Scalable XQuery Type Matching

Jens Teubner
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
teubner@us.ibm.com

ABSTRACT

XML Schema awareness has been an integral part of the XQuery language since its early design stages. Matching XML data against XML types is the main operation that backs up XQuery type expressions, such as `typeswitch`, `instance of`, or certain XPath operators. This interaction is particularly vital in data-centric XQuery applications, where data come with detailed type information from an XML Schema document.

So far there has been little work on the optimization of those operations. This work presents an efficient implementation of the runtime aspects of XML Schema support. We propose *type ranks* as a novel and uniform way to implement all facets of type matching in the W3C XQuery Recommendation. As a concise encoding of the type hierarchy defined by an XML Schema document, type ranks minimize the cost of checking the runtime type of XQuery singleton items. By *aggregating* type ranks, we leverage the grouping capabilities of modern DBMS implementations to efficiently execute type matching on XQuery sequences. In addition, we improve the complexity bounds incurring with `typeswitch` expressions over existing approaches. Experiments on an off-the-shelf database system demonstrate the potential of our approach.

1. INTRODUCTION

While the XQuery language specification has just reached the status of a W3C Recommendation [2], research and industry are already pushing out a number of extensions to the new language. However, we feel that some of the core functionalities of XQuery are still not well understood. In this work, we address the *dynamic typing* features of XQuery and report on means to implement *sequence type matching*, the low-level primitive behind the XQuery `typeswitch`, `instance of`, and XPath operators, in an efficient and scalable manner.

All these operators allow the querying of the runtime type of an XQuery subexpression. The `typeswitch` clause

```
let $x := (x1, x2, ..., xk) return
  typeswitch ($x)
    case τ1 return e1
    case τ2 return e2
     ⋮
    case τn return en
    default return edef
```

compares the runtime type of the item sequence $(x_1, \ldots, x_k)$ against each of the sequence types $\tau_1, \ldots, \tau_n$ in turn. If a match is encountered for type τ_i, its associated subquery e_i is evaluated and immediately returned as the overall expression result. If no match can be found, the result is determined by the expression in the `default` clause e_{def}. With the same underlying definition of type matching, the expression e `instance of` τ returns the Boolean outcome of comparing e's runtime type against the sequence type τ. In XPath, node tests such as `element(`*tag*`,` *type*`)` describe filters on path expression results based on type matching.

Current XQuery engines commonly implement type matching using the late binding facilities of object-oriented programming languages or—even worse—by recursively traversing the XML Schema type derivation hierarchy [8, 16]. Both approaches, however, incur a significant runtime cost, which defeats their use for high-volume XML processing. Instead, here we focus on database-style XQuery processing and leverage existing DBMS capabilities to implement type matching in a scalable manner. It turns out that grouping and aggregation functionalities are a perfect fit for XQuery's `typeswitch` instruction, provided that the encoding of types and values is chosen deliberately.

The key insight of this paper is how *type ranks*, an adaption of Grust's XPath accelerator numbering scheme [9], encode XQuery item types in an elegant and concise manner. In the XQuery Formal Semantics, XQuery type matching is backed up by the matches predicate [6]. Type ranks provide an efficient *constant time* implementation of this predicate for singleton items. At the same time, they integrate seamlessly with database-style XQuery processing, particularly if the back-end system already uses a *pre*/*post*-based encoding for its XML storage.

The same idea can be applied to XQuery sequences of arbitrary length after *type aggregation*. Aggregation and grouping are well-tuned operations in any modern DBMS. Deliberately applied to type ranks, we show how they can also serve as a highly efficient implementation of XQuery's `instance of` predicate. The advantage becomes even more

EDBT'08, March 25–30, 2008, Nantes, France.

apparent, when we show how to use type aggregation for a holistic evaluation of `typeswitch` instructions with multiple `case` branches. The reduction in runtime complexity compared to existing approaches shows up in a significant increase in query performance in real-world scenarios.

The essences of type aggregation are simple enough to be used in any XQuery engine. Since we want to unleash existing aggregation algorithms for large data volumes, however, we exemplify our findings using a relational XQuery setup that is based on the *loop lifting* technique [12], a principle implemented, *e.g.*, in the *Pathfinder* XQuery compiler[1].

We will proceed as follows. The upcoming Section 2 discusses work related to ours. Section 3 then gives a refresh of the relevant aspects of the XQuery type system, including the introduction of type ranks to represent it. The actual implementation of sequence type matching is the topic for Section 4, where we also discuss type aggregation. Sections 5 and 6 describe and evaluate a prototype implementation on top of a standard SQL system, before we summarize in Section 7.

2. RELATED WORK

The interaction with XML Schema definitions is an integral part of the XQuery language, and functionality to inspect the runtime type of an expression result has been a core XQuery feature since its earliest W3C working draft. Yet, efficient support for this feature received only little attention in existing work. Many XQuery implementations do not support type matching at all.

The state of the art to implement type matching seems to be the recursive analysis of the `base` specification in XML Schema type definitions, as it is done, *e.g.*, by the Galax XQuery engine [8]. Saxon [16] additionally caches each analysis of the derivation hierarchy to speed up later inquiries on types. In both systems, sequences are matched item by item. Even if we assume a cache hit rate of 100 %, this implies an effort $O(l)$ to match an l-item sequence. When executing `typeswitch` instructions, both systems perform type matching for each `case` branch in turn and independently. To determine the right `case` for a `typeswitch` with n branches, this incurs an $O(l \cdot n)$ cost. In Section 4.3, we process such instructions holistically and push down the cost to $O(l + n)$ by using *type aggregation*.

An operation related to XQuery type matching is the `instanceof` operator in Java or other object-oriented programming languages, a well-studied problem in the programming languages literature. The proposed solutions can roughly be categorized into *(i)* interval-based type encodings (*e.g.*, Schubert's *number brackets* [18]), *(ii)* array-based encodings (as suggested, *e.g.*, by Cohen [5] and later implemented, *e.g.*, in the Jalapeño JVM [1]), and *(iii)* bit-vector encodings (such as [4]). Our representation of types may be seen as a member of the former category. The latter two categories address the specific needs of some object-oriented programming languages, such as the incremental nature of the Java type hierarchy or multiple inheritance (respectively). They depend on tailor-made data structures incompatible with existing database environments. None of the aforementioned works has investigated the aggregation of types that we pursue in Section 4.2.1.

