On the Power of Walking for Querying Tree-Structured Data

Frank Neven
University of Limburg
frank.neven@luc.ac.be

Abstract

XSLT is the prime example of an XML query language based on tree-walking. Indeed, stripped down, XSLT is just a tree-walking tree-transducer equipped with registers and look-ahead. Motivated by this connection, we want to pinpoint the computational power of devices based on tree-walking. We show that in the absence of unique identifiers even very powerful extensions of the tree-walking paradigm are not relationally complete. That is, these extensions do not capture all of first-order logic. In contrast, when unique identifiers are available, we show that various restrictions allow to capture LOGSPACE, PTIME, PSPACE, *and* EXPTIME. *These complexity classes are defined w.r.t. a Turing machine model working directly on (attributed) trees. When no attributes are present, relational storage does not add power; whether look-ahead adds power is related to the open question whether tree-walking captures the regular tree languages.*

1 Introduction

A main research topic in the theory of databases is characterizing the expressiveness and computational complexity of query languages. Although numerous XML transformation languages have been defined [1, 8, 9], there are almost no complete characterizations of their expressive power or computational complexity.

In this paper, we focus on the navigational part of XML languages. More precisely, we study computation by tree-walking. This is a well-known paradigm from formal language theory studied in the context of attribute grammars and tree-transformations [3, 5, 11]. As XML documents are, by and large, attributed trees, it is no surprise that this paradigm popped up in XML research. Indeed, a first instance of tree-walking is provided by the caterpillar expressions of Brüggemann-Klein and Wood [7]. Further, Milo, Suciu, and Vianu [17] defined a tree-walking tree-transducer model with pebbles as an abstract model for XML transformations. Finally, as argued by Bex, Maneth and Neven [4], stripped down, XSLT is essentially a tree-walking tree-transducer with registers and look-ahead. The present investigation is motivated by the latter model. In brief, registers deal with data values and look-ahead allows sub-computations. As we focus on the computational aspect of tree-walking, we restrict attention to boolean queries and do not consider the transducer part.

Neven, Schwentick, and Vianu [20] showed, essentially, that tree-walking automata with registers are not relationally complete. That is, they cannot compute all first-order logic (FO) definable properties. In the present paper, we push this result quite a bit further by showing that tree-walking is not relationally complete even in the presence of look-ahead and a relational storage (rather than with registers which can only contain a single data value). The latter weakness is due to the absence of unique IDs.

In strong contrast, if every node has a unique ID, we show that various restrictions capture natural complexity classes:

no look-ahead, single-valued registers	LOGSPACE
look-ahead, single-valued registers	PTIME
no-look ahead, relational storage	PSPACE
look-ahead, relational storage	EXPTIME

To obtain the latter results we introduce a Turing machine model (XTMs) operating directly on attributed trees. XTMs are adaptations of the domain Turing machines of Hull and Su [15]. XTMs provide a convenient tool for measuring the expressiveness of tree-walking devices as they work directly on trees. Further, there is a natural time/space-correspondence with ordinary Turing machines working on encodings of attributed trees.

When no attributes are present, it is easy to see that relational storage does not add additional power. In [4] it is shown that adding look-ahead allows to capture all unary monadic second-order logic formulas (MSO) and, therefore, all regular tree languages [18]. Hence, whether look-ahead adds power depends on the open question whether tree-walking captures all regular tree-languages [12, 13, 19].

This paper is further organized as follows. In Section 2, we provide background on attributed trees and logic. In Section 3, we define the most general class $\text{TW}^{r,l}$ of tree-walking programs. In Section 4 we prove the relational incompleteness of $\text{TW}^{r,l}$. In Section 5, we define several restrictions of $\text{TW}^{r,l}$. In Section 6, we define XTMs. In Section 7, we prove characterizations of restrictions of $\text{TW}^{r,l}$ in terms of XTMs. We conclude in Section 8.

2 Background

2.1 Attributed Trees and XML

We start with the necessary definitions regarding trees over a finite alphabet Σ. To use these trees as adequate abstractions of actual XML documents, we extend them with attributes that take values from an infinite (recursively enumerable) domain $\mathbf{D} = \{d_1, d_2, \ldots\}$.

The set of Σ-trees, denoted by $\mathcal{T}_\Sigma$, is inductively defined as follows: (i) every $\sigma \in \Sigma$ is a Σ-tree; (ii) if $\sigma \in \Sigma$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, $n \geq 1$ then $\sigma(t_1, \ldots, t_n)$ is a Σ-tree. Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked.*

For every tree $t \in \mathcal{T}_\Sigma$, the *set of nodes of* t, denoted by $\text{Dom}(t)$, is the subset of $\mathbb{N}^*$ defined as follows: if $t = \sigma(t_1 \cdots t_n)$ with $\sigma \in \Sigma$, $n \geq 0$, and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, then $\text{Dom}(t) = \{\varepsilon\} \cup \{ui \mid i \in \{1, \ldots, n\}, u \in \text{Dom}(t_i)\}$. Thus, ε represents the root while ui represents the i-th child of u. By $\text{lab}^t(u)$ we denote the label of u in t.

Next, we add XML attributes. To this end, for the rest of the paper, we fix a finite set of attributes A.

Definition 2.1 *An* attributed Σ-tree *is a pair* $(t, (\lambda_a^t)_{a \in A})$, *where* $t \in \mathcal{T}_\Sigma$ *and for each* $a \in A$, $\lambda_a^t : \mathit{Dom}(t) \mapsto \mathbf{D}$ *is a function defining the* a*-attribute of nodes in* t.

Of course, in real XML documents, usually, not all element types have the same set of attributes. Obviously, this is just a convenience and not a restriction. Further, XML documents can contain elements with mixed content. As shown in [4], this can be faithfully represented by attributed trees with dummy intermediate nodes.

In the following, when we say tree we always mean attributed Σ-tree.

