Schema Validation and Type Annotation for Encoded Trees

Torsten Grust Stefan Klinger
University of Konstanz
Department of Computer and Information Science
P.O. Box D 188, D-78457 Konstanz, Germany
{grust,klinger}@inf.uni-konstanz.de

ABSTRACT

We argue that efficient support for *schema validation* and *type annotation* in XQuery processors deserves as much attention as efficient evaluation techniques for XPath queries have received in the past. To this end, we describe a validation procedure that operates on an *encoding of trees* that has already been succesfully used for XPath location step evaluation. The validation algorithm works without the construction of finite state automata and can exploit properties of the encoding to speed up the validation of already partially type-annotated trees. First experiments—carried out using the database back-end of the authors' XQuery compiler—indicate that this approach can indeed lead to high-troughput validation support.

1. INTRODUCTION

Although the focus of recent research might suggest so, it is not immediately clear that the evaluation of the XPath (sub)expressions in a given query dominates the dynamic evaluation phase of XQuery. Depending on the query kind, an XQuery runtime system may devote a significant share of query evaluation time to *schema validation* and *type annotation*, resulting either from the use of `element` constructors or explicit `validate` expressions.

The evaluation of the expression `validate` e, in which subexpression e yields an element node[1] v with tag name t, involves

(1) the creation of a copy of v and all its descendants (*i.e.*, the computation of v`/descendant-or-self::node()`),
(2) a check that the structure of the subtree rooted at v matches the in-scope XML Schema definition for t, and, given that this succeeds,
(3) for all attribute and element nodes in the subtree copy, the annotation with the corresponding XML Schema type (name) found during the previous step.

The almost exact effort is required to evaluate the element constructor `element` t `{` e_1`,` ... `,`e_n `}` (in the above, v is the newly constructed node).

[1]For brevity, we omit the treatment of document nodes in this article.

Informal Proceedings of the *First International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P),* June 17-18, 2004, Paris, France.

The annotations produced in step (3) are then subject to inspection by the query, *e.g.*, via `typeswitch` or `instance of` expressions or, put more generally, through *sequence type matching*.

In principle and as remarked in the W3C XQuery specification [3], an implementation of XQuery could perform the computation described in (1)–(3) as follows: serialize node v, parse the serialization result to produce the corresponding XML information set, validate the latter as defined by the XML Schema validation rules [2], and finally map the validated information set back into the implementation's internal tree representation. Performance-wise, this is clearly infeasible.

Here, we propose to perform validation and type annotation using an encoding of XML documents (or fragments thereof) that has been designed to efficiently support the evaluation of XPath location steps, the *pre/post* plane [11, 12]. Internally, the XQuery runtime may thus operate on a *single* representation of trees without the need for mappings of any kind. Being XPath-aware, the encoding naturally supports step (1) above. Furthermore, it turns out that we can adapt the concept of *derivatives of a regular expression* [8] to implement validation and type annotation, *i.e.*, steps (2) and (3), for *pre/post* encoded trees. We will see that we can use properties of the encoding to speed up the validation of already partially annotated trees as well as to support XQuery's validation modes (`strict`, `skip`). Experiments show that this approach can indeed lead to high throughput validation and annotation support for XQuery.

2. ENCODING TREES

```
<a>
  <b>c</b>
  d
  <e>
    <f><g/><h/></f>
    <i>j</i>
  </e>
</a>
```

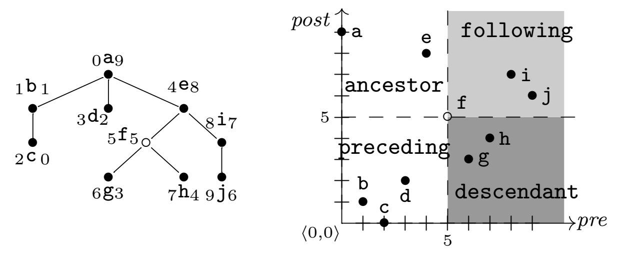
Figure 1: XML sample.

We are using an encoding of XML documents that (a) has been developed with a close eye on the XPath semantics, and (b) is suited to efficiently represent and query trees on all levels of the memory hierarchy, *i.e.*, in main memory as well as on secondary storage.

The encoding process reduces the XML document to its *skeleton tree* (the skeleton abstracts from node kinds, for example). Each node v in the skeleton is assigned its unique *preorder* and *postorder traversal ranks*, *pre*(v) and *post*(v), respectively. Figure 2(a) depicts the decorated skeleton tree of the XML document depicted in Figure 1.