Type matching is easily confused with *subtyping*. During the static analysis of an XQuery expression, only an approximation of the type of each subexpression is known, expressed in terms of a regular expression. The comparison of this regular expression against another type is referred to as subtyping. Algorithms to decide this structural type relationship have been developed in the context of the XDuce [13] and XOBE [15] projects. At runtime, the exact type of an XQuery value is known, which, in combination with syntactical constraints on XQuery sequence types, allows for more efficient algorithms, such as the approach we describe here.

Complementary to type matching is the annotation of types during XML *validation*. An evaluation technique for this operation has been presented, *e.g.*, in [10]. Here, we interpret the annotated type information and evaluate the XQuery `typeswitch` and `instance of` operators, as well as path expressions that depend on sequence type matching.

The separation of validation and type matching in the XQuery language semantics is an outcome of the study in [19]. A conclusion of that work is that once XML data has passed validation, an XQuery engine can efficiently decide type matching without having to implement all idiosyncrasies of XML Schema. Our work may be seen as a delivery of this promise with a simple and efficient implementation for type matching.

3. THE XQUERY TYPE SYSTEM

The W3C XQuery Data Model Recommendation (XDM) [7] defines the *item* as XQuery's principal data type. An item can either be a *node*, a valid instance of one of the six XML node kinds, or an *atomic value*, an instance of a primitive type like `xs:integer` or `xs:string`. Each XQuery item is annotated with a *type*, referencing the name of a defined XML Schema type.[2]

3.1 Type Annotations

Type annotations to atomic values describe the type of the respective value. This restricts the possible type annotations to *atomic types*, a subset of XML Schema's simple types. While, in XML Schema, the latter also cover list and union types built from atomic types, such type variants are not applicable to XQuery atomic type instances. To make this point explicit, the W3C has added an explicit `xs:anyAtomicType` definition to XML Schema's set of predefined types in the XDM Recommendation. It serves as a base type for all built-in atomic types.

Inspired by the notation used in [6], we write

$$x = v \texttt{ of type } t$$

to identify the value v and type annotation t of an XQuery item x in the remainder of this paper.

By contrast, type annotations to nodes are a description of the nodes' *content*. For XML element or document nodes, they can either be the name of a *complex type* definition, constraining associated attributes and child elements, or the name of a *simple type* that describes the lexical structure of text-only elements. Attributes do not have children and, hence, always carry a simple type annotation. The content of text nodes is always one string of type `xs:untypedAtomic`,

[1] `http://www.pathfinder-xquery.org/`

[2] For this purpose, anonymous XML Schema types are implicitly assigned an implementation-dependent name, a process that usually remains transparent to the user.

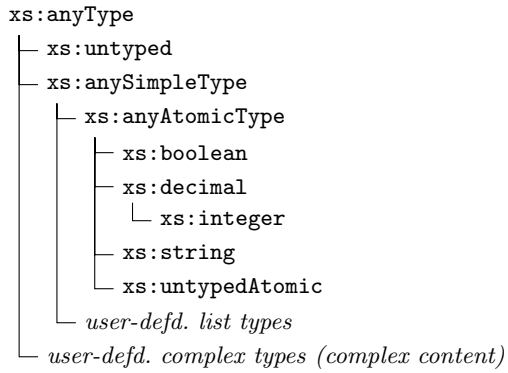


Figure 1: Excerpt of the XDM/XML Schema type hierarchy (user-defined atomic types and complex types of simple content not shown; from [7]).

which is why their type annotation can be omitted. Similarly, we omit type annotations also for comment and processing instructions. Formally, we write

$$x = \texttt{element}\ n\ \texttt{of type}\ t$$

to indicate that the XQuery item x is an element with tag name n and type annotation t.

3.2 The XDM Type Hierarchy

All XML Schema predefined types can be arranged into a type hierarchy, which corresponds to the derivation tree of the built-in types in the XQuery Formal Semantics [6]. An excerpt of the hierarchy is shown in Figure 1.

The common base type for all types is the XML Schema type `xs:anyType`. As mentioned earlier, `xs:anyAtomicType` covers all atomic types. It is derived from `xs:anySimpleType`, which describes arbitrary sequences of atomic types.[3] The type `xs:untyped` is used in XQuery to annotate unvalidated XML nodes.

User-defined types are added to the type hierarchy according to the type they have been derived from. Either built-in primitive types or other user-defined types can be referenced explicitly in an XML Schema type definition. If no base type is given, user-defined simple types are implicitly derived from `xs:anySimpleType`, user-defined complex types from the root type `xs:anyType`, as indicated in Figure 1. It is exactly this hierarchy that we will query in a moment using the XQuery `typeswitch` and `instance of` operators.

Example. *Figure 3(b) shows the type hierarchy that results for the example XML Schema document in Figure 3(a). The XML fragment in Figure 3(c) uses* `xsi:type` *attributes to provide type information to a validating XML parser.*

Observe that the resulting type hierarchy solely depends on each type's `base` specification, but not on the specific mechanism used for derivation (by extension or restriction). In effect, the types defined by an XML Schema document always assemble into a true tree hierarchy. At runtime, type matching thus boils down to querying a pure tree structure, which enables the efficient type encoding and evaluation techniques that we present in this paper. All XML Schema documents referenced in a query have to be declared in the query prolog, such that the full type hierarchy is known at query compile time. We will now exploit both observations and efficiently encode and query the XDM type hierarchy.

[3] `xs:anySimpleType` is defined as `xs:anyAtomicType*` in [6].

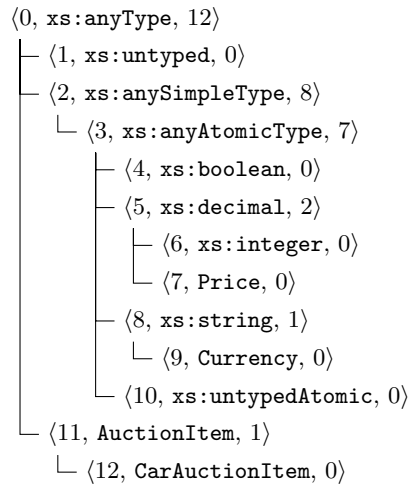


Figure 2: Type hierarchy of our running example, with annotated preorder ranks *pre*(t) (left) and subtype numbers *size*(t) (right).