2.2 Logic

As we will compare our tree-walking language with FO, we explain how a tree is represented as a logical structure. We make use of the vocabulary $\tau_{\Sigma,A} = \{E, <, \prec, (O_\sigma)_{\sigma \in \Sigma}, (\text{val}_a)_{a \in A}\}$. Each val_a is a function, all other vocabulary symbols are relations. The domain of the structure representing t is $\text{Dom}(t)$. The relations are then interpreted in the following way: $E(u, ui)$ for all $u, ui \in \text{Dom}(t)$ and $i \in \mathbb{N}$ (edge relation);[1] $ui < uj$ iff $i < j$ and $i, j \in \mathbb{N}$ (ordering on siblings); $u \prec uv$ for all $u, uv \in \text{Dom}(t)$ (descendant relation); $O_\sigma(u)$ iff $\text{lab}^t(u) = \sigma$; and $\text{val}_a(u) = \lambda_a(u)$. Note that val_a is a function from $\text{Dom}(t)$ to $\mathbf{D}$. The logic at hand is based on the logics accompanying the metafinite structures of Grädel and Gurevich [14].

[1]Recall that ui is a child of u.

An atomic formula is of the form $E(x, y)$, $x < y$, $x \prec y$, $O_\sigma(x)$, $x = y$, $\mathrm{val}_a(x) = \mathrm{val}_b(y)$ or $\mathrm{val}_a(x) = d$ where $a, b \in A$ and $d \in \mathbf{D}$. Such formulas have the obvious semantics. FO is obtained by closing the atomic formulas under the boolean connectives and first-order quantification over $\mathrm{Dom}(t)$. As an example consider the FO sentence $\forall x(\mathrm{val}_a(x) = d \vee \mathrm{val}_a(x) = \mathrm{val}_b(x))$, expressing that the value of every a-attribute is d or is equal to the b-attribute.

2.3 FO($\exists^*$) and XPath.

The pattern language employed by XSLT is XPath. For our inexpressibility result we abstract XPath by binary FO($\exists^*$) formulas over the above mentioned vocabulary. The latter logic is the fragment of first-order logic where all formulas are in prenex normal form and all quantifiers are existential. Additionally, formulas can make use of the unary predicates $\mathrm{root}(x)$, $\mathrm{leaf}(x)$, $\mathrm{first}(x)$, and $\mathrm{last}(x)$ (denoting that x is the root, a leaf, the first and the last child, respectively) and the binary predicate $\mathrm{succ}(x, y)$ (denoting that y is the right sibling of x, respectively). Note that these predicates are FO-definable but not FO($\exists^*$)-definable.

We consider the fragment of XPATH defined by the following grammar:

$$\begin{array}{rcll}
p & := & p_1 | p_1 & \text{(union)} \\
 & & | \; /p & \text{(root)} \\
 & & | \; p_1/p_2 & \text{(child)} \\
 & & | \; p_1//p_2 & \text{(descendant)} \\
 & & | \; p_1[p_2] & \text{(filter)} \\
 & & | \; \sigma & \text{(element test)} \\
 & & | \; * & \text{(wildcard)}
\end{array}$$

We refer the reader unfamiliar with XPath to [10].

Clearly, XPath defined as such can be simulated by FO($\exists^*$). As an example consider the XPath expression `a//b[//c][d]` which is equivalent to the FO($\exists^*$) formula

$$\varphi(x, y) := \exists y_2 \exists y_3 (x \prec y \wedge y \prec y_2 \wedge E(y, y_3) \wedge O_a(x) \wedge O_b(y) \wedge O_c(y_2) \wedge O_d(y_3)).$$

We always take x as the current position and y as the selected position. Note that FO($\exists^*$) can also compare attribute values of nodes.

3 Computation by tree-walking

Before we give the definition of tree-walking automata, let's recall two-way deterministic finite state machines on *strings*: such devices 'walk' in two directions over a string changing state and direction depending on the current state, the current symbol and whether the current position is the left or right-delimiter. An automaton accepts if a final state is reached at some point. Analogously, a *tree-walking* automaton is a finite state device walking a *tree*. Its control is always at one node of the input tree. Based on the label of that node, its state, and its position in the tree (first or last child, root, or leaf), the automaton changes state and moves to one of the neighboring nodes (parent, first child, left or right sibling). The automaton accepts the tree when it enters a final state.

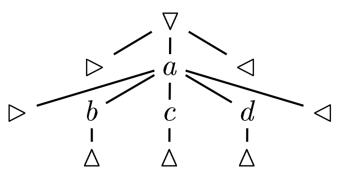
To deal with $\mathbf{D}$-values a tree-walking automaton is equipped with a finite number of registers.[2] In [4] these registers can only store $\mathbf{D}$-values coming from the attributes. In the present paper, we extend the power of tree-walking automata in two ways: (*i*) registers can store finite relations over $\mathbf{D}$; (*ii*) tree-walking automata can start subcomputations.

We explain this in a bit more detail. (*i*) We fix a set of relation names $\overline{X} := X_1, \ldots, X_k$ each of a specific arity. These will serve as the relational store and will be interpreted by finite relations over $\mathbf{D}$. We will use FO to define and manipulate the relational storage. However, the logic has only access to the attribute-values of the current node and the relational storage. Moreover, all quantification is over the active domain of the relational storage. So there is *no* access to the tree structure. (*ii*) A subcomputation computes a relation over $\mathbf{D}$. Such a subcomputation is not necessarily started at the present location: an XPath expression (or in our setting, an FO($\exists^*$) formula) is used to locate various

[2] In XSLT, the counterpart of registers are variables that can be passed between templates.

positions in the tree. From every selected position a subcomputation is started. When such a subcomputation accepts, it returns the content of its first register. The overall result is then the union of the relations computed by the subcomputations. When one subcomputation rejects, the whole computation rejects. To be concrete, a subcomputation is invoked by a construct of the form[3] $\text{atp}(\varphi(x,y),q)$. Suppose $\varphi(x,y) = x \prec y \wedge \text{leaf}(y)$. Then the former starts a subcomputation in state q at all leaves below x (the current node). As already mentioned, the result is then the union of the results of the various subcomputations.

