Efficient Memory Representation of XML Documents

Giorgio Busatto[1], Markus Lohrey[2], and Sebastian Maneth[3]

[1] Department für Informatik, Universität Oldenburg, Germany
giorgio.busatto@informatik.uni-oldenburg.de
[2] FMI, Universität Stuttgart, Germany lohrey@informatik.uni-stuttgart.de
[3] Faculté I & C, EPFL, Switzerland sebastian.maneth@epfl.ch

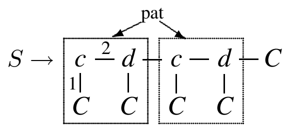
Abstract. Implementations that load XML documents and give access to them via, e.g., the DOM, suffer from huge memory demands: the space needed to load an XML document is usually many times larger than the size of the document. A considerable amount of memory is needed to store the tree structure of the XML document. Here a technique is presented that allows to represent the tree structure of an XML document in an efficient way. The representation exploits the high regularity in XML documents by "compressing" their tree structure; the latter means to detect and remove repetitions of tree patterns. The functionality of basic tree operations, like traversal along edges, is preserved in the compressed representation. This allows to directly execute queries (and in particular, bulk operations) without prior decompression. For certain tasks like validation against an XML type or checking equality of documents, the representation allows for provably more efficient algorithms than those running on conventional representations.

1 Introduction

There are many scenarios in which trees are processed by computer programs. Often it is useful to keep a representation of the tree in main memory in order to retain fast access. If the trees to be stored are very large, then it is important to use a memory efficient representation. A recent, most prominent example of large trees are XML documents which are sequential representations of ordered (unranked) trees, and an example application which requires to materialize (part of) the document in main memory is the evaluation of XML queries. The latter is typically done using one of the existing XML data models, e.g., the DOM. Benchmarks show that a DOM representation in main memory is 4–5 times larger than the original XML file. This can be understood as follows: a node of the form `<a/>` needs 4 bytes in XML; but as a tree node it needs at least 16 bytes: a name pointer, plus three node pointers to the parent, the first child, and the next sibling (see, e.g., Chapter 8 of [20]). There are some improvements leading to more compact representations, e.g., Galax [9] uses only 3–4 times more main memory than the size of the file. Another, more memory efficient data model for XML is that of a binary tree. As shown in [21], the known XML query languages can be readily evaluated on the binary tree model.

In this paper, we concentrate on the problem of representing binary trees in a space efficient way, so that the functionality of the basic tree operations (such as the traversal along edges) are preserved. Instead of compression, this is often called "data optimization" [14]. Our technique is a generalization of the well-known sharing of common subtrees. The latter means to determine during a bottom-up phase, using a hash table, whether the current subtree has occurred already, and if so to represent it by a pointer to its previous occurrence. In this way the minimal unique DAG (directed acyclic graph) of the tree is obtained in amortized linear time. For common XML documents the minimal DAG is about 1/10 of the size of the original tree [3]. Our representation is based on sharing of common subgraphs of a tree. The resulting sizes are 1/2–1/3 of the size of the minimal DAG. To our knowledge, this is the most efficient pointer-based tree representation that is currently available. At the same time, the complexity of querying, e.g. using XQuery, stays the same as for DAGs [18]. We therefore believe that our representation is better suited for in-memory storage of XML documents, than DAG-based representations.

Of course, an XML document consists of more things than just tree nodes: a node may have attributes, and a leaf may have character data. Both type of values we keep in string buffers. When traversing the XML tree, we keep information on how many nodes before (in document order) the current one (i) have attributes and (ii) how many have character data. These numbers determine for a node the correct indices into the attribute and data value buffers, respectively. With this in mind, it is straightforward to implement a DOM proxy for our representations. Note that attribute and character values can be stored more space efficiently using standard techniques [1]. The XML file compression tool XMill [17] separates data values into containers and compresses them individually using standard methods. The result is stored together with the tree structure. It is likely that compressing the tree structure by the technique presented here will further improve XMill's compression ratios.



$C \rightarrow c(A, A)$
$A \rightarrow a$

$$
\begin{aligned}
S &\rightarrow B(B(C)) \\
B(y_1) &\rightarrow c(C, d(C, y_1)) \\
C &\rightarrow c(A, A) \\
A &\rightarrow a
\end{aligned}
$$

Fig. 1. Regular and cf tree grammars generating $\{t\}$.

We now describe our representation in more detail. Consider the tree $c(c(a, a), c(a, a))$, or, in XML `<c><c><a/><a/></c><c><a/><a/></c></c>`. It consists of seven nodes and six edges. The minimal DAG for this tree has three nodes u, v, w and four edges ('first-child' and 'second-child' edges from u to v and from v to w). The minimal DAG can also be seen as the minimal regular tree grammar that generates the tree [19]: the shared nodes correspond to nonterminals of the grammar. For example, the above DAG is generated by the regular tree grammar with productions $S \rightarrow c(V, V)$, $V \rightarrow c(W, W)$, and $W \rightarrow a$. A generalization of sharing of subtrees is the sharing of arbitrary patterns, i.e., connected subgraphs of a tree. In a graph model it leads to the well-known notion of sharing graphs which are graphs with special "begin-sharing" and "end-sharing" edges, called fan-ins and fan-outs [15]. Since fan-in/out pairs can be nested, this structure allows to represent a tree in double-exponentially smaller size.

In contrast, a DAG is at most exponentially smaller than the tree it represents. A sharing graph can be seen as a context-free (cf) tree grammar [19]. In a cf tree grammar nonterminals can appear inside of an intermediate tree (as opposed to at the leaves in the regular case); formal parameters $y_1, y_2, \dots$ are used in productions in order to indicate where to glue the subtrees of the nonterminal which is being replaced. Finding the smallest sharing graph for a given tree is equivalent to finding the smallest cf tree grammar that generates the tree. Unfortunately, the latter problem is NP-hard: already finding the smallest cf (string) grammar for a given string is NP-complete [16]. The first main result of this paper is a linear time algorithm that finds a small cf tree grammar for a given tree. On common XML documents the algorithm performs well, obtaining grammars that are 1.5-2 times smaller than the minimal DAGs. As an example, consider the tree $t = c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))$ which has 18 edges. The minimal DAG, written as tree grammar, can be seen on the left of Fig. 1. It is the initial input to our algorithm "BPLEX" which tries to transform the grammar into a smaller cf tree grammar. It does so by going bottom-up through the right-hand sides of productions, looking for multiple (non-overlapping) occurrences of patterns. In our example, the tree pattern pat (consisting of two nodes labeled c and d and their left children labeled C) appears twice in the right-hand side of the first production. A pattern p in a tree can conveniently be represented by a tree t_p with formal parameters $y_1, \dots, y_r$ at leaves: simply add to t_p all children of nodes of p (and the edges), and label the jth such node (in preorder) by y_j. Thus, $t_{\text{pat}} = c(C, d(C, y_1))$. This tree becomes the right-hand side of a new nonterminal B and the right-hand side of the first production becomes $B(B(C))$. The resulting minimal cf tree grammar is shown on the right of Fig. 1.