3.3 Encoding Type Hierarchies

Incidentally, trees are also the data structure underlying XML documents themselves, and we borrow some ideas from one of the established XML storage techniques in order to represent XDM type hierarchies. *XPath accelerator* [9], is a lightweight tree encoding, initially developed for XML processing on relational databases. To encode the XML tree structure, each node is assigned a pair of integer values (*pre* and *post*) that allows for a constant-time checking of ancestor/descendant relationships in the tree.

We use a variant of this encoding to encode the tree structure that describes the XDM type derivation hierarchy. For each type t in the hierarchy we record its rank in a preorder tree traversal, *pre*(t), as well as the number of types that have (recursively) been derived from t, *size*(t). The same encoding has been used to encode XML documents in [12]. Note, however, that we are using *pre*/*size* here to encode type hierarchies, not XML. Figure 2 shows the *pre*/*size* values (printed left/right of each type name, respectively) that correspond to the types in our example XML instance.

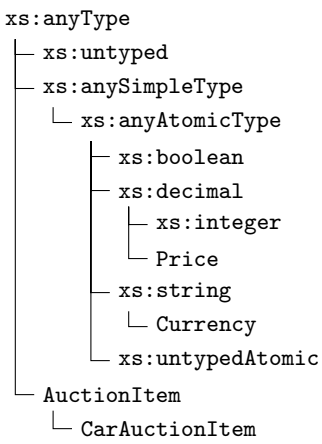
Type ranks can be computed as soon as all `import schema` declarations have been processed at XQuery expression compile time. As shown in [9], rank assignment can be performed during a single tree traversal. Since preorder ranks uniquely identify each type in the hierarchy, they can be used as a very compact representation of the type annotation to XQuery items.

3.3.1 Querying the Type Hierarchy

One of the virtues of the *pre*/*size* encoding scheme is that it provides a simple and efficient test for the ancestor/descendant relationship between arbitrary nodes. As shown in [12], the test for a descendant relationship among any two nodes v and v' in a *pre*/*size*-encoded tree amounts

```
<xs:simpleType name='Currency'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='EUR'/>
    <xs:enumeration value='USD'/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name='Price'>
  <xs:simpleContent>
    <xs:extension base='xs:decimal'>
      <xs:attribute name='currency' type='Currency'/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name='AuctionItem'>
  <xs:sequence>
    <xs:element name='name' type='xs:string'/>
    <xs:element name='price' type='Price'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='CarAuctionItem'>
  <xs:complexContent>
    <xs:extension base='AuctionItem'>
      <xs:sequence>
        <xs:element name='make' type='xs:string'/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name='item' type='AuctionItem'/>
```

(a) Example XML Schema specifications for an online auction site.



(b) Resulting type derivation hierarchy.

```
<item xsi:type='CarAuctionItem'>
  <name>Ford Windstar SEL</name>
  <price currency='USD'>8199.00</price>
  <make>Ford</make>
</item>
<item xsi:type='AuctionItem'>
  <name>Teddy Bear</name>
  <price currency='EUR'>9.95</price>
</item>
```

(c) Example XML fragment.

Figure 3: Interaction between (a) an XML Schema specification and (b) the XDM type hierarchy. The XML fragment (c) shows valid instances for the `CarAuctionItem` and `AuctionItem` types.

to two integer comparisons only:

$$\frac{pre(v') \geq pre(v) \wedge pre(v') \leq pre(v) + size(v)}{v' \text{ is } v \text{ or a descendant of } v} \quad . \qquad (1)$$

Furthermore, queries of this kind are well suited for the access paths provided by R- or B-tree indexes in existing RDBMS implementations [9, 11].

Equation 1 directly translates into an implementation of the derives from judgment in the XQuery Formal Semantics [6], the transitive closure of the derivation relationship between types:

$$\frac{pre(t_1) \geq pre(t_2) \wedge pre(t_1) \leq pre(t_2) + size(t_2)}{t_1 \text{ derives from } t_2} \quad . \qquad (2)$$

$pre(t)$ and $size(t)$ denote the preorder type rank of the named type t and the number of types derived from it, respectively.

4. SEQUENCE TYPE MATCHING

The inspection of runtime type information in XQuery using the `typeswitch` and `instance of` constructs is based on *sequence type matching*. In both cases, the expression under investigation is evaluated first. Then, the resulting value is tested against the given *node kind*, *tag name* (if applicable), and *type annotation* requirements using the matches judgment of the XQuery Formal Semantics [6].

The XQuery operator `instance of`, *e.g.* evaluates its argument expression e to obtain the item sequence x ('$\Rightarrow$' is the "evaluates to" judgement in [6]), then uses the formal matches judgment to match its runtime type against the sequence type τ:

$$\frac{e \Rightarrow x \qquad x \text{ matches } \tau}{e \texttt{ instance of } \tau \Rightarrow \texttt{true}} \quad . \qquad (3)$$

Refer to [6] for the formal definition of the `instance of` and `typeswitch` semantics.

In XPath location steps, the node tests `attribute(`n`,` t`)`, `element(`n`,` t`)`, `document-node(`t`)`, or `schema-element(`n`)` can be used as a filter on the navigation result based on node kind, tag name, and type annotation. See [6] for details.

We will now develop an implementation for all these variants of type matching. We first constrain ourselves to the evaluation of matches for singleton items and show how the judgment can efficiently be implemented using type ranks. As we proceed with the remainder of the section, we will then generalize our ideas to support arbitrary item sequences, `typeswitch` expressions, substitution groups, and XPath navigation.

4.1 Type Matching for Singletons

Two syntactical variants of the matches judgment reflect the different properties to be queried for atomic values and nodes. Both variants delegate the comparison of types to the derives from judgment which we discussed a moment ago.

4.1.1 Type Matching for Atomic Values

An atomic item x matches an atomic type t if its type annotation t' is a type derived from the named atomic type t [6, Sect. 8.3.1]:

$$\frac{x = v \texttt{ of type } t' \qquad t' \text{ derives from } t}{x \text{ matches } t} \ . \tag{4}$$

Since, in any case, the type of an atomic value must be atomic, t is only allowed to be the name of an atomic type (derived by restriction from `xs:anyAtomicType`).