To simplify the definition of two way automata on strings, one usually delimits strings with the start and end symbols $\rhd$ and $\lhd$, respectively. We do the same for trees using the extra symbols $\triangle$ and $\triangledown$. For instance, if t is the tree $a(bcd)$ then $\text{delim}(t)$ is the tree



We assume that every attribute of a delimiter contains $\perp$ where $\perp \notin \mathbf{D}$. We refrain from giving a formal definition of $\text{delim}(t)$.

In the next definition, the r and l in $\text{TW}^{r,l}$ stand for relational storage and look-ahead, respectively.

Definition 3.1 *A k-register $\text{TW}^{r,l}$-automaton $\mathcal{B}$ is a tuple (Q, q_0, q_F, τ_0, P) where Q is a finite set of states; $q_0 \in Q$ is the initial state; $q_F \in Q$ is the final state; $\tau_0 : \{1,\dots,k\} \to \mathbf{D} \cup \{\perp\}$ is the initial register assignment; and, P is a finite set of rules of the form $(\sigma, q, \xi) \to \alpha$. Here, $\sigma \in \Sigma$, $q \in Q$, and ξ is an FO sentence over the vocabulary $\overline{X} \cup \bigcup_{a \in A}\{a\} \cup \bigcup_{d \in \mathbf{D}}\{d\}$ where each a and d are constants. Intuitively, a transition can be applied when the current node carries a σ, the current state is q, and the contents of the relational store satisfies ξ. The righthand side α has one of the following forms:*

1. *(q', d) with $q' \in Q$ and $d \in \{\cdot, \leftarrow, \rightarrow, \uparrow, \downarrow\}$; intuitively, this means* change to state q' and move in direction d *($\cdot$ means 'stay');*
2. *(q', ψ, i) with $q' \in Q$, ψ an FO formula over the vocabulary $\overline{X} \cup \bigcup_{a \in A}\{a\} \cup \bigcup_{d \in \mathbf{D}}\{d\}$, and $i \in \{1,\dots,k\}$; intuitively, this means* change to state q' and replace register i with the relation obtained by applying ψ*;*
3. *$(q', atp(\varphi(x,y),p), i)$ with $q', p \in Q$, $\varphi(x,y)$ an FO($\exists^*$) formula, and $i \in \{1,\dots,k\}$; intuitively, this means* change to state q' and replace register i with the relation obtained by applying $\text{atp}(\varphi(x,y),p)$.

We assume that there is no transition possible from the final state. Further, we assume that the automaton never moves off the input tree. The size of $\mathcal{B}$ is $|Q| + \Sigma_{i=1}^{k}|\tau_0(i)| + \Sigma_{(\sigma,q,\xi)\to\alpha \in P}|\xi|$.

Let $\mathbf{D}_{\text{active}}$ be the set of $\mathbf{D}$-elements occurring in t and $\mathcal{B}$. Given a tree t, a *configuration of $\mathcal{B}$ on t* is a tuple $[u, q, \tau]$ where $u \in \text{Dom}(t)$, $q \in Q$, and $\tau : \{1,\dots,k\} \to \bigcup_{i \geq 1} \mathbf{D}^i_{\text{active}}$. That is, u is the current node, q the current state, and τ the register content. The *initial* configuration is $\gamma_0 := [\varepsilon, q_0, \tau_0]$. A configuration $[u, q_F, \tau]$ is *accepting.* A rule (σ, p, ξ) applies to a configuration $[u, q, \tau]$ iff $\text{lab}^t(u) = \sigma$, $p = q$ and ξ holds under the interpretation induced by τ where in addition each $a \in A$ is interpreted by $\text{val}^t_a(u)$ and each $d \in \mathbf{D}$ is interpreted by d. We assume that automata are deterministic: if (σ, q, ξ_1) and (σ, q, ξ_2) appear as left-hand sides then there is never a configuration such that both ξ_1 and ξ_2 apply.

Before we define the transition relation, we define the (partial) move function m_d for every $d \in \{\cdot, \leftarrow, \rightarrow, \uparrow, \downarrow\}$ as follows. For every node u, $m_{\cdot}(u)$, $m_{\leftarrow}(u)$, $m_{\rightarrow}(u)$, $m_{\uparrow}(u)$, and $m_{\downarrow}(u)$ equals u, the left sibling, the right sibling, the parent and the first child of u, respectively (if they exist).

Denote the set of all configurations by $C^{\mathcal{B},t}$. The transition graph $\vdash^* \subseteq C^{\mathcal{B},t} \times C^{\mathcal{B},t}$ is the smallest graph with nodes in $C^{\mathcal{B},t}$ such that $\vdash^*$ is transitive; further, there is an edge between $\gamma = [u, q, \tau]$ and $\gamma' = [u', q', \tau']$ if there is a rule $(\sigma, q, \beta, \xi) \to \alpha$ in P to which γ applies and

[3]atp stands for *apply templates.* The latter terminology is used by XSLT.

- if $\alpha = (p, d)$ then $q' = p$, $m_d(u) = u'$, and $\tau = \tau'$. Intuitively, this is a move in direction d.

- if $\alpha = (p, \psi, i)$ then $q' = p$, $u' = u$, $\tau'(j) = \tau(j)$ for all $j \neq i$, and $\tau'(i) = \{\bar{d} \mid \psi(\bar{d}) \text{ holds in } \tau\}$. Intuitively, register i is replaced by the result of ψ.

- if $\alpha = (p, \text{atp}(\varphi(x,y), p'), i)$ then $q' = p$, $u' = u$, $\tau'(j) = \tau(j)$ for all $j \neq i$, and $\tau'(i) = \bigcup_{j=1}^{\ell} \tau_j(1)$ where $(u_j, p', \tau) \vdash^* (v_j, q_F, \tau_j)$ for some node v_j and $\{u_1, \ldots, u_\ell\} = \{v \mid t \models \varphi(u,v)\} \subseteq \text{Dom}(t)$.

