches p at position i (after reading w_i). The key idea is that i is exactly the position where two computations of M_1, that start at position $j + 1$ in two states from two of these sets, flow together into the same state (in this case the state p). Hence, on its way to the left, A always remembers two states from different γ sets from the position before (right) and starts its

way back to position i by simulating the behavior of M_1 from position $j+1$ beginning with these two states. ■

We note that Lemma 3.10 is also used extensively by Engelfriet and Hoogeboom [19] to prove connections between MSO definable string transductions and deterministic two-way finite state transducers.

4 Query automata on ranked trees

After the excursion on strings, we turn to trees. Specifically, we define query automata for trees simply as two-way deterministic tree automata extended with a selection function. We will show that in the case of ranked trees such automata compute exactly the queries definable in MSO. Surprisingly, in the unranked case, we have to add more to capture exactly MSO.

We borrow some notation from Brüggemann-Klein, Murata and Wood [9] for the following definitions. All definitions in this section are for Σ-trees with rank at most m, for some fixed natural number m.

4.1 Two-way tree automata

We use the definition of a two-way tree automaton by Moriya [31].

Definition 4.1 A *two-way deterministic ranked tree automaton* (2DTA^r) is a tuple

$$A = (Q, \Sigma, F, s, \delta),$$

where Q is a finite set of states, $F \subseteq Q$ is the set of final states and $s \in Q$ is the initial state. There are disjoint subsets U and D of $Q \times \Sigma$ (U corresponds to up transitions and D to down transitions) such that $\delta_{\text{leaf}} : D \to Q$ is the transition function for leaves,[5] $\delta_{\text{root}} : U \to Q$ is the transition function for the root, $\delta_{\uparrow} : U^* \to Q$ is the transition function for up transitions, and $\delta_{\downarrow} : D \times \{1, \ldots, m\} \to Q^*$ is the transition function for down-transitions. For each $i \leq m$, $\delta_{\downarrow}(q, a, i)$ is a string of length i.

We introduced the disjoint sets U and D to avoid collision between up and down transitions. We come back to this after having defined the computation of 2DTA^{r}s. To this end, we introduce the following notions. A *cut* of $\mathbf{t}$ is

[5]Note that leaves can also take part in up transitions.

a subset of Nodes($\mathbf{t}$), that contains exactly one node of each path from the root to a leaf. A *configuration* of A on $\mathbf{t}$ is a mapping $c : C \to Q$ from a cut C of $\mathbf{t}$ to the set of states of A.

If $\mathbf{v}$ is a node of $\mathbf{t}$, then children($\mathbf{v}$) denotes the set of children of $\mathbf{v}$. Let $c : C \to Q$ be a configuration. If children($\mathbf{v}$) $\subseteq C$, then formally c(children($\mathbf{v}$)) is a subset of Q. We overload this notation so that c(children($\mathbf{v}$)) also denotes the sequence of states in Q which arises from the order of $\mathbf{v}$'s children in $\mathbf{t}$. If children($\mathbf{v}$) $= \mathbf{v}_1, \ldots, \mathbf{v}_n$ (in order) then define $\pi(c, \mathbf{v})$ as the sequence $(c(\mathbf{v}_1), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}_1)) \cdots (c(\mathbf{v}_n), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}_n))$.

The automaton A operating on $\mathbf{t}$ makes a *transition* between two configurations $c_1 : C_1 \to Q$ and $c_2 : C_2 \to Q$, denoted by $c_1 \to c_2$, iff it makes an up transition, a down transition, a leaf transition or a root transition:

1. A makes an *up transition* from c_1 to c_2 if there is a node $\mathbf{v}$ such that
   (i) children($\mathbf{v}$) $\subseteq C_1$,
   (ii) $C_2 = (C_1 - \text{children}(\mathbf{v})) \cup \{\mathbf{v}\}$,
   (iii) $\delta_{\uparrow}(\pi(c_1, \mathbf{v})) = c_2(\mathbf{v})$, and
   (iv) c_1 is identical to c_2 on $C_1 \cap C_2$.
2. A makes a *down transition* from c_1 to c_2 if there is a node $\mathbf{v}$ such that
   (i) $\mathbf{v} \in C_1$,
   (ii) $C_2 = (C_1 - \{\mathbf{v}\}) \cup \text{children}(\mathbf{v})$,
   (iii) $\delta_{\downarrow}(c_1(\mathbf{v}), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}), \mathit{arity}(\mathbf{v})) = c_2(\text{children}(\mathbf{v}))$, and
   (iv) c_1 is identical to c_2 on $C_1 \cap C_2$.
3. A makes a *leaf transition* from c_1 to c_2 if there is a leaf node $\mathbf{v}$ such that
   (i) $\mathbf{v} \in C_1$,
   (ii) $C_2 = C_1$,
   (iii) $\delta_{\text{leaf}}(c_1(\mathbf{v}), \mathrm{lab}_{\mathbf{t}}(\mathbf{v})) = c_2(\mathbf{v})$, and
   (iv) c_1 is identical to c_2 on $C_1 - \{v\}$.
4. A makes a *root transition* from c_1 to c_2 if
   (i) $C_1 = \{\text{root}(\mathbf{t})\}$,

(ii) $C_2 = C_1$, and

(iii) $\delta_{\text{root}}(c_1(\text{root}(\mathbf{t})), \text{lab}_{\mathbf{t}}(\text{root}(\mathbf{t}))) = c_2(\text{root}(\mathbf{t}))$.

The configuration $c : C \to Q$ with $c(\text{root}(\mathbf{t})) = s$ (and hence $C = \{\text{root}(\mathbf{t})\}$) is the *start configuration*. Any configuration with $c(\text{root}(\mathbf{t})) \in F$ is an *accepting configuration*. This means that a 2DTA^r starts at the root and returns there to accept the tree. A *run* is a sequence of configurations $c_1, \ldots, c_n$, $n \geq 1$, such that $c_1 \to \cdots \to c_n$ and c_1 is the start configuration. A run is *maximal* if there does not exist a c such that $c_n \to c$. A run is *accepting* if it is maximal and if c_n is an accepting configuration.

It should be noted that, although there are usually many different runs for the same tree, for all nodes the sequence of states in which they are visited is the same in all these runs. Indeed, the disjointness of $Q \times \Sigma$ into U and D makes sure that a node labeled with a certain state cannot make an up transition in one run and a down transition in another run. Therefore it is justified to consider the behavior of these automata as deterministic. For this reason, we will refer to *the* run of A on a tree rather then the more correct *a* run of A.

A 2DTA^r A *accepts* a tree $\mathbf{t}$ if the run of A on $\mathbf{t}$ is accepting; A *accepts a tree language* $\mathcal{T}$ if it accepts exactly every tree in $\mathcal{T}$.

Note that A can run forever on an input tree $\mathbf{t}$. In this case the run of A on $\mathbf{t}$ is infinite and therefore not accepting. We, however, will only consider automata that always terminate on every input. This is a decidable subclass. Indeed, later we show that the behavior of A can be defined in MSO. One then can construct an MSO sentence that is satisfiable iff A does not terminate on at least one tree. Since satisfiability of MSO sentences on trees is decidable [45], it follows that deciding whether a 2DTA^r halts on every input is also decidable.

We illustrate the above definitions with an example.

Example 4.2 Consider trees that represent Boolean circuits consisting of AND and OR gates having two inputs and one output. The represented Boolean function is evaluated from the leaves to the root. We define a 2DTA^r accepting all trees that evaluate to a 1. For ease of exposition, we only consider full binary trees that indeed represent Boolean circuits. That is, internal nodes are labeled with AND and OR, and leaves are labeled with 0 and 1. Define the 2DTA^r

$$A = (Q, \Sigma = \{\text{AND}, \text{OR}, 0, 1\}, F, s, \delta),$$

with $Q = \{s, u\} \cup \{0,1\}^2$; $D = \{s\} \times \Sigma$; $U = \{u, (0,0), (0,1), (1,0), (1,1)\} \times \Sigma$; $F = \{1\}$; and for $\sigma \in \Sigma, i, j, i_1, j_1, i_2, j_2 \in \{0,1\}$, and $\mathrm{op}, \mathrm{op}_1, \mathrm{op}_2 \in \{\mathrm{AND}, \mathrm{OR}\}$, define

1. $\delta_{\downarrow}(s, \sigma, 2) = (s, s)$;
2. $\delta_{\mathrm{leaf}}(s, \sigma) = u$;
3. $\delta_{\uparrow}((u, i), (u, j)) = (i, j)$;
4. $\delta_{\uparrow}(((i_1, j_1), \mathrm{op}_1), ((i_2, j_2), \mathrm{op}_2)) = (i_1\ \mathrm{op}_1\ j_1, i_2\ \mathrm{op}_2\ j_2)$; and
5. $\delta_{\mathrm{root}}((i, j), \mathrm{op}) = i\ \mathrm{op}\ j$.

Here, i AND j and i OR j define the standard Boolean functions. The automaton first walks to the leaves (1); at the leaves it changes state s into state u (2); hereafter, A assigns the state (i, j) to nodes of height 1 where i is the label of their first child and j is the label of their second child (3); from then on, A assigns to each inner node the pair $(i, j) \in \{0,1\}^2$, where i and j are the result of the evaluation of the left and right subtree of this node (4); finally, the root is assigned the value of the tree (5). ■

To obtain a more uniform two-way tree automaton we have let all transitions depend on the state and the label of the nodes from where this transition originates. That is, up transitions depend on the labels and the states of the children of the node the automaton heads to, while a down transition depends on the state and the label of the parent node. Up transitions of the one-way tree automata defined in Section 2.3 differ from these in that they depend on the states at the children and the label of the parent. Each two-way tree automaton can readily simulate a one-way one. Indeed, let $B = (Q_B, \Sigma, \delta_B, F_B)$ be a bottom-up deterministic tree automaton. For ease of exposition assume all transitions of B are defined. Then define the two-way automaton A simulating B as follows. First, A runs to the leaves of the input tree $\mathbf{t}$. From thereon it uses functions $f : \Sigma \to Q_B$ as states with the following intended meaning: A assigns f to a node $\mathbf{v}$ such that $\delta_B^*(\mathbf{t}_\mathbf{v}) = f(\sigma)$ whenever $\mathrm{lab}_\mathbf{t}(\mathbf{v}) = \sigma$. Thus to each leaf A assigns the state f with $f(\sigma) = \delta_B(\sigma)$ for each $\sigma \in \Sigma$, and up transitions are defined as follows: $\delta(f_1, \sigma_1, \ldots, f_n, \sigma_n) = f$ with $f(\sigma) = \delta_B(f_1(\sigma_1), \ldots, f_n(\sigma_n), \sigma)$ for every $\sigma \in \Sigma$. Furthermore, A accepts when $f(\mathrm{lab}_\mathbf{t}(\mathrm{root}(\mathbf{t}))) \in F_B$ where f is the state assigned to the root.