The BPLEX algorithm is presented in Section 3. In Section 4 we discuss the application of BPLEX to XML documents and present experimental results. In Section 6 we study two problems for our tree grammars G that are important for XML documents: (1) to validate against an XML type and (2) to test equivalence. In fact, we consider both of these problems for so called "straight-line" (for short, SL) context-free tree grammars, which are grammars that are guaranteed to generate at most one tree; the "straight-line" notion is well-known from string grammars (see, e.g., [25, 26]). Since BPLEX generates "SLT grammars" of a more restricted form (additionally: linear in the parameters) we also consider problems (1) and (2) for this restricted case. It is shown that for an XML type T, represented by a (deterministic) bottom-up tree automaton, we can test whether or not the tree represented by G has type T in time $O(a^m \times |G|)$, where m is the maximal number of parameters of the nonterminals of G and a is the size of the automaton. Running a tree automaton is similar to evaluating a query; in [3] it was shown that a 'Core XPath query' Q can be evaluated on an XML document represented by its minimal DAG D in time $O(2^{|Q|} \times |D|)$. Next it is proved that testing the equivalence of two SL cf tree grammars can be done in polynomial space w.r.t. the sum of sizes of the two grammars, and, if the grammars are linear ("SLT") then even in polynomial time w.r.t. their sizes.

2 Preliminaries

For $k \in \mathbb{N}$, the set $\{1, \ldots, k\}$ is denoted by $[k]$. A finite set Σ together with a mapping $\mathrm{rank} : \Sigma \to \mathbb{N}$ is called a *ranked alphabet*. The set of all (ordered, rooted, ranked) *trees* over Σ is denoted by T_Σ. For a set A, $T_\Sigma(A)$ is the set of all trees over $\Sigma \cup A$, where all elements of A have rank 0. We fix a set of parameters $Y = \{y_1, y_2, \ldots\}$ and, for $k \geq 0$, $Y_k = \{y_1, \ldots, y_k\}$. For a ranked tree t, $V(t)$ denotes its set of nodes and $E(t)$ its set of edges. Each node in $V(t)$ can be represented by a sequence u of integers describing the path from the root of t to the desired node (Dewey notation), the node itself is denoted by t_u; for example, $1.1.1$ denotes the left-most leaf of the tree t from the Introduction (labeled a). The label at node u is denoted $t[u]$ and the subtree rooted at u is denoted t/u. For symbols $a_1, \ldots, a_n$ of rank zero and trees $t_1, \ldots, t_n$, $[a_1 \leftarrow t_1, \ldots, a_n \leftarrow t_n]$ denotes the substitution of replacing each leaf labeled a_i by the tree t_i, $1 \leq i \leq n$.

Context-free (cf) tree grammars are a natural generalization of cf grammars to trees (see, e.g., Section 15 in [13]). A cf tree grammar G consists of ranked alphabets N and Σ of nonterminal and terminal symbols, respectively, of a start symbol (of rank zero), and of a finite set of productions of the form $A(y_1, \ldots, y_k) \to t$. The right-hand side t of a production of the nonterminal A is a tree over nonterminal and terminal symbols and over the parameters in Y_k which may appear at leaves, where k is the rank of A, i.e., $t \in T_{N\cup\Sigma}(Y_k)$. Sentential forms are trees s, s' in $T_{N\cup\Sigma}$ and $s \Rightarrow_G s'$ if s' is obtained from s by replacing a subtree $A(s_1, \ldots, s_k)$ by the tree $t[y_1 \leftarrow s_1, \ldots, y_k \leftarrow s_k]$ where t is the right-hand side of an A-production. Thus, the parameters are used to indicate where to glue the subtrees of a nonterminal, when applying a production to it. The language generated by G is $\{s \in T_\Sigma \mid S \Rightarrow_G^* s\}$. Note that a parameter can cause copying (if it appears more than once in a rhs) or deletion (if it does not appear). For example, the cf tree grammar with productions $S \to A(a)$, $A(y_1) \to A(c(y_1, y_1))$, $A(y_1) \to y_1$ generates the language of all full binary trees over the binary symbol c and the constant symbol a.

A cf tree grammar is *regular* if all nonterminals have rank 0. It is *straight-line* (for short, SL) if each nonterminal A has exactly one production (with right-hand side denoted $\mathrm{rhs}(A)$) and the nonterminals can be ordered as $A_1, \ldots, A_n$ in such a way that $\mathrm{rhs}(A_i)$ has no occurrences of A_j for $j \leq i$ (such an order is called "SL order"). Thus, an SL cf tree grammar can be defined by a tuple $(N, \Sigma, \mathrm{rhs})$ where N is ordered and rhs is a mapping from N to right-hand sides. A cf tree grammar is *linear* if for every production $A(y_1, \ldots, y_k) \to t$, each parameter y_i occurs at most once in t.

In the sequel we use *SLT grammar* to stand for "SL linear cf tree grammar".

3 The BPLEX Algorithm

Grammar-based tree compression means to find a small grammar that generates a given tree. The size of such a grammar can be considerably smaller than the size of the tree, depending on the grammar formalism chosen. For example, finding the smallest regular tree grammar that generates a given tree t can be done in (amortized) linear time, and the resulting grammar is isomorphic to the minimal DAG of the tree. The minimal regular tree grammar G_t is also straight-line (any grammar that generates exactly one element can be turned into an SL grammar). The initial input to our compression algorithm BPLEX is the grammar G_t: BPLEX takes an arbitrary SL regular tree grammar as input and outputs a (smaller) SLT grammar. As mentioned in the Introduction, moving from regular to cf tree grammars corresponds to generalizing the sharing of common subtrees to the sharing of arbitrary tree patterns (connected subgraphs of a tree).

The basic idea of the algorithm is to find tree patterns that appear more than once in the input grammar (in a non-overlapping way), and to replace them by new nonterminals that generate the corresponding patterns. We call this technique *multiplexing* because multiple occurrences of the replaced patterns are represented only once in the output. The order in which the algorithm scans the nodes in the right-hand sides of the input grammar corresponds to scanning the generated tree bottom up; for this reason, the algorithm is called BPLEX (for *bottom-up multiplexing*).

BPLEX (see Fig. 2) takes as input an SL regular tree grammar G and three parame-

procedure BPLEX(G: grammar, K_N: int, K_S: int, K_R: int): grammar
begin
    $A_l :=$ last symbol in the SL ordering of G
    $z :=$ leftmost leaf of $\mathrm{rhs}_G(A_l)$
    while true do
        $\mathrm{repM} := \mathrm{RepM}(G, z, K_N)$
        $\mathrm{newM} := \mathrm{NewM}(G, z, K_N, K_S, K_R)$
        if $\mathrm{newM} \neq \emptyset$ **or** $\mathrm{repM} \neq \emptyset$ **then**
            $m := \max(\mathrm{newM}, \mathrm{repM})$
            if $m \in \mathrm{repM}$ **then**
                $G := G[m \leftarrow A]$, with $\mathrm{rhs}_G(A) = p_m$
            else
                $k := \mathrm{rank}(p_m)$
                $A := \mathrm{fresh}(G, k)$
                $G := \mathrm{add}(G, A(y_1, \ldots, y_k) \to p_m)$
                $G := G[m, c_m \leftarrow A]$
            fi
        elseif $\exists w \in V_G^l : z <_G^l w$ **then** $z := \mathrm{next}(<_G^l, z)$
        else break
        fi
    od
    return G
end BPLEX