  Intuitively, register i is replaced by the result of $\text{atp}(\varphi(x,y), p')$; the latter, starts ℓ subcomputations at the nodes $\{u_1, \ldots, u_\ell\} = \{v \mid t \models \varphi(u,v)\} \subseteq \text{Dom}(t)$; these computations are started in state p' and with relational store τ; when they end in a final state, the contents of the first register is returned (that is, $\tau_j(1)$); the content of register i is then the union of the results of all subcomputations.

$\mathcal{B}$ accepts the input tree t if $\gamma_0 \vdash^* \gamma$ for some accepting configuration γ.

Example 3.2 Assume $\Sigma = \{\sigma, \delta\}$ and $A = \{a\}$. We define an automaton that accepts a tree if for every δ-labeled node all its leaf-descendants have the same a-attribute. By leaf-descendants we do not mean nodes labeled with $\vartriangle$ but the parents of those nodes. We define a 1-register automaton where the register X_1 is a set. Let $Q = \{q_0, q_1, q_2, q_3, q_4, q_F\}$, and $\tau_0(1) = \emptyset$. P consists of the following rules:

$$\begin{aligned}
(\triangledown, q_0, \text{true}) &\to (q_1, \text{atp}(\varphi_1, q_2), 1) && (1)\\
(\triangledown, q_1, \text{true}) &\to (q_F, \cdot) && (2)\\
(\delta, q_2, \text{true}) &\to (q_3, \text{atp}(\varphi_2, q_4), 1) && (3)\\
(\delta, q_3, \xi) &\to (q_F, \cdot) && (4)\\
(\delta, q_4, \text{true}) &\to (q_F, x = a, 1) && (5)\\
(\sigma, q_4, \text{true}) &\to (q_F, x = a, 1) && (6)
\end{aligned}$$

where $\varphi_1 \equiv x \prec y \wedge O_\delta(y)$, $\varphi_2 \equiv \exists y_1(x \prec y \wedge E(y, y_1) \wedge O_\vartriangle(y_1)$, and $\xi \equiv \forall x \forall y (X_1(x) \wedge X_1(y) \rightarrow x \neq y)$.

The automaton works as follows: (1) a subcomputation is initiated that selects all δ-labeled descendants of the root; (2) when all subcomputations return, that is, state q_1 is reached, the tree is accepted; (3) every δ-labeled node selects all leaves (recall that we work with delimited trees); (4) when the returned set is a singleton, the subcomputation accepts (otherwise, the subcomputation gets stuck and the main computation rejects); as (5) and (6) make sure that every leaf returns the value of its a attribute, the computations initiated by (3) accepts in (4) iff every leaf has the same a-attribute. Note that $x = a$ is the formula that defines the set containing the value of the a-attribute of the current node. □

In the next section we show that $\text{TW}^{r,l}$, as such, is actually very weak. That is, the language is not even relationally complete. In contrast, when unique IDs are present, various restrictions of $\text{TW}^{r,l}$ capture natural complexity classes. So, we show in Section 7 that under very reasonable assumptions $\text{TW}^{r,l}$ is in fact a very robust language.

4 Inexpressibility

The present section is devoted to the proof of the following theorem which shows that $\text{TW}^{r,l}$ is not relationally complete.

Theorem 4.1 *$\text{TW}^{r,l}$ can not simulate FO over the vocabulary $\tau_{\Sigma,A}$ when $\Sigma \neq \emptyset$ and $A \neq \emptyset$.*

For ease of exposition, we restrict attention to strings, that is, monadic trees. In particular, we take $\Sigma = \{\sigma\}$ and $A = \{a\}$. As we are proving an inexpressibility result, this is no loss of generality. For convenience, we write a tree $\sigma(\sigma(\sigma(\sigma)))$ with $\lambda_a^t(\varepsilon) = d_0$, $\lambda_a^t(1) = d_1$, $\lambda_a^t(2) = d_2$, and $\lambda_a^t(3) = d_3$ simply as the string $d_0 d_1 d_2 d_3$.

Before we state the string language not computable with a $\text{TW}^{r,l}$-program, we introduce some notation. A *1-hyperset over* $\mathbf{D}$ is a finite subset of $\mathbf{D}$. For $i > 1$, an *i-hyperset over* $\mathbf{D}$ is a finite set of $(i-1)$-hypersets over $\mathbf{D}$. We often denote i-hypersets with a superscript i, as in $S^{(i)}$.

For ease of presentation, we assume that $\mathbf{D}$ contains all natural numbers. For each $j > 0$, let $\mathbf{D}_j$ be $\mathbf{D} - \{1, \ldots, j\}$. Next, let $j > 0$ be fixed. We inductively define *encodings* of i-hypersets over $\mathbf{D}_j$. First, a string $w = 1d_1d_2 \cdots d_n1$ over $\mathbf{D}_j$ is an encoding of the 1-hyperset $H(w) = \{d_1, \ldots, d_n\}$ over $\mathbf{D}_j$. For each $i \leq j$, and encodings $w_1, \ldots, w_n$ of $(i-1)$-hypersets, $iw_1iw_2 \cdots iw_ni$ is an encoding of the i-hyperset $\{H(w_i) \mid i \leq n\}$. Define $L^m_=$ as the language $\{f\#g \mid f$ and g are encodings of m-hypersets over $\mathbf{D}_m - \{\#\}$ and $H(f) = H(g)\}$.

The following is shown in [20].

Lemma 4.2 *For each m, $L^m_=$ is definable in FO.*

We next show that no $\text{TW}^{r,l}$-program can compute $L^m_=$ for sufficiently large m. The proof is based on communication complexity. More precisely, (i) we show that every $\text{TW}^{r,l}$-program can be computed by a specific communication protocol (Lemma 4.5); (ii) we show that no protocol can compute $L^m_=$ for sufficiently large m (Lemma 4.6). Step (i) is the most involved one.

