Nordic Journal of Computing

BINARY QUERIES FOR DOCUMENT TREES

ALEXANDRU BERLEA and HELMUT SEIDL
Technische Universität München, Germany
{berlea,seidl}@in.tum.de

Abstract. Motivated by XML applications, we address the problem of answering *k*-ary queries, i.e. simultaneously locating *k* nodes of an input tree as specified by a given relation. In particular, we discuss how binary queries can be used as a means of navigation in XML document transformations. We introduce a grammar-based approach to specifying *k*-ary queries. An efficient tree-automata based implementation of unary queries is reviewed and the extensions needed in order to implement *k*-ary queries are presented. In particular, an efficient solution for the evaluation of binary queries is provided and proven correct. We introduce fxgrep, a practical implementation of unary and binary queries for XML. By means of fxgrep and of the fxt XML transformation language we suggest how binary queries can be used in order to increase expressivity of rule-based transformations. We compare our work with other querying languages and discuss how our ideas can be used for other existing settings.

CR Classification: F.2.2, I.7, H.3

Key words: XML, tree automata, querying, transforming

1. Introduction

Locating parts of documents with specific properties is a fundamental task in document processing and in particular in XML applications. In this work we refer to this process as querying. Querying is used on its own in order to extract information from documents. Furthermore, especially in the context of XML, where documents are often dynamically created from different XML sources, querying accomplishes the basic function of locating sub-components used for creating new content.

The importance of query-languages becomes apparent if one notes that XPath [21], the XML query language proposed by the W3C Consortium, is integral part of many other important specifications, for example XML Schema Language [23], XSLT [22] or XQuery [25]. Various other query languages have been proposed, see for example a survey on this [6].

XML documents are textual representations of trees. Most of the attention in the study of XML query languages has been drawn by *unary queries*, which locate individual nodes from the input tree. As opposed to this, in this work we address *k-ary queries*, which are able to locate *k* nodes which simultaneously satisfy a specific property. In particular, we consider binary queries and how they can be efficiently implemented.

Received 5th July 2004.

Binary queries turn out to be especially useful in rule-based transformation languages like XSLT or fxt [2]. There, queries are used for two purposes. The first purpose is to specify which are the nodes to which a rule is applicable and is accomplished by so-called *match patterns*. Secondly, within a rule, queries are used for selecting nodes for further processing, relative to the node to which the rule is applied (the *match node*). Queries used for this purpose are called *select patterns*.

The evaluation of select patterns may be problematic for two reasons. Firstly, they are to be evaluated in the context of the match node, i.e. dynamically for each match node. In contrast, match patterns can be evaluated once before the transformation begins, as the applicable rule for a node is given by the match pattern fulfilled by the node in the (static) context of the root.

Secondly, as the nodes to be selected for further processing can be anywhere around the match node, a dynamic implementation of select patterns has to allow for arbitrary navigation in the input tree, which might be a source of inefficiency. Many of the nodes to be selected however, have been visited and tested for the required properties, by the time the match pattern has been evaluated. This led us to the idea of simultaneously locating the match node and the selected nodes.

We can even remove select patterns by combining the match pattern of a rule and a select pattern within this rule into a match pattern expressed by a binary query. It turns out, as binary queries can be efficiently evaluated, that this also solves the first mentioned problematic aspect of select patterns.

Consider the following XML input document:

XML Example 1

```
<company>
  <url>spice.girls</url>
  <empl><name>Mel A.</name></empl>
  <empl><name>Mel B.</name></empl>
  <empl><name>Mel C.</name></empl>
</company>
```

The following XSLT rule produces a `homepage` element for each employee:

XML Example 2

```
<xsl:template match="company[url]/empl">
  <homepage>
    <body>Under construction.
       See the company's page:
          <link><xsl:copy-of select="../url"/></link>
    </body>
  </homepage>
</xsl:template>
```

A binary match could simultaneously locate an employee and the `url` of her company. Let us suppose that binary queries were possible in XPath. Let the second element of a binary match be specified by preceding the corresponding

node in the pattern with the % symbol, and referred within the rule by using the same symbol. The rule above could then be expressed as follows:

XML Example 3

```
<xsl:template match="company[%url]/empl">
  <homepage>
    <body>Under construction.
       See the company's page:
          <link>%</link>
    </body>
  </homepage>
</xsl:template>
```

Given the wide range of use of a query language for XML, it is desirable that it is as expressive and efficient as possible. A powerful formalism for expressing unary queries on tree-structured documents is the forest grammar formalism introduced by Neumann [14]. Neumann and Seidl [16] introduce a class of tree automata, the *pushdown forest automata*, and show how they can be efficiently used to evaluate unary queries.

The main contribution of this work is extending the grammar formalism by proposing a concept of *recognizable k-ary queries* and presenting techniques for the efficient implementation for the special case of *binary queries*, based on pushdown forest automata.

The presented techniques have been implemented in our XML querying tool fxgrep [15]. The binary queries of fxgrep are used as suggested above as select patterns in our XML transformation tool fxt. We address some challenges arising in practical implementations.

Querying with unary and binary patterns is an essential task in fxt. Therefore, the used techniques must not only be efficient but also reliable. It is for this reason that we have put some effort into proving the correctness of our main algorithm.

We firstly introduced binary queries in a contribution presented in the Extreme Markup Languages 2002 Conference [3]. The present paper is a completely revised and updated version, containing more detailed explanations of the involved algorithms, as well as the relevant proofs.

The rest of the paper is organized as follows. Section 2 formally defines trees and forests used to model XML documents, and introduces a set of notations to be used throughout the paper. Section 3 introduces regular forest grammars as a generalization of XML schema languages. Section 4 presents pushdown forest automata which can be used to efficiently check conformance to a schema. In Section 5 we show how forest grammars can express queries of arbitrary arities. Section 5.1 and Section 5.2 present each an efficient algorithm based on pushdown forest automata for answering unary and binary queries, respectively. Section 6 discusses the practical implementation of the algorithms in fxgrep and Section 7 addresses related work. We conclude in Section 8.

2. Preliminaries

Let Σ be an alphabet. $\mathcal{R}_\Sigma$ is the set of regular expressions over Σ and $[\![r]\!]_\mathcal{R}$ is the regular string language defined by some regular expression r.

The sets $\mathcal{T}_\Sigma$ of *trees* t and $\mathcal{F}_\Sigma$ of *forests* f over Σ are given as follows:

$$t \;::=\; a\langle f\rangle,\;\; a \in \Sigma \qquad f \;::=\; \varepsilon \mid t_1 \ldots t_n,\;\; n > 0\;\;,$$

where ε denotes the *empty forest*.

We write $t = x\langle\_\rangle$ or $lab(t) = x$ iff $t = x\langle f\rangle$ for some f.

Let f be a forest. Then $\Pi(f) \subseteq \mathbb{N}^*$ is the set of all paths π in f and is defined as follows:

$$\begin{aligned}\Pi(\varepsilon) &= \{\lambda\}\\ \Pi(t_1 \ldots t_n) &= \{\lambda\} \cup \{i\pi \mid 1 \le i \le n, \pi \in \Pi(f_i) \text{ for } t_i = a_i\langle f_i\rangle\}\end{aligned}$$

where λ denotes the empty string.

$N(f) = \Pi(f) \setminus \{\lambda\}$ is the set of nodes in f. A node identifies one of f's subtrees. For $\pi \in N(f)$, $f[\pi]$ is called the *subtree of* f *located at* π and is defined as follows:

$$(t_1 \ldots t_n)[i\pi] = \begin{cases} t_i \quad , & \text{if } \pi = \lambda \\ f_i[\pi], & \text{if } \pi \neq \lambda \text{ and } t_i = a\langle f_i\rangle \end{cases}$$

For a path π, we define $last_f(\pi)$ as the number of children of the node π:

$$last_f(\pi) = max(\{n \mid \pi n \in N(f)\} \cup \{0\})$$

Note that $last_f(\pi) = 0$ iff π identifies a leaf. Also note that a path always locates a tree in a *forest*, not in a tree. Given a tree t, $t[\pi]$ denotes the tree located by π in the forest which consists of t only. One can see by definition that in this case π always begins with 1. In particular, one can use the subtree $t = f[\pi_1]$ located by a path π_1 in a forest f to further locate a subtree of t. In this case we have that $f[\pi_1][1\pi_2] = f[\pi_1\pi_2]$.

3. Regular Forest Languages

An important task in document processing consists in verifying a structural property of a document tree. For example, XML validation means checking that a document has a required structure. The structure of a document can be specified by using various so-called *schema languages*. Besides the document type definition of a document, there exist various more precise schema languages like XML Schema Language, DSD [18] or RelaxNG [19].

In essence, all these languages specify *regular forest languages* as noted by Murata *et al.* [13]. Regular forest languages, also called regular hedge languages by Brüggemann-Klein *et al.* [5], constitute a very expressive and theoretically robust formalism for specifying properties of forests . Validating a document against a schema is therefore a test of membership in a regular forest language.

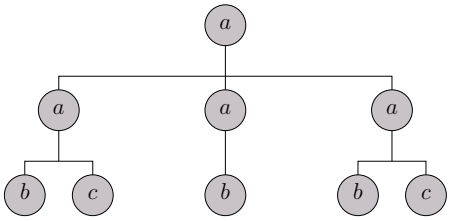


Figure 1: The tree representation of t in Example 2

One modality to specify forest regular languages is by using forest grammars, as presented by Neumann [14]. Among other possibilities of specifying regular forest languages, forest grammars have the advantage of being more comprehensible.

A *forest grammar* over Σ is a tuple $G = (R, r_0)$ where R is a set of productions (also named rules) using non-terminals from a set X and terminal symbols from Σ and $r_0 \in \mathcal{R}_X$ is the *start expression*.

The productions in R have the form $x \to a\langle r\rangle$ with $x \in X$, $a \in \Sigma$ and $r \in \mathcal{R}_X$. A set of productions R together with a distinguished non-terminal $x \in X$ or a regular expression $r \in \mathcal{R}_X$ defines a *tree derivation* relation $\mathcal{D}eriv_{R,x} \in \mathcal{T}_\Sigma \times \mathcal{T}_X$ or a *forest derivation* relation $\mathcal{D}eriv_{R,r} \in \mathcal{F}_\Sigma \times \mathcal{F}_X$, respectively, as it follows:

$$
\begin{array}{lll}
(a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x} & \text{iff} & x \to a\langle r\rangle \in R \text{ and } (f, f') \in \mathcal{D}eriv_{R,r} \\
(t_1 \dots t_n, t'_1 \dots t'_n) \in \mathcal{D}eriv_{R,r} & \text{iff} & x_1 \dots x_n \in [\![r]\!]_{\mathcal{R}} \text{ and } (t_i, t'_i) \in \mathcal{D}eriv_{R,x_i} \\
 & & \text{for } i = 1, \dots, n \\
(\varepsilon, \varepsilon) \in \mathcal{D}eriv_{R,r} & \text{iff} & \lambda \in [\![r]\!]_{\mathcal{R}}
\end{array}
$$

If $(f, f') \in \mathcal{D}eriv_{R,r}$ we say that f' is a *derivation* of f w.r.t. R and r. In the following we omit R when it is clear from the context which set of productions is meant. If $(R, r) = G$ we write $(f, f') \in \mathcal{D}eriv_G$ and say that f' is a derivation of f w.r.t. the grammar G.

Note that a derivation f' is a relabeling of f. If $lab(f'[\pi]) = x$ we say that f' *labels* $f[\pi]$ with x.

E 1. Let R be the set of following productions:

$$
\begin{aligned}
x_a &\to a\langle (x_a | x_b)^* \rangle \\
x_b &\to b\langle\rangle
\end{aligned}
$$

Let $f = a\langle a\langle\rangle b\langle\rangle\rangle$ and suppose we want to check whether there is a derivation of f w.r.t. R and x_a. We can proceed in a bottom-up manner.