4.2 Query automata

A ranked query automaton is simply a two-way deterministic tree automaton over ranked trees extended with a selection function.

Definition 4.3 A *ranked query automaton* (QA^r) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a $\mathrm{2DTA}^r$ and λ is a function from $Q \times \Sigma$ to $\{\bot, 1\}$; λ is the *selection function*.

We define the semantics of a QA^r A. If $\mathbf{t}$ is a tree and $\mathbf{v}$ is a node of $\mathbf{t}$, then A *selects* $\mathbf{v}$ *in configuration* $c : C \to Q$, if $\mathbf{v} \in C$ and $\lambda(c(\mathbf{v}), \mathrm{lab}_{\mathbf{t}}(\mathbf{v})) = 1$. A *selects* $\mathbf{v}$ if the run $c_1, \ldots, c_n$ of A on $\mathbf{t}$ is accepting and if there is an $i \in \{1, \ldots, n\}$ such that $\mathbf{v}$ is selected by A in c_i. The *query computed by* A is defined as $A(\mathbf{t}) := \{\mathbf{v} \in \mathrm{Nodes}(\mathbf{t}) \mid A \text{ selects } \mathbf{v}\}$. Furthermore, A *accepts the tree language* that is accepted by the underlying tree automaton.

Example 4.4 An automaton selecting all nodes evaluating to 1 in a Boolean circuit, is obtained from the automaton of Example 4.2 by changing F to Q and adding the selection function λ defined by, for $i, j \in \{0, 1\}$ and $\mathrm{op} \in \{\mathrm{AND}, \mathrm{OR}\}$, $\lambda((i, j), \mathrm{op}) := 1$ iff i op $j = 1$. ■

Remark 4.5 Although two-way deterministic tree automata are equivalent to deterministic bottom-up tree automata (see, e.g., Moriya [31]), not all *query* automata are equivalent to deterministic *query* automata that are only top-down or only bottom-up. Consider for example queries of the following kind: *select the root if there is a leaf labeled with* σ and *select all leaves if the root is labeled with* σ.

In other words: two-way and one-way query automata are equivalent with respect to defining tree languages but not with respect to computing queries. ■

In preparation of the proof of Lemma 4.7, we extend the notion of a behavior function used in the proof of Theorem 3.9 to two-way tree automata.

Definition 4.6 Let A be a QA^r with state set Q. The *behavior function* $f^A_{\mathbf{t}} : Q \to Q$ of A on a tree $\mathbf{t}$ is the partial function defined as follows

$$f^A_{\mathbf{t}}(q) := \begin{cases} q & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \text{ is in } U \\ q' & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \text{ is in } D \text{ and} \\ & \text{whenever } A \text{ starts its computation on } \mathbf{t} \text{ in state } q \text{ then } q' \\ & \text{is the first state in which it returns at } \mathrm{root}(\mathbf{t}). \end{cases}$$

It should be noted that, as we always assume that automata do not enter infinite cycles, $f^A_{\mathbf{t}}(q) = q$ always implies that $(q, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \in U$. We call a partial function f from Q to Q *admissible*, if the graph of f contains no (directed) cycles of length > 1. If $c_1, \ldots, c_n$ is the run of A on $\mathbf{t}$, then the set of states A *assumes* at a node $\mathbf{v}$ is defined as

$$\mathrm{Assumed}^A(\mathbf{t}, \mathbf{v}) := \{c_i(\mathbf{v}) \mid i \in \{1, \ldots, n\} \text{ and } \mathbf{v} \text{ belongs to the cut of } c_i\}.$$

We introduce some more notation. If $f_1, \ldots, f_n$ are admissible functions from Q to Q and $q \in Q$, then the set of states reachable from q by using the functions $f_1, \ldots, f_n$, denoted by $\mathrm{States}(f_1, \ldots, f_n, q)$, is the smallest set of states containing q and closed under applications of every f_i. We define $\mathrm{up}(f, q)$ as the unique state q' in $\mathrm{States}(f, q)$ for which $f(q') = q'$. If there is no such state, then $\mathrm{up}(f, q)$ is undefined. Intuitively, when f corresponds to the behavior function $f^A_{\mathbf{tv}}$, then $\mathrm{up}(f, q)$ is the state in which A makes an up transition at $\mathbf{v}$ when started at $\mathbf{v}$ in state q.

4.3 Expressiveness

We characterize the expressiveness of ranked query automata in terms of MSO. First, we show how the query computed by a ranked query automaton can be defined in MSO.

Lemma 4.7 *Every query computed by a ranked query automaton can be defined in MSO.*

Proof. Let $A = (Q, \Sigma, F, s, \delta, \lambda)$ be a QA^r. Like in the proof of Theorem 3.9, we will construct an MSO formula that guesses sets and then verifies the consistency of these sets. We make use of the sets $Z_{f,B}$, where f is a partial mapping $f : Q \to Q$ and $B \subseteq Q$. On input $\mathbf{t}$ they have the following intended meaning: a node $\mathbf{v} \in Z_{f,B}$ iff $f = f^A_{\mathbf{tv}}$, and $B = \mathrm{Assumed}^A(\mathbf{t}, \mathbf{v})$. Again, like in the proof of Theorem 3.9, the correctness of these guesses is easily verified in FO since they are determined by local conditions only. Indeed,

1. the behavior function of every leaf node only depends on its label;
2. the behavior function of every non-leaf node $\mathbf{v}$ with n children only depends on $f^A_{\mathbf{tv}1}, \ldots, f^A_{\mathbf{tv}n}$, $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}1), \ldots, \mathrm{lab}_{\mathbf{t}}(\mathbf{v}n)$, and $\mathrm{lab}_{\mathbf{t}}(\mathbf{v})$. Specif-

ically, let $\mathbf{v}$ be a node of $\mathbf{t}$ of arity n. Then, for every $q \in Q$,

$$f^A_{\mathbf{t_v}}(q) := \begin{cases} q & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v})) \in U \\ q' & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v})) \notin U,\ \delta_{\downarrow}(q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v})) = (q_1, \ldots, q_n) \text{ and} \\ & \delta_{\uparrow}(\mathrm{up}(f^A_{\mathbf{t_{v1}}}, q_1), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}1), \ldots, \mathrm{up}(f^A_{\mathbf{t_{v}}n}, q_n), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}n)) = \\ & q'; \qquad (*) \end{cases}$$

We use the convention that $f^A_{\mathbf{t_v}}(q)$ is undefined whenever in $(*)$, $\delta_{\downarrow}$, $\delta_{\uparrow}$, or one of the $\mathrm{up}(f^A_{\mathbf{t_v}i}, q_i)$ is undefined;

3. $\mathrm{Assumed}^A(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ only depends on $f^A_{\mathbf{t}}$, the label of the root, and the start state. Specifically,[6]

$$\mathrm{Assumed}^A(\mathbf{t}, \mathrm{root}(\mathbf{t})) = \mathrm{States}(f^A_{\mathbf{t}}, \delta_{\mathrm{root}}(\cdot, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))), s);$$

4. for every non-root node $\mathbf{v}i$ in $\mathbf{t}$, the set $\mathrm{Assumed}^A(\mathbf{t}, \mathbf{v}i)$ only depends on $\mathrm{Assumed}^A(\mathbf{t}, \mathbf{v})$, the label of $\mathbf{v}$, and the behavior function of $\mathbf{v}i$. Specifically, let $\mathbf{v}$ be a node with n children. Then[7]

$$\mathrm{Assumed}^A(\mathbf{t}, \mathbf{v}i) = \bigcup \{\mathrm{States}(f^A_{\mathbf{t_v}i}, q) \mid \exists q' \in \mathrm{Assumed}^A(\mathbf{t}, \mathbf{v}) \wedge \delta_{\downarrow}(q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v})).i = q\}.$$

The above conditions uniquely determine the behavior functions and the sets Assumed. In particular, the behavior functions are fixed bottom-up, whereafter the sets Assumed are fixed top-down. Furthermore, the above conditions can clearly be expressed in FO. Together with the verification of these conditions, the formula verifies whether A halts in an accepting state (this only depends on $f^A_{\mathbf{t}}$) and, if so, selects those nodes that are visited in a selecting state, which now only depends on the B's. ■

For the proof of the other direction we construct an automaton computing, for some fixed k, the type $\tau^{\mathrm{MSO}}_k(\mathbf{t}, \mathbf{v})$ for each node $\mathbf{v}$ of the input tree.

Theorem 4.8 *A query is computable by a ranked query automaton if and only if it is definable in MSO.*

[6] Here, for each $\sigma \in \Sigma$, $\delta_{\mathrm{root}}(\cdot, \sigma)$ is the function mapping each q to $\delta_{\mathrm{root}}(q, \sigma)$.

[7] We denote by $\delta_{\downarrow}(q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v})).i$ the i-th entry of $\delta_{\downarrow}(q, \mathrm{lab}_{\mathbf{t}}(\mathbf{v}))$.

Proof. The only-if direction is given in Lemma 4.7.

For notational simplicity we describe the proof of the other direction only for trees of rank 2. The proof of the general case is a straightforward generalization.

Let $\varphi(x)$ be an MSO-formula of quantifier depth k. We describe a QA^r that computes the query which is defined by φ. The automaton has to find out, for each vertex $\mathbf{v}$ of a tree $\mathbf{t}$, whether $\mathbf{t} \models \varphi(\mathbf{v})$. This depends only on $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$, the $\equiv_k^{\mathrm{MSO}}$-type of the structure $(\mathbf{t}, \mathbf{v})$. By Proposition 2.9(1), $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$ is uniquely determined by $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{v}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$. Hence, the QA^r only has to compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{v}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ to decide whether $\mathbf{v}$ should be selected. We first describe an algorithm that computes these $\equiv_k^{\mathrm{MSO}}$-types for every node of a *complete* binary tree. Next we explain how this algorithm can be translated to a QA^r. Finally, we sketch how the QA^r has to be modified to deal also with possibly non-complete trees.

(i) The underlying algorithm of the QA^r for complete binary trees is outlined in Figure 5.

(ii) All objects computed by the algorithm in Figure 5 are of bounded size, depending only on φ and not on the size of $\mathbf{t}$. Hence, a QA^r can store them in its state. To simulate the outer loop of the algorithm a QA^r can proceed in cuts that consist of all vertices of level i.[8] It follows from Proposition 2.9 that steps 2, 3 and 5 only involve the application of fixed finite functions. Hence steps 2–5 can be performed in parallel at all vertices of the same depth i. Step 1 is the only one that involves non-local computation. We next discuss this step.