Define $\exp_0(n) := n$ and $\exp_i(n) := 2^{\exp_{i-1}(n)}$, for $i > 0$. We say that two strings s_1 and s_2 are k-equivalent, denoted $s_1 \equiv_k s_2$ if they satisfy the same FO($\exists^*$)-formulas with k variables. We write $(s; i_1, \ldots, i_n)$, $i_1, \ldots, i_n \in \text{Dom}(s)$, for the structure where each i_j is taken as a constant. By $\text{tp}_k(s; i_1, \ldots, i_n)$ we denote the $\equiv_k$-equivalence class to which $(s; i_1, \ldots, i_n)$ belongs. We also call $\text{tp}_k(s)$ the *k-type* of s.

Lemma 4.3 *Let D be a finite subset of $\mathbf{D}$. Let f and g be two strings over D.*

1. *$tp_k(f\#g; i, j)$ with $i < j$ only depends on $tp_k(f; i, j)$ and $tp_k(g)$ when i and j refer to positions in f; on $tp_k(f; i)$ and $tp_k(g; j)$ when i and j refer to positions in f and g, respectively; and, on $tp_k(f)$ and $tp_k(f; i, j)$ when i and j refer to positions in g;*

2. *the number of equivalence classes of $\equiv_k$ is not more than $exp_3(p(k+|D|))$ for some polynomial p.*

Proof. (sketch) The proof of (1) is a standard Ehrenfeucht-game argument. The proof of (2) is a counting argument. Indeed, the number of atomic formulas is bounded by $p(k + |D|)$ for some polynomial p. As every FO($\exists^*$) is equivalent to one with quantifier-free part in disjunctive normal form, the total number of formulas is bounded by $\exp_2(p(k + |D|))$. As a $\equiv_k$-class is determined by the set of formulas its members satisfy, the number of $\equiv_k$-classes is bounded by $\exp_3(p(k + |D|))$. $\square$

Definition 4.4 *Let N be a natural number. Let P be a binary predicate on i-hypersets over $\mathbf{D}$. We say that P can be* computed by an N-communication protocol *between two parties (denoted by* I *(male) and* II *(female)) if there is a polynomial p such that for all i-hypersets $X^{(i)}$ and $Y^{(i)}$ over a finite set D there is a finite alphabet Δ of size at most $\exp_3(p(N+|D|))$ such that $P(X^{(i)}, Y^{(i)})$ can be computed as follows:* I *gets $X^{(i)}$ and* II *gets $Y^{(i)}$; both know D and Δ; they send elements from Δ back and forth; and, after $2|\Delta|$ rounds of message exchanges, both* I *and* II *have enough information to decide whether $P(X^{(i)}, Y^{(i)})$ holds.*

We refer to strings of the form $f\#g$, where f and g do not contain $\#$, as split strings. *A communication protocol computes on such strings by giving f to* I *and g to* II. *Note that* I *and* II *have unlimited power on the strings assigned to them.*

Lemma 4.5 *On split strings, the strings recognized by $\text{TW}^{r,l}$-programs of size N can be computed by an N-protocol.*

Proof. (sketch) Let $\mathcal{B} = (Q, q_0, F, \tau_0, P)$ be a $\text{TW}^{r,l}$-program and $f\#g$ be an input string over alphabet D. Assume $\mathcal{B}$ never changes direction at the marker $\#$. Assume the position of $\#$ in $f\#g$ is b.

We take Δ as the set containing all tuples of the form

- $\langle \varphi, q, \theta, \tau \rangle$ (*atp-request*);
- $\langle R \rangle$ (*reply*);
- $\langle q, \tau \rangle$ (*send main configuration*)
- $\langle q, \tau, \text{NeedAnswer} \rangle$ (*send configuration of subcomputation*)

- $\langle\theta\rangle$ (*send N-type*);
- $\langle$reject$\rangle$ and $\langle$accept$\rangle$,

where φ is an FO($\exists^*$) formula occurring in $\mathcal{B}$, $q \in Q$, θ is an $\equiv_N$-equivalence class and each R_i and R are relations over D. By Lemma 4.3(2), Δ is of the required size.

Recall that `I` is given f while `II` is given g. The two parties simulate the behavior of $\mathcal{B}$. As an initialization step, `I` sends the N-type of f to `II` and `II` in turn answers with the N-type of g. Hereafter, `I` starts computation in the start configuration $[\varepsilon, q_0, \tau_0]$.

Both `I` and `II` make use of a stack during their computation (which we denote by $\text{Stack}^{\mathtt{I}}$ and $\text{Stack}^{\mathtt{II}}$, respectively). The stack alphabet of `I` (resp., `II`) consists of the symbol `ReturnAns` together with all symbols $\mathtt{Compute\&Return}(S, R; p, \tau)$ and $\mathtt{Compute}(S, R; u, p, \tau, i)$ where S is a set of positions in $f\#$ (resp., $\#g$), $p \in Q$, R is a relation over $\mathbf{D}$, $i \in \{1, \ldots, k\}$, τ is the contents of a relational store, and u is a position in $f\#$ (resp., $\#g$). When `ReturnAns` is on top of the stack, this means that when the computation reaches the final state, the content of the first register should be sent to the other party. The symbols $\mathtt{Compute\&Return}(S, R; p, \tau)$ and $\mathtt{Compute}(S, R; u, p, \tau, i)$ are used to compute atp-request from the other party and the present party, respectively. More precisely, S contains the position which still have to be processed; R is the intermediate result; the other information is used to start new subcomputations.

We only explain the behavior of `I`, `II`'s behavior is analogous. Assume that at some point the computation in f is in configuration $\gamma = [u, q, \tau]$. That is, `I` is simulating the computation and `II` is waiting. Suppose first that $q = q_F$.

1. Assume $\text{Stack}^{\mathtt{I}}$ is empty. This means that the current computation is the main one. Hence, the string is accepted. Therefore, `I` sends $\langle$accept$\rangle$ to `II` and both parties know the string is accepted.