4.1.2 Type Matching for Nodes

matches can also be used to query the kind and tag name properties of XML tree nodes. To simplify matters here, we only consider element nodes here. Attribute and document nodes, the remaining two node kinds that interact with types, can be handled in much the same way. For element nodes, matches can be defined as

$$\frac{x = \texttt{element } n \texttt{ of type } t' \qquad t' \text{ derives from } t}{x \text{ matches } \texttt{element}(n, t)} \ . \tag{5}$$

XQuery allows the use of the wildcard `*` instead of the element name n in the sequence type specification. The type name t may be omitted to inspect only the kind and tag name properties of a node. If present, it may reference a complex or simple type for `element(·)` and `document-node(·)` sequence types, whereas `attribute(·)` only allows the specification of a simple type (in line with the content types allowed for the respective node kinds).

Example. *Evaluated over the XML fragment in Figure 3(c), the following* `instance of` *expressions return* `true`*:*

1. `/item[1] instance of element(*, AuctionItem)`
2. `/item[2]/price instance of element(*, Price)`
3. `/item[2]/price instance of element(*, xs:decimal)`
4. `(//@currency)[1] instance of element(*, Currency)`
5. `data(/item[2]/price) instance of xs:string`
6. `data((//@currency)[1]) instance of Currency`.

4.1.3 Implementing Type Matching

The XML Schema type t on the right-hand side of all matches predicates, including its $pre(t)$ and $size(t)$ values, is known statically at query compile time. With the preorder type rank $pre(t')$ as an implementation of the type annotation to the XQuery item subject to type matching, we can now inline our implementation of the derives from test (Equation 2) to obtain a constant-time implementation for matches, based on simple integer comparisons. For atomic types, *e.g.*, we get

$$\frac{x = v \texttt{ of type } t' \qquad pre(t') \geq pre(t) \wedge pre(t') \leq pre(t) + size(t)}{x \text{ matches } t} \ . \tag{6}$$

Thus, we have just avoided the recursive analysis of the type derivation tree at query runtime as sequence type matching is implemented in existing systems.

4.2 Sequences and Occurrence Indicators

So far we only considered the case that a singleton item is to be matched against a given type. In practice, XQuery expressions may return sequences of arbitrary length. An *occurrence indicator*, syntactically located after the sequence type specification, allows testing the length of the sequence of the `typeswitch` or `instance of` argument.

For occurrence indicators $\square \in \{\sqcup, \texttt{?}, \texttt{+}, \texttt{*}\}$, a sequence $x = (x_1, x_2, \ldots, x_l)$ matches the sequence type $\tau\square$ if

1. x_i matches τ for all x_i in x and
2. the sequence length l is compatible with the occurrence indicator $\square$ (*i.e.*, $l = 1$ for $\square = \sqcup$, $l \leq 1$ for $\square = \texttt{?}$, $l \geq 1$ for $\square = \texttt{+}$; no restriction on l for $\square = \texttt{*}$).

For ease of presentation, we restrict ourselves to atomic types for a moment. We can then use Rule 6, to rephrase the first part of the above definition based on type ranks:

$$\begin{gathered} x_i \text{ matches } t \text{ for all } x_i \text{ in } x \\ \Leftrightarrow \\ \forall\, (x_i = v_i \texttt{ of type } t_i) \in x : \\ pre(t_i) \geq pre(t) \wedge pre(t_i) \leq pre(t) + size(t) \ . \end{gathered} \tag{7}$$

Informally, we test each x_i's type annotation $pre(t_i)$ against the range prescribed by the $pre/size$ values of type t.

4.2.1 Type Aggregation

Instead of testing the two inequalities for each sequence item in separation, however, we could as well compute the minimum and maximum preorder ranks over all sequence items x_i first, then test against the aggregates once:

$$\begin{gathered} \forall\, (x_i = v_i \texttt{ of type } t_i) \in x : \\ pre(t_i) \geq pre(t) \wedge pre(t_i) \leq pre(t) + size(t) \\ \Leftrightarrow \\ \min_{(x_i = v_i \texttt{ of type } t_i) \in x} (pre(t_i)) \geq pre(t) \\ \wedge \max_{(x_i = v_i \texttt{ of type } t_i) \in x} (pre(t_i)) \leq pre(t) + size(t) \end{gathered} \tag{8}$$

The aggregated pre values now describe a range of types that covers all item types t_i found in the sequence x. Precomputing aggregates this way resembles the rewrites performed in [11], where aggregation functions took the role of a pruning operator for accelerated XPath processing.

Aggregate functions of this kind have been highly tuned in existing database implementations, mainly driven by their importance in other application domains, such as online analytical processing (OLAP). It is once more interesting to see how XQuery processing can benefit from database operators that have originally been designed to answer OLAP workloads. We have seen similar applications of OLAP functionality in [12].

Testing the compatibility of a sequence's length against a given occurrence indicator (and, hence, answering the second criterion for sequence-aware type matching) straightforwardly leads to another aggregation function: *counting* the items in x yields the sequence length l:

$$l = \text{count}(x) \ . \tag{9}$$

For atomic item sequences we can now implement matches based on type aggregates ($\lesseqgtr \square$ denotes the compatibility

check of the sequence length with the occurrence indicator $\square$):

$$\frac{\begin{array}{rcl}\min\limits_{(x_i=v_i \text{ of type } t_i)\in x}(pre(t_i)) & \geq & pre(t)\\ \max\limits_{(x_i=v_i \text{ of type } t_i)\in x}(pre(t_i)) & \leq & pre(t)+size(t)\\ \text{count}(x) & \begin{matrix}\leq\\ \geq\end{matrix} & \boxdot\end{array}}{x \text{ matches } t\square}\ . \quad (10)$$