The type $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{w}, \mathbf{w})$ can be computed in a bottom-up fashion for a subtree $\mathbf{t}_\mathbf{w}$ of $\mathbf{t}$. Indeed, we just use the automaton of Lemma 2.10. As discussed at the end of Section 4.1 this automaton can be readily simulated by a two-way automaton that now starts at $\mathbf{w}$. The problem, however, is to detect when the root of the subtree $\mathbf{t}_\mathbf{w}$, i.e., the starting point, is reached.

Our QA^r remembers this starting point by a kind of pebbling trick. To compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}2}, \mathbf{v}2)$ it first makes a down transition: to $\mathbf{v}2$ the automaton assigns a U-state which keeps $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ in mind and waits until the computation in the left subtree has finished. To this end, the QA^r goes down to the leaves of $\mathbf{t}_{\mathbf{v}1}$ and computes $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$ in one bottom-up traversal as described above. It "recognizes" that the subtree-

[8] It should be noted that we are describing here only one special run of the automaton. But, as mentioned before, all possible runs are equivalent.

Input: $\mathbf{t}$
Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathrm{root}(\mathbf{t})}}, \mathrm{root}(\mathbf{t}))$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$,
for $i := 0$ **to** depth of $\mathbf{t}$ **do**
begin
  for all vertices $\mathbf{v}$ of level i **do**
  begin
    % the root is level 0
    % $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ has already been computed
    % now compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$
    1. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$
       Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}2}, \mathbf{v}2)$
    2. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$ from $\mathrm{lab}_{\mathbf{t}}(\mathbf{v})$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$,
       and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}2}, \mathbf{v}2)$
    3. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$
    4. Deduce from $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$ whether $\mathbf{t} \models \varphi(\mathbf{v})$ holds
       If so, select $\mathbf{v}$
    5. Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}1}}, \mathbf{v}1)$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}2}}, \mathbf{v}2)$ from
       $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}2}, \mathbf{v}2)$
  end
end

Figure 5: The algorithm for computing the query defined by $\varphi(x)$ over complete binary trees.

evaluation is finished by meeting the U-state at $\mathbf{v}2$. Next it makes an up transition, followed by a down transition. Hereafter, $v1$ has a U-state which contains $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$, and waits for the termination of the evaluation of the right subtree which is done analogously to the case of the left subtree. This finishes the description of the QA^r for the case of complete binary trees.

(*iii*) We now explain how a QA^r can deal with non-complete binary trees. We cannot use the above described pebbling trick when a node $\mathbf{v}$ has only one child. To remedy this we make use of Lemma 3.10. If a node $\mathbf{v}$ has only one child (while its parent node $\mathbf{p}$ has at least two children), then we view the part of the tree between $\mathbf{v}$ and the first descendant $\mathbf{w}$ of $\mathbf{v}$ with more than one child (or no child) as a string, where $\mathbf{p}$ and $\mathbf{w}$ play the role of begin and endmarker, respectively. Since $\mathbf{w}$ has more than one child or is a leaf, we can compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{w}, \mathbf{w})$ inductively.

Consider the deterministic string automata M_1 and M_2, where, for all vertices $\mathbf{c}$ between $\mathbf{p}$ and $\mathbf{w}$, M_1 computes $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{c}}, \mathbf{c})$ starting from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ and M_2 computes $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{c}, \mathbf{c})$ starting from $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{w}, \mathbf{w})$. On the string between $\mathbf{p}$ and $\mathbf{w}$ the QA^r then behaves as the two-way string automaton that combines the two automata M_1 and M_2 as specified in Lemma 3.10.

Hereafter, the automaton walks to $\mathbf{w}$ arriving there in state $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{w}}, \mathbf{w})$ and continues.

If we consider m-ary trees, then in step (1) we just need to compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}), \ldots, \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}n})$, where n is the arity of $\mathbf{v}$. Because $n \leq m$ and m is fixed, we can compute these one after the other. Steps (2–5) again consist of the application of fixed finite functions. ■

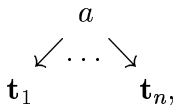
5 Query automata on unranked trees

We next turn to query automata over unranked trees. Surprisingly, the equivalence with MSO obtained in the previous section does not generalize smoothly to unranked trees. Indeed, to obtain the expressiveness of MSO we have to add so-called "stay transitions" to our model.

5.1 Tree automata over unranked trees

We start by recalling the definition of bottom-up tree automata over unranked trees.

Definition 5.1 A *nondeterministic bottom-up unranked tree automaton*, denoted by NBTA^u, is a tuple $B = (Q, \Sigma, F, \delta)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular language for every $a \in \Sigma$ and $q \in Q$. The semantics of B on a tree $\mathbf{t}$, denoted by $\delta^*(\mathbf{t})$, is defined inductively as follows: if $\mathbf{t}$ consists of only one node labeled with a then $\delta^*(\mathbf{t}) = \{q \mid \varepsilon \in \delta(q, a)\}$; if $\mathbf{t}$ is of the form



then

$$\delta^*(\mathbf{t}) = \{q \mid \exists q_1 \in \delta^*(\mathbf{t}_1), \ldots, \exists q_n \in \delta^*(\mathbf{t}_n) \text{ and } q_1 \cdots q_n \in \delta(q, a)\}.$$

A tree $\mathbf{t}$ over Σ is *accepted* by the automaton B if $\delta^*(\mathbf{t}) \cap F \neq \emptyset$. The tree language *defined* by B, denoted by $L(B)$, consists of the trees accepted by B. A tree language $\mathcal{T}$ is *recognizable* if there exists a NBTA^u B such that $\mathcal{T} = L(B)$.

Note that we use recognizable both for ranked as well as unranked trees. It will always be clear from the context, however, whether we are considering ranked or unranked trees. We represent the string languages $\delta(q, a)$ by NFAs. The *size* of B then is the sum of the sizes of Q, Σ, and the NFAs defining the transition function.

We need the following lemma in Section 6. Its proof is a straightforward generalization of the ranked case (see, e.g., the survey paper by Vardi [46]).

Lemma 5.2 *Deciding whether the tree language accepted by an NBTA is non-empty is in PTIME.*

Proof. Let $B = (Q, \Sigma, F, \delta)$ be an NBTA^u. We inductively compute the set of *reachable* states R defined as follows: $q \in R$ iff there exists a tree $\mathbf{t}$ with $q \in \delta^*(\mathbf{t})$. Obviously, $L(B) \neq \emptyset$ if and only if $R \cap F \neq \emptyset$. Define for all $n > 0$,

$$\begin{aligned} R_1 &:= \{q \in Q \mid \exists a \in \Sigma : \varepsilon \in \delta(q, a)\}; \\ R_{n+1} &:= \{q \in Q \mid \exists a \in \Sigma : \delta(q, a) \cap R_n^* \neq \emptyset\}. \end{aligned}$$

Note that for all n, $R_n \subseteq R_{n+1} \subseteq Q$. Hence, $R_{|Q|} = R_{|Q|+1}$. Thus, define R as $R_{|Q|}$.

Clearly, R_1 can be computed in time linear in the size of B. Since testing non-emptiness of $\delta(q,a) \cap R_n^*$ can be done in time polynomial in the sum of the sizes of these (see, e.g., [25]), each R_{n+1} can be computed in time polynomial in the size of B. This concludes the proof of the lemma. ■

A *deterministic* bottom-up unranked tree automaton, abbreviated by DBTA^u, is an NBTA^u as above where $\delta(q,a) \cap \delta(q',a) = \emptyset$ for every $a \in \Sigma$ and $q, q' \in Q$ with $q \neq q'$.

We mention that a detailed study of tree automata over unranked trees has been initiated by Brüggemann-Klein, Murata and Wood [9, 32].

We next generalize Theorem 2.8 to unranked trees by showing that an unranked tree language is recognizable if and only if it is definable in MSO. A DBTA^u B can be defined in MSO in the usual manner: the MSO sentence defining the behavior of B just guesses states and verifies the consistency of its guesses with the transition function. The latter can now no longer be done in FO, as was the case for ranked trees, because the transition functions are now determined by regular languages. However, by Theorem 2.5 this check can be readily done in MSO.

For the other direction, we again show that the $\equiv_k^{\mathrm{MSO}}$-type of a tree can be computed by a DBTA^u $B = (\Phi_k, \Sigma, \delta, F)$, for some fixed k. The idea is the same as for the ranked case. Again, the type of the children of a node $\mathbf{v}$ of a tree $\mathbf{t}$ plus the label of $\mathbf{v}$ determine $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{v})$. The problem is that now, as there is no bound on the number of children of a vertex, the correspondence between the children's types and the type of the whole subtree is no longer given by a finite function, as was the case for ranked trees. Instead, this correspondence is controlled by a regular language. Therefore, for each $\sigma \in \Sigma$ and $\theta \in \Phi_k$, we define $\delta(\theta, \sigma)$ as the set of strings $\theta_1 \cdots \theta_n$ where for $i = 1, \ldots, n$, $\theta_i \in \Phi_k$, and whenever for a tree $\mathbf{t}$ and a node $\mathbf{v}$ labeled with σ with n children, $\tau_k(\mathbf{t}_{\mathbf{v}i}) = \theta_i$, for each $i = 1, \ldots, n$, then $\tau_k(\mathbf{t}_\mathbf{v}) = \theta$. We now show that $\delta(\theta, \sigma)$ is indeed a regular language. To this end we state the following proposition.

Proposition 5.3 *Let k be a natural number, $\sigma \in \Sigma$, and let $\mathbf{t}_1, \ldots, \mathbf{t}_n$, $\mathbf{s}_1, \ldots, \mathbf{s}_m$, $\mathbf{t}$, $\mathbf{s}$ be trees. If $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m)$ and $\mathbf{t} \equiv_k^{\mathrm{MSO}} \mathbf{s}$, then $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{t}) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m, \mathbf{s})$.*

Proof. The proof is almost identical to the proof of Proposition 2.7. We just combine the winning strategies in the subgames

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m))$$

and $G_k^{\mathrm{MSO}}(\mathbf{t};\mathbf{s})$ to obtain a winning strategy in

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n,\mathbf{t});\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m,\mathbf{s})).$$

At the end of the game, the selected nodes define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the relations E and $<$ between selected nodes coming from different substructures. There is only one technicality in showing this. To this end, we note the following. If the spoiler picks the root of a $\mathbf{t}_a$ ($a \in \{1,\ldots,n\}$) in his l-th move with $l < k$ in $G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n);\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m))$, then the duplicator is forced to answer with the root of an $\mathbf{s}_b$ ($b \in \{1,\ldots,m\}$). Indeed, if she does not do so and picks another node, say $\mathbf{e}$, then in the next round the spoiler just picks the parent of $\mathbf{e}$ to which the duplicator has no answer. The same holds when the spoiler picks the root of $\mathbf{t}$, then the duplicator is forced to pick the root of $\mathbf{s}$.