2. Assume $\text{top}(\text{Stack}^{\mathtt{I}}) = \mathtt{ReturnAns}$. This means that `II` has sent a configuration and expects an answer. Therefore, `I` pops the stack and sends $\langle\tau(1)\rangle$ to `II`.

3. Assume $\text{top}(\text{Stack}^{\mathtt{I}}) = \mathtt{Compute\&Return}(S, R; p, \tau')$. This means that the current computation is computing the `I`-part of an atp-request of `II`. If $S = \emptyset$ then the request is completed. So, `I` pops the stack, sends $\langle R \cup \tau(1)\rangle$ to `II`, and waits for an answer. Otherwise, `I` takes an element v from S, pops the stack, pushes $(S - \{v\}, R \cup \tau(1); p, \tau')$ on the stack and continues in configuration $[v, p, \tau']$.

4. Finally, assume $\text{top}(\text{Stack}^{\mathtt{I}}) = \mathtt{Compute}(S, R; u, p, \tau', i)$. This means that the current computation is computing the `I`-part of an atp issued by `I` himself. If $S = \emptyset$ then the computation is finished. So, `I` pops the stack and continues in configuration $[u, p, \tau'']$ where $\tau''(j) = \tau'(j)$ for $j \neq i$ and $\tau''(i) = R \cup \tau(1)$. Otherwise, if S is not empty, `I` takes an element v from S, pops the stack, pushes $(S - \{v\}, R \cup \tau(1); p, \tau')$ on the stack and continues in configuration $[v, p, \tau']$.

Suppose $q \neq q_F$. If no rule applies. Then `I` sends $\langle$reject$\rangle$ to `II`. Both parties then know that the automaton rejects. Conversely, let $(\sigma, q, \xi) \to \alpha$ be the rule that applies to $[u, q, \tau]$.

1. If $\alpha = (p, d)$ then `I` computes the new configuration $[p, m_d(u), \tau]$ as $\mathcal{B}$ would do. If u is a position in f then `I` continues as before. Otherwise, the computation leaves f. If $\text{Stack}^{\mathtt{I}}$ is empty, then the main computation leaves f and `I` sends $\langle p, \tau\rangle$ to `II` and waits for an answer. If $\text{top}(\text{Stack}^{\mathtt{I}})$ contains `ReturnAns`, this means that `I` should send a relation to `II`. However, the computation hasn't finished yet, but returns to `II`. She will now take care of this subcomputation. Therefore, `I` pops the stack, sends $\langle p, \tau\rangle$ to `II` and waits for an answer. Otherwise, if the stack contains a `Compute` or a `Compute&Return`, `I` needs the result of the subcomputation. Therefore, he `I` sends $\langle p, \tau, \text{NeedAnswer}\rangle$ to `II`.

2. If $\alpha = (p, \psi, i)$ then I computes the new configuration $[p, u, \tau']$ as $\mathcal{B}$ would do, and continues as before.

3. If $\alpha = (p, \text{atp}(\varphi, p'), i)$ then I pushes $\texttt{Compute}(S, \emptyset; u, q, \tau, i)$ on the stack where $S = \{v \mid t \models \varphi(u, v)\}$. Note that by Lemma 4.3(2), the latter only depends on $\text{tp}_N(f; u)$ and $\text{tp}_N(g)$. Next, he sends $\langle \varphi, p', \text{tp}_N(f; u), \tau \rangle$ to II and waits for an answer.

Suppose the computation resides in g and I receives a message β.

1. If $\beta = \langle \text{accept} \rangle$ ($\langle \text{reject} \rangle$) then I knows the string is accepted (rejected).

2. If $\beta = \langle \varphi, q, \theta, \tau \rangle$ then I pushes $\texttt{Compute\&Return}(S - \{v'\}, \emptyset; q, \tau)$ on the stack where v' is an arbitrary element in S. Here, S is is the set of position in $f\#$ selected by φ. By Lemma 4.3(1), the latter only depends on f and θ. Next, I starts computation in configuration $[v', q, \tau]$.

3. If $\beta = \langle R' \rangle$ and $\text{top}(\text{Stack}^I) = \texttt{Compute}(S, R; u, p, \tau, i)$ then I pops the stack, takes a $v \in S$, pushes $\texttt{compute}(S - \{v\}, R \cup R'; u, p, \tau, i)$ on the stack, and starts computation in configuration $[v, p, \tau]$. If $\text{top}(\text{Stack}^I)$ contains a $\texttt{Compute\&Return}$, I acts similarly. The top of the stack can not contain $\texttt{ReturnAns}$.

4. If $\beta = \langle q, \tau \rangle$, then I starts computation in configuration $[b, q, \tau]$.

5. If $\beta = \langle q, \tau, \text{NeedAnswer} \rangle$, then I pushes $\texttt{ReturnAns}$ on the stack and continues computation in configuration $[b, q, \tau]$.

It remains to argue that we need at most $2|\Delta|$ rounds of message exchanges. Thereto, we slightly alter the protocol. Suppose a party (say I) needs to send a request (an atp-request or a configuration of a subcomputation), then (i) if I has already sent this request and obtained an answer, then the request isn't sent but the already obtained value is used; (ii) if I has already sent the request but no answer is obtained yet, then the computation got stuck in a cycle and I sends $\langle \text{reject} \rangle$ to II; (iii) if no such request has been made, I simply sends it to II. Hence, each request will only be sent at most once. By the definition of the protocol, each request will be answered once. Further, each configuration can only be sent at most once in every direction. Finally, an N-type will only be sent once. This means that there are at most $2|\Delta|$ rounds of message exchanges. □

Lemma 4.6 *For $m > 6$, $L^m_=$ cannot be computed by an N-communication protocol for any N.*