4.2.2 Aggregating Node Properties

To implement type matching for an XQuery node sequence x, the necessary test on tag names can be expressed using a similar pre-aggregation approach: determine the minimum and maximum tag names occurring in x first, then compare both aggregates to the requested tag name n. The aggregation-based implementation of type matching for element nodes then reads (lines 3 and 4 implement tag name aggregation):

$$\frac{\begin{array}{rcl}\min\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in x}(pre(t_i)) & \geq & pre(t)\\ \max\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in x}(pre(t_i)) & \leq & pre(t)+size(t)\\ \min\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in x}(n_i) & = & n\\ \max\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in x}(n_i) & = & n\\ \text{count}(x) & \begin{matrix}\leq\\ \geq\end{matrix} & \boxdot\end{array}}{x \text{ matches } \texttt{element}(n,t)\square}\ . \quad (11)$$

Example. *Using type aggregation, the XQuery expression*

```
/item instance of element(item,CarAuctionItem)+
```

can be implemented by testing

$$\begin{array}{rcl}\min\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in \texttt{/item}}(pre(t_i)) & \overset{?}{\geq} & 2\\ \max\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in \texttt{/item}}(pre(t_i)) & \overset{?}{\leq} & 2+0\\ \min\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in \texttt{/item}}(n_i) & \overset{?}{=} & \texttt{item}\\ \max\limits_{(x_i=\texttt{element } n_i \text{ of type } t_i)\in \texttt{/item}}(n_i) & \overset{?}{=} & \texttt{item}\\ \text{count}(\texttt{/item}) & \overset{?}{\geq} & 1\end{array}$$

For the XML fragment in Figure 3(c), the first two conditions are not met such that the query evaluates to `false`.

4.3 Multiple `case` Branches

Observe that type aggregation does *not* depend on the sequence type an expression is matched against. This means that even for `typeswitch` expressions that contain multiple `case` branches,

```
typeswitch (e)
  case τ1 return e1
  case τ2 return e2
   ⋮
  case τn return en
  default return edef ,
```

it is sufficient to compute the type aggregation of the argument expression e only once. Once the type of the item sequence returned by e has been aggregated, deciding the individual `case` matches does not depend on the length of the sequence returned by e. Contrasted to iterating over e for each `case` branch independently, this can reduce the cost of evaluating `typeswitch`es significantly.

4.4 Runtime Costs

The naïve—and still predominant—way of implementing the `matches` primitive for XQuery singletons is the recursive analysis of the XDM derivation hierarchy. This analysis requires $O(d)$ recursion steps, where d is the depth of the derivation hierarchy. An evaluation based on type ranks, by contrast, only requires an $O(1)$ operation, independent of the derivation depth.

Another $O(1)$ implementation has been published in the context of the Jalapeño Java compiler [1]. There, types are encoded using variable-length arrays ("displays" in Jalapeño speak). The entries of these arrays describe the inheritance path from the Java `Object` type to the respective instance type. Executing o `instanceof` t in Java (with an object o and a type t) then amounts to the comparison of t's last array item with the item in the display of o at the same array position.

While seemingly this involves less work than two integer comparisons in the case of type ranks, the use of variable-length arrays requires an additional bounds checking. The resulting control hazard can block pipelined processing in modern computer CPUs and significantly harm performance. In contrast to that, type ranks can be compared fully independently—even in parallel if supported by the underlying system.

4.4.1 Matching Sequences

The matching of an entire sequence inherently requires the inspection of all sequence items. $O(l)$ is thus a lower bound for the average cost to match a sequence of length l. Type aggregation reaches this limit, while providing a database-compatible evaluation strategy at the same time.

Existing work spent the $O(l)$ cost also individually for each possible `case` branch of a `typeswitch` expression. For a `typeswitch` with n branches, this implies an overall cost of $O(l \cdot n)$ for matching. By applying type aggregation, we separated the traversal of the argument sequence from the type comparison. The resulting cost, $O(l+n)$, can be significantly more efficient than existing approaches, particularly for `typeswitch` expressions with multiple branches.

Also note that we consistently use type aggregation here in a way that allows the system to perform lazy evaluation. If, while aggregating the ranks of a sequence e, an item exceeds the *pre* constraints of the target type(s), the system may immediately decide the match to fail and abandon any further processing of e. This corresponds to an early-out strategy used in some existing XQuery engines, but does not affect the cost assessment above.

4.5 Substitution Groups

Instead of supplying type information using `xsi:type` attributes, the XML Schema *substitution group* instrument may be used to make the XML data representation more descriptive. In Figure 4(a), we have declared a global element `auction-item` of type `AuctionItem`. Each occurrence of `auction-item` in an XML Schema content model, however, may be instantiated by a `car-auction-item`, as defined by the `substitutionGroup` attribute in the declaration of `car-auction-item`.

```
<xs:element name='auction-item'
            type='AuctionItem'/>

<xs:element name='car-auction-item'
            substitutionGroup='auction-item'
            type='CarAuctionItem'/>
```

(a) XML Schema substitution group definition.

```
<car-auction-item>
  <name>Ford Windstar SEL</name>
  <price currency='USD'>8199.00</price>
  <make>Ford</make>
</car-auction-item>
<auction-item>
  <name>Teddy Bear</name>
  <price currency='EUR'>9.95</price>
</auction-item>
```

(b) Corresponding XML Fragment. Substitution groups have made `xsi:type` annotations redundant.

Figure 4: Substitution groups allow for descriptive element names while still ensuring subtype substitutability.

This mechanism is reflected in the `schema-element(·)` sequence type specification in XQuery. For an element node x, the match

$$x \text{ matches } \texttt{schema-element}(n)$$

succeeds whenever

1. the tag name of x is n or any of its substitution group members and
2. x's type annotation is a type derived from the XML schema type in the global element declaration for n.

Example. *Evaluated over the example data in Figure 4, the XQuery expression*

```
for $i in /* return
  $i instance of schema-element(auction-item)
```

returns `(true, true)`.

The semantics of type matching for `schema-element(·)` sequence types is captured by the inference rule

$$\frac{\texttt{define element } n \texttt{ of type } t \in \mathsf{elemDecl} \quad n' \text{ substitutes for } n \quad t' \text{ derives from } t}{\texttt{element } n' \texttt{ of type } t' \text{ matches schema-element}(n)}\ , \tag{12}$$

where we used elemDecl to denote the static environment that contains all XML Schema global element declarations.