It is easy to see that $(a\langle\rangle, x_a\langle\rangle) \in \mathcal{D}eriv_{x_a}$ and $(b\langle\rangle, x_b\langle\rangle) \in \mathcal{D}eriv_{x_b}$. Since $x_a x_b \in [\![(x_a|x_b)^*]\!]_{\mathcal{R}}$ we have that $(a\langle\rangle b\langle\rangle, x_a\langle\rangle x_b\langle\rangle) \in \mathcal{D}eriv_{(x_a|x_b)^*}$. It follows that $(a\langle a\langle\rangle b\langle\rangle\rangle, x_a\langle x_a\langle\rangle x_b\langle\rangle\rangle) \in \mathcal{D}eriv_{x_a}$.

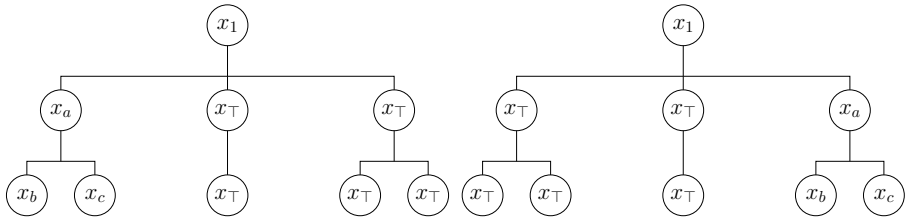


Figure 2: Possible derivations of t from Example 2

E 2. Let R_2 be the set of following productions:

(1) $x_\top \to a\langle x_\top^* \rangle$ (4) $x_1 \to a\langle x_\top^* (x_1|x_a) x_\top^* \rangle$ (6) $x_b \to b\langle x_\top^* \rangle$
(2) $x_\top \to b\langle x_\top^* \rangle$ (5) $x_a \to a\langle x_b x_c \rangle$ (7) $x_c \to c\langle x_\top^* \rangle$
(3) $x_\top \to c\langle x_\top^* \rangle$

Let t be the tree textually represented by the following XML document:

XML Example 4

```
<a>
  <a><b/><c/></a>
  <a><b/></a>
  <a><b/><c/></a>
</a>
```

The tree t is graphically presented in Fig. 1. Two possible derivations of t w.r.t. R and the regular expression $x_1|x_a$ are depicted in Fig. 2.

The *meaning* $[\![R]\!]$ of a set of productions R assigns sets of trees to the non-terminals $x \in X$ and sets of forests to regular expressions in $r \in \mathcal{R}_X$ and is defined by:

$$t \in [\![R]\!]\, x \text{ iff there is } t' \in \mathcal{T}_X \text{ and } (t, t') \in \mathcal{D}eriv_{R,x}$$
$$f \in [\![R]\!]\, r \text{ iff there is } f' \in \mathcal{F}_X \text{ and } (f, f') \in \mathcal{D}eriv_{R,r}$$

If $t \in [\![R]\!]\, x$ or $f \in [\![R]\!]\, r$ we say that t can be derived from x or f can be derived from r, respectively.

E 3. Let R be the set of productions from Example 1. It is easy to see that $[\![R]\!]\, x_b$ is the set consisting only of the tree $b\langle\rangle$. $[\![R]\!]\, x_a$ is the set of all trees whose internal nodes are all labeled a and whose leaves are labeled either a or b.

The *regular forest language* specified by a forest grammar $G = (R, r_0)$ is the set of forests $\mathcal{L}_G = [\![R]\!]\, r_0$.

E 4. Consider the grammar $G = (R_2, x_1|x_a)$ over $\{a, b, c\}$ with the productions R_2 as defined in Example 2.

$\mathcal{L}_G$ is the set of documents in which there is a path from the root to a node labeled a, whose children are a node labeled b and a node labeled c , and whose

ancestors are all labeled *a*. The first three productions make $x_\top$ account for trees with arbitrary content. As specified by production (5), x_a stands for the *a* element with the *b* and the *c* children. Productions (6) and (7) say that these children can have arbitrary content. Finally, production (4) specifies that the *a* specified by (5) can be at arbitrary depth in the input, and all its ancestors must be labeled *a*.

4. Recognizing Forest Languages

It is well known that regular ranked tree languages are recognizable by the class of bottom-up tree automata [9]. Also, every unranked tree can be encoded to a unique ranked tree representation and the notion of regular tree language is invariant under these encodings (see e.g. the proof by Neumann [14]). Therefore, bottom-up tree automata can be used to recognize regular forest languages.

In order to efficiently implement bottom-up automata, they have to be made deterministic. Deterministic bottom-up automata may have an exponential number of states. Therefore, their implementation can be prohibitively expensive.

Pushdown forest automata, proposed by Neumann and Seidl [16, 14], are equally expressive with bottom-up automata but much more concise and efficient to implement in practice. Any implementation of bottom-up automata has to traverse the input tree. The idea of pushdown automata is based on the observation that when reaching a node during the traversal, the information gained from the already visited part of the tree can be used in order to reduce the number of possible transitions of the automaton at that node. Intuitively, in the case of a depth-first, left-to-right traversal, the advantage is that the complete left context can be taken into account before processing the current node.

The name of the automata (pushdown forest automata) is due to the fact that information from the context is stored on the stack (pushdown) which is implicitly used for the tree traversal.

Also, rather than working on ranked encodings of unranked trees, the pushdown forest automata directly recognize unranked trees and forests. Besides saving the time needed for encoding, this also has the advantage of making the construction of the automata more straightforward and intelligible.

4.1 Pushdown Forest Automata

Supplementary to the tree states of classical tree automata, a pushdown automaton (PA) also has *forest states*. Intuitively, a forest state contains the information gained from the context at any point during the tree traversal. Let us consider first a left-to-right, depth-first traversal. Later, we also will consider right-to-left traversals. The following notations are essentially those introduced by Neumann [14].

The behaviour of a *left-to-right pushdown automata* is depicted in Fig. 3. When arriving at some node *n* labeled *a*, the context information is available in the forest state *q* by which the automaton reaches the node. The automaton has to traverse *n* and compute a tree state *p*, which describes *n* within the context *q*. In order to do so, the children of *n* are recursively processed. The context information for the first child, q_1, is obtained (via a *Down* transition) by refining *q* by taking into account

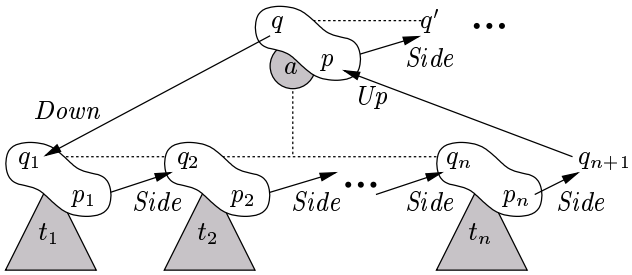


Figure 3: The processing model of a pushdown forest automaton

that the father is labeled a. Subsequently the q_2 context information for the second child is obtained (via a *Side* transition) from q_1 and the information p_1 gained from the traversal of t_1. Proceeding in this manner, after visiting all n children, enough context-information is collected in q_n in order to compute p (via an *Up* transition). After processing n the context information for the subsequent node is updated into q'.

Formally, a *left-to-right pushdown forest automaton* (LPA) $A = (P, Q, I, F, Down, Up, Side)$ consists of a set of *tree states* P, a set of *forest states* Q, a set of *initial states* $I \subseteq Q$, a set of *final states* $F \subseteq Q$, a *down-relation* $Down \subseteq Q \times \Sigma \times Q$, an *up-relation* $Up \subseteq Q \times \Sigma \times P$ and a *side-relation* $Side \subseteq Q \times P \times \Sigma \times Q$.

Based on *Down*, *Up* and *Side*, the behavior of A is described by the relations $\delta^A_{\mathcal{F}} \subseteq Q \times \mathcal{F}_\Sigma \times Q$ and $\delta^A_{\mathcal{T}} \subseteq Q \times \mathcal{T}_\Sigma \times P$ as it follows:

$$
\begin{aligned}
(q, \varepsilon, q) \in \delta^A_{\mathcal{F}} \quad & \text{for all } q \in Q \\
(q_1, f a\langle f_1 \rangle, q_2) \in \delta^A_{\mathcal{F}} \quad & \text{iff } (q_1, f, q) \in \delta^A_{\mathcal{F}},\ (q, a\langle f_1 \rangle, p) \in \delta^A_{\mathcal{T}} \text{ and} \\
& (q, p, a, q_2) \in Side \text{ for some } q \in Q, p \in P \\
(q, a\langle f \rangle, p) \in \delta^A_{\mathcal{T}} \quad & \text{iff } (q, a, q_1) \in Down,\ (q_1, f, q_2) \in \delta^A_{\mathcal{F}} \\
& \text{and } (q_2, a, p) \in Up \text{ for some } q_1, q_2 \in Q
\end{aligned}
$$

The language accepted by the automaton A is given by:

$$\mathcal{L}_A = \{ f \in \mathcal{F}_\Sigma \mid \exists\, q_1 \in I, q_2 \in F \text{ and } (q_1, f, q_2) \in \delta^A_{\mathcal{F}} \}$$

Similarly, if we consider a right-to-left depth-first traversal we obtain a *right-to-left pushdown forest automaton* (RPA). An RPA $A = (P, Q, I, F, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$ is similar to an LPA but it proceeds on a forest from the right to the left, i.e. the second case of $\delta^A_{\mathcal{F}}$ above is replaced by:

$$
\begin{aligned}
(q_1, a\langle f_1 \rangle f, q_2) \in \delta^A_{\mathcal{F}} \quad & \text{iff } (q_1, f, q) \in \delta^A_{\mathcal{F}},\ (q, a\langle f_1 \rangle, p) \in \delta^A_{\mathcal{T}} \text{ and} \\
& (q, p, a, q_2) \in Side^{\leftarrow} \text{ for some } q \in Q, p \in P
\end{aligned}
$$

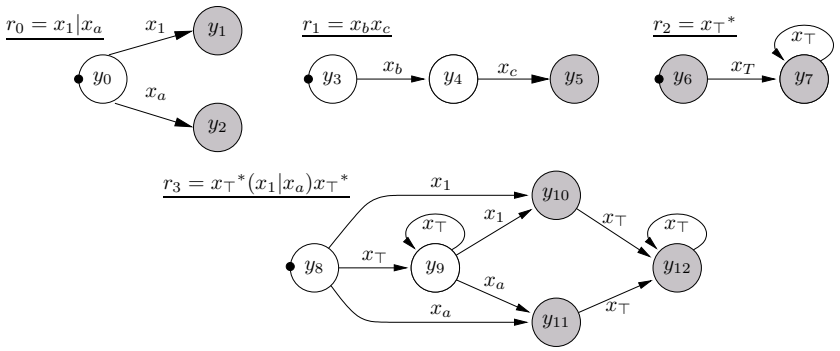


Figure 4: NFAs obtained by Berry-Sethi construction for regular expressions in Example 4

4.1.1 Compiling Forest Grammars into Pushdown Automata

Neumann and Seidl [16] show that every non-deterministic PA can be made deterministic. Neumann [14] gives a compilation schema for translating a forest grammar into a deterministic LPA (DLPA) accepting the same regular forest language. In this section we briefly recall this compilation schema.

For a forest grammar $G = (R, r_0)$ over an alphabet Σ and with non-terminals from a set X, let $r_1, \ldots, r_l$ be the regular expressions occurring on the righthand sides in the productions R, where l is the number of productions. Moreover, for $0 \leq j \leq l$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the regular string language defined by r_j as obtained by the Berry-Sethi construction [4]. Here Y_j is the set of NFA states, $y_{0,j}$ the start state, F_j the set of final states and $\delta_j \in Y_j \times \Sigma \times Y_j$ is the transition relation.

An NFA obtained by the Berry-Sethi construction has the important property that all transitions coming into the same state are labeled by the same symbol. This property is used in the querying algorithms based on PAs.