Fig. 2. The BPLEX algorithm.

ters specifying (1) the maximum number K_N of nodes and productions that are examined when computing patterns matching at a given node, (2) the maximum size K_S of a new pattern, and (3) the maximum rank K_R of a new pattern. If $A_1, \ldots, A_l$ are the nonterminals of G (in SL order) and G is an SLT grammar containing these nonterminals, then $<_G^l$ indicates the ordering over all nodes of $\mathrm{rhs}_G(A_1)$,..., $\mathrm{rhs}_G(A_l)$ obtained by scanning $\mathrm{rhs}_G(A_l)$ through $\mathrm{rhs}_G(A_1)$, each in postorder, and z is the current position with respect to this ordering. At each step, BPLEX computes a set of *repeated matches* by comparing the patterns occurring at z with the right-hand sides of the last K_N productions of G with index greater than l, and a set of *new matches* by finding pairs of non-overlapping occurrences of patterns at z and at the K_N most recently visited nodes (thus exploiting the well-known idea of a sliding window that appears e.g. in many implementations of the LZ77 compression scheme, cf. the discussion in Section 7). If at least one match is found, BPLEX performs the sharing that provides the highest size reduction for the grammar, it moves to the next node otherwise. If there is no next node, then it returns the current SLT grammar.

We now examine the algorithm in detail. We describe the progress of the computation through a sequence of configurations $(G_1, z_1), \ldots, (G_h, z_h)$ where, for each $i \in [h]$, G_i is an SLT grammar generating the uncompressed tree, and z_i is an *address* (see below) denoting the node that is examined during the i-th iteration (the *current* node). $G_1 = G$ is the input to the algorithm; G_h is the output. For $i \in [h]$, grammar G_i has nonterminals $A_1, \ldots, A_{l_i}$, with $l_1 = l$ and, for $i > 1$, either $l_i = l_{i-1}$ or $l_i = l_{i-1} + 1$ and $A_{l_i} = \mathrm{fresh}(G_{i-1}, k)$ for some $k > 0$. By $\mathrm{fresh}(G, k)$ we denote a nonterminal of rank k that does not occur in G. Given $i \in [h]$ and a grammar G_i, scanning the nodes of $\mathrm{rhs}_{G_i}(A_l)$ through $\mathrm{rhs}_{G_i}(A_1)$ in postorder induces a total order $<^l_{G_i}$ on the set of nodes $V^l_{G_i} = \bigcup_{j \in [l]} V(\mathrm{rhs}_{G_i}(A_j))$. For $i \in [h]$ and $j \in [l]$, a node in $V(\mathrm{rhs}_{G_i}(A_j))$ is denoted by the address $z = (j, u)$, where u is the path to that node in the tree $\mathrm{rhs}_{G_i}(A_j)$. If z is a node in $V^l_{G_i}$ that is not the root of $\mathrm{rhs}_{G_i}(A_1)$, then $\mathrm{next}(<^l_{G_i}, z)$ is the node following z in the order $<^l_{G_i}$. The starting address z_1 is the left-most leaf of $\mathrm{rhs}_{G_1}(A_l)$ and the final address $z_h = (1, \varepsilon)$ is the root of $\mathrm{rhs}_{G_h}(A_1)$.

A tree pattern can be described by a tree with parameters at leaves (parameters denote connected subtrees that are not part of the pattern). Formally, a (*tree*) *pattern* p (*of rank* k) is a tree in which each $y \in Y_k$ occurs exactly once. Given a tree t and a node u of t, the pattern p *matches* t in u if there are trees $t_1, \ldots, t_k$ and a pattern p' isomorphic to p such that $t/u = p'\Theta$ where Θ is the substitution $[y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k]$. The pair (p', Θ) is called a *match* of p (in t) at u. Given a match m, p_m denotes the corresponding pattern. Two matches (p', Θ'), (p'', Θ''), are *overlapping* if p' and p'' have at least one common node. Two matches $m' = (p', [y_1 \leftarrow t'_1, \ldots, y_k \leftarrow t'_k])$, $m'' = (p'', [y_1 \leftarrow t''_1, \ldots, y_k \leftarrow t''_k])$ of the same pattern p are *maximal* if, for all $i \in [k]$, $t'_i[\varepsilon] \neq t''_i[\varepsilon]$ (intuitively: there is no possibility to extend m', m'' to matches of some larger common pattern). Given a grammar G with nonterminals $A_1, \ldots, A_h$ and $j \in [h]$, a pattern p *matches* G in $z = (j, u)$ if p matches $\mathrm{rhs}_G(A_j)$ in u; if $m = (p', \Theta)$ is the match of p in $z = (j, u)$, then z is the address of m in G.

The replacement of patterns is defined as follows. Let G be an SLT grammar, p a pattern of rank k with a corresponding production $A(y_1, \ldots, y_k) \to p$ in G, and $m = (p', [y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k])$ a match of p in the right-hand side of some other production of G. The match m is replaced by A by replacing the subtree rooted at the root of p' and with the tree $A(t_1, \ldots, t_k)$. The resulting grammar is denoted by $G[m \leftarrow A]$. Similarly, for two non-overlapping matches m_1, m_2 of p in G, $G[m_1, m_2 \leftarrow A]$ is the grammar obtained from G by replacing each match m_1 and m_2 by A.

We now discuss how the size of an SLT grammar changes when occurrences of a tree pattern are replaced by a nonterminal that generates the pattern. The *size of a tree* (without parameters) is its *number of edges*. Since the SLT grammars that are generated by BPLEX have the property that all k parameters of a nonterminal appear exactly once in the right-hand side of its rule, and in the order $y_1, y_2, \ldots, y_k$, we do not need to explicitly represent the parameters as nodes of the tree. Hence, we do not count the edges to parameters; thus in general, for a tree t, $\mathrm{size}(t)$ is defined as $|E(t)| - |E_y(t)|$ where $E_y(t)$ are the edges to parameters in t. For a tree grammar G, $\mathrm{size}(G)$ is the sum of sizes of the right-hand sides of the productions of G. Clearly, $\mathrm{size}(G) - \mathrm{size}(G[m \leftarrow A]) = \mathrm{size}(p)$ and $\mathrm{size}(G) - \mathrm{size}(G[m_1, m_2 \leftarrow A]) = 2 \times \mathrm{size}(p)$. If prod is not in G already then the size of the grammar $\mathrm{add}(G, \mathrm{prod})$ obtained by adding prod to G is $\mathrm{size}(G) + \mathrm{size}(\mathrm{rhs}(\mathrm{prod}))$.

Let us turn our attention to the computation of pattern sets. At stage i, BPLEX computes the set $\mathrm{RepM}(G_i, z_i, K_N)$ of all matches in z_i of patterns that are isomorphic to some right-hand side $\mathrm{rhs}_{G_i}(A_j)$ for $l < j \leq l_i$, $l_i - j < K_N$. This computation considers at most K_N productions of index greater than l. Note that one can check whether $p = \mathrm{rhs}_{G_i}(A_j)$ matches G_i in z_i in at most $\mathrm{size}(p) \leq K_S$ steps by comparing the two trees top-down and binding parameters of p to descendants of z_i. The total cost of computing $\mathrm{RepM}(G_i, z_i, K_N)$ is bounded by $K_N \times K_S$ because (see below) a production with index $j > l$ has size at most K_S.