According to the XPath semantics, given any context node c, the XPath axes `ancestor`, `preceding`, `following`, and `descendant` *partition* the document containing c. This property is preserved by the *pre/post* encoding. If we map the encoded nodes into a two-dimensional *pre/post plane*, the four axes coincide with disjoint rectangular regions partitioning the plane (Figure 2(b)). In the plane, a location step along these axes may be evaluated by means of a simple conjunctve range predicate, *e.g.*,

$$v \in c/\texttt{following} \Leftrightarrow pre(v) > pre(c) \land post(v) > post(c)$$
$$v \in c/\texttt{descendant} \Leftrightarrow pre(v) > pre(c) \land post(v) < post(c) \ .$$



(a) XML skeleton tree and pre/postorder ranks. (b) Resulting *pre/post* plane.

Figure 2: Encoding trees in the *pre/post* plane (context node `f`).

The *pre/post* encoding naturally leads to a tabular representation of XML documents as shown in Figure 3(a). In addition to the *pre|post* table, more two-column tables (Figure 3(a)) may be maintained to hold further node properties as defined by the XQuery data model [10]. Note that, in all tables, the nth row has a *pre* value of n. An implementation may thus omit the *pre* columns altogether, provided that the tables are maintained in an ordered data structure (*e.g.*, an in-memory array). An RDBMS-based implementation may choose to fuse the tables into a single table to avoid the need for joins and to maintain the *pre* property as a virtual column by means of a `ROW_NUMBER` function or similar. In [11–13] we have shown how XPath axis traversals lead to efficient sequential scans in this tabular XML encoding.

2.1 Validation and Type Annotation

In view of Figure 3, validating an encoded document leads to the population of an additional table *pre|type* in which the

pre	*post*	*pre*	*kind*	*pre*	*name*	*pre*	*value*
0	9	0	*elem*	0	a	0	·
1	1	1	*elem*	1	b	1	·
2	0	2	*text*	2	·	2	c
3	2	3	*text*	3	·	3	d
4	8	4	*elem*	4	e	4	·
5	5	5	*elem*	5	f	5	·
6	3	6	*elem*	6	g	6	·
7	4	7	*elem*	7	h	7	·
8	7	8	*elem*	8	i	8	·
9	6	9	*text*	9	·	9	j

(a)

pre	*type*
0	t1
1	xs:string
2	xdt:untypedAtomic
3	xdt:untypedAtomic
4	t2
5	t3
6	xs:string
7	xs:string
8	xs:string
9	xdt:untypedAtomic

(b)

Figure 3: Tabular representation of *pre/post* encoding of the XML document in Figure 1. Table (b) is populated by the validation/annotation process.

```
<xs:element name="a" type="t1"/>
<xs:complexType name="t1" mixed="true">
 <xs:sequence>
  <xs:element name="b" type="xs:string"/>
  <xs:element name="e" type="t2"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="t2">
 <xs:sequence>
  <xs:element name="f" type="t3" maxOccurs="unbounded"/>
  <xs:element name="i" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="t3">
 <xs:all>
  <xs:element name="g" type="xs:string" minOccurs="0"/>
  <xs:element name="h" type="xs:string"/>
 </xs:all>
</xs:complexType>
```

Figure 4: XML Schema description (namespace prefix `xs` bound to `http://www.w3.org/2001/XMLSchema`).

type column holds the name of the (user-defined or built-in) XML Schema [2] type assigned to each node. Initially, text nodes are annotated with the type `xdt:untypedAtomic` while not yet validated element nodes carry an `xdt:untyped` annotation. Validating the sample document of Figure 1 against the schema description of Figure 4 results in the type annotations shown in Figure 3(b). At query runtime, the type of a validated node v is thus available via a lookup in the *pre|type* table in row $pre(v)$.

In what follows, our aim is to develop an efficient validation and annotation procedure that is able—given only the *pre/post* encoding and the associated node property tables—to correctly populate the *pre|type* table.

3. DERIVING VALIDITY

Clearly, the most efficient way to access the tabular representation of a *pre/post* encoded tree is to forward scan the tables of Figure 3(a) *sequentially*, simulating a preorder traversal of the tree. Since, consequently, the validation process operates on a *sequence of encoded nodes*, we capture the gist of the type definitions contained in an XML Schema description by *regular expressions over node sequences.* In our context, regular expressions (1) provide a more tractable representation of types than the XML Schema syntax itself and (2) are also sufficiently general to express other schema languages, *e.g.*, DTDs or RelaxNG [9].