Proof. Suppose there is a protocol computing $L^m_=$. For any given input $f\#g$, the number of different possible messages is $|\Delta| = \exp_3(p(|D| + N))$ with D the symbols in $f\#g$. Call a complete sequence of exchanged messages $a_1 b_1 a_2 b_2 \ldots$ a dialogue. Every dialogue has at most $2 \exp_3(p(|D| + N))$ rounds. Suppose for ease of exposition that it has exactly that many rounds. Hence, there is less than $\exp_3((p(|D| + N) + 1) \cdot \exp_3(p(|D| + N) + 1))$ different dialogues. However, the number of different m-hypersets over D is $\exp_m(|D|)$. Hence, for $m > 6$ and D large enough, there are m-hypersets $f \neq g$ such that the protocol gives the same dialogue for $f\#f$ and $g\#g$, and therefore also on $f\#g$ and $f\#g$. This leads to the desired contradiction. □

5 Restrictions

We consider several restrictions of $\text{TW}^{r,l}$: TW^r has no look-ahead (only relational storage), TW^l has registers only containing single $\mathbf{D}$-values; TW is TW^l without look-ahead. Formally, we have the following definition.

Definition 5.1

- TW^r *is* $\text{TW}^{r,l}$ *where no transitions of the form* $\beta \to (q, atp(\varphi, p), i)$ *are allowed.*

- TW^l *is the restriction of* $\text{TW}^{r,l}$ *where all relations are unary and contain at most* one *data value during every execution . It is also possible to give a syntactic definition. Indeed, every* *formula ψ in a rule $\beta \to (q, \psi, i)$ is quantifier-free and defines only one value. Further, every φ in a rule $\beta \to (q, atp(\varphi, p), i)$ should select only one node (for instance, select parent or first child). So, the look-ahead will compute one data-value rather than a set of values or a relation.*

- TW *is* TW^l *where no transitions of the form $\beta \to (q, atp(\varphi, p), i)$ are allowed.*

6 XML Turing Machines

As a technical vehicle we introduce XML Turing machines (XTM). These are inspired by the domain Turing machines of Hull and Su [15]. The input to such machines is an attributed tree. A machine is equipped with a finite number of registers and a one-way infinite work-tape.

Definition 6.1 *An* XTM *is a* TW *with a one-way infinite work-tape. The alphabet of the work tape is finite. The size of the input is the number of nodes in the tree.*[4] PTIME^X *(*EXPTIME^X*) is the class of* XTM*s that make at most a polynomial (exponential) number of transitions in the size of the input.* LOGSPACE^X *and* PSPACE^X *is the class of* XTM*s that use at most an amount of space on the worktape that is logarithmic/polynomial in the size of the input. Alternating complexity classes, denoted by an* A *in front of their name, are defined w.r.t. alternating* TW*s.*

We omit the proof of the following theorem.

Theorem 6.2 *Every tree language that is recognizable in* LOGSPACE^X, ALOGSPACE^X PTIME^X, PSPACE^X, APSPACE^X *is recognizable by an ordinary TM working on the encoding of trees in* LOGSPACE, ALOGSPACE, PTIME, PSPACE, APSPACE *and vice versa.*

[4]It does not matter for the results whether we count the sizes of the **D**-values or not.

7 Unique IDs

In this section we assume the existence of an attribute ID whose value is unique among the attribute values of the nodes in the tree. That is, for every tree t and for all nodes $u, v \in \text{Dom}(t)$, if $\lambda^t_{\text{ID}}(u) = \lambda^t_{\text{ID}}(v)$ then $u = v$.

This attribute will only be used for navigational purposes to the benefit of $\text{TW}^{r,l}$s. Storing these values in registers can be seen as placing pebbles on the corresponding nodes. We use this analogy in the proofs. When an XTM has at least logspace workspace at its disposal it does not matter whether the ID-attribute can be used. Indeed, the XTM can assign on the fly to each node its number in the in-order of the tree and recompute it when necessary. Therefore, we assume w.l.o.g. that XTMs have access to IDs.

Due to space limitation we only briefly mention the techniques used in the proofs below.

Theorem 7.1 *1.* TW *captures* LOGSPACE^X*;*

2. TW^l *captures* PTIME^X*;*

3. TW^r *captures* PSPACE^X*; and,*

4. $\text{TW}^{r,l}$ *captures* EXPTIME^X.

Proof. (sketch) (1) Clearly, every TW can be simulated in LOGSPACE^X. The reverse direction is similar to the proof that multi-head automata capture LOGSPACE. [22] For ease of exposition let M be a LOGSPACE^X XTM such that on every tree the computation needs at most $\log_2(|t|)$ space (rather than the more general $k \cdot \log_2(|t|)$ space). Assume further that the tape can only contain the symbols 0 and 1. As we have unique IDs, we can place a finite number of pebbles on positions of the input tree by storing their IDs in designated registers. The tape content can be represented by a number between 0 and $|t| - 1$. We consider the nodes in in-order. So, if the tape contains j then a pebble is placed on the $(j + 1)$th node in the in-order of the tree (the root represents zero). We refer to this pebble as the *tape pebble.* As the tape initially contains 0, the tape pebble is placed on the root. We also need to remember the position of the head. This is done by a second pebble in an analogous fashion. We refer to the latter pebble as the *head pebble*. If the head is placed on the ith cell from the right, the head pebble is placed on the ith node. We only need to be able to do two things: (a) check whether the symbol under the head is 0 or 1; and, (b) overwrite the symbol under the head.

(a) We simply have to check whether j divided by 2^{i-1} is even. Node $j/2$ can be found by placing a pebble on the root and one on j and letting them walk towards each other. The latter can be repeated $i-1$ times. It remains to check whether the resulting node is even or not. This can be done by walking towards the root and counting modulo two.

(b) If position i has to be changed from 0 to 1 we add 2^i to j otherwise we substract. Obtaining the number 2^i, addition and substraction is done in the obvious way.

(2) We first sketch that every TW^l is in PTIME^X. Indeed, there are only polynomially many configurations in the size of the input tree. One can first construct the configuration graph in a bottom-up manner (this is an inflationary process as only edges will be added) and then check whether a final configuration can be reached from the initial one. Of course, we cannot directly write **D**-values onto the tape, but we can assign a unique number to each **D**-value by considering the first occurrence in the in-order of the tree in which it appears.