BPLEX also computes the set $\mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$ of all matches in z_i such that, for each $m \in \mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$, we have

- there exists a (non-overlapping) *companion match* c_m of the same pattern in some node w among the last K_N nodes preceding z_i in the order $<^l_{G_i}$;
- $0 < \mathrm{size}(p_m) \leq K_S$ and, if $\mathrm{size}(p_m) < K_S$, then either (1) m and c_m are maximal, or (2) m and c_m can only be extended to larger matches that overlap;
- the rank of p_m is at most K_R.

The set $\mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$ can be computed by comparing top-down the tree rooted at z_i with trees rooted at nodes preceding z_i. Since the computation stops whenever it encounters a pattern that is larger than K_S, the cost of computing $\mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$ is bounded by $K_N \times K_S$.

BPLEX chooses a match $m \in \mathrm{RepM}(G_i, z_i, K_N) \cup \mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$ with maximal size, denoted by $\max(\mathrm{repM}, \mathrm{newM})$. If $m \in \mathrm{RepM}(G_i, z_i, K_N)$, then the match is replaced by the right-hand side of the corresponding production. If $m \in \mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$, BPLEX adds a production $A \to p_m$ to the grammar, with $A = \mathrm{fresh}(G_i, \mathrm{rank}(p_m))$, and replaces the matches m, c_m by A. In both cases, the size of the grammar is reduced by $\mathrm{size}(p_m)$. If no matches are found, BPLEX tries to move the address z_i to the next node with respect to the order $<^l_{G_i}$. The linearity of BPLEX derives from the fact that, for an input grammar G, the loop cannot be executed more than $2 \times |G|$ times (each run through the loop either moves the address forward or reduces the size of the grammar), and from the fact that the sets $\mathrm{RepM}(G_i, z_i, K_N)$ and $\mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$ can be computed in constant time. Note that each nonterminal in the output grammar has rank at most K_R (see also Section 6). Finally, note that the indices of nonterminals in the generated grammars do not reflect the SL order; in the examples we have renamed nonterminals to indicate an SL order.

We now illustrate the computation of BPLEX on the regular tree grammar on the left of Fig. 1. BPLEX does not perform any sharing in the third and second production; it then scans the first production. When the highest d is encountered (address $(1,2)$) a match m of pattern $d(C, y_1)$ is found, together with a companion c_m matching in $(1, 2.2.2)$. This has size 1 and is chosen for replacement. The new nonterminal D of rank 1 is added to the grammar together with production $D(y_1) \to d(C, y_1)$, and the two matches are replaced so that the first production becomes $S \to c(C, D(c(C, D(C))))$. The new pattern $\mathrm{rhs}(D)$ does not match the new grammar in $z = (1,2)$ and no pairs of new matches are found either. Therefore z is changed to the root of the S production ($z = (1, \varepsilon)$). Here, the right-hand side of D does not match, while the maximal pattern $c(C, D(y_1))$ matches in $(1, \varepsilon)$ and in $(1, 2.1)$. Therefore a new nonterminal E of rank 1 is added together with the production $E(y_1) \to c(C, D(y_1))$, and the matches are replaced by E, producing the output grammar shown in Fig. 3. Both this grammar and the cf tree grammar on the right of Fig. 1 have size

$$\begin{array}{lll} S & \to E(E(C)) & \qquad C \to c(A, A) \\ E(y_1) & \to c(C, D(y_1)) & \qquad A \to a \\ D(y_1) & \to d(C, y_1) & \end{array}$$

Fig. 3. Cf tree grammar generating $\{c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))\}$.

7. Note that BPLEX has not detected pattern $p = c(C, d(C, y_1))$ appearing in Fig. 1, because the smaller pattern $d(C, y_1)$ is replaced before p has been scanned completely.

4 XML Compression using BPLEX

In this section we explain how BPLEX can be used to generate a small representation of the tree structure of an XML document. This tree structure can be conveniently modeled as unranked or binary tree. While BPLEX performs (almost) equally well in both models, this is *not* the case for the minimal DAG.

An XML document is a sequential representation of a nested list structure. As mentioned in the Introduction, there are different data models for XML, which vary in their sizes. For example, DOM trees contain bidirectional pointers between a node and its children, its parent node, and its direct left and right sibling; the resulting size is approximately 4-5 times more than the size of the original XML document. Another data model are (ordered) unranked trees which are like DOM trees, but without pointers between siblings. As an example, consider the following XML document skeleton (i.e., without data values).

```
<agenda>
   <person><name/><street/></person>  }
   ...                                 } 5 times
   <person><name/><street/></person>  }
</agenda>
```

An (ordered) unranked tree representation of this XML document consists of a root node labeled `agenda` which has associated with it an array of five pointers, each to a node labeled `person` which in turn has an array of two pointers to nodes labeled `name` and `street`, respectively. For each pointer to a child node we can additionally also keep the inverse pointer from the child to its parent node. This doubles the number of pointers in the representation. Our investigations are independent of this choice: we always count in number of edges (these numbers have to be multiplied by the implementation cost of an edge, which possibly involves the cost of two pointers). The size of the unranked tree representation of the above XML document is 15 edges.

The BPLEX algorithm works on (ranked) trees; it is well-known that every unranked tree can be turned into a binary ranked tree without changing the number of edges: delete all edges to non-first children, and add a (second child) edge from any node to its next sibling. Note that a leaf (resp. the last sibling) in the unranked tree has no left (resp. no right) child edge in the binary tree representation; this is denoted by the superscript 2 (resp. 1), and by 0 for a last sibling leaf. In Fig. 4 the binary representation of the unranked tree for the XML document above is shown (with second

$$
\begin{array}{ccccccccc}
\text{agenda}^1 & & & & & & & & \\
| & & & & & & & & \\
\text{person} & - & \text{person} & - & \text{person} & - & \text{person} & - & \text{person}^1 \\
| & & | & & | & & | & & | \\
\text{name}^2 & & \text{name}^2 & & \text{name}^2 & & \text{name}^2 & & \text{name}^2 \\
| & & | & & | & & | & & | \\
\text{street}^0 & & \text{street}^0 & & \text{street}^0 & & \text{street}^0 & & \text{street}^0
\end{array}
$$

Fig. 4. Binary tree representation of an unranked tree.

child edges of person-nodes drawn horizontally). As before, we first turn a (ranked) tree into its minimal DAG, represented as a regular tree grammar, and then apply BPLEX to the grammar. In our example, the corresponding regular tree grammar has the three productions $S \to \text{agenda}^1(\text{person}(A, \text{person}(A, \text{person}(A, \text{person}(A, \text{person}^1(A))))))$, $A \to \text{name}^2(B)$, $B \to \text{street}^0$ and its size is 11. Consider the S-production of this grammar. Its right-hand side contains four occurrences of the pattern $p = \text{person}(A, y_1)$. Thus, given a production $C(y_1) \to \text{person}(A, y_1)$, each of the occurrences can be replaced by the nonterminal C. However, there is one further occurrence of a similar pattern $p' = \text{person}^1(A)$, which can be obtained by removing the parameter y_1 from the pattern p. Note that, since A is a first child in p, removing y_1 changes person into person^1. In general, we allow a nonterminal K of rank m to appear with any rank $0 \leq r \leq m$ in the right-hand sides of productions, provided it is indicated which parameters are to be deleted; in the implementation, missing parameters are marked by a special "empty tree marker". With this "overloading" semantics of productions in mind, BPLEX turns the above regular tree grammar into:

$$
\begin{array}{lll}
S & \to \text{agenda}^1(C(D(D))) & \quad A \to \text{name}^2(B) \\
D(y_1) & \to C(C(y_1)) & \quad B \to \text{street}^0 \\
C(y_1) & \to \text{person}(A, y_1) &
\end{array}
$$