In Figure 5(a), ε matches the empty tree (*i.e.*, the empty sequence of encoded nodes) while $\varnothing$ matches no tree at all. The binary, associative operators $\_ \cdot \_$ and $\_ \mid \_$ (sequence and choice, respectively) are defined as usual. $e^{m,n}$ with $n \geqslant m \geqslant 0$, $n \geqslant 1$ matches m-to-n-fold repetitions of e ($n = *$ allows for infinitely many repetitions). Note that interleave (&) is a non-associative n-ary operator which is why we define its application to $n \geqslant 2$ arguments (such *all groups* are discussed in more detail in Section 3.3). $\mathsf{elem}\,t$, $\mathsf{attr}\,t$, and text match single nodes of the indicated kind (and name, in case of element and attribute nodes). Regular expression $\mathsf{elem}\,t\,\{e\}$ matches an element node (with tag t) whose content subtree matches e.

After schema import and translation into regular expressions, the XML Schema description of Figure 4 takes the form of Figure 5(b). Local element declarations are registered with the context in which they may be validated (the

$e ::= \varepsilon$	empty
$\mid\ \varnothing$	none
$\mid$ elem t	element node
$\mid$ attr t	attribute node
$\mid$ text	text node
$\mid\ e \cdot e$	sequence
$\mid\ e \mid e$	choice
$\mid\ e^{m,n}$	repetition
$\mid\ e \,\&\, \cdots \,\&\, e$	interleave
$\mid$ elem $t\,\{e\}$	element + content
$\mid$ attr $t\,\{e\}$	attribute + content
$\mid\ {}^{q}_{p}[\![e]\!]^{s}_{r}$	*pre/post* guard
$\mid\ id$	type name

(a) Regular expression syntax.

val. context	schema declaration
	elem a {t1}
a	elem b {xs:string}
a	elem e {t2}
a/e	elem f {t3}
a/e	elem i {xs:string}
a/e/f	elem g {xs:string}
a/e/f	elem h {xs:string}

type	type definition
xs:string →	ε | text
t1 →	xs:string · elem b {xs:string} · xs:string · elem e {t2} · xs:string
t2 →	(elem f {t3})^{1,*} · elem i {xs:string}
t3 →	(elem g {xs:string})^{0,1} & elem h {xs:string}

(b) Schema declarations and type definitions after schema import.

Figure 5: Representing XML Schema descriptions by regular expressions.

context is empty for the globally declared element a). Type definitions occur for user-defined types as well as those built into XML Schema (here: `xs:string`).

3.1 *Pre/Post* Guards

In a preorder traversal, the descendants of a node v are enumerated immediately after v. This suggests the correspondence elem $t\,\{e\} \equiv$ elem $t \cdot e$.

To see that this is not yet correct, consider the regular expression

$$\text{elem b}\,\{\text{elem c}\} \equiv \text{elem b} \cdot \text{elem c}$$

which matches element node b in both tree fragments in Figure 6 (a scan of both tree encodings yields the element node sequence a · b · c) while only the match in tree (a) would be valid. Instead, the validation procedure uses the equivalence

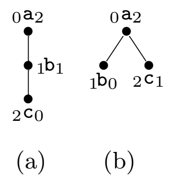


Figure 6: Tree Fragments.

$$\text{elem b}\,\{\text{elem c}\} \equiv \text{elem b} \cdot {}^{\infty}_{pre(\text{b})}[\![\text{elem c}]\!]^{post(\text{b})}_{-\infty}$$

in which $[\![$elem c$]\!]$ denotes a *pre/post guarded regular expression.* An encoded tree t matches against the guarded regular expression ${}^{q}_{p}[\![e]\!]^{s}_{r}$ if (1) t matches against e recursively, and, (2) for all nodes v in t,

$$p < pre(v) < q \quad \wedge \quad r < post(v) < s$$

(we will shortly see that the validation process solely relies on guards for which either the root of t already violates condition (2) or all nodes in t satisfy (2)—a test of all nodes is not required). In a sense, *pre/post* guards re-emboss the tree structure on the flat node sequence generated by the scan of the encoding.

3.2 Derivatives

The core of the validation procedure is based on computing *derivatives* ∂ *of regular expressions* [8]. The derivatives are regular expressions again—the procedure does *not* construct state automatons of any kind. Let $v_1 \cdot v_2 \cdots v_n$ denote a node sequence, v_0 a single node, and e a regular expression, then

$$e \text{ matches } v_0 \cdot v_1 \cdot v_2 \cdots v_n \Leftrightarrow \partial_{v_0}(e) \text{ matches } v_1 \cdot v_2 \cdots v_n \ .$$

Repeated derivation leads to the following validation procedure for a tree t encoded by the node sequence $v_0 \cdot v_1 \cdots v_n$:

$$t \text{ validates against } e \Leftrightarrow \nu(\partial_{v_n}(\cdots \partial_{v_1}(\partial_{v_0}(e)) \cdots)) \ ,$$

where predicate $\nu(e)$ (Figure 7(b)) indicates whether e matches the empty node sequence. Given a current node v, Figure 7(a) defines the derivation for the various regular expression kinds with respect to v. Note, how cases (5) and (8) use *pre/post* guards[2] to enforce that a regular expression is to be matched against the nodes in the `following` and `descendant` axes of v, respectively (cf. Figure 2(b)). Case (9) enforces the *pre/post* guard semantics.