For the converse direction, we first note that Theorem 6.2 and the known fact that ALOGSPACE equals PTIME implies that $\text{PTIME}^X = \text{ALOGSPACE}^X$. One can easily adapt the simulation in (1) to alternating TW^ls with logspace worktape. Indeed, when a universal state is entered the TW^l uses a subcomputation for each branch. Every branch returns a value indicating whether that branch accepts or not.

(3) Every TW^r is in PSPACE^X. Indeed, when no look-ahead is present, the configuration graph is a chain. As every configuration is of size polynomial in the size of the input tree, we can only keep the last configuration on the work tape.

Every XTM in PSPACE^X can be simulated by a TW^r by encoding the tape into a relation in the standard way (see, e.g, [16, 23]) and then using FO to compute the new configuration from the current one.

(4) The proof is analogous to (2) taking into account that there are an exponential number of configurations and that APSPACE equals EXPTIME. $\square$

For the sake of completeness, we examine the situation when $A = \emptyset$. Obviously, when $A = \emptyset$ there are no IDs.

Proposition 7.2 *When* $A = \emptyset$, $\text{TW}^{r,l} = \text{TW}^l =$ *MSO and* $\text{TW}^r = \text{TW}$.

Proof. Clearly, when $A = \emptyset$ there are only a finite number of register contents. These contents can therefore be kept in the state. Hence, $\text{TW}^{r,l} = \text{TW}^l$ and $\text{TW}^r = \text{TW}$. In [4], it is shown that TW^l can simulate MSO. Further, when $A = \emptyset$, TW^l can be rather directly simulated in MSO. $\square$

It is an open problem whether TW can simulate MSO [12, 13, 19]. We point out that TW^l is not included in MSO when $A \neq \emptyset$ [20].

8 Discussion and related work

We presented a formal analysis of the tree-walking paradigm. The main motivation is the XML transformation language XSLT. We showed that in the absence of unique IDs even very powerful enhancements of this paradigm fail to capture all of first-order logic. This result is a strengthening of a result by Neven, Schwentick, and Vianu [20] to look-ahead and relational storage. The present proof is again based on communication complexity, but is more subtle than the latter one. Indeed, although the proof in [20] takes alternation into account, the protocol is memory-less and keeps track of the complete configuration tree as a whole. Here, the protocol needs to follow a strict evaluation order where new communications depend on previous ones.

Communication complexity was first applied in databases by Abiteboul, Herr, and Van den Bussche [2]. The only other applications we are aware of are [6] and [20]. The used protocols are mostly similar. The main difficulty is actually the simulation lemma. It would be desirable to come up with general criteria when communication complexity can be applied.

In contrast to the negative result, when IDs are present in XML documents, we exhibited natural fragments that capture precisely LOGSPACE^X, PTIME^X, PSPACE^X and EXPTIME^X. Although the proofs of these results are combinations of known techniques in complexity, finite model theory, and formal languages, they provide a quite complete picture of the expressiveness of query languages based on tree-walking. The most surprising might be that TW^r which is the abstraction of XSLT defined in [4] captures in fact precisely PTIME. Of course, one immediate drawback of the current approach is that the formalisms under consideration do not generate output. This is the subject of further research.

Finally, we recall the relational attribute grammars (RAGs) studied by Neven and Van den Bussche [21] which also have a relational storage. As the latter capture linear parallel time they are included in TW^{r,l}, TW^l, and TW^r, and this inclusion is probably strict.

Acknowledgment

I thank Thomas Schwentick and Stijn Vansummeren for comments on a previous version of this paper. A suggestion of Thomas leading to a great simplification of the TW^{r,l} formalism is gratefully acknowledged.

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, L. Herr, and J. Van den Bussche. Temporal connectives versus explicit timestamps to query temporal databases. *Journal of Computer and System Sciences*, 58(1):54–68, 1999.

[3] A. V. Aho and J. D. Ullman. Translations on a context-free grammar. *Information and Control*, 19(5):439-475, 1971.

[4] G. J. Bex, S. Maneth and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1): 21-39, 2002.

[5] R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *Journal of Computer and System Sciences*, 61(1):1–50, 2000.

[6] N. Bidoit and S. De Amo. Implicit temporal query languages: towards completeness. C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 245–257. Springer, 1999.

[7] A. Brüggemann-Klein and Derick Wood. Caterpillars: a context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[8] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. `http://www.w3.org/TR/xquery/`

[9] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`

[10] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0 W3C Recommendation 16 November 1999 `http://www.w3.org/TR/xpath`

[11] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definition, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science.* Springer, 1988.

[12] J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In J. Karhum ki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.

[13] J. Engelfriet, H.J. Hoogeboom, and J.-P. van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[14] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140(1):26–81, 1998.

[15] R. Hull and J. Su. Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, 47(1):121–156, 1993

[16] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, 1986.

[17] T.Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. To appear in *Journal of Computer and System Sciences.* Extended abstract in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS 2000)*, pp. 11-22, 2000.

[18] F. Neven and T. Schwentick. Query automata. To appear in *Theoretical Computer Science.* Extended abstract in *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.

[19] F. Neven and T. Schwentick. On the power of tree-walking automata. To appear in *Information and Computation.* Extended abstract in *27th International Colloquium on Automata, Languages and Programming (ICALP 2000)*, pages 547–560, *Lecture Notes in Computer Science*, volume 1853. Springer, 2000.

[20] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. J. Sgall, A. Pultr, P. Kolman, editors, *Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 2001.

[21] F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. To appear in *Journal of the ACM.* Extended abstract in *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems (PODS 1998)*, pages 11–17. ACM Press, 1998.

[22] I. H. Sudborough. On tape-bounded complexity classes and multihead finite automata. *Journal of Computer and System Sciences*, 10(1):62–76, 1975.

[23] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 1982*, pages 137-146. ACM Press, 1982.