In this grammar, the D-production generates copies along a path of the binary tree. Repeated applications of such copying productions cause exponential size increase. In this way, the size of the input grammar can, in certain cases, be reduced exponentially. Consider our example, but now with 10000 person entries (thus, a binary tree with

30000 edges). The corresponding minimal regular tree grammar G_{10000} has size 20001 while BPLEX outputs the following grammar of size 20:

$$\begin{array}{lcl}
S & \to & \text{agenda}^1(A_8(A_5(A_4(A_3(A_1(A_1)))))) \\
A_1(y_1) & \to & A_2(A_2(y_1)) \\
A_2(y_1) & \to & A_3(A_3(y_1)) \\
 & \vdots & \\
A_{12}(y_1) & \to & A_{13}(A_{13}(y_1)) \\
A_{13}(y_1) & \to & \text{person}(A_{14}, y_1) \\
A_{14} & \to & \text{name}^2(A_{15}) \\
A_{15} & \to & \text{street}^0
\end{array}$$

In this grammar, the symbol A_{13} generates the tree $\text{person}(\text{name}(\text{street}, y_1))$. More generally, for $j = 1, \ldots, 13$, A_j generates a chain with 2^{13-j} occurrences of this pattern and one parameter y_1 at the end of the chain. It is easy to see that S generates the correct tree with 10000 person entries.

Unranked DAGs with Multiplicities Before presenting experimental results with BPLEX, we discuss its relation to another tree compression method that has been applied to XML. Recall that we applied BPLEX to the minimal regular tree grammar of a binary tree representation of an unranked tree. An unranked tree has itself a unique minimal DAG (minimal regular tree grammar) which can be obtained in the same way as for ranked trees. However, the size of the minimal DAG of an unranked tree can be different from the one of the minimal DAG of its binary representation! In most cases the minimal unranked DAG is smaller than the binary one. The reason is that chains of second child edges in the binary tree become sibling subtrees in the unranked tree. To see this, consider the binary tree in Fig 4. Clearly, its minimal DAG has only one copy of the subtree $\text{name}^2(\text{street})$ and hence has only 11 edges. On the other hand, the minimal DAG of the corresponding unranked tree has only one copy of the subtree $\text{person}(\text{name}, \text{street})$ and therefore has only 7 edges. As an example of a binary tree with a minimal DAG that is smaller than the one of the corresponding unranked tree, consider the unranked tree $t_u = u(p(x, b, c, b, c), p(y, b, c, b, c), p(z, b, c, b, c))$. Its minimal unranked DAG has 18 edges, but the minimal binary DAG has only 12, because only one copy of the subtree $b^2(c^2(b^2(c^0)))$ appears.

In fact, the size of the minimal DAG representation can even be further reduced by using "multiplicity" counters for consecutive equal subtrees [3]. Then the DAG for the unranked tree of the agenda-example can be represented using only 3 edges, or equivalently, by an (unranked) regular tree grammar with multiplicity counters and productions

$$A \to \text{agenda}([5]P), \quad P \to \text{person}(\text{name}, \text{street}).$$

Of course, multiplicity counters take up space, but following Koch et al. this space is neglected (similar to the fact that we do not count edges to parameters in cf tree grammars, see Section 3). Thus, BPLEX produces the grammar dsiplayed on the previous page, which is smaller (size 6) than the minimal DAG of the unranked tree (size 7), but such a minimal DAG has a smaller representation (size 3) when multiplicity counters are added. From now on, we call this DAG representation for an unranked tree its mDAG (minimal DAG with multiplicities). Such representation can easily be turned into a regular tree grammar with the *same size* that generates the binary representation of the original unranked tree. This grammar also contains multiplicity counters at nodes, which are expanded to chains of nodes. We implemented a version of BPLEX which works on such grammars (and does not change the multiplicity counters). As it turns out, only in a few cases we obtained small improvements over BPLEX on the binary regular tree grammar corresponding to the minimal DAG. Thus, the advantage of counters is compensated for, by the ability of BPLEX to exponentially compress chains of nodes. On a few files, the minimal binary DAG was even smaller than the mDAG, due to similar chains as in the tree t_u of above; cf. in Tab. 1 the two catalog files and the file NCBI_gene.chr1.

5 Experimental Results

We implemented BPLEX in C using gcc and the Expat XML parsing library (see

input file	size of tree in #edges	min. binary DAG size		min. unranked mDAG size		BPLEX output size	
SwissProt (457,4 MB)	10,903,568	1,437,445	13.2%	1,100,648	10.1%	311,328	2.9%
DBLP (103.6 MB)	2,611,931	533,183	20.4%	222,754	8.5%	115,902	4.4%
Treebank (55.8 MB)	2,447,727	1,454,494	59.4%	1,301,688	53.2%	519,542	21.2%
1998statistics (657 KB)	28,306	2,403	8.5%	726	2.6%	410	1.4%
catalog-02 (104M)	2,240,231	52,392	2.3%	32,267	1.4%	26,774	1.2%
catalog-01 (11M)	225,194	6,990	3.1%	8,503	2.8%	3,817	1.7%
dictionary-02 (104M)	2,731,764	681,130	24.9%	441,322	16.2%	160,329	5.9%
dictionary-01 (11M)	277,072	77,554	28.0%	46,993	17.0%	20,150	7.3%
JST_snp.chr1 (36M)	655,946	40,663	6.2%	25,047	2.3%	12,858	1.8%
JST_gene.chr1 (11M)	216,401	14,606	6.7%	5,658	2.6%	4,000	1.8%
NCBI_snp.chr1 (190M)	3,642,225	809,394	22.2%	15	<0.1%	59	<0.1%
NCBI_gene.chr1 (24M)	360,350	14,356	4.0%	11,767	3.3%	7,160	2.0%
medline_0378 (123M)	2,790,421	629,853	22.6%	695,505	24.9%	132,733	4.8%

Table 1. BPLEX in highest compression mode. All sizes are in number of edges.

http://expat.sourceforge.net/). See http://bplex.sourceforge.net/ for a preliminary version of BPLEX. Our experiments were done on a Pentium 3Ghz running Linux. We tested BPLEX on three different sets of XML documents. The first one contains documents used in [3]: SwissProt (protein data), DBLP (a bibliographic database), Treebank (a linguistic database), and 1998statistics (baseball statistics). The second set contains XML documents generated by XBench [29], and the third contains documents from the Japanese Single Nucleotide Polymorphism database (see http://snp.ims.u-tokyo.ac.jp).