Suppose in a play elements c_i and c_j are chosen such that $c_i < c_j$, c_i is the root of a $\mathbf{t}_a$, and c_j is the root of $\mathbf{t}$, then the above discussion implies that d_i is the root of a $\mathbf{s}_b$ and d_j is the root of $\mathbf{s}$ simply because they are picked before the k-th move in their subgames. Hence, $d_i < d_j$, as had to be shown.

Suppose in a play c_i and c_j are chosen such that c_i is the root of $\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)$ and c_j is the root of $\mathbf{t}$. Then, $E(c_i,c_j)$. By a similar argument as above, it can be shown that d_i has to be the root of $\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m)$ and d_j has to be the root of $\mathbf{s}$. Hence, $E(d_i,d_j)$. ■

The above proposition implies that we can compute the $\equiv_k^{\mathrm{MSO}}$-type of a tree $\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)$, by incrementally reading the $\equiv_k^{\mathrm{MSO}}$-types of the $\mathbf{t}_i$, starting from the state $\tau_k^{\mathrm{MSO}}(\sigma)$. Indeed, define $M_{\theta,\sigma} = (\Phi_k,\Phi_k,s_M,\delta_M,F_M)$ where $s_M = \{\tau_k^{\mathrm{MSO}}(\sigma)\}$, $F_M = \{\theta\}$, and for each $\theta_1,\theta_2 \in \Phi_k$, $\delta_M(\theta_1,\theta_2) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n,\mathbf{t}))$ where $\mathbf{t}_1$, ..., $\mathbf{t}_n$, and $\mathbf{t}$ are trees such that

$$\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)) = \theta_1$$

and $\tau_k^{\mathrm{MSO}}(\mathbf{t}) = \theta_2$. Now clearly, $L(M_{\theta,\sigma}) = \delta(\theta,\sigma)$.

From the above the following theorem readily follows.

Theorem 5.4 *An unranked tree language is recognizable if and only if it is definable in MSO.*

In Section 5.4, we will be needing a tree automaton computing the type $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ for each input tree. Such an automaton is just a slight extension of the automaton discussed above. To show this we have the following proposition. To be precise, we only need item 2, the other items are used in Section 5.4. We abbreviate $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ by $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root})$.

Proposition 5.5 *Let k be a natural number, σ be a label, $\mathbf{t}$ and $\mathbf{s}$ be two trees, $\mathbf{v}$ be a node of $\mathbf{t}$ with children $\mathbf{v}_1, \ldots, \mathbf{v}_n$, and $\mathbf{w}$ be a node of $\mathbf{s}$ with children $\mathbf{w}_1, \ldots, \mathbf{w}_m$.*

1. *If $(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{w}}, \mathbf{w})$ and $(\mathbf{t}_\mathbf{v}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{w}, \mathbf{w})$, then $(\mathbf{t}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}, \mathbf{w})$.*

2. *If $(\sigma(\mathbf{t}_{\mathbf{v}_1}, \ldots, \mathbf{t}_{\mathbf{v}_{n-1}}), \mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{w}_1}, \ldots, \mathbf{s}_{\mathbf{w}_{m-1}}), \mathrm{root})$ and*

$$(\mathbf{t}_{\mathbf{v}_n}, \mathbf{v}_n) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{w}_m}, \mathbf{w}_m),$$

   then $(\mathbf{t}_\mathbf{v}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{w}, \mathbf{w})$.

3. *Let the label of $\mathbf{v}$ and $\mathbf{w}$ be σ. For $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, if*

   - *$(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{w}}, \mathbf{w})$,*
   - *$(\sigma(\mathbf{t}_{\mathbf{v}_1}, \ldots, \mathbf{t}_{\mathbf{v}_{i-1}}), \mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{w}_1}, \ldots, \mathbf{s}_{\mathbf{w}_{j-1}}), \mathrm{root})$,*
   - *$(\sigma(\mathbf{t}_{\mathbf{v}_{i+1}}, \ldots, \mathbf{t}_{\mathbf{v}_n}), \mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{w}_{j+1}}, \ldots, \mathbf{s}_{\mathbf{w}_m}), \mathrm{root})$, and*
   - *the label of $\mathbf{v}_i$ equals the label of $\mathbf{w}_j$,*

   then $(\overline{\mathbf{t}_{\mathbf{v}_i}}, \mathbf{v}_i) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_{\mathbf{w}_j}}, \mathbf{w}_j)$.

Proof. We focus on the third case where there are altogether 4 subgames including the trivial game in which one structure consists only of $\mathbf{v}i$ and the other of $\mathbf{w}j$. The winning strategy in the game on $(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i)$ and $(\overline{\mathbf{s}_{\mathbf{w}j}}, \mathbf{w}j)$ just combines the winning strategies in those 4 subgames. At the end of the game, the selected vertices define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures again one only has to check the relations $<$ and E between the chosen elements, and the distinguished constants $\mathbf{v}i$ and $\mathbf{w}i$. similarly to the proof of Proposition 5.3, this immediately follows from the following observation. The distinguished constants in the subgames make sure that (i) whenever in the game on $(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i)$ and $(\overline{\mathbf{s}_{\mathbf{w}j}}, \mathbf{w}j)$ a child of $\mathbf{v}$

($\mathbf{w}$) is chosen, the duplicator has to reply with a child of $\mathbf{w}$ ($\mathbf{v}$); and, (*ii*) whenever $\mathbf{v}$ ($\mathbf{w}$) is chosen, the duplicator has to reply with $\mathbf{w}$ ($\mathbf{v}$). ■

We will need the following lemma in Section 5.4.

Lemma 5.6 *Let k be a natural number. There exists a DBTA^u $B = (Q, \Sigma, \delta, F)$ such that $\delta^*(\mathbf{t}) = \tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$, for every unranked tree $\mathbf{t}$.*

Proof. Define Q as the set Φ_k. Here we take Φ_k as the set of $\equiv_k^{\mathrm{MSO}}$-types of trees with one distinguished node. For each $\theta \in \Phi_k$ and $\sigma \in \Sigma$, define $\delta(\theta, \sigma)$ as the regular language defined by the automaton $M_{\theta,\sigma} = (\Phi_k, \Phi_k, s_M, \delta_M, F_M)$. This automaton is defined as follows:

$$s_M = \{\tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma), \mathrm{root})\};$$

$F_M = \{\theta\}$; and for each $\theta_1, \theta_2 \in \Phi_k$, $\delta_M(\theta_1, \theta_2) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{t}), \mathrm{root})$ where $\mathbf{t}_1, \ldots, \mathbf{t}_n$, and $\mathbf{t}$ are trees such that $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n), \mathrm{root}) = \theta_1$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}) = \theta_2$. By Proposition 5.5(2), it does not matter which members of the classes θ_1 and θ_2 we choose. ■

5.2 First approach

A first approach to define query automata for unranked trees is to add a selection function to the two-way deterministic automata for unranked trees as defined by Brüggemann-Klein, Murata and Wood [9]. However, it will turn out that these automata cannot even compute all first-order logic definable queries.

Definition 5.7 A *two-way deterministic unranked tree automaton* (2DTA^u) is a tuple $A = (Q, \Sigma, F, s, \delta)$, where Q, F, s, U, D, δ_{leaf} and δ_{root} are as in Definition 4.1. The transition function for up transitions is now of the form $\delta_\uparrow : U^* \to Q$, and the transition function for down transitions is of the form $\delta_\downarrow : D \times \mathbb{N} \to Q^*$. For each $(q, a) \in D$, $L_\downarrow(q, a) := \{\delta_\downarrow(q, a, i) \mid i \in \mathbb{N}\}$ is regular; for each $j \in \mathbb{N}$, $\delta_\downarrow(q, a, j)$ must be a string of length j; and for each $q \in Q$ the language $L_\uparrow(q) := \{w \in U^* \mid \delta_\uparrow(w) = q\}$ must be regular. To assure determinism, we require that $L_\uparrow(q) \cap L_\uparrow(q') = \emptyset$ for all $q \neq q'$.

The definitions of configuration, leaf, root, up and down transitions,[9] run, and accepting run carry over from QA^{r}s.

[9]Note that $\delta_\downarrow$ is uniquely determined by the regular languages $L_\downarrow(q, a)$.

We next argue that each transition in a run of the automaton takes linear time. To this end we elaborate on the structure of the regular languages $L_\downarrow(q,a)$. Each such language contains for each $n \in \mathbb{N}$, at most one string of length n. Shallit [40] has shown that such languages can be described by finite unions of regular expressions of the form xy^*z, where x, y, and z are strings. Hence, we can assume all languages $L_\downarrow(q,a)$ are represented by such languages. Suppose the automaton makes a down transition in state q at a node $\mathbf{v}$ with label a and arity n. Then all we have to do is look up in $L_\downarrow(q,a)$ the string of length n, if it exists. This can clearly be done in time linear in the size of the input tree when $L_\downarrow(q,a)$ is represented by finite unions of regular expressions of the above simple form. We represent all regular languages $L_\uparrow(q)$ by deterministic finite acceptors. Suppose in a configuration c the automaton makes an up transition at the children of a node $\mathbf{v}$. Then we just have to check for each q whether $\pi(c,\mathbf{v})$ (cf. Section 4.1) belongs to $L_\uparrow(q)$. This can also be done in time linear in the size of the input tree.

Each two-way tree automaton can readily simulate a one-way one. Indeed, let $B = (Q_B, \Sigma, \delta_B, F_B)$ be a bottom-up deterministic tree automaton over unranked trees. For ease of exposition assume all transitions of B are defined, that is, for each $q_1 \cdots q_n \in Q_B^*$ there exists $q \in Q_B$ and $\sigma \in \Sigma$ such that $q_1 \cdots q_n \in \delta_B(q,\sigma)$. Then define the two-way automaton $A = (Q, \Sigma, F, s, \delta)$ simulating B as follows. First, A runs to the leaves of the input tree $\mathbf{t}$. From thereon it uses functions $f : \Sigma \to Q_B$ as states with the following intended meaning: A assigns f to a node $\mathbf{v}$ such that $\delta_B^*(\mathbf{t}_\mathbf{v}) = f(\sigma)$ whenever $\mathrm{lab}_\mathbf{t}(\mathbf{v}) = \sigma$. Thus to each leaf A assigns the state f with $f(\sigma) = q$ such that $\varepsilon \in \delta_B(q,\sigma)$ for each $\sigma \in \Sigma$. Up transitions are defined as follows: $(f_1,\sigma_1)\cdots(f_n,\sigma_n) \in L_\uparrow(f)$ whenever for every $\sigma \in \Sigma$ we have that $f_1(\sigma_1)\cdots f_n(\sigma_n) \in \delta_B(q,\sigma)$, where $f(\sigma) = q$. Clearly, each $L_\uparrow(f)$ is regular. Furthermore, A accepts when $f(\mathrm{lab}_\mathbf{t}(\mathrm{root}(\mathbf{t}))) \in F_B$ where f is the state assigned to the root.