The substitutes for relationship between element names describes, once again, a true hierarchy. Hence, we can implement the required substitutes for check in much the same way that we implemented derives from: assign $pre(n)$ and $size(n)$ values to each element name n in the substitution group hierarchy, then use $pre/size$ comparisons to decide type matching for `schema-element(·)` sequence types.

$$\frac{\begin{array}{rcl} \min\limits_{(x_i=\texttt{element } n_i \texttt{ of type } t_i) \in x} (pre(t_i)) & \geq & pre(t) \\ \max\limits_{(x_i=\texttt{element } n_i \texttt{ of type } t_i) \in x} (pre(t_i)) & \leq & pre(t) + size(t) \\ \min\limits_{(x_i=\texttt{element } n_i \texttt{ of type } t_i) \in x} (pre(n_i)) & \geq & pre(n) \\ \max\limits_{(x_i=\texttt{element } n_i \texttt{ of type } t_i) \in x} (pre(n_i)) & \leq & pre(n) + size(n) \\ \text{count}(x) & \overset{\leq}{\geq} & \boxdot \end{array}}{x \text{ matches } \texttt{schema-element}(n)\,\square}\ . \tag{13}$$

Figure 5: Matching rule for `schema-element(·)` sequence types, including substitution groups.

This seamlessly mixes with the way we implemented tag name tests above. Preorder ranks as an implementation for XML tag names can easily be aggregated to their minimum and maximum values, as required to match `element(·)` above. The support for `schema-element(·)` sequence types now merely requires the replacement of the equality predicates there by inequalities that test $pre/size$ constraints. We have listed the resulting implementation in Figure 5.

4.6 XPath Navigation

Sequence types in XQuery can also be used as node test operators in XPath navigation steps. The expression

```
/descendant::element(*, xs:string) ,
```

e.g., selects all string-valued elements in a given document (for the example document in Figure 3(c), this returns the `name` and `make` elements).

Node tests of this kind can straightforwardly be implemented as a postprocessing filter (based on type ranks) on the result of the XPath navigation operator. The filter may also be pushed into the navigation step itself, however, and exploit multi-dimensional index support in the underlying XML storage. The XPath accelerator encoding in [9], *e.g.*, already suggests the use of R-trees to evaluate XPath navigation. Filtering by sequence type then only requires an additional range constraint along the type and/or tag name dimensions of the index, allowing a holistic processing of path navigation and sequence type filtering.

5. IMPLEMENTING TYPE RANKS

Our approach is meant to support type matching even for large-scale XML processing. We can particularly benefit from back-end processors that provide efficient implementations of aggregation functions. Relational databases are known to be strong in both aspects and, hence, a good choice to implement type matching in a scalable fashion. All operations that we require are readily provided by existing RDBMS implementations.

5.1 Type Ranks in a Loop-Lifting Compiler

Loop lifting is a compilation procedure that brings full XQuery functionality to any relational database back-end [12]. The open-source XQuery compiler Pathfinder, now part of the MonetDB/XQuery distribution[4], is a complete implementation of this paradigm. In such a compiler, type

[4] `http://www.monetdb-xquery.org/`

iter	pos	item	type
1	1	γ_1	12
1	2	γ_2	11
2	2	γ_1	12

(a) Sequence encoding.

iter	tmin	tmax	cnt
1	11	12	2
2	12	12	1

(b) Aggregated types.

iter	pos	item	type
1	1	`false`	4
2	1	`true`	4

(c) `instance of` result.

Figure 6: Type-annotated sequence encoding of variable `$x` in Query Q_1, its aggregated types, and the loop-lifted result of the `instance of` operation in Query Q_1.

ranks can easily be integrated to accelerate the execution of type matches, as we will see after a short recap of the principles behind loop lifting.

The loop lifting compilation procedure is built around a relational encoding for XQuery item sequences that bridges the gap between the XQuery and RDBMS processing models. If a subexpression e is evaluated within the body of a `for` iteration, its values taken in individual iterations are all collected into a single relation. This relation is termed the *loop-lifted representation* of the sequence returned by e. In this representation (an example is shown on the left), a tuple $\langle i, p, v\rangle$ may be read as "in the ith iteration, the value of e at sequence position p is v." Evaluation remains purely relational: since each item v is processed with the iteration it belongs to, XQuery FLWOR expressions can be evaluated fully set-oriented. See [12] for a full introduction into loop lifting.

iter	pos	item
1	1	$x_{1,1}$
⋮	⋮	⋮
1	l_1	x_{1,l_1}
⋮	⋮	⋮
k	1	$x_{k,1}$
⋮	⋮	⋮
k	l_k	x_{k,l_k}

5.1.1 Type Annotations

The loop-lifted sequence encoding is easily extensible. Annotations to XQuery items need only be put into a new column added to the encoding. The compilation procedure will then naturally make sure that the information is propagated through the query execution flow.

Here, we use this extensibility to keep track of type annotations to XQuery values. For each item, the new column `type` holds the *pre* value of the item's type annotation, as illustrated on the left. Note that an optimizing compiler (such as Pathfinder) will never actually generate this column for any subexpression whose type information is never inspected by any upstream operator.

iter	pos	item	type
⋮	⋮	⋮	⋮

Example. *The loop-lifted encoding $q_{\$x}$ of the subexpression* `$x` *in the query*

```
for $min-price in (5, 500) return
  let $x := /item[price > $min-price] return
    $x instance of element(*, CarAuctionItem)+ ,
```
(Q_1)

including type information, is the relation shown in Figure 6(a). γ_1 and γ_2 are surrogates standing for the two `item` *nodes of the XML fragment in Figure 3(c). Values 11 and 12 are the preorder ranks of the XML Schema types* `AuctionItem` *and* `CarAuctionItem`*, respectively, as shown in Figure 2.*

5.1.2 Type Aggregation

Given the type-enriched variant of the loop-lifted sequence encoding, type aggregation is straightforward to express in terms of relational operators. In SQL, *e.g.*, this operation reads[5]

```
SELECT iter, MIN(type), MAX(type), COUNT(*)
  FROM q_e
 GROUP BY iter .
```

Figure 6(b) illustrates the result of running this query over the encoding of `$x` in Query Q_1. Based on this, we can now extend the rule set of the compiler in [12] to deal with type-related XQuery features. In Figure 7, the Rule InstanceOf defines the "compiles to" function '$\mapsto$' using the SQL `CASE` construct. The resulting SQL code generates `true` or `false`, depending on the outcome of the match. Note how the rule also populates column `type` with a reference to the XML Schema type `xs:boolean` (preorder rank 4). Figure 6(c) shows the outcome of evaluating the `instance of` expression in Query Q_1 (Boolean values `false` and `true` in the first and second `for` iteration, respectively).