Table 1 shows for each document the size of its tree structure (in number of edges) together with the sizes in three different representations. The minimal unranked DAG (with multiplicities) is consistently smaller than the minimal binary DAG. The smallest sizes are generated by BPLEX, ranging between 0.1% and 21% of the original tree structure; they were obtained by running BPLEX with large input parameters (window size 30.000, maximal pattern size 20, maximal rank 10). Only late we were informed by the authors of [3] that their DAG compression does not perform well on the medical bibliographies of medline; note, this is the only example in the table for which a binary DAG is slightly smaller than the mDAG. As seen in the last entry of the table, BPLEX performs surprisingly well on medline.

We also implemented a version of BPLEX that runs on unranked trees, instead of binary trees. The results are not shown in the table, because, roughly, they are the same as BPLEX on binary encodings. This means that tree compression by BPLEX is *not* sensible to un-/rankedness of the input. This is interesting, because, as shown in the previous section, this is *not* true for tree compression by DAGs.

Claim. BPLEX is unsensible to unrankedness/bin.-encoding of input.

Performance Recall from Fig. 2 the three parameters of BPLEX: the window size K_N, the maximal rank K_R of a pattern, and the maximal size K_S of a pattern. Our experiments show that the algorithm performs well with small values of K_R and K_S and that values above 5 and 10 respectively do not increase compression anymore. The main factor for good compression is the window size. BPLEX achieves best compression with a window size of > 100; values above $20,000$ do not change compression. Our current implementation runs slow on large window sizes, requiring several hours to obtain the results shown in Tab. 1. This is mainly due to the way in which matches of patterns are found and recorded; the part of the program should be improved in the future. Interestingly, even with a small window size, BPLEX already compresses considerably better than binary DAGs and unranked mDAGs. If we use $K_N = K_R = K_S = 3$ then all our examples compress in less than one minute; compression rates are SwissProt 4.1%, Treebank 34%, and dictionary-01 12%. It remains to test on a real machine the impact of our compression wrt the total memory consumption for an XML document in main memory.

6 Algorithms on SLT Grammars

SLT grammars are well suited to efficiently represent XML documents. Consider now a grammar in memory which represents a large XML document. How can we process the XML tree, without decompressing the grammar? Any read access like, e.g., reading the label of the root node, or moving along an edge from one node to another, can be realized on the grammar representation with an additional per-step overhead of at most the size h of the grammar [19]. Additionally, a stack of height at most h must be maintained at all times. Thus, the price to be payed for having a small representation that can be accessed without decompression, is a slow down for each read operation. For some special applications, however, it is possible to eliminate the slow-down, or to even achieve speed ups. In this section we investigate such applications.

XML Type Validation The first application we consider is XML type validation: an XML document represented by an SL cf tree grammar should be validated against an XML type. There are several formalisms for describing XML types, with varying expressiveness, e.g., DTDs, XML Schema, or RELAX NG. All of these can conveniently be modeled by the regular tree languages [23], a classical concept well known from formal language theory [13]. Our first result states that XML type checking can be done in time linear in the size of the grammar G, if the maximal number of parameters m is fixed. The involved constant depends on the size of the XML type definition, and on the maximal number m of parameters of the nonterminals in G; in fact, m appears as an exponent. In BPLEX, m is controlled by the input parameter K_R. Practical experiments show that small values of m already achieve competitive compression ratios; in fact, we observed that for all the files shown in Tab. 1 taking K_R bigger than 10 does not improve the compression anymore. It can therefore be assumed that m is very small with respect to the size of G. As formal model for regular tree languages we use (deterministic bottom-up finite) tree automata. Such an automaton can be defined by a tuple $A = (Q, \Sigma, \{\delta_\sigma\}_{\sigma\in\Sigma}, F)$ where Q is a finite set of states, Σ is a ranked alphabet, $\delta_\sigma : Q^k \to Q$ for $\sigma \in \Sigma$ of rank k, and $F \subseteq Q$ is a set of final states. The transition function δ of A is extended to trees over Σ in the usual way: $\delta(\sigma(t_1, \ldots, t_k)) = \delta_\sigma(\delta(t_1), \ldots, \delta(t_k))$ for $\sigma \in \Sigma$ of rank k and $t_1, \ldots, t_k \in T_\Sigma$. The language accepted by A is $\{s \in T_\Sigma \mid \delta(s) \in F\}$.

Theorem 1. *Given an SL cf tree grammar G and a tree automaton B it can be checked whether $L(G) \cap L(B) = \varnothing$ in worst case time $O(s^m \times |G|)$, where s is the number of states of B and m is the maximal number of parameters of nonterminals of G.*

The proof of Theorem 1 can be found in the Appendix. Note that in [18] it is shown already that the problem of Theorem 1 is PSPACE-complete. The intention of the proof above was to present a more efficient algorithm. Note further that in order to use Theorem 1 in the context of XML types, the corresponding type definition has to first be transformed into a (deterministic bottom-up finite) tree automaton. If the type is given as DTD or as XML Schema, then the transformation into a deterministic tree automaton can be done in time linear in the size of the representation; the reason is that these formalisms are deterministic: there is only one rule per nonterminal, and the regular expressions which are used in right-hand sides are also deterministic (which implies that the corresponding Glushkov automaton is deterministic and can be constructed in time linear in the size of the expression). Hence, the algorithm of the proof of Theorem 1 is highly practical for DTDs and XML Schemas. For RELAX NG (which employs full regular tree languages) it might be less practical, because the size of the corresponding deterministic tree automaton can be exponential in the size of the representation r. However, if the SL cf tree grammar is linear (=SLT, which it is, if it was produced by BPLEX), then Theorem 1 can be extended to the case that the automaton B is nondeterministic: the Ψ_A are now functions from Q^k to 2^Q, where k is the rank of A; they are computed by checking for every state p and states $p_1, \ldots, p_k$ of B whether there is a run on $\mathrm{rhs}(A_n)[y_1 \leftarrow p_1, \ldots, y_k \leftarrow p_k]$ arriving in p. Thus the problem can be solved in time $O(s^{m+1} \times |G|)$.

Equality Test Consider two SL cf tree grammars G_1 and G_2. Is it possible to test whether both G_1 and G_2 generate the same tree t, without fully uncompressing the grammars, i.e., without deriving the tree t? More precisely, we are interested in the time complexity of testing equivalence of G_1 and G_2.

In the string case, i.e., if G_1, G_2 are SL cf string grammars, then the problem can be solved in polynomial time with respect to the sum of the sizes of G_1 and G_2 [25]. The proof relies on the fact that, for an SL cf string grammar G (in Chomsky nf) of size n, the length of the string derivable from a nonterminal of G is $\leq 2^n$, and therefore can be stored in n bits. Since basic operations (comparing, addition, subtraction, multiplication, etc.) on such numbers work in polynomial time with respect to n, we can compute in polynomial time the length of the word generated by any nonterminal of G. Since in the tree case this property does *not* hold anymore (because the size of t generated by an SL cf tree grammar of size n can be 2^{2^n}) it looks unlikely that the equivalence problem can also be solved by an algorithm running in polynomial time. In fact, we do not know whether such an algorithm exists. The following theorem shows that the problem can be solved using polynomial space, and hence in exponential time. On the other hand, if the grammars G_1, G_2 are linear, then they can be transformed into SL cf string grammars generating a depth-first left-to-right traversal of the corresponding tree; then, the result of [25] can be used to show that in this case testing equivalence can be done in polynomial time. The proof of Theorem 2 can be found in the Appendix.