E 5. Consider the regular expressions occurring in the productions in Example 4. The corresponding NFAs as obtained by the Berry-Sethi construction are depicted in Fig. 4. Initial states are marked by the • symbol. Final states are depicted in gray.

By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_l$ and $\delta = \delta_0 \cup \ldots \cup \delta_l$. A DLPA $A^{\vec{}}_G$ accepting $\mathcal{L}_G$ can be defined as $A^{\vec{}}_G = (2^X, 2^Y, \{q_0\}, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$, where:

$$
\begin{aligned}
q_0 &= \{y_{0,0}\} \\
F &= \{q \mid q \cap F_0 \neq \emptyset\} \\
\mathit{Down}(q, a) &= \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle \text{ for some } x, y_1\} \\
\mathit{Up}(q, a) &= \{x \mid x \to a\langle r_j \rangle \text{ and } q \cap F_j \neq \emptyset\} \\
\mathit{Side}(q, p, a) &= \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\}
\end{aligned}
$$

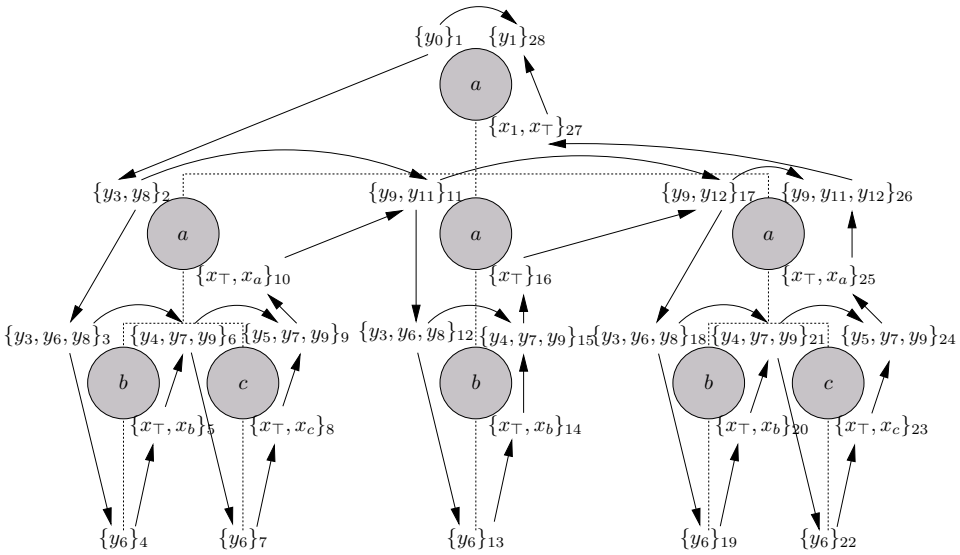


Figure 5: The run of $A_G^{\rightarrow}$ on the input document in Example 6

As the *Side* transition of $A_G^{\rightarrow}$ does not use the a parameter, we will omit it in the following.

E 6. Consider the grammar G from Example 4. The NFAs for the regular expressions occurring in G are depicted in Fig. 4. As input consider the XML document from XML Example 4. The run of $A_G^{\rightarrow}$ on the tree representation of the input is shown in Fig. 5, where the sets containing x-s are tree states and the sets containing y-s are forest states. The order in which the tree and forest states are computed is denoted by the index at their right. Observe that the input tree, which is in the regular forest language specified by G, is accepted by $A_G^{\rightarrow}$ as it stops in the state $\{y_1\}$, which is a final state of the LPA.

5. Recognizable Queries

A *recognizable k-ary query* is a pair $Q = (G, T)$ consisting of a forest grammar $G = (R, r_0)$ and a k-ary relation $T \subseteq X^k$ where X is the set of non-terminals in R. The *matches* of Q in an input forest f are given by the k-ary relation $\mathcal{M}_{Q,f} \subseteq N(f)^k$:

$$(\pi_1, \ldots, \pi_k) \in \mathcal{M}_{Q,f} \text{ iff } \exists (f, f') \in \mathcal{D}eriv_G, \exists (x_1, \ldots, x_k) \in T \text{ and } lab(f'[\pi_i]) = x_i \text{ for } i = 1, \ldots, k$$

We say that $(\pi_1, \ldots, \pi_k)$ is a match of Q in f w.r.t. to the derivation f'. We call the non-terminals in T *targets*.

For $k = 1$ and $k = 2$ we obtain unary and binary queries, respectively.

E 7. Consider the grammar G from Example 4.

The unary query $Q_1 = (G, \{x_a\})$ locates the a nodes within a tree over $\{a, b, c\}$, whose ancestors are all nodes labeled a and whose children are a node labeled b followed by a node labeled c. For the tree t depicted in Fig. 1, these are the leftmost and the rightmost a nodes. One can see that they fulfill the definition by looking at the first and the second derivation of t w.r.t. G as depicted in Fig. 2.

The binary query $Q_2 = (G, \{(x_b, x_c)\}$ locates pairs of nodes b and c having as father the same node a, and only a ancestors. The leftmost b and c in t form a match pair, as one can see by definition by looking at the first derivation. Similarly, the rightmost b and c form a match pair as defined by the second derivation w.r.t. G.

5.1 Recognizing Unary Queries

Specifying which are the subtrees of interest in a query typically consists of two conceptual parts. The contextual part constrains the surrounding context of the subtrees of interest, whereas the structural part describes the properties of the subtrees themselves.

E 8. Supposing we have an XML document which represents a conference article, where sections and subsections are encoded as XML elements, we might be interested in *subsections containing the word "automata"* occurring *in sections whose title contain the word "query"*. The two emphasized parts denote the structural and the contextual part, respectively.

E 9. Let G be the grammar in Example 4. $Q_1 = (G, \{x_b\})$ is a unary query locating the b nodes (structure) which have only a ancestors and a right c sibling (context).

Neumann and Seidl [14, 16] show how unary queries can be specified by using forest grammars and implemented by using pushdown forest automata. In the remaining of this section, we briefly review their approach.

The idea is that a grammar $G = (R, r_0)$ together with a distinguished non-terminal x of it can specify both the desired structure and context of some subtree t in a forest f. The structure is specified by the productions which can be used in order to derive t from x. The remaining productions of the grammar, which constrain the locations where x can occur in a derivation of f from r_0, capture the context part of the specification.

As argued in Section 4.1 a PA uses its forest states to remember information from the already visited part of the input. Therefore, by looking into the forest state of the PA after visiting a subtree t it should be possible to check a structural property of t as well as whether a contextual property can be satisfied considering the part of the context seen so far.

E 10. Let Q_1 be the unary query from Example 9. Consider the run of the corresponding LPA on the input as depicted in Fig. 5. One can see that by the time the automata has seen any of the b nodes, each of them fulfills the structural part (it

is a b node) and the upper-left contextual part (all ancestors are a nodes). This is reflected in the forest states of the LPA when it leaves each of the b nodes, depicted at the upper right of each of them, respectively. In each of these forest states, the NFA state y_4, which is reached after reading an x_b, denotes that a derivation of the input forest may exist in which the respective node is labeled x_b.

However, since the right part of the context has not yet been seen, the LPA can not decide at the time it leaves the b nodes whether they are indeed matches.

In order to decide whether a node is a match, in general, the remaining part of the context has to be also seen. The idea is to *remember* for each node the information collected after seeing only a part of the context and to let a second automaton proceed from the opposite direction (i.e. depth-first right-to-left traversal if the first PA does a left-to-right traversal) in order to account for the remaining context.

Pushdown Forest Automata as Relabelings

A run of a PA on an input forest f can be seen as a *relabeling* of each node in f with the triple of states involved in the transitions at that node during the run.

Consider the DLPA $A^{\vec{}}_G$ as defined in Section 4.1. Formally, the relabeling of f by $A^{\vec{}}_G$ is a mapping $\vec{\alpha} : N(f) \to Q \times P \times Q$, $\vec{\alpha}(\pi i) = (\vec{q}_{\pi(i-1)}, \vec{p}_{\pi i}, \vec{q}_{\pi i})$, where, for the node πi, $\vec{q}_{\pi(i-1)}$, $\vec{p}_{\pi i}$ and $\vec{q}_{\pi i}$ are the forest state in which the node is reached, the tree state synthesized for the node and the forest state in which the node is left respectively, by $A^{\vec{}}_G$, i.e.:

$$\begin{aligned}
\vec{q}_{\lambda 0} &= \vec{q}_0 \text{ (the initial state)} \\
\vec{q}_{\pi 0} &= \mathit{Down}(\vec{q}_\pi, a) \\
\vec{p}_\pi &= \mathit{Up}(\vec{q}_{\pi n}, a), \text{ if } n = \mathit{last}_f(\pi) \\
\vec{q}_{\pi i} &= \mathit{Side}(\vec{q}_{\pi(i-1)}, \vec{p}_{\pi i}, a)
\end{aligned}$$

where $a = \mathit{lab}(f[\pi])$.

Similarly, a deterministic RPA (DRPA) can be seen as a relabeling $\overleftarrow{\alpha}(\pi i) = (q_{\pi(i-1)}, p_{\pi i}, q_{\pi i})$.

In the following, given a node π, we denote by $\vec{p}_\pi$ and $\vec{q}_\pi$ the tree state synthesized for π and the forest state in which π is left by $A^{\vec{}}_G$, respectively. Given a DRPA, we denote by q_π and p_π, the forest state in which π is reached and the tree state synthesized for π by a DRPA, respectively.

5.1.1 Locating Unary Matches

The state $\vec{q}_\pi$ in which the $A^{\vec{}}_G$ DLPA leaves the node π synthesizes all the information collected after seeing the upper left context and all the content of π. Given this information, a second (DRPA) automaton $B^{\overleftarrow{}}_G$, proceeding from right to left, will have at every node the information necessary in order to decide whether the node fulfills the structural and contextual requirements of a query.

Thus, by remembering $\vec{q}_\pi$ one can locally decide at each node during a second traversal of the input by $B^{\overleftarrow{}}_G$ whether the node is a match of a query. Also, to avoid

unnecessary re-computations by $B^{\leftarrow}_G$, $\vec{p}_\pi$ is also remembered as to account for the structure information collected at π.

The automaton $B^{\leftarrow}_G$ runs thus on an *annotation* f_a of the input forest f by $A^{\rightarrow}_G$, $f_a \in \mathcal{F}_{\Sigma\times P\times Q}$, $N(f_a) = N(f)$ and $lab(f_a[\pi]) = (lab(f[\pi]), \vec{p}_\pi, \vec{q}_\pi)$ for all $\pi \in N(f)$.

The construction of $B^{\leftarrow}_G$ is similar to that of $A^{\rightarrow}_G$ but follows the NFA transitions in reverse and considers corresponding NFA final states at rightmost siblings, as the input to the NFAs is seen from the right to the left. Supplementary, $B^{\leftarrow}_G$ takes into account information collected by $A^{\rightarrow}_G$ in order to avoid considering NFA transitions which are not relevant for the acceptance. The automaton $B^{\leftarrow}_G = (2^X, 2^Y, \{F_0\}, \emptyset, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$, where Y and F_0 are as in the definition of $A^{\rightarrow}_G$, is given by:

$$\begin{aligned} Down^{\leftarrow}(q,(a,\vec{p},\vec{q})) &= \{y_2 \mid y \in q \cap \vec{q},\ (y_1,x,y) \in \delta,\ x \to a\langle r_j\rangle \text{ and } y_2 \in F_j\} \\ Up^{\leftarrow}(q,(a,\vec{p},\vec{q})) &= \vec{p} \\ Side^{\leftarrow}(q,p,(a,\vec{p},\vec{q})) &= \{y \mid (y,x,y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\} \end{aligned}$$

Note that $p_\pi = \vec{p}_\pi$ for all π. When it is clear from the context which is the label $(a, \vec{p}, \vec{q})$ at a transition we will omit this argument.