Type annotation is carried out as a side-effect of validation. In case (8), the current element node v is annotated with the named type id $[type(v) := id]$ before the actual derivation result is returned. Here we assume that all element content types have been assigned some name id as in Figure 5(b)—anonymous XML Schema type definitions are associated with a system-generated type name during schema import.

Note how $\partial_v(e)$ yields $\varnothing$ in those cases where v cannot possibly lead to successful validation against e. Since $\varnothing$ is a fixpoint of ∂ (case (2)), validation may be aborted as soon as $\varnothing$ has been derived. In this case, the current derivative e can be used to generate an error message indicating what would have been acceptable content in place of v.

3.3 Attributes and `xs:all` Groups

Both, XML Schema attribute (`xs:attribute`, `xs:attributeGroup`) and `xs:all` declarations, yield &-groups (*all groups*) during translation into regular expression types. The permutation semantics of such groups can introduce significant complexity: naive expansion of an &-group of size n leads to (1) an |-group of size $O(n!)$, and (2) considerable non-determinism: for each e_i in the &-group, the expansion has $(n-1)!$ branches starting with e_i:

$$e_1 \,\&\, e_2 \,\&\, e_3 \equiv e_1e_2e_3 \mid e_1e_3e_2 \mid e_2e_1e_3 \mid e_2e_3e_1 \mid e_3e_1e_2 \mid e_3e_2e_1 .$$

The impact of non-determinism on the derivation is discussed in Section 5.

If we expand &-groups with more care and follow the scheme below we can get around the complexity ($m_i \in \{0,1\}$ as required by XML Schema):

$$e_1{}^{m_1,1} \,\&\, \cdots \,\&\, e_n{}^{m_n,1} \equiv \Big|_{i=1}^{n} \; e_i \cdot \left(\mathop{\&}_{\substack{j=1 \\ j \neq i}}^{n} e_j{}^{m_j,1} \right)^{(\max_{\substack{j=1 \\ j\neq i}}^{n} m_j),1} .$$

Figure 7(c) exemplifies the expansion for $n = 3$. In this scheme, all |-branches start with distinct symbols which is guaranteed by the XML Schema requirement that the e_i are of the form elem t_i with distinct t_i (a consequence of

[2]Here, and in the sequel, we omit ∞ and $-\infty$ guard limits.

$$
\begin{array}{lll}
(1) & \partial_v(\varepsilon) & = \varnothing \\
(2) & \partial_v(\varnothing) & = \varnothing \\
(3) & \partial_v(\mathsf{elem}\,t) & = \begin{cases} \varepsilon & \text{if } \mathit{kind}(v) = \mathit{elem} \wedge \mathit{name}(v) = t \\ \varnothing & \text{otherwise} \end{cases} \\
(4) & \partial_v(\mathsf{text}) & = \begin{cases} \varepsilon & \text{if } \mathit{kind}(v) = \mathit{text} \\ \varnothing & \text{otherwise} \end{cases} \\
(5) & \partial_v(e_1 \cdot e_2) & = \begin{cases} (\partial_v(e_1) \cdot {}_{\mathit{pre}(v)}[\![e_2]\!]_{\mathit{post}(v)}) \mid \partial_v(e_2) & \text{if } \nu(e) \\ \partial_v(e_1) \cdot {}_{\mathit{pre}(v)}[\![e_2]\!]_{\mathit{post}(v)} & \text{otherwise} \end{cases} \\
(6) & \partial_v(e_1 \mid e_2) & = \partial_v(e_1) \mid \partial_v(e_2) \\
(7) & \partial_v(e^{m,n}) & = \begin{cases} \partial_v(e) & \text{if } n = 1 \\ \partial_v(e) \cdot {}_{\mathit{pre}(v)}[\![e^{\mathit{max}(0,m-1),n-1}]\!]_{\mathit{post}(v)} & \text{if } n > 1 \end{cases} \\
(8) & \partial_v(\mathsf{elem}\,t\,\{\mathit{id}\}) & = \begin{cases} [\mathit{type}(v) := \mathit{id}]\; \partial_v(\mathsf{elem}\,t \cdot {}_{\mathit{pre}(v)}[\![e]\!]^{\mathit{post}(v)}) & \text{if } \mathit{id} \to e \\ \varnothing & \text{otherwise} \end{cases} \\
(9) & \partial_v({}^{q}_{p}[\![e]\!]^{s}_{r}) & = \begin{cases} {}^{q}_{p}[\![\partial_v(e)]\!]^{s}_{r} & \text{if } p < \mathit{pre}(v) < q \wedge r < \mathit{post}(v) < s \\ \varnothing & \text{otherwise} \end{cases} \\
(10) & \partial_v(\mathit{id}) & = \begin{cases} \partial_v(e) & \text{if } \mathit{id} \to e \\ \varnothing & \text{otherwise} \end{cases}
\end{array}
$$