Theorem 2. *Testing equivalence of two SL cf tree grammars G_1 and G_2 can be done in* PSPACE*, and in polynomial time if G_1 and G_2 are linear.*

7 Related Work

There are succinct pointer-less representations of trees, see, e.g., [14]. In this way, an n-node tree can be represented by $2n + o(n)$ bits, while allowing $O(1)$ time for most read operations on a tree [12]. In the context of XML, pointer-less tree representations can, e.g., be found in XPRESS [22]: label paths in an XML document are encoded by real number intervals following an arithmetic encoding; this allows to run path queries directly on the compressed instance. This method is typically applied directly to XML documents on the file system. While XPRESS has smaller query evaluation times than other systems working on compressed XML files (like, e.g., XGrind [28]), it is unclear how well it compares to other approaches (like ours) when documents are loaded into memory. It is also possible to use strings to represent XML trees in memory [30]; their experiments show that this offers good compression, while still being able to query efficiently the representation. XQueC uses a queriable XML representation that is based on compression of data values [1]. An advanced implementation which basically uses DAG sharing together with compression of data values is presented in [8]; their results are convincing, which strengthens belief in our approach, because replacing DAG sharing by SLT grammars should immediately improve their system.

Consider now the problem of finding the smallest cf string grammar for a given string. This problem is NP-complete and various approximation algorithms have been studied [16]. In particular, the size of the smallest cf grammar is lower bounded by the size of the smallest LZ77 representation of the string (when the size of the sliding window is unbounded) [4, 27]. The question arises whether a similar result holds in the tree case. But for trees it is unclear how an efficient LZ77 representation would look like. The problem is how to specify tree prefixes that have appeared before [7]. In [27] a technique to decrease the size of an SL cf grammar is presented; the idea is to change the grammar in such a way that its derivation trees become balanced trees, in the sense of AVL trees. This technique gives good compression ratios, when applied to an SL cf grammar obtained from the minimal LZ77 representation of the string. Even though there is no obvious way to extend LZ77 to trees, it might be possible to apply the technique of [27] to SL cf tree grammars. Another variation of Lempel-Ziv compression, known as LZ78, can more readily be extended to trees. For LZ78 on strings, new patterns are composed by adding a letter to already existing patterns. A pattern is specified as a pair (i, a) where i is the index of a previous pattern and a is a letter; the case $i = 0$ represents the one-letter pattern a. In this scheme the string $abbbaabbabbb$ is compressed to $(0, a)(0, b)(2, b)(1, a)(3, a)(3, b)$. Thus, the pair $(2, b)$ is the concatenation bb of b (the second pattern) and b, and similarly $(3, a)$ represents bba. The LZ78 encoding has a natural interpretation as an SL cf string grammar (see e.g. [16]). LZ78 can be extended to trees by using a dictionary of tree patterns where, during a top-down scan of the input tree, new patterns are obtained from existing ones by appending subpatterns at parameter positions; in the simplest case, only a one-node subpattern is appended. Such a technique is presented in [5]; other variations, each using a different method for extending the patterns, are presented in [6]. In [5] no experimental results are provided. In [6] the proposed algorithms are applied to term compression, and the best performance is a size reduction to about 50% of the original. It remains to be investigated how these techniques perform on XML documents.

In [10] it was shown that evaluation of Core XPath queries on DAGs is PSPACE complete. Recently we have shown that this result can be extended to linear SL cf tree grammars; this means that, while achieving better compression than DAGs by using BPLEX, the complexity of evaluating a Core XPath still remains the same for outputs of BPLEX as it is for DAGs.

8 Conclusions and Future Work

A linear time algorithm was presented that transforms a given tree into a small SLT grammar. The algorithm can be used to "compress" the tree structure of an XML document into a highly efficient memory representation. The representation preserves the basic tree operations and can be accessed via DOM (using an appropriate proxy). On average, the size of a compressed instance is one half of the size of the minimal unique DAG of the tree, which in turn is about 1/10 of the size of the original tree [3]. Some problems can, under certain conditions, even be solved more efficiently on the compressed instances than on conventional tree presentations; in particular we considered (1) validation against XML types and (2) testing equality of documents. In [18] we considered XQuery evaluation. It remains to implement these ideas and test how well they behave on practical queries. To further increase memory efficiency, our representation could be combined with a (mild) compression of data values (e.g., similar to the one of [1]). It is also possible to directly keep results of queries in compressed format; this idea has been considered for DAG compression and a fragment of XQuery [2]. It also has been considered for compression by SLT grammars, and macro tree transducers as query formalism [19]. It is not difficult to change BPLEX to take arbitrary SL cf tree grammars as input; in this way it might be possible to achieve further compression by running BPLEX on its on output.

Several recent programming languages allow to process XML documents via pattern matching constructs. Such constructs are compiled into automata which carry out the matching in the document. It seems straightforward to extend this compilation to automata which directly work on SLT grammars. In this way an efficient XML query evaluator is obtained because XQueries and XSLTs can be translated to pattern matching statements. In this context, other optimization might become important (e.g. lazy sequences [11]).

We would like to test how our technique can be used for XML file compression. Maybe the performance of existing compressors, like XMill, can be further improved by using BPLEX for the compression of tree structure.

References

1. A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing queries to compressed XML data. In *Proc. VLDB*, pages 1065–1068, 2003.
2. P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large XML repositories. To appear in *Proc. ICDE*, 2005.
3. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proc. VLDB*, pages 141–152, 2003.
4. M. Charikar et.al. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. STOC'02*, pages 792–801. ACM Press, 2002.
5. S. Chen and J. H. Reif. Efficient lossless compression of trees and graphs. In *Proc. DCC'96*, page 428. IEEE Computer Society Press, 1996.
6. J. R. Cheney. First-order term compression: techniques and applications. Master's thesis, Carnegie Mellon University, August 1998.
7. J. R. Cheney. Personal communication. 2004.
8. J. Cheng and W. Ng. XQzip: Querying compressed xml using structural indexing. In *Proc. EDBT*, pages 219–236, 2004.
9. M. F. Fernandez, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing xquery 1.0: The galax experience. In *Proc. VLDB*, pages 1077–1080, 2003.
10. M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Proc. LICS*, pages 188–197. IEEE, 2003.
11. V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. To appear in *Proc. CC.*, 2005.
12. R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. SODA*, pages 1–10, 2004.
13. F. Gécseg and M. Steinby. Tree languages. In *Handbook of Formal Languages, Volume 3*, chapter 1. Springer-Verlag, 1997.
14. J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Intern. J. of Foundations of Comput. Sci.*, 1:425–447, 1990.
15. J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proc. POPL'1990*, pages 16–30. ACM Press, 1990.