Definition 5.8 An *unranked query automaton* (QA^u) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a $\mathrm{2DTA}^u$ and λ is a mapping $\lambda : Q \times \Sigma \to \{\bot, 1\}$.

The query computed by a QA^u and the tree language defined by a QA^u are defined analogously to QA^rs.

Example 5.9 Consider Boolean circuits consisting of AND and OR gates that have one output but can have an arbitrary number of inputs. The

following query automaton selects all nodes of the input tree that evaluate to a 1. Again, we only consider trees as inputs that represent Boolean circuits.

Define the $\mathrm{QA}^u A = (Q, \Sigma = \{\mathrm{AND}, \mathrm{OR}, 0, 1\}, F, s, \delta)$, with

$$Q = \{s, u, \mathit{all\_one}, \mathit{all\_zero}, \mathit{mixed}\},$$

$D = \{s\} \times \Sigma$, $U = \{u, \mathit{all\_one}, \mathit{all\_zero}, \mathit{mixed}\} \times \Sigma$, and $F = Q$. Define

1. for any natural number n and $\sigma \in \Sigma$, $\delta_{\downarrow}(s, \sigma, n) = s \cdots s$ (n times);
2. for all $\sigma \in \Sigma$, $\delta_{\mathrm{leaf}}(s, \sigma) = u$;
3. $(q_1, \sigma_1) \cdots (q_n, \sigma_n) \in L_{\uparrow}(\mathit{all\_one})$ iff for all $i \in \{1, \ldots, n\}$
   - if $q_i = u$ then $\sigma_i = 1$;
   - if $\sigma_i = \mathrm{AND}$ then $q_i = \mathit{all\_one}$; and
   - if $\sigma_i = \mathrm{OR}$ then $q_i = \mathit{mixed}$ or $q_i = \mathit{all\_one}$.
4. $(q_1, \sigma_1) \cdots (q_n, \sigma_n) \in L_{\uparrow}(\mathit{all\_zero})$ iff for all $i \in \{1, \ldots, n\}$
   - if $q_i = u$ then $\sigma_i = 0$;
   - if $\sigma_i = \mathrm{AND}$ then $q_i = \mathit{all\_zero}$ or $q_i = \mathit{mixed}$; and
   - if $\sigma_i = \mathrm{OR}$ then $q_i = \mathit{all\_zero}$.
5. $L_{\uparrow}(\mathit{mixed}) := U^* - (L_{\uparrow}(\mathit{all\_one}) \cup L_{\uparrow}(\mathit{all\_zero}))$.

The automaton first walks to the leaves (1) and then changes state s into state u (2). Hereafter, it walks back up again assigning to each inner node the state *all_one*, *all_zero* or *mixed*, depending on whether the evaluation of the subtrees of this node returns only ones, only zeros, or both ones and zeros, respectively (3–5). Consider for example (3): an internal node is assigned *all_one* if all the trees rooted at its children evaluate to 1. That is, first, if a child is a leaf then it should be labeled with a 1. Next, if a child is labeled with AND then it should be assigned the state *all_one*, as all his children in turn should evaluate to 1. Finally, if a child is labeled with OR then it should be assigned the state *all_one* or *mixed*, as at least one of this node's children should evaluate to 1.

The selection function is now defined as follows: for all $q \in Q$ and $\mathrm{op} \in \Sigma$, $\lambda(q, \mathrm{op}) = 1$ if and only if $q = \mathit{all\_one}$ and $\mathrm{op} \in \{\mathrm{AND}, \mathrm{OR}\}$, or $q = \mathit{mixed}$ and $\mathrm{op} = \mathrm{OR}$. ■

Even though QA^{u}s can accept all recognizable tree languages, they cannot even compute all first-order logic definable queries as is illustrated next.

Proposition 5.10 *Unranked query automata cannot compute all queries definable in first-order logic.*

Proof. Let Σ be the alphabet $\{0, 1\}$. Consider the query "select all 1-labeled leaves for which there is no node among their left siblings that is labeled with a 1". Towards a contradiction, suppose there exists a QA^u A that computes this query. Let Q be the set of states of A and let $m = |Q|$. The crucial observation is that there exist at most $m!$ different sequences of states that A can take at the root of a tree. We set $n := m!$. For $i = 0, \ldots, n$, let $\mathbf{t}_i$ be the tree consisting of a root (say, labeled 0) with $n + 1$ children, where the first i children are labeled with 0 and the others are labeled with 1. There now exist $j, j' \in \{0, \ldots, n\}$ such that $j < j'$ and A goes through the same sequence of root states for $\mathbf{t}_j$ and $\mathbf{t}_{j'}$. Since for each state and for each arity there is only one string of states that can be assigned to the children, the set of states assumed by A at the $(j' + 1)$-th leaf of $\mathbf{t}_j$ is the same as the set of states assumed by A at the $(j' + 1)$-th leaf of $\mathbf{t}_{j'}$. Since both leaves carry a 1, A selects them both or does not select them at all. This leads to the desired contradiction. ■

QA^{u}s cannot compute the query in the proof of Proposition 5.10 because they cannot pass enough information from one sibling to another. Indeed, when the automaton makes a down transition at some node $\mathbf{v}$, it assigns a state to every child of $\mathbf{v}$; even though every child knows its own state, it cannot know in general which states are assigned to its siblings. To resolve this, we introduce in the next section query automata with "stay transitions". Such transitions are represented by two-way string-automata which process the string formed by the states and the labels of the children of a certain node, and then output a new state for each child.

5.3 Strong query automata

Tree automata with stay transitions are defined next.

Definition 5.11 A *generalized two-way deterministic unranked tree automaton* (G2DTA^u) is a tuple $A = (Q, \Sigma, F, s, \delta)$, where Q, F, s, U, D, δ_{leaf}, δ_{root} and $\delta_{\downarrow}$ are defined as in Definition 5.7. Let U_{up} and U_{stay} be two disjoint

regular subsets of U^*. Then $\delta_\uparrow$ is a function $\delta_\uparrow : U_{\text{up}} \to Q$ (here the same conditions apply as in Definition 3.1), and $\delta_- : U_{\text{stay}} \to Q^*$ is the transition function for stay transitions. We require that this function is computed by a generalized string query automaton (cf. Definition 3.5).

Most definitions remain the same as for QA^{u}s. Only, we now also have stay transitions: A makes a *stay transition* from a configuration $c_1 : C_1 \to Q$ to a configuration $c_2 : C_2 \to Q$ if there is a node $\mathbf{v}$ in $\mathbf{t}$ such that

(i) $\text{children}(\mathbf{v}) \subseteq C_1$,

(ii) $C_2 = C_1$,

(iii) $\delta_-(\pi(c_1, \mathbf{v})) = c_2(\text{children}(\mathbf{v}))$, and

(iv) c_1 is identical to c_2 on $C_1 \cap C_2$.

The above defined automata are much more expressive than MSO. Indeed, they can for instance simulate linear space Turing Machines on trees of depth one. Therefore, we restrict them in the following way:

Definition 5.12 A *strong* two-way deterministic unranked tree automaton (S2DTA^u) is a G2DTA^u that makes at most one stay transition for the children of each node.

In Lemma 5.16 we show that the behavior of a S2DTA^u can be defined in MSO. It is then not difficult to construct an MSO sentence asserting that a particular G2DTA^u makes two stay transitions at the children of a particular node. Since satisfiability of MSO sentences on trees is decidable, again we can conclude that it is decidable whether a G2DTA^u is a S2DTA^u.

A strong query automaton is an S2DTA^u extended with a selection function.

Definition 5.13 A *strong query automaton* (SQA^u) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a S2DTA^u and λ is a function from $Q \times \Sigma$ to $\{\bot, 1\}$.

We illustrate the above with an example.

Example 5.14 Recall the query of the proof of Proposition 5.10, *select all 1-labeled leaves for which there is no node among their left siblings labeled with a 1.* This query can be computed by a SQA^u. Indeed, let $A = (Q, \Sigma, F, s, \delta, \lambda)$ be the SQA^u with $Q = F = \{s, \mathit{stay}, \mathit{up}, 1\}$, $D = \{s\} \times \Sigma$, $U = \{\mathit{stay}, \mathit{up}, 1\} \times \Sigma$, and where

- $U_{\text{stay}} = (\{stay\} \times \Sigma)^*$,
- $U_{\text{up}} = U^* - U_{\text{stay}}$,
- for each natural number n and $\sigma \in \Sigma$, $\delta_{\downarrow}(s, \sigma, n) = s \cdots s$ (n times),
- for each $\sigma \in \Sigma$, $\delta_{\text{leaf}}(s, \sigma) = stay$;
- δ_{-} is computed by the GSQA that assigns 1 to the first 1-labeled node and up to the others, and
- $L_{\uparrow}(up) = up^* 1 up^* + up^*$.

The automaton walks to the leaves, makes one stay transition, and then walks back to the root. The selection function is defined as follows: for each $\sigma \in \Sigma$ and $q \in Q$, $\lambda(q, \sigma) = 1$ iff $q = 1$. ■

5.4 Expressiveness

We next prove that a query is computable by a SQA^u if and only if it is definable in MSO. But first, we emphasize the remarkable difference between tree automata and query automata over unranked trees. Indeed, as shown in the next proposition, stay transitions do not increase the expressiveness with respect to defining tree languages. However, stay transitions do make a difference with respect to computing queries, as was shown in Proposition 5.10 and Example 5.14.

Proposition 5.15 *Every* S2DTA^u *is equivalent to a* 2DTA^u *accepting the same tree language.*

Proof. This follows directly from Theorem 5.4 above and Lemma 5.16 below. ■

We first generalize Lemma 4.7 to query automata over unranked trees.