The following proposition by Neumann [14] shows how for every node π, the forest state q_π in which $B^{\leftarrow}_G$ arrives at π, containing information from the right context can be combined with the information for the rest part of the input given in the annotation $\vec{q}_\pi$ in order to find matches of a unary query. A node is a match if both the forest states in which $A^{\rightarrow}_G$ leaves the node and in which $B^{\leftarrow}_G$ arrives at the node contain an NFA state reachable after seeing a target non-terminal from T.

T 1. *Let $Q = (G,T)$ be a unary query and $f \in \mathcal{L}_G$. With $A^{\rightarrow}_G$ and $B^{\leftarrow}_G$ as above, $\pi \in \mathcal{M}_{Q,f}$ iff $y_1 \in q_\pi \cap \vec{q}_\pi$ and $(y,x,y_1) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.*

P . This theorem is proven in [14] as Theorem 7.1 using different definitions and notations, which are equivalent to those introduced in this work. □

Directly from Theorem 1 follows the corollary:

C 1. *$(f, f') \in \mathcal{D}eriv_{r_0}$ and $lab(f'[\pi]) = x$ iff $y \in q_\pi \cap \vec{q}_\pi$, $(y_1,x,y) \in \delta$ for some $y, y_1 \in Y$.*

This further implies that:

C 2. *If $(f, f') \in \mathcal{D}eriv_{r_0}$ and $lab(f'[\pi]) = x$ then $x \in p_\pi$.*

P . By Corollary 1 there are $y \in q_\pi \cap \vec{q}_\pi$, $(y_1,x,y) \in \delta$. Since $y \in \vec{q}_\pi$, it follows by the definition of *Side* that there is $(y', x_1, y) \in \delta$ and $x_1 \in p_\pi$. By the Berry-Sethi construction $x_1 = x$. □

E 11. Consider the run of $A^{\rightarrow}_G$ depicted in Fig. 5. The run of $B^{\leftarrow}_G$ on the tree annotated by the $A^{\rightarrow}_G$ is presented in Fig. 6. The order in which the tree and forest states are computed is denoted by the index at their right. Note how the rightmost *b* node is recognized as a match of the query $Q_1 = (G, x_b)$. As noted in Example 10,

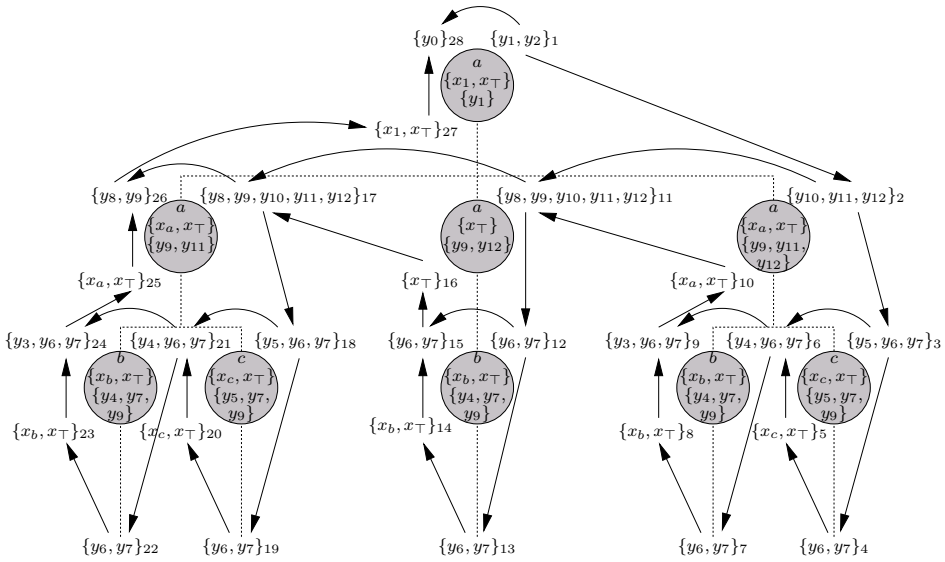


Figure 6: The run of the $B^{\leftarrow}_G$ on the input document annotated by the $A^{\rightarrow}_G$ in Example 6

y_4 in the annotation denotes the node as a potential match after accounting for the upper left context and the content. The conformance of the right context is also fulfilled as the forest state in which $B^{\leftarrow}_G$ arrives at the node contains y_4 as well. Similarly, the leftmost *b* node is a match. On the contrary, the node *b* in the middle is not a match, as its right context does not contain a *c* sibling as required by the query.

5.2 Recognizing Simple Binary Queries

In the following we present the central contribution of this work.

Let $Q = (G, B)$ be a binary query. For convenience, we will first assume that $B = \{(x^1, x^2)\}$ for some $x^1, x^2 \in X$, where X is the set of non-terminals from G. We call such a query a *simple binary query*. In this section we show how simple binary queries can be implemented. In the next section we show how the approach works for general binary queries.

According to the definition, a pair (π_1, π_2) is a match for an input f iff there is a derivation f' of f w.r.t. G and $f'[\pi_1] = x^1$, $f'[\pi_2] = x^2$.

Observe that this implies that π_1 and π_2 are matches of the unary queries (G, x^1) and (G, x^2), respectively. Therewith, (π_1, π_2) is a binary match for Q iff:

(p) π_1 is a match of the unary query (G, x^1) and

(s) π_2 is a match of the unary query (G, x^2) and

(r) π_1 and π_2 are unary matches w.r.t. the same derivation f'.

We call the nodes fulfilling (p) and (s) *primary* and *secondary* matches, or, for short, *primaries* and *secondaries*, respectively.

We have already seen how unary matches can be located. Thus, testing (p) and (s) can be done by an automata construction as in Section 5.1. In order to implement binary queries, however, one must supplementary be able to test (r).

5.2.1 Construction

In the following we show that, similarly as in the case of unary queries, binary queries can be efficiently answered by using a run of an DLPA $A^{\rightarrow}_G$ followed by a run of a DRPA $B^{\leftarrow}_G$. $A^{\rightarrow}_G$ and $B^{\leftarrow}_G$ are defined exactly as in Section 5.1. Primary and secondary matches can be thus recognized in the same way as in Section 5.1 and we keep the same notations as there.

In order to locate binary matches, we have to remember during the run of $B^{\leftarrow}_G$ which of the already visited nodes are primary or secondary matches, as potential components of binary matches. We accumulate these primaries and secondaries in set attributes l_1 and l_2, respectively, with which we equip each element of the tree and forest states of $B^{\leftarrow}_G$.

For a tree state p at node π and $x \in p$, $x.l_1$ contains primary matches and $x.l_2$ secondary matches which are found below π and are defined w.r.t. derivations which label $f[\pi]$ with x.

Similarly, for a forest state q at node π and $y \in q$, $y.l_1$ contains primary and $y.l_2$ secondary matches collected from the already visited right-sibling subtrees of $f[\pi]$. These are the matches defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by an NFA reaching the current location in state y.

Similarly to attribute grammars, the values of the l_1 and l_2 attributes are defined by a set of local rules, as it follows:

- For the elements of a forest state in which $B^{\leftarrow}_G$ arrives at a node π which has no right-siblings, the sets of primaries and secondaries collected from the right sibling subtrees is obviously empty. This is the case for the initial state F_0 at the root and for the states obtained by executing a $Down^{\leftarrow}$ transition:

$$\text{If } y \in F_0 \text{ or } y \in Down^{\leftarrow}(q,(a,\vec{p},\vec{q})) \text{ then } y.l_1 = \emptyset,\ y.l_2 = \emptyset$$

- After finishing visiting the children of a node π, the sets of primaries and secondaries found below π are propagated and possibly updated with π if π is a primary or secondary match, respectively:

$$\text{If } x \in Up^{\leftarrow}(q,(a,\vec{p},\vec{q})) \text{ then}$$

$$x.l_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \rightarrow a\langle r_j\rangle\}, & \text{if } x = x^1 \\ \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \rightarrow a\langle r_j\rangle\} & \text{, otherwise} \end{cases}$$

$$x.l_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \rightarrow a\langle r_j\rangle\}, & \text{if } x = x^2 \\ \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \rightarrow a\langle r_j\rangle\} & \text{, otherwise} \end{cases}$$

- At side transitions over a node π, the list of primaries and secondaries found so far are obtained by combining the matches below π with the matches from

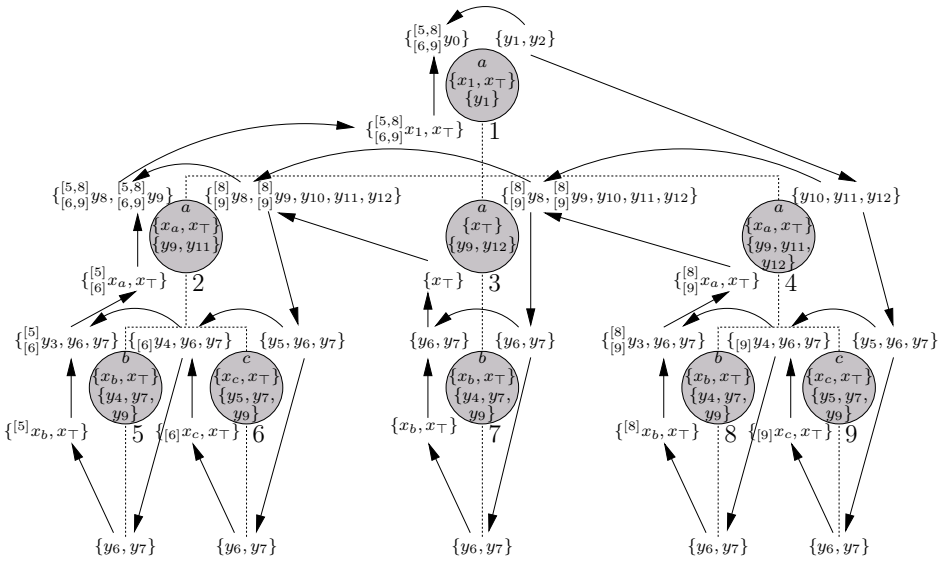


Figure 7: Evaluation of the l_1 and l_2 attributes

the already visited part to the right:

$$\text{If } y \in \mathit{Side}^{\leftarrow}(q, p, (a, \vec{p}, \vec{q})) \text{ then}$$

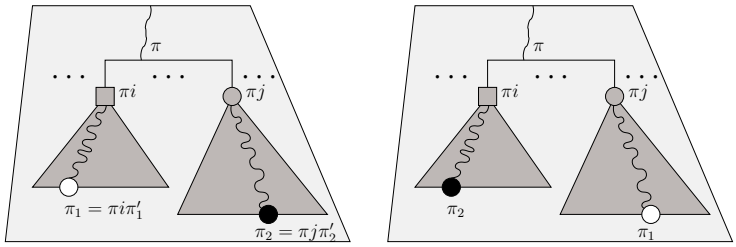
$$y.l_1 = \bigcup\{y_1.l_1 \cup x.l_1 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$
$$y.l_2 = \bigcup\{y_1.l_2 \cup x.l_2 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$

Note that the rules allow a bottom-up, right-to-left evaluation of the attributes. Therefore, they can be evaluated directly along the run of $B^{\leftarrow}_G$, which does a depth-first, right-to-left traversal. Moreover, the information used for the evaluation of attributes at a node π is the same as the information needed to compute the transitions at π. In our practical implementation (see Section 6), where transitions are computed as they are needed during the run of $B^{\leftarrow}_G$, the attributes can be thus computed at minimal costs.

E 12. Consider the binary query $Q_2 = (G, \{(x_b, x_c)\}$ from Example 7 on the input document in XML Example 4. Fig. 7 depicts how the l_1 and l_2 attributes are computed along the run of $B^{\leftarrow}_G$ on the input annotated by the run of $A^{\rightarrow}_G$. The order of computation is the same as in Fig. 6. Note that nodes are identified by ordinal numbers rather than by paths in order to increase readability. The attributes l_1, l_2 for an element x are depicted as $\begin{smallmatrix} l_1 \\ l_2 \end{smallmatrix} x$. Attributes with value $\emptyset$ are omitted.