16. E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proc. SODA*, pages 205–212. SIAM Press, 2002.
17. H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In W. Chen et. al., editor, *Proc. SIGMOD*, pages 153–164. ACM, 2000.
18. M. Lohrey and S. Maneth. Tree automata on compressed trees. Submitted manuscript, 2005.
19. S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proc. FOSSACS'04*, volume 2987 of *LNCS*, pages 363–377. Springer-Verlag, 2004.
20. D. Megginson. *Imperfect XML: Rants, Raves, Tips, and Tricks ... from an Insider*. Addison-Wesley, 2004.
21. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comp. Syst. Sci.*, 66:66–97, 2003.
22. J. Min, M. Park, and C. Chung. XPRESS: A queriable compression for XML data. In *Proc. SIGMOD*, pages 122–133. ACM Press, 2003.
23. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages*, 2000.
24. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.
25. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Proc. ESA'94*, volume 855 of *LNCS*, pages 460–470. Springer-Verlag, 1994.
26. W. Rytter. Algorithms on compressed strings and arrays. In *Proc. SOFSEM 1999*, volume 1725 of *LNCS*, pages 48–65. Springer-Verlag, 1999.
27. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302:211–222, 2002.
28. P. M. Tolani and J. R. Hartisa. XGRIND: A query-friendly XML compressor. In *Proc. ICDE 2002*, pages 225–234. IEEE Computer Society, 2002.
29. B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proc. ECDE 2004*, pages 621–633. IEEE Computer Society, 2004.
30. N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. ICDE*, pages 54–65, 2004.

Appendix

Proof of Theorem 1 Let $G = (N, \Sigma, \mathrm{rhs})$ with $N = \{A_1, \ldots, A_n\}$ and $B = (Q, \Sigma, \delta, F)$. We assume that G is reduced, i.e., each nonterminal is used in a (successful) derivation of G. We now run the tree automaton B on the right-hand sides of G. If we do this bottom-up, starting with the right-hand side $\mathrm{rhs}(A_n)$, then for the parameters in $\mathrm{rhs}(A_n)$ we have to try all possibilities of states, obtaining a finite function Ψ_{A_n} from Q^k to Q, where k is the rank of A_n. This function is now used as transition for A_n, when running B on right-hand sides $\mathrm{rhs}(A_m)$ with $m < n$. In this way, for each nonterminal of rank k, $|Q|^k$ many values of Ψ are computed. Hence, in total at most $s^m \times |G|$ computations steps are needed.

This number can be greatly decreased by going *top-down* in a 'lazy' manner through G, starting with $\mathrm{rhs}(A_1)$. Note though, that the price for the improvement is the necessity to maintain recursive calls. Consider the run of B on $\mathrm{rhs}(A_1)$. If B arrives at a nonterminal A_i ($i > 1$) of rank k, in states $q_1, \ldots, q_k$, then we issue a recursive call to compute $\Psi_{A_i}(q_1, \ldots, q_l)$. Such a call means to substitute q_j for y_j, $1 \leq j \leq k$ in $\mathrm{rhs}(A_i)$ and then to run B on this tree. During the run further recursive calls may be generated. Clearly, in the worst case again at most $s^m \times |G|$ computations are needed. On average, however, the top-down procedure is far more efficient than the above bottom-up algorithm. □

Proof of Theorem 2 Let $G_1 = (\{A_1, \ldots, A_m\}, \Sigma, \mathrm{rhs}_1)$ and $G_2 = (\{B_1, \ldots, B_n\}, \Sigma, \mathrm{rhs}_2)$ be SL cf tree grammars. By Savitch's Theorem (see, e.g., [24]) and the complement closure of PSPACE, it suffices to give a nondeterministic algorithm that tests *in*equivalence. Roughly speaking, the algorithm guesses corresponding paths in the DAGS d_1 and d_2, generated by G_1 and G_2 respectively, and accepts if the labels of the corresponding nodes are different. The DAG d_i (for $i = 1, 2$) is obtained from G_i by identifying all nodes in a right-hand side that are labeled with the same parameter. The key issue is now that a node in d_i can be represented in polynomial space w.r.t. the size of G_i. This representation is discussed in the end of [19]. It consists of a sequence $(i_1, u_1), (i_2, u_2) \ldots, (i_p, u_p)$ where $i_1 = 1$, $i_1 < \cdots < i_p$ are indices in $\{1, \ldots, m\}$, and for $1 \le \nu < p$, u_ν is a node in $\mathrm{rhs}_1(A_{i_\nu})$ with label $A_{i_{\nu+1}}$; moreover $\mathrm{rhs}_1(A_{i_p})[u_p] \in \Sigma$. The first pair $(1, u_1)$ denotes that we start a derivation of G_1 with the right-hand side of A_1 and node u_1 marked; the next pair (i_2, u_2) means u_1 is labeled A_{i_2} and that we apply its production with u_2 marked, etc. Since u_p is terminal, the sequence represents a derivation of a node of d_1. Given such a sequence h representing a node u of d_1 it is straightforward to construct a sequence h' representing the i-th child ui of u in d_1 [19]. Note that any such sequence has length $< n$. The algorithm starts with two empty sequences. It then generates the sequences h_1, h_2 representing the root nodes of d_1, d_2, respectively. If their labels are different we accept. Otherwise, we guess a child number i and move down to the i-th child, resulting in h'_1, h'_2. If the corresponding labels are different we accept, etc. If there is no child number (we are at a leaf) we reject.

Now let G_1, G_2 be linear. This means that for any nonterminal A of G_1, G_2, of rank k, the tree $A(y_1, \ldots, y_k)$ derives to a tree t over $\Sigma \cup Y_k$ in which y_j occurs at most once, $1 \le j \le k$. In fact, it is straightforward to change the grammars in such a way that (1) every y_j occurs exactly once in t and (2) the order of the parameters in t (going depth-first left-to-right) is $y_1, \ldots, y_k$. The idea is now to construct cf string grammars H_1, H_2 which generate depth-first left-to-right traversals of t_1 and t_2, respectively. Let $i \in \{1, 2\}$. For every nonterminal X of G_i of rank $k > 0$ let $X_{0,1}, X_{1,2}, \ldots, X_{k-1,k}, X_{k,0}$ be new nonterminals of H_i, and for every $\sigma \in \Sigma$ of rank $k > 0$ let $\sigma_{0,1}, \sigma_{1,2}, \ldots, \sigma_{k-1,k}, \sigma_{k,0}$ be new terminals of H_i. Nonterminals and terminals of rank zero are taken over to H_i. The right-hand side of the nonterminal $A_{0,1}$ is the traversal starting at the root of the right-hand side of A (indicated by the index 0) up to the first parameter y_1 in the right-hand side of A (indicated by the parameter 1); The right-hand side of $A_{\nu,\nu+1}$ is the traversal starting at the parameter y_ν in the right-hand side of A up to the parameter $y_{\nu+1}$. Similarly, a terminal symbol $g_{2,3}$ means that g was entered coming from its second child and was exited by moving to its third child. It should be clear how to construct the productions of H_i. As an example, consider the tree grammar production $A(y_1, y_2, y_3) \to B(g(y_1, a, b), h(B(y_2, y_3)))$ and the nonterminal $A_{1,2}$ of the constructed string grammar; its production is $A_{1,2} \to g_{1,2}\, a\, g_{2,3}\, b\, g_{3,0}\, B_{1,2}\, h_{0,1}\, B_{0,1}$. Clearly, $t_1 = t_2$ if and only if the string w_1 generated by H_1 equals w_2 (gen. by H_2). Moreover, H_1, H_2 are SL cf string grammars of polynomial size w.r.t. G_1, G_2, respectively. By the result of [25], testing $w_1 = w_2$ can be done in polynomial time w.r.t. the sizes of H_1, H_2. $\square$