5.2 XML DBMSs with a Nested Data Model

Whereas the loop lifting procedure compiles XQuery into a purely relational execution plan, operating on 1NF relations only, others have given up the 1NF constraint and allow attributes to be sequence-valued. Algebraic compilers that target such a data model have been developed, *e.g.*, for the Galax [17] and Natix [3] systems.

Much like in the loop lifting setup, these processors could use type ranks to annotate XQuery item values with type information. In addition, since nesting and unnesting are expressed explicitly in nested algebras for XQuery, they provide direct hooks to perform type aggregation. Aggregated types could then be maintained along with the nested sequence values. Type aggregates are easy to compute incrementally, which might open the way for interesting optimizations that avoid the re-computation of type aggregates.

6. EXPERIMENTAL ASSESSMENT

Since loop lifting seems to be the only XQuery compilation strategy that can be implemented on top of a commodity database system, our experimental assessment is based on the loop-lifted sequence encoding sketched above. Type ranks could equally be implemented in tailor-made XML database back-ends, and we would expect to see similar performance advantages there. The back-end system we used was an IBM DB2 9 ESE installation on a 2.33 GHz Intel Core 2 machine running a version 2.6 Linux kernel. The system was equipped with 3 GB of main memory and 100 GB secondary SATA storage. Our experiments are SQL-only and do not use the XML processing capabilities of DB2 9.

6.1 Experimental Setup

Our experiments focus on the type matching performance only. To this end, we simulated an intermediate query result and materialized it as a persistent database table val in DB2 (schema ⟨iter|pos|item|type⟩). Since, in an actual XQuery

[5]For ease of presentation, we assume an atomic sequence here. Matching node sequences requires implementation-dependent access to node properties and aggregation as sketched in Section 4.2.2.

$$\frac{\Gamma; \mathsf{loop}; \mathsf{doc} \vdash e \Rightarrow q_e \qquad t \text{ is a named atomic type}}{\Gamma; \mathsf{loop}; \mathsf{doc} \vdash e \texttt{ instance of } t\square \Rightarrow} \text{(InstanceOf)}$$

$$\left(\begin{array}{l}
\texttt{SELECT iter, 1 AS pos,} \\
\quad \texttt{CASE WHEN (MIN(type) >= } pre(t) \texttt{ AND MAX(type) <= } pre(t) + size(t) \\
\quad\quad \texttt{AND COUNT(*) IS COMPATIBLE WITH } \square \texttt{)} \\
\quad\ \texttt{THEN 'true' ELSE 'false' END AS item,} \qquad\qquad\qquad\qquad\qquad\qquad \texttt{, doc} \\
\quad pre\texttt{(xs:boolean) AS type} \\
\texttt{FROM } q_e \\
\texttt{GROUP BY iter}
\end{array}\right)$$

Figure 7: Translation of `instance of` for atomic types in a loop-lifting compiler (notation adapted from [12]).

run, table val would be a computed result that does not provide index support, we intentionally refrained from creating any indexes. Each item in val referenced a random type from the XML Schema definition for the Financial products Markup Language (FpML) [14], which contains 777 complex and simple type definitions.

On the generated data, we compared three different evaluation strategies for the XQuery `instance of` operator, each of them implemented in SQL:

1. Implementation `NAIVE` uses a recursive traversal to analyze the type derivation hierarchy (as it is done, *e.g.*, in Galax). The SQL code for this implementation is listed in Figure 8(a).

2. The `RANKS` implementation uses type ranks to decide matches for each sequence item in turn (see Eq. 7). See Figure 8(b) for its SQL implementation.

3. `AGGR` finally exploits the back-end's grouping functionalities to implement type aggregation (see Eq. 8). This implementation uses the SQL code shown in Figure 7.

Each implementation was run multiple times with random types from the FpML schema as the operation's right-hand-side type.

6.2 Results

We simulated a workload where `instance of` is evaluated in a loop with 10,000 iterations (*i.e.*, table val contained 10,000 distinct iter values). For each iteration, the loop-lifted sequence encoding contained a random XQuery item sequence with an average length ranging from 5 to 50 (*i.e.*, a total of 50,000–500,000 tuples in table val).

Figure 9 illustrates the total execution performance we observed for the three type matching strategies. For all strategies, execution times grow linear with the sequence length l, which confirms our discussion on runtime costs in Section 4.4. The aggregation-based implementation outperforms the `NAIVE` and `RANKS` implementations by more than two orders of magnitude. Type aggregation is indeed a perfect match for the grouping functionality in the relational back-end. Strategy `AGGR` is essentially evaluated by a single `GRPBY` operator in DB2.

The analysis of type relationships via recursion in strategy `NAIVE` incurs less runtime overhead than one might expect judging from the discussion in Section 4.4. A look into the respective execution plan clarifies why DB2 can handle the recursive case so well. In this plan, DB2 uses materialization to avoid the recursive computation at runtime. Instead, the entire derives from relationship ($\approx$ 3603 type pairs) is

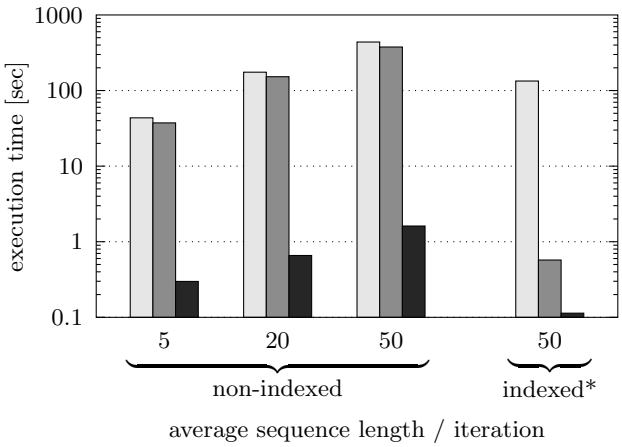


Figure 9: Type matching performance observed for the ☐ `NAIVE`, ▒ `RANKS`, and ■ `AGGR` strategies, run over a loop-lifted sequence encoding that describes 10,000 XQuery `for` iterations (*see Section 6.3).

generated and materialized before running the actual query. Testing individual types then amounts to a temporary table lookup only. This somewhat relates to the caching mechanism in the Saxon XQuery processor. In contrast to Saxon, however, DB2 pre-materializes the derives from relationship eagerly, for all types in the XDM type hierarchy.