Lemma 5.16 *Every query computed by an unranked query automaton can be defined in MSO.*

Proof. The proof is similar to the proof of Lemma 4.7. We use some of the notation introduced there. Given an SQA^u $A = (Q, \Sigma, F, s, \delta)$, we again guess sets $Z_{f,B}$ and check their consistency. On input $\mathbf{t}$ these sets have the following

intended meaning: a node $\mathbf{v} \in Z_{f,B}$ iff $f = f^A_{\mathbf{t}_\mathbf{v}}$ and $B = \text{Assumed}^A(\mathbf{t}, \mathbf{v})$. As opposed to the proof of Lemma 4.7, the consistency check can no longer be specified in first-order logic because the correctness of the guesses depends on the transition functions $\delta_\uparrow$, $\delta_\downarrow$ and δ_- which are no longer finite functions, but are given by regular languages and by a GSQA. However, the correctness can easily be verified in MSO because, by Theorem 2.6 and Theorem 2.12, regular languages and GSQAs can be defined in MSO. Further, the correctness of the behavior functions crucially depends on our assumption that at most one stay transition can occur at the children of each node. Indeed, suppose a node $\mathbf{v}$ is labeled with transition function f and its n children are labeled with $f_1, \ldots, f_n$. Then we have to check for all states q and q' with $f(q) = q'$ that[10]

1. $q = q'$ if $(q, \text{lab}_\mathbf{t}(\mathbf{v})) \in U$;
2. if $(q, \text{lab}_\mathbf{t}(\mathbf{v})) \in D$ then there exist states $q_1, \ldots, q_n$ such that
$$\delta_\downarrow(q, \sigma, n) = q_1 \cdots q_n;$$
and either
   (a) for each $i \in \{1, \ldots, n\}$, $\text{up}(f_i, q_i) \in U$: in this case we should have
   $$\delta_\uparrow((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)) = q';$$
   or
   (b) for each $i \in \{1, \ldots, n\}$, $\text{up}(f_i, q_i) \in U_{\text{stay}}$: in this case there should exist $q'_1, \ldots, q'_n$ with
   $$\delta_-((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)) = q'_1 \cdots q'_n,$$
   and
   $$\delta_\uparrow(\text{up}(f_1, q'_1), \sigma_1) \cdots (\text{up}(f_n, q'_n), \sigma_n)) = q'.$$

   Finally, if $f(q)$ is undefined then $\delta_\downarrow(q, \sigma, n)$ should be undefined; or in case (2a) $\delta_\uparrow((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n))$ or one of the $\text{up}(f_i, q_i)$ should be undefined; in case (2b)
   $$\delta_-((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)),$$

[10] As in the ranked case, $\text{States}(f_1, \ldots, f_n, q)$ is the smallest set of states containing q and closed under applications of every f_i. We then define $\text{up}(f, q)$ as the unique state q' in $\text{States}(f, q)$ for which $f(q') = q'$. If there is no such state, then $\text{up}(f, q)$ is undefined.

$\delta_\uparrow(\mathrm{up}(f_1, q_1'), \sigma_1) \cdots (\mathrm{up}(f_n, q_n'), \sigma_n))$, or one of the $\mathrm{up}(f_i, q_i)$ or $\mathrm{up}(f_i, q_i')$ should be undefined.

From our assumption that an SQA^u can make at most one stay transition at the children of each node, it follows that the case distinctions (2a) and (2b) suffice. ■

We are now ready to prove the main result of this section.

Theorem 5.17 *A query is computable by a* SQA^u *if and only if it is definable in MSO.*

Proof. The only-if direction was given in Lemma 5.16.

Let $\varphi(x)$ be an MSO-formula of quantifier depth k. We describe an SQA^u that computes the query which is defined by φ. This automaton has to find out, for each node $\mathbf{v}$ of a tree $\mathbf{t}$, whether $\mathbf{t} \models \varphi[\mathbf{v}]$. This depends only on $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$, which in turn, by Proposition 5.5(1), depends only on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_\mathbf{v}, \mathbf{v})$. The case where trees can also have nodes with one child can be treated as in the proof of Theorem 4.8; hence, we can assume that all inner nodes have more than one child. In Figure 6, we describe an algorithm that evaluates φ.

The $\equiv_k^{\mathrm{MSO}}$-type of a subtree $(\mathbf{t}_\mathbf{w}, \mathbf{w})$ can be computed in a bottom-up manner by the automaton of Lemma 5.6. This automaton can be transformed to an equivalent two-way automaton as discussed at the end of Section 5.2. Note that the two-way automaton starts at $\mathbf{w}$.

Step 1 is now done in two phases. We re-use the pebbling idea from the proof of Theorem 4.8. First, the automaton makes a down transition. All children of $\mathbf{v}$, besides $\mathbf{v}1$ enter a U-state which remembers $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ and waits until the computation in the subtree $\mathbf{t}_{\mathbf{v}1}$ has finished. The $\equiv_k^{\mathrm{MSO}}$-type of this subtree is computed bottom-up. The automaton "recognizes" that the subtree-evaluation is finished by meeting the U-states at the siblings of $\mathbf{v}1$. Next it makes an up transition, followed by a down transition. After this, $\mathbf{v}1$ has a U-state which remembers $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_\mathbf{v}}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)$ and waits for the termination of the evaluation of the other subtrees which are computed in parallel. This evaluation simultaneously computes $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}j}, \mathbf{v}j)$, for each $j > 1$.

Step 2 is just a special case of step 1. Indeed, since the types of the subtrees of $\mathbf{t}_\mathbf{v}$ are present at the children of $\mathbf{v}$, they can be combined to the type of $\mathbf{t}_\mathbf{v}$ by making an up transition. Step 3 and 4 only involve information that is available at vertex $\mathbf{v}$.

Input: $\mathbf{t}$
Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathrm{root}(\mathbf{t})}}, \mathrm{root}(\mathbf{t}))$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$
for $i := 0$ **to** depth of $\mathbf{t}$ **do**
begin
  for all vertices $\mathbf{v}$ of level i **do**
  begin
    % the root is level 0
    % the type of $(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ *has already been computed*
    % now compute the type of $(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$
    1. **for** $j = 1, \ldots,$ *arity*$(\mathbf{v})$ **do** compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}j}, \mathbf{v}j)$
    2. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$ from $\mathrm{lab}_{\mathbf{t}}(\mathbf{v})$ and the $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}j}, \mathbf{v}j)$
    3. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}}, \mathbf{v})$
    4. Deduce from $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{v})$ whether $\varphi(\mathbf{v})$ holds
       If so, select $\mathbf{v}$
    5. **for** $j = 1, \ldots,$ *arity*$(\mathbf{v})$ **do**
         Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}j}}, \mathbf{v}j)$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ and the $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}j'}, \mathbf{v}j')$
  end
end

Figure 6: The algorithm for computing the query defined by $\varphi(x)$ over unranked trees.

It then only remains to show how step 5 can be done by an SQA^u. It should be noted that after the up transition of step 2 the information about the types of the subtrees of $\mathbf{v}$ is lost. Therefore the SQA^u first recomputes the $\equiv_k^{\mathrm{MSO}}$-types $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}j}, \mathbf{v}j)$, as described above (also keeping $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$ in mind).

We now show that there is a GSQA B computing the sequence

$$\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}1}}, \mathbf{v}1) \cdots \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}n}}, \mathbf{v}n)$$

on input

$$(\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v}), \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}1}, \mathbf{v}1)) \cdots (\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v}), \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}n}, \mathbf{v}n)),$$

for a tree $\mathbf{t}$ and $\mathbf{v}$ a node of $\mathbf{t}$ of arity n. Let σ be the label of $\mathbf{v}$. Then, by Proposition 5.5(3), for each $i = 1, \ldots, n$, $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i)$ only depends on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}}}, \mathbf{v})$, $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{v}1}, \ldots, \mathbf{t}_{\mathbf{v}(i-1)}), \mathbf{v})$, $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}i)$ (which depends only on $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{v}i}, \mathbf{v}i)$), and $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{v}(i+1)}, \ldots, \mathbf{t}_{\mathbf{v}n}), \mathbf{v})$.

Now, B is defined as the automaton combining, as specified in Lemma 3.10, the automata B_1 and B_2, where B_1 computes $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{v}1} \ldots \mathbf{t}_{\mathbf{v}(i-1)}), \mathbf{v})$ and B_2 computes $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{v}(i+1)} \ldots \mathbf{t}_{\mathbf{v}n}), \mathbf{v})$. So, at each position i the automaton B has enough information to output $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{v}i}}, \mathbf{v}i)$.

Hence, step 5 can be done by recomputing the $\equiv_k^{\mathrm{MSO}}$-types $\tau_k(\mathbf{t}_{\mathbf{v}j}, \mathbf{v})$ (in the same way as in step 2) and making one stay transition. ■

Remark 5.18 Allowing an SQA^u to make any constant number of stay transitions at the children of each node does not increase the expressiveness of the formalism. Indeed, like in the proof of Theorem 5.17 such an automaton can be simulated in MSO.

6 Decision problems

Optimization of queries is one of the most studied subjects in database theory [3]. It involves, for instance, the rewriting of given queries into equivalent ones that can be evaluated more efficiently, as also the detection of subqueries that always evaluate to the empty set. Checking equivalence and nonemptiness of queries are, therefore, fundamental operations. Although the general problem of deciding whether two queries are equivalent or the result

of a query is always empty, is usually undecidable, their language-theoretic counter parts, equivalence and emptiness of automata, are well-known to be decidable. We next establish the complexity of these decision problems for our automata who express queries. The above mentioned problems, as well as a related one, are defined as follows:

Non-emptiness: Given a query automaton A, is there a tree $\mathbf{t}$ such that $A(\mathbf{t}) \neq \emptyset$?

Containment: Given two query automata A_1 and A_2, is the query computed by A_1 contained in the query computed by A_2? That is, is $A_1(\mathbf{t}) \subseteq A_2(\mathbf{t})$ for all trees $\mathbf{t}$?

Equivalence: Given two query automata A_1 and A_2, do they compute the same query? That is, is $A_1(\mathbf{t}) = A_2(\mathbf{t})$ for all trees $\mathbf{t}$?

We show that these problems are EXPTIME-complete for QA^{r}s, QA^{u}s and SQA^{u}s. EXPTIME-hardness of all these problems follows from EXPTIME-hardness of the non-emptiness problem for QA^{r}s. EXPTIME-membership follows from EXPTIME-membership of the non-emptiness problem for SQA^{u}s, since we will show that containment and equivalence can be reduced to non-emptiness in polynomial time and query automata on ranked trees are special cases of query automata on unranked trees.

The *size* of an SQA^u is the sum of the sizes of the DFAs representing the up transitions and U_{stay}, the sizes of the automata for the stay transitions, the sizes of the regular expressions representing the down transitions, and the size of the set of states of the SQA^u. We point out in the proof of Theorem 6.3, why we need DFAs, as opposed to NFAs, for the representation of up transitions.

We start by observing that deciding whether the tree language defined by a 2DTA^r is non-empty is EXPTIME-hard. We use a reduction from the TWO PERSON CORRIDOR TILING which is known to be hard for EXPTIME [11].