(a) Derivative of a regular expression over encoded trees, cases for $\mathsf{attr}\,t$ analogous to $\mathsf{elem}\,t$ (in case (7): $* - 1 = *$).

$$
\begin{array}{lll}
\nu(\varepsilon) & = & \mathit{true} \\
\nu(\varnothing) & = & \mathit{false} \\
\nu(\mathsf{elem}\,t) & = & \mathit{false} \\
\nu(\mathsf{text}) & = & \mathit{false} \\
\nu(e_1 \cdot e_2) & = & \nu(e_1) \wedge \nu(e_2) \\
\nu(e_1 \mid e_2) & = & \nu(e_1) \vee \nu(e_2) \\
\nu(e^{m,n}) & = & m = 0 \vee \nu(e) \\
\nu(\mathsf{elem}\,t\,\{e\}) & = & \mathit{false} \\
\nu({}^{q}_{p}[\![e]\!]^{s}_{r}) & = & \nu(e) \\
\nu(\mathit{id}) & = & \nu(e) \quad \text{with } \mathit{id} \to e
\end{array}
$$

(b) Predicate $\nu(e)$.

$$
\begin{array}{rcl}
e_1 \,\&\, {e_2}^{0,1} \,\&\, {e_3}^{0,1} & = & e_1 \cdot ({e_2}^{0,1} \,\&\, {e_3}^{0,1})^{0,1} \\
 & & \mid e_2 \cdot (e_1 \,\&\, {e_3}^{0,1}) \\
 & & \mid e_3 \cdot (e_1 \,\&\, {e_2}^{0,1}) \\
 & = & e_1 \cdot (e_2 \cdot {e_3}^{0,1} \mid e_3 \cdot {e_2}^{0,1})^{0,1} \\
 & & \mid e_2 \cdot (e_1 \cdot {e_3}^{0,1} \mid e_3 \cdot e_1) \\
 & & \mid e_3 \cdot (e_1 \cdot {e_2}^{0,1} \mid e_2 \cdot e_1)
\end{array}
$$

(c) Modified &-group expansion.

Figure 7: Computing derivatives, predicate ν, and all group expansion.

the *unique particle attribution* constraint [2]). During validation, at most one branch does not yield $\varnothing$ when the expanded &-group is derived by the next encoded node (case (3), Figure 7(a)). Since $\varnothing \mid e = e \mid \varnothing = e$, the $\varnothing$ branches may be pruned. If the above scheme is implemented *lazily*, *i.e.*, in each derivation expansion is applied to the outermost &-group only, this enables validation against all groups in $O(n)$ steps with a regular expression of size $O(n^2)$. Note that a minimal deterministic automaton for the same &-group has $O(2^n)$ states and transitions [14].

4. SUPPORTING THE XQUERY SEMANTICS

As described here, the validation procedure is subject to simplifications due to its strict 'forward table scan' discipline. First, note that all *pre*/*post* guards introduced by ∂_v either identify the nodes in the `descendant` or `follwing` axis of the current node v and thus are of one of two forms, namely

$$
{}_{\mathit{pre}(v)}[\![e]\!]^{\mathit{post}(v)} \quad \text{or} \quad {}_{\mathit{pre}(v)}[\![e]\!]_{\mathit{post}(v)} \ .
$$

Only nodes v' with $\mathit{pre}(v) < \mathit{pre}(v')$ will be encountered during the derviation of e. This renders the guard condition $\mathit{pre}(v) < \mathit{pre}(v') < \infty$ trivially fulfilled and thus the *pre* guards obsolete: *post* guards of the form $[\![e]\!]^{s}_{r}$ suffice in our context.

4.1 Copy Semantics

Further, recall that an XQuery `element` t `{` e_1`,` ... `,`e_n `}` constructor—in addition to the validation of the newly constructed tree—generates fresh copies of the subtrees e_i, $i = 1 \ldots n$. To create the new tree

$$
\begin{tikzcd}[column sep=small]
 & & t \arrow[dll, dash] \arrow[dl, dash] \arrow[rr, dash] & & \\
e_1 & e_2 & \cdots & & e_n
\end{tikzcd}
\ ,
$$

an XQuery runtime based on *pre*/*post* encoded trees

(1) adds the representation of the new element node t to the encoding tables, appends copies of those rows representing the e_i to the individual tables, and then

(2) updates the postorder ranks in the extended $\underline{\mathit{pre}\,|\mathit{post}}$ table to ensure the consistency of the encoding.