6.3 Optimization

As a conservative approximation of a computed subquery result, the above experiments left table val without index support and in random tuple order. Although this is a valid assumption with respect to indexes, computed results may still provide other features beneficial for query optimization. Order properties in particular may help the evaluation of the grouping operators. In fact, to evaluate query `AGGR`, DB2 applied a `SORT` operator first.

While we cannot provide the system any information about the physical tuple order of a given table, sorted indexes (B-trees) can be used to access a relation in a given order. To judge the potential of index- and order-related optimizations, we let DB2's design advisor tool `db2advis` choose suitable indexes for our workload (an $\langle \mathsf{iter}, \mathsf{type} \rangle$ B-tree) and then re-ran our experiments.

The rightmost measurement in Figure 9 shows how particularly the two type rank-based implementations can benefit from ordered data access. In both cases, the availability

```
WITH ancestor(id, anc) AS
  (SELECT id, id AS base FROM types
    UNION ALL
   SELECT a.id AS id, t.base AS base FROM ancestor AS a, types AS t WHERE a.base = t.id)
SELECT iter, 1 AS pos,
       CASE WHEN EXISTS (SELECT * FROM q_e AS inner
                         WHERE inner.iter = q.iter
                           AND NOT EXISTS (SELECT * FROM ancestor AS a
                                           WHERE a.id = inner.type AND a.base = t))
        THEN 'false' ELSE 'true' END AS item,
       pre(xs:boolean) AS type
  FROM q_e AS q
 GROUP BY iter
```

(a) Strategy `NAIVE` recursively accesses the types table which encodes the XML Schema derivation hierarchy.

```
SELECT iter, 1 AS pos,
       CASE WHEN EXISTS (SELECT * FROM q_e AS inner
                         WHERE inner.iter = q.iter
                           AND NOT (inner.type >= pre(t) AND inner.type <= pre(t) + size(t))
        THEN 'false' ELSE 'true' END AS item,
       pre(xs:boolean) AS type
  FROM q_e AS q
 GROUP BY iter
```

(b) Strategy `RANKS` decides the derives from relationship based on type ranks.

Figure 8: SQL code to implement the `NAIVE` and `RANKS` strategies (queries assume a $*$ occurrence indicator).

of a suitable index now avoided an explicit `SORT` of the input data. The use of type ranks enabled such optimizations through a query representation that depends on purely relational database operators only.

7. SUMMARY

Although XML Schema awareness has been an integral part of the XQuery language since its earliest design stages, the support for it has received only little attention in existing work. It turns out that its runtime aspect, XQuery type matching, can be implemented very efficiently once deliberate representations have been determined for XDM type hierarchies and type annotations to XQuery items.

Type ranks provide such a representation in terms of simple integer attributes that could easily be integrated into existing XQuery processors. Once implemented, they provide uniform and complete support for all runtime type tests defined in the XQuery language. Most importantly, however, type ranks pave the way for efficient, database-style execution of XQuery type matching. By aggregating types, we leverage well-studied database (grouping) functionality to provide highly scalable implementations for XQuery type matching.

Since type ranks integrate well into the execution paradigms of existing database technology, the resulting query plans are highly susceptible to established query optimization techniques. With preliminary experiments that use indexes to implement ordered data access, we demonstrated how DB2's optimizer acknowledges this opportunity and improves query performance by an order of magnitude. We expect even further performance advances if type-based queries can be optimized in the context of a complete XQuery expression.

We look forward to an implementation of type ranks in a full-scale XQuery processor.

Acknowledgements

Parts of this work have been done while the author was a member of the database group at the Technische Universität München.

8. REFERENCES

[1] Bowen Alpern, Anthony Cocchi, and David Grove. Dynamic Type Checking in Jalapeño. In *Java™ Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, USA, April 2001.

[2] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, January 2007.

[3] Matthias Brantner, Carl-Christian Kanne, Sven Helmer, and Guido Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. of the 21st IEEE Conference on Data Engineering (ICDE)*, Tokyo, Japan, April 2005.

[4] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In *Proc. of the 8th Conference*

on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Washington D.C., USA, 1993.

[5] Norman H. Cohen. Type-Extension Type Tests Can Be Performed In Constant Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4), October 1991.

[6] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, January 2007.

[7] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, January 2007.

[8] Galax. `http://www.galaxquery.org/`.

[9] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, Madison, WI, USA, June 2002.

[10] Torsten Grust and Stefan Klinger. Schema Validation and Type Annotation for Encoded Trees. In *Proc. of the 1st Int'l Workshop on XQuery Implementation, Experience, and Perspective*, Paris, France, June 2004.

[11] Torsten Grust, Jan Rittinger, and Jens Teubner. Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect. In *Proc. of the 2007 ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[12] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[13] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1), January 2005.

[14] International Swaps and Derivatives Association, Inc. FpML Financial product Markup Language Recommendation, version 4.2, May 2007. `http://www.fpml.org/spec/2007/rec-fpml-4-2-2007-05-14`.

[15] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In *Proc. of the 2003 BTW Conference (Datenbanksysteme für Business, Technologie und Web)*, Leipzig, Germany, February 2003.

[16] Michael Kay, Saxonica. Saxon-SA 8.8J. `http://www.saxonica.com/`.

[17] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of the 22nd IEEE Conference on Data Engineering (ICDE)*, Atlanta, GA, USA, April 2006.

[18] Lenhart K. Schubert, Mary A. Papalaskaris, and Jay Taugher. Determine Type, Part, Color, and Time Relationships. *IEEE Computer*, 16(10):53–60, October 1983.

[19] Jérôme Siméon and Philip Wadler. The Essence of XML. In *Proc. of the 30th Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, USA, January 2003.