5.2.2 Locating Binary Matches

Fig. 8 (a) and (b), and Fig. 9 (c), (d) and (e) show all possible relative positions of the primary (depicted in white) and the secondary component (depicted in black)



(a) (b)

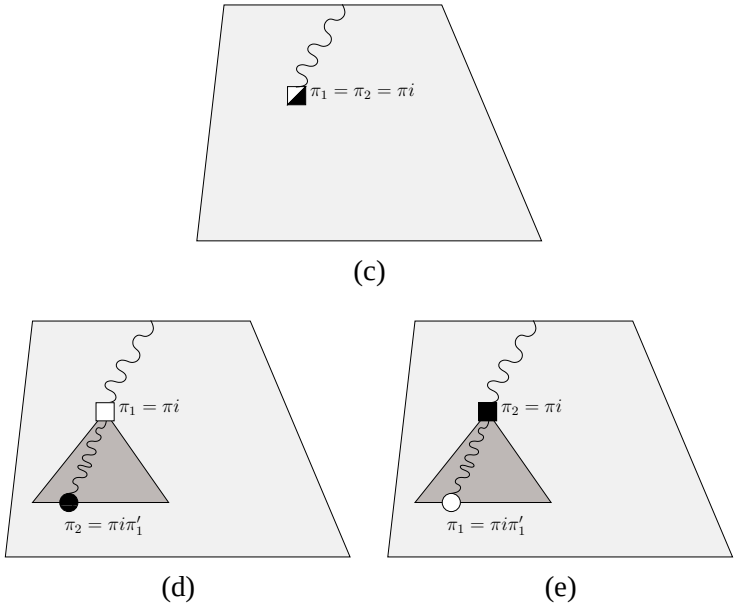
Figure 8: Relative positions of matches: π is least common ancestor or λ

of one binary match (π_1, π_2). In all five situations, due to the construction above, π_1 and π_2 belong to the attributes of one of the tree state $p_{\pi i}$ or forest state $q_{\pi i}$ in which the automaton reaches node πi (depicted by a square). This is where the binary match (π_1, π_2) will be detected at the $Side^{\leftarrow}(q_{\pi i}, p_{\pi i})$ transition.

To see how, we need to observe that our construction ensures the following invariants:

(i_1) A node π_1 belongs to the l_1 or l_2 attribute of an element x of a tree state computed for a node πi iff π_1 is below πi and there is a derivation of the input forest which labels πi with x and π_1 with x^1 or x^2, respectively.

(i_2) A node π_2 belongs to the l_1 or l_2 attribute of an element y of a forest state



(c)

(d) (e)

Figure 9: Relative positions of matches: equal, or one is a proper ancestor of the other

in which $B^{\leftarrow}_G$ arrives at a node πi iff pth_2 is in some right sibling subtree and there is a derivation of the input forest which labels πi with x, the label of the NFA transitions coming into y, and π_2 with x^1 or x^2, respectively.

This is formally expressed by the following theorem in which the involved nodes are named as in Fig. 8 (a) (or (b)):

T 2.

(i_1) *If* $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ *for some* y', x *then*
$\pi_1 \in x.l_1$ *(or* $\pi_1 \in x.l_2$*)* *iff*
$\pi_1 = \pi i \pi'_1$, $\exists f_1$ *s.t.* $(f, f_1) \in \mathcal{D}eriv_{r_0}$, $lab(f_1[\pi i]) = x$ *and*
$lab(f_1[\pi_1]) = x^1$ *(or* $lab(f_1[\pi_1]) = x^2$*, respectively).*

(i_2) $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ *and* $\pi_2 \in y.l_2$ *(or* $\pi_2 \in y.l_1$*)* *iff*
$\pi_2 = \pi j \pi'_2$, $j > i$, $\exists f_2$ *s.t.* $(f, f_2) \in \mathcal{D}eriv_{r_0}$, $lab(f_2[\pi i]) = x$ *and*
$lab(f_2[\pi_2]) = x^2$ *(or* $lab(f_2[\pi_2]) = x^1$*, respectively)*

P . The proof is given in Appendix Appendix B. □

Let $x \in p_{\pi i}$, $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $(y', x, y) \in \delta$. Let $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$. It is easy to see that (i_1) directly implies (p) and (i_2) implies (s). Less obvious but still true is that (i_1) and (i_2) also imply (r). It results that every pair formed with $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$ is a binary match.

To see why (i_1) and (i_2) imply (r), let us define a function which given a forest f, a node π and a tree t constructs a forest f_1 by replacing in f the subtree located at π with t, formally $f_1 = f/^{\pi} t$ where:

$$(t_1 \dots t_i \dots t_n)/^{i} t = t_1 \dots t \dots t_n$$
$$(t_1 \dots t_i \dots t_n)/^{i\pi} t = t_1 \dots a\langle f/^{\pi} t\rangle \dots t_n, \text{ if } t_i = a\langle f\rangle$$

If $f_1 = f/^{\pi} t$, we say that f_1 is obtained by *grafting* t into f at π.

The following theorem observes that given two derivations of a forest f which label a node π with the same symbol, a new derivation can be obtained by doing a relabeling of f in which the nodes below π are labeled as in one of the derivations and the rest of nodes as in the other.

T 3. *If* $(f, f_1) \in \mathcal{D}eriv_r$, $(f, f_2) \in \mathcal{D}eriv_r$ *and* $lab(f_1[\pi]) = lab(f_2[\pi])$ *then* $(f, f_1/^{\pi} f_2[\pi]) \in \mathcal{D}eriv_r$ *and*

$$lab((f_1/^{\pi} f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi_1]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f_1[\pi_1]), & \text{otherwise} \end{cases}$$

P . The proof is given in Appendix Appendix A. □

With the notations of Theorem 2, let $f' = f_2/^{\pi i} f_1[\pi i]$. It follows that $(f, f') \in \mathcal{D}eriv_{r_0}$, $f'[\pi_1] = x^1$ and $f'[\pi_2] = x^2$, thus (r) also holds for (π_1, π_2). It follows that (π_1, π_2) is a binary match.

E 13. Consider the side transition at node 8 in Fig. 7. $_{[9]}y_4$ denotes that node 9 is a secondary match in the part of the tree already visited. $^{[8]}x_b$ denotes that 8 is a primary match found in the subtree 8. The fact that 8 and 9 are defined with respect to the same derivation can be seen from the fact that x_b is the label of the incoming transitions into y_4. Thus $(8, 9)$ is a binary match.

Similarly, $(5, 6)$ is detected as a match at the side transition at node 5.

Therewith, we obtain how binary matches can be detected:

(a) Every pair (π_1, π_2) with $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ is a binary match, as presented above.

(b) Completely similarly, one can show that every pair (π_1, π_2) with $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ is a binary match.

(c) If $x = x^1 = x^2$ it is easy to see in the invariant (i_1) that by definition $(\pi i, \pi i)$ is a binary match.

(d) If $x = x^1$ we also have by (i_1) that every pair $(\pi i, \pi_2)$ with $\pi_2 \in x.l_2$ is a binary match.

(e) Similarly, if $x = x^2$ we have by (i_1) that every pair $(\pi_1, \pi i)$ with $\pi_1 \in x.l_1$ is a binary match.

To see that all binary matches are detected as above, let, reciprocally, (π_1, π_2) be a binary match. If $\pi_1 = \pi i \pi_1'$ and $\pi_2 = \pi j \pi_2'$, $j > i$ then there is f', $(f, f') \in \mathcal{D}eriv_{r_0}$, $f'[\pi i \pi_1'] = x^1$ and $f'[\pi j \pi_2'] = x^2$. Let $f'[\pi i] = x$. It follows by Corollary 1 that there are $y' \in q_{\pi i} \cap \vec{q}_{\pi i}$, $(y_1', x, y') \in \delta$. By Corollary 2 we have that $x \in p_{\pi i}$. By (i_1) it follows that $\pi_1 \in x.l_1$. By (i_2) there are $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ and $\pi_2 \in y.l_2$. It follows that there is πi, $x \in p_{\pi i}$, $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $(y', x, y) \in \delta$, $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$.

Similarly, for $\pi_2 = \pi i \pi_2'$, $\pi_1 = \pi j \pi_1'$, $j > i$, or $\pi_1 = \pi_2$, or $\pi_2 = \pi_1 i \pi_2'$, or $\pi_1 = \pi_2 i \pi_1'$ we obtain the reciprocals of (b), (c), (d) or (e), respectively.

We have thus proven the following theorem:

T 4. *A pair (π_1, π_2) is a binary match iff there is $\pi \in N(f)$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y', x, y) \in \delta$ and either:*

(a) $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ *or*

(b) $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ *or*

(c) $\pi_1 = \pi_2 = \pi$, $x = x^1 = x^2$ *or*

(d) $\pi_1 = \pi$, $x = x^1$, $\pi_2 \in x.l_2$ *or*

(e) $\pi_2 = \pi$, $x = x^2$, $\pi_1 \in x.l_1$.

Complexity

Let n be the size of the input forest f, i.e. the number of nodes in f. The complexity of answering a binary query is given by the complexities of running $A^{\rightarrow}_G$ and $B^{\leftarrow}_G$, computing the l_1 and l_2 attributes and that of locating binary matches.

The automaton $A^{\rightarrow}_G$ executes at each node one *Down*, one *Side* and one *Up* transition. As one can see in the definitions of the transitions, the time cost of each of these transitions does not depend on f. The run of $A^{\rightarrow}_G$ requires thus time $O(n)$. Similarly, the run of $B^{\leftarrow}_G$ needs time $O(n)$.

The l_1 and l_2 attributes have to be computed for each component of the state obtained by a $Side^{\leftarrow}$ and $Up^{\leftarrow}$ transition. For the complexity assessment let us suppose that m is the maximum between the number of primary and secondary matches in f.

Consider now an $Up^{\leftarrow}$ transition. The set $x.l_1$ of primaries for each component is computed as the union of the sets $y.l_1$ of primaries. As the number of sets $y.l_1$ does not depend on f, and a set union can be computed in time $O(m)$, the time for computing $x.l_1$ is in $O(m)$. Similarly, $x.l_2$ is computed in time $O(m)$. As the number of elements in the computed state does not depend on f either, executing $Up^{\leftarrow}$ can be done in time $O(m)$. The sets $y.l_1$ and $y.l_2$ computed at $Side^{\leftarrow}$ transition for each component of the state are similarly computed in time $O(m)$. It follows that the attributes can be computed in time $O(n \cdot m)$.

As for the complexity of locating matches, let p be the number of binary matches in f. Note that each of the binary matches is located at exactly one of the $Side^{\leftarrow}$ transitions, namely at the $Side^{\leftarrow}$ transitions over the ancestor of one of the primary or secondary, which is a sibling of an ancestor of the other. As remembering each binary match only requires constant time, locating binary matches has the overall time cost in $O(p)$.

The total time cost of answering binary queries is thus in $O(n \cdot m + p)$. Since $p \leq m^2$ and $m \leq n$, the theoretical worst cost is in $O(n^2)$. This corresponds to the case in which every pair of nodes from f is a binary match. In practice, however, the number of primary, secondary and binary matches tend to be irrelevant as compared to the input size. In this case, the time consumed is rather linear in the input size and binary queries can be answered almost as efficiently as unary queries.

5.3 Recognizing General Binary Queries

Let $Q = (G, T)$, where $T \subseteq X^2$, be a binary query. The construction is similar to that for simple binary queries but has to keep a set attribute for each non-terminal occurring in T.

Formally, let $X_1 = \{x \mid (x, x') \in T \text{ or } (x', x) \in T\} = \{x_1, \ldots, x_n\}$.