While (1) seems unavoidable due to the inherent XQuery copy semantics, (2) has been a frequent point of critique: the update cost of the *pre*/*post* encoding appear to be significant.

A change of encoding is one possible remedy. We trade the $\underline{\mathit{pre}\,|\mathit{post}}$ table for a $\underline{\mathit{pre}\,|\mathit{size}}$ table which, for each node v, maintains the number $\mathit{size}(v)$ of nodes in the subtree below v [15]. Support for XPath location step evaluation is preserved. We have

$$
\begin{array}{rcl}
v \in c/\texttt{following} & \Leftrightarrow & \mathit{pre}(v) > \mathit{pre}(c) + \mathit{size}(c) \\
v \in c/\texttt{descendant} & \Leftrightarrow & \mathit{pre}(c) < \mathit{pre}(v) \leqslant \mathit{pre}(c) + \mathit{size}(c) \ ,
\end{array}
$$

i.e., document nodes are effectively placed in a *pre*/*pre*+*size* plane. Validation now operates with *pre*/*size* guards $[\![e]\!]^{s}_{r}$: a node v passes the guard if

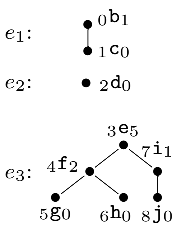
$$
r < \mathit{pre}(v) \leqslant r + s \ .
$$

To complete the shift to the *pre*/*size* encoding, cases (5) and (8) in the definition of ∂ are modified to use guards of the form

$$
[\![e_2]\!]^{\infty}_{\mathit{pre}(v)+\mathit{size}(v)} \quad \text{and} \quad [\![e]\!]^{\mathit{pre}(v)+\mathit{size}(v)}_{\mathit{pre}(v)} \ ,
$$

respectively (the former identifies the `following`, the latter the `descendant` nodes of the current node v).

Since the property $\mathit{size}(v)$ is invariant to the actual placement of v and its descendants in the containing tree, element construction in the *pre*/*size* encoding is merely a matter of



pre	*size*	*pre*	*size*	
		0	9	a
0	1	1	1	e_1
1	0	2	0	
2	0	3	0	e_2
3	5	4	5	e_3
4	2	5	2	
5	0	6	0	
6	0	7	0	
7	1	8	1	
8	0	9	0	

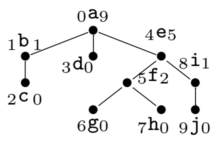


Figure 8: Constructing a new tree (right) from three fragments e_1, e_2, e_3 (left) in the *pre*/*size* encoding.

pasting fragment encodings (remember that the *pre* column is virtual). Figure 8 visualizes the evaluation of the XQuery element constructor `element a` $\{e_1, e_2, e_3\}$.

4.2 XQuery Validation Modes and Partially Validated Trees

It is typical for an XQuery expression to apply multiple element constructors in succession to build complex trees from simpler fragments. With the XML Schema description of Figure 4, suppose the XML document of Figure 1 is the result of the following query:

```
1   let $fs := for $f in fn:doc("f.xml")//f
2               return
3                 validate context a/e { $f } ,
4   return
5     element a {
6       element b {text {"c"}} ,
7       text {"d"} ,
8       element e {
9         $fs ,
10        element i {text {"j"}}
11      }
12    }
```

Expression evaluation proceeds bottom-up. When the constructor `element e` is evaluated, the runtime system has already validated the `f` element nodes and their subtrees as requested in line 3. Depending on the number of `f` element nodes in the input document `"f.xml"`, this validation effort might already be substantial. Re-validating these tree fragments when the containing constructor `element e` is evaluated (and, again, when the outer constructor `element a` is applied) surely seems wasteful.

As an alternative to re-validation, the system could also copy the relevant rows in the *pre|type* table when the subtree copies are pasted during element construction (Section 4.1). This strategy coincides with a recent proposal for a new XQuery `skip preserve` validation mode made by Don Chamberlin on the `public-qt-comments@w3.org` mailing list[3] as well as the XSLT `preserve` validation mode semantics. When the system finally initiates the validation of the newly created tree, the validation procedure will potentially encounter nodes v for which $\mathit{type}(v) \neq$ `xdt:untyped`. This implies that the nodes in the subtree below v have already received type annotations, too. Validation might as well *skip* v and its descendants.

In a runtime system based on the *pre/size* encoding, skipping is particularly easy to integrate into ∂. We redefine ∂ to return the derivative e' as well as $s \in \{0, 1, \dots\}$:

$$\partial_v(e) = \langle s, e' \rangle \quad ,$$

where s indicates how far the encoding table scan may skip ahead. The next derivative computed is $\partial_{v'}(e')$ with $\mathit{pre}(v') = \mathit{pre}(v) + 1 + s$.