For natural numbers n and m we view $\{1, \ldots, n\} \times \{1, \ldots, m\}$ as a rectangle consisting of m rows of width n. Let T be a finite set of tiles, let $H, V \subseteq T \times T$ be horizontal and vertical constraints, and let $\bar{b} = b_1, \ldots, b_n, \bar{t} = t_1, \ldots, t_n \in T^n$ be the bottom and the top row. A *corridor tiling* from $\bar{b}$ to $\bar{t}$ is a mapping $\lambda : \{1, \ldots, n\} \times \{1, \ldots, m\} \to T$, for some natural number m, such that

- the first row is $\bar{b}$, that is, $\lambda(1,1) = b_1, \ldots, \lambda(1,n) = b_n$;
- the m-th row is $\bar{t}$, that is, $\lambda(m,1) = t_1, \ldots, \lambda(m,n) = t_n$;
- for $i = 1, \ldots, n-1$ and $j = 1, \ldots, m$, $(\lambda(i,j), \lambda(i+1,j)) \in H$; and
- for $i = 1, \ldots, n$ and $j = 1, \ldots, m-1$, $(\lambda(i,j), \lambda(i,j+1)) \in V$.

In a two person corridor tiling game from $\bar{b}$ to $\bar{t}$, two players, on turn, place tiles row wise from bottom to top, and from left to right in each row. The first player starts and each newly placed tile should be consistent with the tiles already placed. The first player tries to make a corridor tiling from $\bar{b}$ to $\bar{t}$, whereas the second player tries to prevent this. If the first player always can achieve such a tiling no matter how the second player plays, then we say that player one *wins* the corridor game. A player that puts down a tile not consistent with the tiles already placed, immediately looses.

TWO PERSON CORRIDOR TILING is the problem to decide, given a set of tiles T, $H, V \subseteq T \times T$, a sequence of tiles $\bar{b} = b_1, \ldots, b_n$ and $\bar{t} = t_1, \ldots, t_n \in T^n$, whether player one wins the corridor game.

Proposition 6.1 *Deciding whether a deterministic two-way ranked tree automaton accepts any tree is hard for EXPTIME.*

Proof. We reduce TWO PERSON CORRIDOR TILING to non-emptiness of 2DTA^r. A strategy for player one can be represented by a tree where the nodes are labeled with tiles. Indeed, if we put the rows of a tiling next to each other rather than on top of each other, then every branch, i.e., the sequence of labels from the root to a leaf, of a tree represents a possible tiling. If we forget about the start row $\bar{b}$ for a moment, then the odd depth nodes have no siblings and represent moves of player one and the even depth nodes do have siblings and represent all the choices of player two. A strategy is then winning when every branch is either a corridor tiling or is a tiling where player two made a false move. The 2DTA^r A we construct will only accept trees that correspond to winning strategies for player one. The automaton essentially only has to check the horizontal and vertical constraints. The vertical constraints at a node $\mathbf{v}$ of the input tree can be checked by moving up n nodes (the width of the corridor), while the horizontal constraints can be checked by looking at the tile carried by the parent of $\mathbf{v}$.

We formally define when a tree represents a winning strategy. Take Σ as $\{0,1,2\} \times \{1, \ldots, n\} \times T$. If a node is labeled with (i,j,t) and $i \neq 0$, then

this means that player i places tile t on the j-th position of the current row. The case $i = 0$ is just to define the first n nodes which are labeled with b_1, $\ldots$, b_n. We say that a Σ-tree $\mathbf{t}$ represents a winning strategy for the first player if the following holds:

1. $\mathbf{t}$ starts with a monadic tree labeled with $\bar{b}$. That is, the root carries the label $(0, 1, b_1)$, the only child of the root carries the label $(0, 2, b_2)$, and so on.

2. If there exists a node of depth n (recall that the root has depth 0) then there exists only one such node (say $\mathbf{v}$) and additionally $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}) = (1, 1, t)$ for some $t \in T$: player one places the tile on the first column of the second row whenever there is a second row; indeed, $\bar{b}$ and $\bar{t}$ can already form a corridor tiling.

3. For every internal node $\mathbf{v}$ of $\mathbf{t}$, if $\mathrm{lab}_{\mathbf{t}}(\mathbf{p}) = (i, j, t)$ with $i \in \{1, 2\}$ and $\mathbf{p}$ the parent of $\mathbf{v}$, then $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}) = ((i \bmod 2) + 1, (j + 1) \bmod n, t')$ for some $t' \in T$; this means that players one and two place tiles on turn.

4. No two siblings are labeled with the same label; and, nodes corresponding to moves of player one have no siblings.

5. Every alternative of player two should be present: for every node $\mathbf{v}$ with $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}) = (1, j, t)$ and for every $t' \in T$ there is a child $\mathbf{w}$ of $\mathbf{v}$ labeled with $(2, (j + 1) \bmod n, t')$.

6. Each branch extended with $\bar{t}$ corresponds to a corridor tiling from $\bar{b}$ to $\bar{t}$ or should contain a false move by player two.

The 2DTA^r A works on trees of rank n. Let N be $|T| + |H| + |V| + n$. Clearly (1)–(5) can be checked by A by using a number of states linear in N. We now consider (6). Suppose A arrives at $\mathbf{v}$. In order to check the horizontal constraints at $\mathbf{v}$, A already remembered the tile of the parent of $\mathbf{v}$ in its state when it moved down. To check the vertical constraints, A just has to move up n nodes to get the tile that is placed immediately below the square corresponding to $\mathbf{v}$. However, moving up requires the cooperation of all siblings. To this end, A moves through the tree level by level, and for each level makes n up transitions to get the required tile. Again, only a number of states that is linear in N is needed. Moreover, A can be computed in LOGSPACE. ■

To show that non-emptiness for SQA^{u}s is in EXPTIME we use the following device.

A *two-way deterministic finite automaton with one pebble* is a 2DFA that has one pebble which it can lay down on the input string and pick back up later. We refrain from giving a formal definition of such automata as we will only use them informally in the following to describe algorithmic computations. Blum and Hewitt [8] showed that such automata can only define regular languages. We will need the following stronger result obtained by Globerman and Harel [21, Proposition 3.2].

Proposition 6.2 *Every two-way deterministic finite automaton M with one pebble is equivalent to an NFA M' whose size is exponential in the size of M. In fact, the size of M' can be uniformly bounded by a function $p(|\Sigma|) \cdot 2^{q(|S|)}$, where p and q are polynomials, Σ is the alphabet, and S is the set of states of M. Additionally, M' can be constructed in time exponential in the size of M.*

We are now ready to prove the next theorem.

Theorem 6.3 *Non-emptiness of* SQA^u*s is in EXPTIME.*

Proof. We describe an EXPTIME algorithm which decides whether the SQA^u A is non-empty. The proof consists of two parts. First, we define a two-way deterministic unranked tree automaton A' such that A is non-empty iff A' accepts at least one tree (we then also say that A' is non-empty). Moreover, the size of A' is linear in the size of A. Subsequently, we show that testing non-emptiness of two-way deterministic unranked tree automata is in EXPTIME. This then implies that non-emptiness of SQA^{u}s is in EXPTIME.

Construction of A'. The two-way deterministic automaton A' works over the alphabet $\Sigma \cup (\Sigma \times \{1\})$. On input $\mathbf{t}$, it first checks whether there is exactly one node with a label in $\Sigma \times \{1\}$. This can be done by one traversal of the tree from the root to the leaves. If there is more than one such node or none at all, then A' rejects. Otherwise, A' walks back to the root and starts simulating A, that is, it just behaves like A would but without actually selecting nodes. Let $\mathbf{v}$ be the unique node with a label in $\Sigma \times \{1\}$. Then A' accepts when A does and when, additionally, A selects $\mathbf{v}$. The latter can be achieved by keeping a flag in the state of A' from the moment A selects $\mathbf{v}$. Clearly, the size of A' is linear in the size of A.

Testing non-emptiness of two-way deterministic unranked tree automata is in EXPTIME. Let $A' = (Q, \Gamma, F, s_0, \delta)$ be such an automaton. We construct a nondeterministic bottom-up automaton (NBTA^u) $B = (Q_B, \Gamma, F_B, \delta_B)$ (cf. Section 5.1) whose size is exponential in the size of A' with the additional property that B is non-empty iff A' is non-empty. By Lemma 5.2 we know that testing non-emptiness of NBTA^{u}s is in PTIME. Hence, testing non-emptiness of two-way deterministic automata is in EXPTIME.

The set of states Q_B consists of all tuples of the form (f, d, s, σ) where

- $f : Q \to Q$ is a partial function;
- $d : Q \to Q$ and $s : Q \to Q$ are total functions; and
- $\sigma \in \Gamma$.

To describe the intuition behind the components in the states of Q_B we introduce the following notion. A *state assignment* for a tree $\mathbf{t}$ is a mapping $\rho : \mathrm{Nodes}(\mathbf{t}) \to Q$. A state assignment ρ for $\mathbf{t}$ is *semi-valid* if for every node $\mathbf{v}$ of $\mathbf{t}$ of arity n, $\rho(\mathbf{v}1) \cdots \rho(\mathbf{v}n) \in \delta(\rho(\mathbf{v}), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}))$, and for every leaf node $\mathbf{v}$, $\varepsilon \in \delta(\rho(\mathbf{v}), \mathrm{lab}_{\mathbf{t}}(\mathbf{v}))$. We say that a state assignment ρ for a tree $\mathbf{t}$ is *valid* iff it is *semi-valid* and $\rho(\mathrm{root}(\mathbf{t})) \in F$. Clearly, a tree $\mathbf{t}$ is *accepted* by B if there exists a valid state assignment for it.

The intuition behind the states in Q_B is that for each semi-valid state assignment ρ for a tree $\mathbf{t}$, if $\rho(\mathbf{v}) = (f, d, s, \sigma)$ then $f^A_{\mathbf{t}_\mathbf{v}} = f$ and $\mathrm{lab}_{\mathbf{t}}(\mathbf{v}) = \sigma$. The functions d and s are just to facilitate the definition of the transition function of A' which we define next.