Rather than with two attributes as in the case of simple binary queries, we equip each element of a state in which $B^{\leftarrow}_G$ visits the input with n attributes $l_1, \ldots, l_n$. The attributes l_i are computed as it follows:

- If $y \in F_0$ (the initial state of $B^{\leftarrow}_G$) or $y \in Down^{\leftarrow}(q, (a, \vec{p}, \vec{q}))$ then $x.l_i = \emptyset$

- If $x \in Up^{\leftarrow}(q,(a,\vec{p},\vec{q}))$ then

$$x.l_i = \begin{cases} \{\pi\} \cup \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x_i \\ \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} & \text{, otherwise} \end{cases}$$

- If $y \in Side^{\leftarrow}(q,p,(a,\vec{p},\vec{q}))$ then

$$y.l_i = \bigcup\{y_1.l_i \cup x.l_i \mid (y,x,y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$

for $i = 1,\ldots,n$.

Similarly as in the case of simple binary queries, matches are found at $Side^{\leftarrow}$ transitions of $B^{\leftarrow}_G$. Let $Side^{\leftarrow}(q_\pi, p_\pi)$ be such a transition and let $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y_1,x,y) \in \delta$. In order to find binary matches, one has to look for every $(x_i,x_j) \in T$ into the l_i and l_j attributes. The pairs are found similarly as in the case of simple binary matches.

T 5. *A pair (π_1,π_2) is a binary match iff there is $\pi \in N(f)$, $(x_i,x_j) \in T$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y_1,x,y) \in \delta$ and either:*

(a) $\pi_1 \in x.l_i$, $\pi_2 \in y.l_j$ *or*

(b) $\pi_1 \in x.l_j$, $\pi_2 \in y.l_i$ *or*

(c) $\pi_1 = \pi_2 = \pi$, $x = x_i = x_j$ *or*

(d) $\pi_1 = \pi$, $x = x_i$, $\pi_2 \in x.l_j$ *or*

(e) $\pi_1 = \pi$, $x = x_j$, $\pi_2 \in x.l_i$.

P . By definition, (π_1,π_2) is a binary match iff there is $(x_i,x_j) \in T$ and (π_1,π_2) is a simple binary match for $(G,(x_i,x_j))$. The proof follows immediately from Theorem 4 by noticing that the attributes l_1 and l_2 from the construction for $(G,(x_i,x_j))$ equal l_i and l_j, respectively. □

In a similar manner as in the case of simple binary queries one obtains that the complexity of answering binary queries is quadratic in the input size in the worst case and rather linear in the average case.

5.4 Recognizing k-ary Queries

In order to locate matches of a query $(G,(x_1,\ldots,x_k))$ with pushdown-automata, the construction has to keep a separate set attribute for each non-empty subset $A \subset \{x_1,\ldots,x_k\}$. The set attribute for A then contains all tuples of nodes which form a partial match corresponding to the elements in A. This is necessary because a complete match can be obtained by considering any pair of complementary partial matches. For example, for a query $(G,(x_1,x_2,x_3))$, one need to consider putting together the partial matches corresponding to $\{x_1\}$ and $\{x_2,x_3\}$, or $\{x_2\}$ and $\{x_1,x_3\}$, or $\{x_3\}$ and $\{x_1,x_2\}$, respectively. The complexity of the construction thus grows exponentially with k.

In the XML practice however many queries are expressed via XPath select patterns which conceptually are binary relations (namely, between the context node

for the evaluation of the pattern and the set of nodes selected in that context). Therefore binary queries can be satisfactory used to cover a wide range of actual XML applications.

Nevertheless, it is possible to implement k-ary queries very efficiently if one adopts a *one-match* semantics for queries. Our queries so far, have an *all-matches* semantics. That is, we considered all possible ways how a query can be answered, i.e. all possible derivations w.r.t. the given grammar, possibly yielding thus more than one match-tuple. A one-match semantics can be obtained from an all-matches semantics by supplementary specifying a *disambiguating* policy, which allows to choose one *best* match. This could be for example a *left-longest match* policy as in XDuce [11, 12], which can be implemented in our framework by always considering at the $Side^{\leftarrow}$ transition of $B^{\leftarrow}_G$ at a node π only one NFA transition (y_1, x, y) conforming to the policy. In this case, x is the label of π for the sought-after derivation. The k match nodes can be thus directly read from the annotation by the second automata, getting thus even linear time complexity.

6. Practical Implementation

The algorithms presented here for answering unary and binary queries have been successfully implemented in the fxgrep XML querying tool [15]. The efficient implementation of unary queries was presented in detail by Neumann [14]. We briefly review here a few aspects which are considered in the practical implementation in order to support efficiency and ease of use.

The pushdown automata are efficiently implemented by computing their transitions only as they are needed. Transitions which are not required for the traversal of the input are not computed. This avoids the computation of possibly exponentially large transition tables. The number of transitions that are actually computed is at most linear in the size of the input document.

However, the automata do not need to compute transitions at every node, as many transitions are repeatedly executed. The first time a transition is needed, its computed value is cached, and the cached value is simply looked up for its subsequent uses. In practice only few transitions need to be computed even for large XML documents.

Even more, information which is repeatedly used for the computation of transitions, and which does not depend on the input document can be computed by a preprocessor of the query and directly accessed when needed. For example, a transition $Down(q, a)$ is computed only when the automaton $A^{\rightarrow}_G$ arrives in forest state q at a node labeled a, and only if the transition was not already computed, using the definition:

$$Down(q, a) = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle \text{ for some } x, y_1\}$$

To do so it can use the following preprocessed information:

$$y_0 s\_for\_y\ y = \{y_{0,j} \mid (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle\} \text{ for all } y \in Y$$
$$y_0 s\_for\_a\ a = \{y_{0,j} \mid x \to a\langle r_j \rangle\} \text{ for all } a \text{ occurring in } G$$

Therewith:

$$Down(q,a) = \begin{cases} y_0 s\_for\_a\ a \cap \bigcup_{y \in q} y_0 s\_for\_y\ y, & \text{if } a \text{ occurs in } G \\ \emptyset & \text{, otherwise} \end{cases}$$

Similar information is computed by the preprocessor for supporting the other transitions of the pushdown automata.

Even though queries specified using forest grammars can be very expressive, their power is not easily exploitable by users who are not familiar with grammar formalisms. Therefore, our querying tool fxgrep allows to specify queries also by using a more intuitive pattern language. Internally, patterns are automatically translated to forest grammars.

The pattern language of fxgrep resembles in its syntax to XPath. However XPath can only express unary queries, while fxgrep can also express binary queries, Aside from that, despite their similar syntax, none of XPath or fxgrep can be subsumed in terms of expressivity by the other. XPath can express non-regular features like counting of matches, e.g `//a[42]` for the 42nd `a` node in document order, or data value comparisons, e.g. `//a[b=c]` for an element `a` having a `b` and a `c` children with the same content.

On the other side XPath can hardly, if at all, express the regular features of fxgrep. In particular, fxgrep allows a more precise specification of paths. Structural conditions for a node may be expressed by using regular expressions over the children of the node. Structural conditions are given between brackets following the node to which they refer. For example, the pattern `a[(b b)* b[c*]]` is fulfilled by an `a` element which has an odd number of `b` children, and such that the last `b` has only `c` children. Contextual conditions for a node may be specified as structural conditions for nodes lying on the path from the root to that node. For example, `//appendix[# corollary]/theorem` identifies `theorem` nodes appearing inside the `appendix` which are followed by a `corollary`. A `#` in a structural condition for a node denotes the child node where the path to the match continues. Furthermore, paths can be also specified with regular expressions. For example, `(a/)+b` identifies a `b` node, where each ancestor (at least one) is an `a` node. The unary matches of Q_1 in Example 9 are located by `(a/)+a[# c]/b`.

In order to make the specification of binary queries as simple and intuitive as possible, we provide one extra symbol `%` which may be placed anywhere inside the pattern to indicate the secondary match position. Thus, the binary query Q_2 in Example 7 can be expressed as `(a/)+a[# %c]/b`. As another example, consider the unary query `//book[(author/"escu$")]/title`. The query locates all book titles whose author's names end in `escu`. The binary query to simultaneously report the titles as above and their authors is: `//book[(%author/"escu$")]/title`.

As suggested in Section 1, we provide binary queries of fxgrep as a means of selection in the fxt rule-based XML transformation language. In previous versions of fxt, only nodes below the current node could be selected, via an fxgrep unary pattern. When the selected nodes are to be recursively processed, this ensures termination. However, when the selected nodes are to be copied into the output, only allowing them to be below the current node can be a serious limitation. We

therefore provided a sort of dynamic variables which allow nodes from the already visited part of the input to be stored and used later. This workaround, as well as the explicit navigation of the XPath select patterns used in XSLT, affect the intended declarativeness of rule-based transformation languages. In contrast, binary queries increased both the expressivity and declarativeness of our rule-based transformation language.

Another advantage of using binary queries in rule-based transformation is decoupling navigation from the transformation rules. Consider an input document in which an `author` element contains all the `book`-s written by the author. The following fxt rule produces for each author a table row containing the `name` of the author and the books written by him:

XML Example 5

```
<fxt:pat>//author[(//%book)][%name]</fxt:pat>
   <tr>
     <fxt:copyContent select="2"/>
     <fxt:copyContent select="1"/>
   </tr>
```

1 and 2 refer to the binary relations specified by the first and the second occurrence of the % symbol, respectively. Rather than pairing the primary node (the author) with every corresponding secondary (each of his books, and each of his names, respectively) as in the case of binary querying, for the purpose of selection in transformations, we pair the primary (the match node) with all its secondaries, as the nodes to be selected.

If the structure of the input document changes s.t. the books of every author follow after the `author` element, only the match pattern has to be modified to `//*[#_ %book]/author[%name]`[1] as to account for the new relation between the author and his books, in order to achieve the same transformation.

7. Related Work

There exists a number of formalisms for expressing queries on trees based on formal languages and logic. A survey on these was done by Neven and Schwentick [17]. Their expressive power is in general subsumed by the monadic second-order logic (MSO), which, in particular, is known to have exactly the same expressive power as regular tree languages. Most of the formalisms only consider the case of unary queries and the proposed evaluations mechanisms are rather theoretical solutions. Neven and Schwentick [17] show that unary queries using their logic formalism can be evaluated in linear time in the size of the input, which is also the complexity of unary querying with pushdown forest automata [16, 14]. Another formalism for expressing unary queries using tree automata was given by Frick and Grohe [7]. It is shown that this formalism is equally expressive as MSO and that their queries can be also evaluated in linear time in the size of the input.

[1] _ is a wild-card denoting an arbitrary sequence of nodes; * is here a wild-card denoting an arbitrary element.

In principle, a logical approach can be easily extended from unary to k-ary queries, by using formulas with k free variables instead of formulas with one free variable. Schwentick [20] defines a logic whose expressivity is between first order logic and MSO. It is shown that an algorithm exists which checks in linear time whether a tuple of nodes verifies a formula on some input. Answering queries using this algorithm implies generating all the k-tuples of nodes from the input, incurring $O(n^k)$ time, where n is the size of the input. This gives the evaluation of k-ary queries the $O(n^{k+1})$ complexity. In particular, binary queries can be answered thus in time $O(n^3)$, which is worse than the complexity of our algorithm.

Gottlob *et al.* [10] show that XPath queries can be evaluated in time $O(n^3)$. They further show that XPath queries without arithmetical and string operations can be evaluated in linear time in the size of the input.

There also exists a number of effective approaches to XML processing which exploit techniques from the tree-theory. XDuce extends the traditional pattern-matching from functional languages with regular expression constructs. Basically, the XDuce patterns are forest grammars. XML values can be de-constructed into their component parts by using patterns with variables. A variable in a pattern is a name for a distinguished sub-pattern and allows to individually address sequences of nodes of arbitrary length. Evaluating a pattern with k variables simultaneously binds the k variables, and can be thus seen as a k-ary query. XDuce adopts a one-match policy, which is well suited for pattern matching in a functional programming language. An *all-matches* semantics, as the one implemented by us, is however more suitable for a querying language, both as a stand-alone tool or embedded within a rule-based transformation language. Nevertheless, as mentioned in Section 5.4, a one-match semantics can be efficiently implemented using pushdown automata.