Normally, $s = 0$, *i.e.*, no node is skipped. If the evaluation encounters an annotated element node v whose annotation $\mathit{type}(v)$ equals the expected type id, ∂ returns $s = \mathit{size}(v)$ and validation for the subtree below v is skipped:

$$\partial_v(\mathsf{elem}\, t\, \{\mathit{id}\}) =$$
$$\begin{cases} \langle \mathit{size}(v), \varepsilon \rangle & \text{if } \mathit{type}(v) = \mathit{id} \\ [\mathit{type}(v) := \mathit{id}]\, \langle 0, \partial_v(\mathsf{elem}\, t \cdot [\![e]\!]^{\mathit{pre}(v)+\mathit{size}(v)}_{\mathit{pre}(v)}) \rangle & \text{if } \mathit{id} \rightarrow e \\ \langle 0, \varnothing \rangle & \text{otherwise} \end{cases}$$

[3]`http://lists.w3.org/Archives/Public/public-qt-comments/2004Feb/0222.html`.

$$\begin{aligned} \varnothing \,|\, e &= e & \qquad e \cdot \varepsilon &= e \\ e \,|\, \varnothing &= e & \varepsilon \cdot e &= e \\ e \cdot \varnothing &= \varnothing & \varepsilon^{m,n} &= \varepsilon \\ \varnothing \cdot e &= \varnothing & [\![\varepsilon]\!]^s_r &= \varepsilon \\ \varnothing^{m,n} &= \varnothing & [\![\, [\![e]\!]^q_p]\!]^s_r &= [\![e]\!]^{\mathit{min}(q,s)}_{\mathit{max}(p,r)} \\ [\![\varnothing]\!]^s_r &= \varnothing & & \end{aligned}$$

Figure 9: Equivalent regular expression types ($[\![e]\!]$ denotes a *pre/size* guarded expression).

Note that we return the derivative ε in this case which effectively simulates the successful derivation of the skipped subtree. It is not immediately clear to us how such "shortcutting" could be smoothly integrated into the operation of a state automaton.

5. HIGH THROUGHPUT VALIDATION

Any XML Schema description is subject to the *unique particle attribution constraint.* The constraint guarantees that an XQuery implementation can form *1-unambiguous* regular expressions [7] from the description when the schema is imported. Additionally, if the regular expression types are brought into *star normal form* after schema import, applications of ∂ preserve the 1-unambiguity property [6]. To exemplify, the star normal form for

$$\left(e_1{}^{0,*} \cdot e_2{}^{0,*}\right)^{0,*} \quad \text{is} \quad (e_1 \mid e_2)^{0,*} \quad .$$

In [6], Brüggemann-Klein and Wood describe an algorithm to compute the star normal form of a regular expression in linear time.

For such unambiguous types, it is sensible to attach semantic actions—the type annotation side effects built into ∂, see Figure 7(a)—to its constituents: during the scan of the encoding, each element node is either unambiguously matched by a specific $\mathsf{elem}\, t$ subexpression of the current derivative or the node does not match at all.

Furthermore, *1*-unambiguity implies *immediate* deterministic choice: availability of the current node v suffices to see that at least one branch in the derivation $\partial_v(e_1 \mid e_2) = \partial_v(e_1) | \partial_v(e_2)$ will yield $\varnothing$ and may thus be pruned. Together with the simple equivalences of Figure 9, 1-unambiguity effectively ensures that the size of the intermediate derivative types remains unaffected by the size of the document being validated.

5.1 Validation Experiments

A primary goal of this work was to design the validation procedure such that the infrastructure of an XQuery runtime system based on a tree encoding similar to that of Section 2 could be reused without change. We were indeed able to hook the validation and type annotation algorithm into our MonetDB-based XQuery back-end with minimal effort. MonetDB [4] is an extensible relational database kernel whose internals have been optimized for in-memory operation. The principal storage structure provided by MonetDB are *two-column relational tables* which fit nicely with the tree encoding scheme of Section 2.

We implemented the regular expression derivation (Figure 7(a)) straightforwardly and with no particular twists. The above observation about the limited size of derivatives in star normal form enabled us to maintain the types in an array of static size whose contents were garbage collected on demand.