For all $n \geq 1$, $\sigma \in \Gamma$, and every state $(f, d, s, \sigma') \in Q_B$,

$$w = (f_1, d_1, s_1, \sigma_1) \cdots (f_n, d_n, s_n, \sigma_n) \in \delta_B((f, d, s, \sigma'), \sigma)$$

iff

1. $\sigma' = \sigma$;
2. $f(q) = q$ for each $q \in Q$ with $(q, \sigma) \in U$;
3. $\delta_\downarrow(q, \sigma, n) = d_1(q) \cdots d_n(q)$ for each $q \in Q$ with $(q, \sigma) \in D$ and for which $f(q)$ is defined; there are now two possibilities:

(a) for each $i \in \{1, \ldots, n\}$, $(f_i(d_i(q)), \sigma_i) \in U$: in this case we should have

$$\delta_\uparrow((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = f(q);$$

or

(b)

$$(f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n) \in U_{\text{stay}} :$$

in this case we should have

$$\delta_-((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = s_1(q) \cdots s_n(q),$$

and

$$\delta_\uparrow((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q))), \sigma_n) = f(q).$$

If $f(q)$ is undefined then $\delta_\downarrow(q, \sigma, n)$ should be undefined; or in case (3a) one of the $f_i(d_i(q))$ or $\delta_\uparrow(f_1(d_1(q)) \cdots f_n(d_n(q)))$ should be undefined; or in case (3b)

$$\delta_-(f_1(d_1(q)) \cdots f_n(d_n(q))),$$

$\delta_\uparrow((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q)), \sigma_n))$, or one of the $f_i(d_i(q))$ or $f_i(s_i(q))$ should be undefined.

From our assumption that an SQA^u can make at most one stay transition at the children of each node, it follows that the case distinctions (3a) and (3b) suffice.

Further, $\varepsilon \in \delta_B((f, d, s, \sigma'), \sigma)$ iff $f = f^A_{\mathbf{t}(\sigma)}$ and $\sigma = \sigma'$. Finally, define $F_B = \{(f, d, s, \sigma) \mid \text{States}(f, \delta_{\text{root}}(\cdot, \sigma), s_0) \cap F \neq \emptyset\}$.[11] It is now readily checked that for each semi-valid state assignment ρ for a tree $\mathbf{t}$, $\rho(\mathbf{v}) = (f, d, s, \sigma)$ iff $f^A_{\mathbf{t}_\mathbf{v}} = f$ and $\text{lab}_\mathbf{t}(\mathbf{v}) = \sigma$. By definition of F_B, we have that A' accepts $\mathbf{t}$ iff there exists a valid state assignment for $\mathbf{t}$. Consequently, A' is non-empty iff B is non-empty.

It remains to show that each $\delta_B((f, d, s, \sigma'), \sigma)$ can be accepted by an NFA whose size is exponential in the size of A'. We will define a two-way deterministic automaton M whose size is polynomial in A' that accepts $\delta_B((f, d, s, \sigma'), \sigma)$. By Lemma 6.2, M is equivalent to an NFA whose size is at most exponential in A'.

(1) does not depend on the input;

[11] Here, for each $\sigma \in \Gamma$, $\delta_{\text{root}}(\cdot, \sigma)$ is the function mapping each q to $\delta_{\text{root}}(q, \sigma)$.

(2) does not depend on the input;

(3) For each $q \in Q$ with $(q, \sigma) \in D$, we do the following. We only describe the case where $f(q)$ is defined, the converse case is similar. To test whether

$$\delta_{\downarrow}(q, \sigma, n) = d_1(q) \cdots d_n(q);$$

we just simulate the finite union of regular expressions representing $L_{\downarrow}(q, \sigma)$ on the string $d_1(q) \cdots d_n(q)$. This can be done by subsequently trying to match each regular expression in this union. Due to the very simple form of these regular expressions (namely xy^*z) this only needs a number of states linear in the size of the expressions. Hereafter, M tests whether

$$(f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n) \in U_{\text{stay}},$$

or whether $(f_i(d_i(q)), \sigma_i) \in U_{\text{stay}}$ for each $i \in \{1, \ldots, n\}$. This test is performed by another sweep through the input string w. Depending on this test M does the following.

(a) M simply simulates the DFA for $L_{\uparrow}(q)$; that is, M tests by another sweep through w whether

$$(f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n) \in L_{\uparrow}(q);$$

This only needs a number of states linear in the size of the automaton for $L_{\uparrow}(q)$.

(b) In the second case, M verifies that

$$\delta_{-}((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = s_1(q) \cdots s_n(q),$$

and that

$$((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q))), \sigma_n)) \in L_{\uparrow}(q).$$

The former can be done by simulating the GSQA for δ_{-}. Recall our convention that each GSQA only outputs one symbol at each position. The latter can be done by one sweep through the input string simulating the DFA for $L_{\uparrow}(q)$. Again only a linear number of states in the size of A is needed.

We briefly come back to why we need DFAs rather than NFAs: in the case where $f(q)$ is undefined we must check that an up transition is undefined for a certain sequence of states. When up transitions are represented by DFAs this is easy: we just simulate the automaton and see whether it gets stuck. For an NFA, however, this is much harder as we have to check that all computations are undefined.

■

We next show how to reduce containment to non-emptiness. Let A_1 and A_2 be two SQA^{u}s working over Σ-trees. Define the SQA^u B working over trees labeled with symbols from the alphabet $\Sigma \cup (\Sigma \times \{1\})$, as follows. On input $\mathbf{t}$, B first checks whether there is exactly one node with a label in $\Sigma \times \{1\}$. This can be done by one traversal of the tree from the root to the leaves. If there is more than one such node or none at all, then B rejects. Otherwise, let $\mathbf{v}$ be the unique node with a label in $\Sigma \times \{1\}$. Then B walks back to the root, first simulates A_1 and then A_2, and remembers which automaton selects $\mathbf{v}$. Recall our convention that we only consider automata that terminate on every input. If A_1 selects $\mathbf{v}$ and A_2 does not, then B selects $\mathbf{v}$. Otherwise, B does not select anything and halts. Hence, B is non-empty iff A_1 is *not* contained in A_2. As the size of B is linear in the sizes of A_1 and A_2, and non-emptiness is in EXPTIME, it follows that containment is in EXPTIME. As, clearly, A_1 is equivalent to A_2 iff each of the automata is contained in the other, we have the following theorem.

Theorem 6.4 *Equivalence and containment of* QA^r*s,* QA^u*s, and* SQA^u*s is in EXPTIME.*

7 Discussion

We introduced query automata for expressing unary queries on structured documents. We investigated their expressiveness and established the complexity of several decision problems relevant for optimization. We considered both ranked and unranked trees. The latter only recently received new attention in the context of SGML and XML. The theory of automata for unranked trees has been further developed by Brüggemann-Klein, Murata and Wood [9], and has been applied by Maneth and Neven [30], Murata [33], Neumann and Seidl [34], and Neven [35, 36]. In particular, we pointed out

a subtle difference between query automata on ranked and unranked trees. Indeed, while the extension of two-way automata on ranked trees with a selection function sufficed to capture all unary MSO queries, we needed special stay transitions for the unranked case.

Acknowledgements

We thank Joost Engelfriet, Wolfgang Thomas, Jan Van den Bussche, and Moshe Vardi for stimulating discussions. The initial idea of query automata is gratefully acknowledged to Jan Van den Bussche. Joost Engelfriet pointed us to Lemma 3.10.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *VLDB Journal*, 7(2):96–114, 1998.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. A general theory of translation. *Mathematical Systems Theory*, 3:193–221, 1969.

[5] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, March 1996.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[7] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In P. Buneman C. Beeri, editor, *Database Theory – ICDT99*, volume 1540 of *Lecture Notes in Computer Science*, pages 296–313. Springer-Verlag, 1998.

[8] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Conference Record of 1967 Eighth Annual Symposium on Switching and Automata Theory*, pages 155–160. IEEE, 1967.

[9] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets (draft 1). Unpublished manuscript, 1998.

[10] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[11] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[12] J. Clark and S. Deach. Extensible stylesheet language (XSL). http://www.w3.org/TR/WD-xsl.

[13] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[14] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5. Elsevier, 1990.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. http://www8.org/.

[16] J. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[17] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1995.

[18] J. Engelfriet. Two-way automata and checking automata. *Mathematical Centre Tracts*, pages 1–69, 1979.

[19] J. Engelfriet and H. J. Hoogeboom. Two-way finite state transducers and monadic second-order logic. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 311–320. Springer, 1999.

[20] F. Gécseg and M. Steinby. Tree languages. In Rozenberg and Salomaa [39], chapter 1.

[21] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.

[22] G.H. Gonnet and F.W. Tompa. Mind your grammar: a new approach to modelling text. In *Proceedings 13th Conference on VLDB*, pages 339–346, 1987.

[23] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. *SIAM Journal on Computing*, 23(6):1093–1137, 1994.

[24] J. E. Hopcroft and J. D. Ullman. An approach to a unified theory of automata. *The Bell Systems Technical Journal*, 46:1793–1829, 1967.

[25] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[26] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured text database system. In R. Furuta, editor, *Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, The Cambridge Series on Electronic Publishing, pages 139–151. Cambridge University Press, 1990.

[27] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the Sixteenth International Conference on Research and Development in Information Retrieval*, pages 214–222. ACM Press, 1993.

[28] P. Kilpeläinen and H. Mannila. Query primitives for tree-structured data. In M. Crochemore and D. Gusfield, editors, *Proceedings of the fifth Symposium on Combinatorial Pattern Matching*, pages 213–225. Springer-Verlag, 1994.

[29] R. E. Ladner. Application of model theoretic games to discrete linear orders and finite automata. *Information and Control*, 33(4):281–303, April 1977.

[30] S. Maneth and F. Neven. A formalization of tree transformations in XSL. To appear in the proceedings of the Seventh International Workshop on

Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[31] E. Moriya. On two-way tree automata. *Information Processing Letters*, 50:117–121, 1994.

[32] M. Murata. Forest-regular languages and tree-regular languages. Unpublished manuscript, 1995.

[33] M. Murata. Data model for document transformation and assembly. In *Proceedings of the workshop on Principles of Digital Document Processing*, 1998. To appear in LNCS.

[34] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 134–145. Springer, 1998.

[35] F. Neven. *Design and Analysis of Query Languages for Structured Documents — A Formal and Logical Approach.* Doctor's thesis, Limburgs Universitair Centrum (LUC), 1999.

[36] F. Neven. Extensions of attribute grammars for structured document queries. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[37] F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems*, pages 11–17. ACM Press, 1998.

[38] C. Pair and A. Quere. Définition et etude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.

[39] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3. Springer, 1997.

[40] J. Shallit. Numeration systems, linear recurrences, and regular sets (extended abstract). In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92*, volume 623 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 1992.

[41] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, pages 198–200, 1959.

[42] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27(1):1–36, 1975.

[43] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[44] W. Thomas. Ehrenfeucht games, the composition method, and the monadic theory of ordinal words. volume 1261 of *Lecture Notes in Computer Science*, pages 118–143. Springer-Verlag, 1997.

[45] W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa [39], chapter 7.

[46] M. Y. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 83–92. ACM Press, 1989.