XDuce focuses on static type-checking and does not provide any efficient algorithm for pattern-matching evaluation, other than naive backtracking. CDuce [1] is based on XDuce and improves its pattern matching evaluation by an implementation based on a combination of top-down and bottom-up tree automata [8] similar to the pushdown forest automata and optimized to take static type information into account.

We have already mentioned in Section 1 how binary queries could be used in a rule based-language like XSLT. Namely, any select queries relative to the dynamic current context can be collected into one binary query whose evaluation can be performed statically, i.e., preceding the transformation of the input document.

A similar usage pattern can be encountered in other cases also. XSLT keys contribute one special case of binary matches. Basically, a key is a pair consisting of the node which has the key and the value of the key (a string). The node is identified using a match pattern, while the value is given by a select pattern evaluated in the context of the node. Thus, binary queries could be also used to implement XSLT keys.

The latest drafts of XPath [24] and XQuery provide a `for` and a `FLWOR` expression, respectively, which allow variables to be bound to nodes which are matches of unary queries. These nodes can be used in the scope of the expressions as context for evaluation of further unary queries. This use of `for` expressions also qualifies

for an implementation which uses binary queries to subsume two unary queries.

8. Conclusions and Future Work

We have introduced forest grammars as a method for specifying queries of arbitrary arities in document trees. We have reviewed how unary queries can be implemented by pushdown forest automata and shown how the automata construction can be extended in order to implement k-ary queries. In particular, we have shown that binary queries can be efficiently implemented and proven that our algorithm is correct.

We have shortly discussed how the algorithm has been implemented in the XML querying tool fxgrep. We have suggested how binary queries can be used as a means of navigation in XML transformation languages and presented the advantages of binary queries over unary select patterns. We have illustrated how we effectively made use of this in the XML transformation tool fxt. Finally, we have mentioned how binary queries could be used in other settings for XML querying and transforming.

For the future it is interesting to study how k-ary and binary queries can be systematically used to implement the constructs provided in the well established XML processing languages. Also, the idea of accumulating potential matches and eventually reporting or dropping them, as enough relevant input is seen, is very useful for one-pass querying, as our recent, not yet published work has proven. One-pass querying allows to find matches without building the document tree in memory, which can be prohibitively expensive for very large documents. Furthermore, it is challenging to investigate how this ideas may support one-pass document transformations.

Appendix A. Proof of Theorem 3

We start by showing that if a derivation f' of a forest f labels a node π with x, then the trees $f[\pi]$ and $f'[\pi]$ are in the derivation relation $\mathcal{D}eriv_x$.

L 1. *If* $(f, f') \in \mathcal{D}eriv_r$ *and* $lab(f'[\pi]) = x$ *then* $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_x$.

P . The proof is by induction on the length of π.

Let $\pi = i$ and $last_f(\lambda) = n$. Thus $f = f[1] \dots f[n]$ and $f' = f'[1] \dots f'[n]$. From the definition of $\mathcal{D}eriv_r$ it follows that there is some $x_1 \dots x_n \in [\![r]\!]_R$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$.

Now let $\pi = \pi_1 i$, $last_f(\pi_1) = n$ and let $lab(f[\pi_1]) = a$, $lab(f'[\pi_1]) = x'$. By the induction hypothesis, $(f[\pi_1], f'[\pi_1]) \in \mathcal{D}eriv_{x'}$. By the definition of $\mathcal{D}eriv_{x'}$ there is some $x' \to a\langle r_1 \rangle \in R$ with $(f[\pi_1 1] \dots f[\pi_1 n], f'[\pi_1 1] \dots f'[\pi_1 n]) \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{r_1}$ there is some $x_1 \dots x_n \in [\![r_1]\!]_R$ with $(f[\pi_1 k], f'[\pi_1 k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[\pi_1 i], f'[\pi_1 i]) \in \mathcal{D}eriv_{x_i}$.

In either case, from $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_{x_i}$ it follows by the definition of $\mathcal{D}eriv_{x_i}$ that $x_i = lab(f'[\pi]) = x$. □

In the following we show that if a derivation f' of a forest f labels a node π with x, and there is a derivation t' of the tree $f[\pi]$ from the same x, then we obtain another derivation of f' by grafting t' into f' at π.

L 2. *Assume* $(f, f') \in \mathcal{D}eriv_r$, $lab(f'[\pi]) = x$ *and* $(f[\pi], t') \in \mathcal{D}eriv_x$. *Then* $(f, f'/^{\pi} t') \in \mathcal{D}eriv_r$.

P . The proof is by induction on the length of π.

If $\pi = i$ then let $f = t_1 \dots t_n$ and let $(t_1 \dots t_i \dots t_n, t'_1 \dots t'_i \dots t'_n) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]_R$ with $(t_k, t'_k) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. Since $t'_i = f'[i] = x\langle\_\rangle$ it follows that $x_i = x$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ we have that $(t_1 \dots t_i \dots t_n, t'_1 \dots t' \dots t'_n) \in \mathcal{D}eriv_r$ which is $(f, f'/^{i} t') \in \mathcal{D}eriv_r$.

If $\pi = ij\pi_1$ we have that $(f[1] \dots f[i] \dots f[n], f'[1] \dots f'[i] \dots f'[n]) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]_R$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ it follows that $f[i] = a\langle f_1\rangle$, $f'[i] = x_i\langle f'_1\rangle$ and there is $x_i \to a\langle r_1\rangle \in R$ and $(f_1, f'_1) \in \mathcal{D}eriv_{r_1}$. As $f_1[j\pi_1] = f[ij\pi_1]$ and $f'_1[j\pi_1] = f'[ij\pi_1]$ we have that $(f_1[j\pi_1], t') \in \mathcal{D}eriv_x$ and $f'_1[j\pi_1] = x\langle\_\rangle$. It follows by the induction hypothesis that $(f_1, f'_1/^{j\pi_1} t') \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{x_i}$, $(a\langle f_1\rangle, x_i\langle f'_1/^{j\pi_1} t'\rangle) \in \mathcal{D}eriv_{x_i}$ which is $(f[i], x_i\langle f'_1/^{j\pi_1} t'\rangle) \in \mathcal{D}eriv_{x_i}$. Therewith, $(f[1] \dots f[i] \dots f[n], f'[1] \dots x_i\langle f'_1/^{j\pi_1} t'\rangle \dots f'[n]) \in \mathcal{D}eriv_r$ which is $(f, f'/^{ij\pi_1} t') \in \mathcal{D}eriv_r$. □

Now we show that the forest obtained by grafting t into f at π has the nodes below π labeled as in t and all other nodes as in f.

L 3.

$$lab((f/^{\pi} t)[\pi_1]) = \begin{cases} lab(t[1\pi_2]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1]), & \text{otherwise} \end{cases}$$

P . First, observe the definition of the subtree located in a grafted forest:

$$(f/^{i\pi_1} t)[j\pi_2] = \begin{cases} f[j\pi_2] & , \text{ if } i \neq j \\ t[1\pi_2] & , \text{ if } i = j, \pi_1 = \lambda \\ a\langle f_1/^{\pi_1} t\rangle & , \text{ if } i = j, \pi_1 \neq \lambda, \pi_2 = \lambda, f[i] = a\langle f_1\rangle \\ (f_1/^{\pi_1} t)[\pi_2], & \text{if } i = j, \pi_1 \neq \lambda, \pi_2 \neq \lambda, f[i] = a\langle f_1\rangle \end{cases}$$

The proof is by induction on the length of π.

If $\pi = i$ then if $\pi_1 = i\pi_2$, $(f/^{i} t)[\pi_1] = t[1\pi_2]$ thus $lab((f/^{i} t)[\pi_1]) = lab(t[1\pi_2])$. If $\pi_1 = j\pi_2$, $j \neq i$ then $(f/^{i} t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{i} t)[\pi_1]) = lab(f[\pi_1])$.

We consider now the case where $\pi = i\pi'$, $\pi' \neq \lambda$.

If $\pi_1 = i\pi_2$ then $(f/^{\pi} t)[\pi_1] = (f_1/^{\pi'} t)[\pi_2]$, where f[i]=$a\langle f_1\rangle$. If $\pi_1 = \pi\pi_3$, i.e. if $i\pi_2 = i\pi'\pi_3$, $\pi_2 = \pi'\pi_3$ then by the induction hypothesis $lab((f_1/^{\pi'} t)[\pi_2]) = lab(t[1\pi_3])$. Thus $lab(f/^{i\pi'} t)[i\pi_2]) = lab(t[1\pi_3])$ and therewith we obtain that $lab(f/^{\pi} t[\pi\pi_3]) = lab(t[1\pi_3])$ as required.

Otherwise, also by the induction hypothesis $lab((f_1/^{\pi'} t)[\pi_2]) = lab(f_1[\pi_2])$. Since $f_1[\pi_2] = f[i\pi_2] = f[\pi_1]$ it follows that $lab((f/^{\pi} t)[\pi_1]) = lab(f[\pi_1])$.

If $\pi_1 = j\pi_2$ and $j \neq i$ then $(f/^{\pi} t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{\pi} t)[\pi_1]) = lab(f[\pi_1])$. □

Using the lemmas above we prove now Theorem 3.

Let $lab(f_1[\pi]) = lab(f_2[\pi]) = x$. By Lemma 1 we have that $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_x$. From Lemma 2 it follows that $(f, f_1/^{\pi} f_2[\pi]) \in \mathcal{D}eriv_r$. By Lemma 3:

$$lab((f_1/^{\pi} f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi][1\pi_2]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1]) \quad , & \text{otherwise} \end{cases}$$

With $f_2[\pi][1\pi_2] = f_2[\pi\pi_2]$ we obtain now the result of our theorem.

Appendix B. Proof of Theorem 2

We start by showing that the nodes collected in the attributes of a tree state at π are from the subtree located at π.

L 4. *If* $x \in p_\pi$, $\pi_1 \in x.l_1$ *then* $\pi_1 = \pi\pi'$.

P . The proof is by induction on the height of $f[\pi]$.

If $f[\pi] = a\langle\varepsilon\rangle$ then $p_\pi = Up^{\leftarrow}(Down^{\leftarrow}(q_\pi, a), a)$. By the definition of $Down^{\leftarrow}$, $Up^{\leftarrow}$ and attributes it follows that $\pi_1 = \pi$.

Otherwise, by the definition of attributes we have that $\pi_1 = \pi$ or there is $y \in q_{\pi 0}$, $y = y_{0,j}$, $x \rightarrow a\langle r_j\rangle$ and $\pi_1 \in y.l_1$. From $\pi_1 \in y.l_1$ it follows by straightforward induction on $n = last_f(\pi)$ that there is $x_1 \in p_{\pi i}$ and $\pi_1 \in x_1.l_1$. By the induction hypothesis it follows that $\pi_1 = \pi i\pi'$. □

Appendix B.1 Proof of (i_1)

Let $\pi' = \pi i$ and $n = last_f(\pi')$.

Left-to-right: From $\pi_1 \in x.l_1$ it follows by Lemma 4 that $\pi_1 = \pi'\pi'_1$. In the following we do the proof by induction on the length of π'_1.

If $\pi'_1 = \lambda$ then $\pi_1 = \pi'$ and by the definition of attributes it follows that $x = x^1$. Our conclusion follows now by Theorem 1.