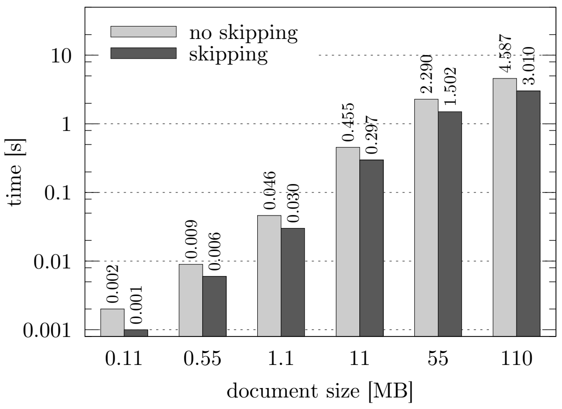


Figure 10: Validation throughput for XMark XML instances of varying size.

To assess the throughput achievable with this approach, the back-end was loaded with XMark XML instances of varying size. We imported the XMark DTD to yield 74 regular expression type definitions. The database was hosted on a 2.2 GHz Intel Pentium 4 based computer with 2 GB RAM running a version 2.4 Linux kernel. Figure 10 reports on the execution times. As expected, validation time grows linearly with the number of nodes (ranging from approx. 3,000 to 3,000,000) in the validated instance. We then ran the validation experiments a second time with selected nodes[4] marked to be skipped during validation. Execution times drop since a significant amount of derivation effort is saved (∂ simply returns ε) and, for each skipped subtree, the sequential encoding scan is immediately guided to the first node following the subtree (via the *size* property of the subtree's root node, see Section 4.2).

6. RELATED TECHNIQUES AND CONCLUSIONS

In [5] and [1], the authors develop validation algorithms based on a stack of Glushkov automata and extended nondeterministic tree automata, respectively. The algorithms exhibit remarkable complexity, introduced mainly to support *incremental validation* of documents after updates have occurred. Update operations are defined following the XML DOM paradigm (*e.g.*, a node is inserted before/after a given reference node, a node is replaced). For each node, both algorithms maintain the current state of the automaton in an auxiliary data structure to enable incremental validation. We believe such updates and the corresponding incremental validation algorithms to be of limited usefulness in the context of an XQuery runtime system where trees are *constructed* from existing fragments. The notion of *skipping* partially validated trees (Section 4.2), which also coincides with the XQuery `skip` and XSLT `preserve` validation modes, seems to be a more simple yet more pratical fit for the XQuery semantics.

In [9], James Clark describes the Haskell implementation of a validation algorithm for RelaxNG which is also based on Brzozowski derivatives.

Since validation and type annotation is an integral concept of XQuery, we believe it is compelling to design validation algorithms with a close eye on the tree encodings which are in use in XQuery runtime systems. This work seems to be among the first to view the validation problem from this angle. It is unrealistic to assume that an XQuery implementation can afford the cost to perform validation and type annotation on an internal representation of XML fragments other than the one used during the dynamic evaluation phase. To this end, this work brought an efficient XPath-aware tree encoding and simple regular expression derivation techniques together.

[4]Element nodes with tag `open_auction` were marked which effectively made the derivation skip approx. 400 nodes in the smallest and up to 1,000,000 nodes in the largest XMark instance.

7. REFERENCES

[1] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *Proc. of the 20th Int'l ICDE Conference*, Boston, Massachusetts, USA, March 2004. IEEE.

[2] P.V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. World Wide Web Consortium, May 2001. W3C Recommendation `http://www.w3.org/TR/xmlschema-2/`.

[3] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, November 2003. W3C Last Call Working Draft `http://www.w3.org/TR/xquery/`.

[4] Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[5] B. Bouchou and M. Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In *Proc. of the 9th Int'l DBLP Conference*, pages 216–232, Potsdam, Germany, September 2003. Springer Verlag.

[6] A. Brüggemann-Klein. Regular Expressions into Finite Automata. *Theoretical Computer Science*, 120(2):197–213, November 1993.

[7] A. Brüggemann-Klein and D. Wood. One-unambiguous Regular Languages. *Information and Computation*, 142(2):182–206, May 1998.

[8] J.A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[9] J. Clark. An Algorithm for RelaxNG Validation. `http://www.thaiopensource.com/relaxng/derivative.html`, February 2002.

[10] M.F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. World Wide Web Consortium, November 2003. W3C Last Call Working Draft `http://www.w3.org/TR/xpath-datamodel/`.

[11] T. Grust. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l SIGMOD Conference*, pages 109–120, Madison, Wisconsin, USA, June 2002. ACM.

[12] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proc. of the 29th Int'l VLDB Conference*, Berlin, Germany, September 2003. Morgan Kaufmann Publishers.

[13] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath Location Steps in Any RDBMS. *ACM TODS*, 29(1), March 2004.

[14] P. Kilpeläinen. SGML & XML Content Models. *Markup Languages: Theory & Practise*, 1(2):53–76, 1999.

[15] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l VLDB Conference*, pages 361–370, Rome, Italy, September 2001.