If $\pi'_1 = l\pi''_1$ then $l \leq n$. By Theorem 1 there is f_a s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$ and $lab(f_a[\pi']) = x$. From $\pi_1 \in x.l_1$ and $\pi' \neq \pi_1$ it follows by the definition of attributes that there is $x \rightarrow a\langle r_h\rangle$, $y_{0,h} \in q_{\pi' 0}$ and $\pi_1 \in y_0.l_1$. By the definition of attributes it follows by straightforward induction on n that there is m, $0 < m \leq n$ and $x_1, \ldots, x_m$, $y_1, \ldots, y_m$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \vec{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = 1, \ldots, m$ and $\pi_1 \in x_m.l_1$. By Lemma 4 $m = l$. By the induction hypothesis it follows that there is f_c s.t. $(f, f_c) \in \mathcal{D}eriv_{r_0}$, $lab(f_c[\pi' l]) = x_l$ and $lab(f_c[\pi_1]) = x^1$.

From $y_l \in q_{\pi' l} \cap \vec{q}_{\pi' l}$ it follows from the definition of $Side^{\leftarrow}$ by straightforward induction on n that there are $x_l, \ldots, x_n, y_l, \ldots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \vec{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = m+1, \ldots, n$. Also by the definitions of $Down^{\leftarrow}$ and $Up^{\leftarrow}$ $y_0 = y_{0,h}$ and $y_n \in F_p$. As NFA transitions are done only inside one NFA we have that $p = h$ and it follows that $x_1, \ldots, x_n \in [\![r_h]\!]_R$.

By Theorem 1 there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi' k]) = x_k$ for all k, and by Lemma 1, $(f[\pi' k], f_k[\pi' k]) \in \mathcal{D}eriv_{x_k}$. Thus $(f[\pi' 1] \dots f[\pi' n], f_1[\pi' 1] \dots f_n[\pi' n]) \in \mathcal{D}eriv_{r_h}$ and with $x_1 \dots x_n \in [\![r_h]\!]_{\mathcal{R}}$, $(f[\pi'], x\langle f_1[\pi' 1] \dots f_n[\pi' n]\rangle \in \mathcal{D}eriv_x$. Let $t = x\langle f_1[\pi' 1] \dots f_n[\pi' n]\rangle$ and let $f_b = f_a/^{\pi}\, ' t$. By Lemma 2, $(f, f_b) \in \mathcal{D}eriv_{r_0}$, $lab(f_b[\pi']) = x$, $lab(f_b[\pi' l]) = x_l$.

Let $f_d = f_b/^{\pi' l}\, f_c[\pi' l]$. By Theorem 3 we now have that $(f, f_d) \in \mathcal{D}eriv_{r_0}$, $lab(f_d[\pi']) = x$ and $lab(f_d[\pi_1]) = x^1$.

Right-to-left: The proof is by induction on the length of π'_1.

If $\pi'_1 = \lambda$ it follows that $x = x^1$ and by the definition of attributes $\pi_1 \in x.l_1$.

If $\pi'_1 = l\pi''_1$ then $l \leq n$ and let $x_k = lab(f_1[\pi' k])$ for $k = 1, \dots, n$. By Corollary 2, $x_k \in p_{\pi' k}$. By Lemma 1 $(f[\pi'], f_1[\pi']) \in \mathcal{D}eriv_x$ and by the definition of $\mathcal{D}eriv_x$ we have that there is $x \to lab(f[\pi'])\langle r_h\rangle$ and $x_1 \dots x_n \in [\![r_h]\!]_{\mathcal{R}}$. Thus there are $y_0, \dots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \dots, n$, $y_0 = y_{0,h}$ and $y_n \in F_h$. Also, by hypothesis there are $y \in q_{\pi'} \cap \vec{q}_{\pi'}$ and y' s.t. $(y', x, y) \in \delta$. Therewith, one can show by using the definition of *Down*, *Side*, and $Down^{\leftarrow}$, $Side^{\leftarrow}$ that for $k = 0, \dots, n$, $y_k \in \vec{q}_{\pi' k}$ and $y_k \in q_{\pi' k}$, respectively.

By the induction hypothesis $\pi_1 \in x_l.l_1$. By straightforward induction on l, using the definition of $Side^{\leftarrow}$ and of the attributes, it follows that $\pi_1 \in y_0.l_1$. Now by the definition of $Up^{\leftarrow}$ and of the attributes it follows that $\pi_1 \in x.l_1$.

Appendix B.2 Proof of (i_2)

Let $n = last_f(\pi)$.

Left-to-right: Let $y_i = y$.

From $\pi_2 \in y.l_2$ it follows from the definition of $Side^{\leftarrow}$ and of attributes by straightforward induction on n that there are j, $i < j \leq n$, $y_{i+1}, \dots, y_j$, $x_{i+1}, \dots, x_j$, s.t. $(y_{k-1}, x_k, y_k) \in \delta_p$ for $k = i+1, \dots, j$ with $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for all k and $\pi_2 \in x_j.l_2$. By (i_1) it follows that $\pi_2 = \pi j \pi'_2$ and there is f_a s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$, $lab(f_a[\pi j]) = x_j$ and $lab(f_a[\pi_2]) = x^2$.

From $y_i \in q_{\pi i} \cap \vec{q}_{\pi i}$ it follows from the definitions of *Side* and $Side^{\leftarrow}$ that there are $y_0, \dots, y_{i-1}$, $x_1, \dots, x_i$ s.t. $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for $k = 0, \dots, i-1$, $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \dots, i$ and $y_0 = y_{0,h}$ for some h. By the Berry-Sethi construction, since $(y', x, y_i) \in \delta$ and $(y_{i-1}, x_i, y_i) \in \delta$, it follows that $x = x_i$. Similarly, from $y_j \in q_{\pi j} \cap \vec{q}_{\pi j}$ it follows that there are $y_j, \dots, y_n$ s.t. $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for $k = j, \dots, n$, $(y_{k-1}, x_k, y_k) \in \delta_g$ for $k = j+1, \dots n$ and $y_n \in F_g$ for some g. Because transitions in δ can be made only inside the same NFA we have that $p = g = h$. We further get that $x_1 \dots x_n \in [\![r_h]\!]_{\mathcal{R}}$.

By Theorem 1 it follows that there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi k]) = x_k$ and by Lemma 1 $(f[\pi k], f_k[\pi k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. Let the forest $f_b = f_1[\pi 1] \dots f_n[\pi n]$. It follows that $(f[\pi 1] \dots f[\pi n], f_b) \in \mathcal{D}eriv_{r_h}$. Let $f_c = f_b/^{j}\, f_a[\pi j]$. By Lemma 1 $(f[\pi j], f_a[\pi j]) \in \mathcal{D}eriv_{x_j}$ and by Lemma 3 we have that $(f[\pi 1] \dots f[\pi n], f_c) \in \mathcal{D}eriv_{r_h}$, $lab(f_c[i]) = lab(f_b[i]) = x_i = x$ and $lab(f_c[j\pi'_2]) = lab(f_a[\pi_2]) = x^2$.

Now, if $\pi = \lambda$ then $h = 0$ and $f = f[\pi 1] \dots f[\pi n]$. As above $(f, f_c) \in \mathcal{D}eriv_{r_0}$ with the required properties.

If $\pi \neq \lambda$ then by the definition of $Down^{\leftarrow}$ there are $y'' \in q_\pi \cap \vec{q}_\pi$, $(y''', x', y'') \in \delta$, $x' \to a\langle r_h \rangle$. By Theorem 1 there is f_d s.t. $(f, f_d) \in \mathcal{D}eriv_{r_0}$ and $lab(f_d[\pi]) = x'$. Let $t = x'\langle f_c \rangle$. We have that $(f[\pi], t) \in \mathcal{D}eriv_{x'}$. Let $f_e = f_d/^\pi t$. By Lemma 2 we have that $(f, f_e) \in \mathcal{D}eriv_{r_0}$ with the required properties.

Right-to-left: Let $x_k = lab(f_2[\pi k])$ for $k = 1, \ldots, n$. By (i_1) $\pi_2 \in x_j.l_2$.

We first show that $x_1 \ldots x_n \in [\![r_h]\!]_{\mathcal{R}}$ for some h. If $\pi = \lambda$ then by the definition of $\mathcal{D}eriv_{r_0}$ it follows that $x_1 \ldots x_n \in [\![r_0]\!]_{\mathcal{R}}$. If $\pi \neq \lambda$ let $lab(f_2[\pi]) = x'$. It follows by Theorem 1 that there is $(y''', x', y'') \in \delta$ and $y'' \in q_\pi \cap \vec{q}_\pi$. By Lemma 1 $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_{x'}$. By the definitions of $\mathcal{D}eriv_{x'}$ there is $x' \to a\langle r_h \rangle$ and $x_1 \ldots x_n \in [\![r_h]\!]_{\mathcal{R}}$.

There are thus $y_0, \ldots, y_n$ s.t. $y_0 = y_{0,h}$, $y_n \in F_h$ and $(y_{k-1}, x_k, y_k) \in \delta_h$ for all k. From the definitions of transitions it follows that $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$.

By Corollary 2, $x_k \in p_{\pi k}$. From $\pi_2 \in x_j.l_2$ it follows by the definitions of attributes by straightforward induction on j that $\pi_2 \in y_i.l_2$. With $y = y_i$ we get the desired result.

References

[1] B , V , C , G , F , A . 2003. CDuce: An XML-centric General-purpose Language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. ACM Press, 51–63.

[2] B , A . 2004. fxt - The Functional XML Transformer. http://www2.informatik.tu-muenchen.de/~berlea/Fxt/.

[3] B , A S , H . 2002. Binary Queries. In *Extreme Markup Languages 2002*.

[4] B , G´ S , R . 1986. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science Journal 48*, 117–126.

[5] B ¨ -K , A , M , M , W , D . 2001. Regular Tree and Regular Hedge Languages over Non-Ranked Alphabets. Research report, HKUST Theoretical Computer Science Center.

[6] F , M , S ´ , J , W , P . 1999. XML Query Languages: Experiences and Exemplars. Draft manuscript: http://www.w3.org/1999/09/ql/docs/xquery.html.

[7] F , M , G , M , K , C . 2003. Query Evaluation on Compressed Trees. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, 188–197.

[8] F , A . 2004. Regular Tree Language Recognition with Static Information. In *Programming Language Technologies for XML (PLAN-X) 2004*.

[9] G´ , F S , M . 1997. Tree Languages. In *Handbook of Formal Languages*, Rozenberg, Grzegorz and Salomaa, Arto, Editors. Volume 3. Springer, Heidelberg, chapter 1, 1–68.

[10] G , G , K , C , P , R . 2002. Efficient Algorithms for Processing XPath Queries. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)*. Morgan Kaufmann, Hong Kong, China, 95–106.

[11] H , H P , B C. 2000. XDuce: A Typed XML Processing Language. In *Proceedings Of The Third International Workshop on the Web and Databases (WebDB2000), Dallas, Texas*, 111–116.

[12] H , H P , B C. 2003. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech. 3*, 2, 117–148.

[13] M , M , L , D , M ., M . 2001. Taxonomy of XML Schema Languages Using Formal Language Theory. In *Extreme Markup Languages 2001, Montreal, Canada*.

[14] N , A . 2000. *Parsing and Querying XML Documents in SML*. PhD thesis, University of Trier, Trier.
[15] N , A B , A . 2004. fxgrep 4.0. http://www2.informatik.tu-muenchen.de/~berlea/Fxgrep/.
[16] N , A S , H . 1998. Locating Matches of Tree Patterns in Forests. In *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, Volume 1530 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 134–145.
[17] N , F S , T . 2002. Automata- and Logic-based Pattern Languages for Tree-structured Data. In *Semantics in Databases*, Volume 2582 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 160–178.
[18] N.K , A.M , M.I.S . 2000. DSD: A Schema Language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*.
[19] OASIS. 2001. RelaxNG Specification. http://www.relaxng.org/.
[20] S , T . 2000. On Diving into Trees. In *Proceedings of the 25-th Symposium on Mathematical Foundations of Computer Science 2000*. ACM Press, 660–669.
[21] W3C. 1999. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath.
[22] W3C. 1999. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt.
[23] W3C. 2001. XML Schema Language. http://www.w3.org/TR/xmlschema-0/.
[24] W3C. 2003. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20.
[25] W3C. 2003. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.