TypEx: A Type Based Approach to XML Stream Querying

George Russell
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
george@cis.strath.ac.uk

Mathias Neumüller
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
mathias@cis.strath.ac.uk

Richard Connor
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
richard@cis.strath.ac.uk

ABSTRACT

We consider the topic of query evaluation over semistructured information streams, and XML data streams in particular. Streaming evaluation methods are necessarily event-driven, which is in tension with high-level query models; in general, the more expressive the query language, the harder it is to translate queries into an event-based implementation with finite resource bounds.

We consider an alternative model by introducing a two-phase evaluation strategy. A query Q is decomposed into an event driven primary filter query Q', which incrementally gathers relevant data from the input stream, and another query Q'' which consumes this data as it becomes available. Evaluation of $Q(s)$ is then equivalent to $Q''(Q'(s))$. The importance of the separation is that it allows the first phase Q' to be expressed in a non-Turing complete algebra which may therefore be generally amenable to event-based interpretation. The second phase Q'' may be expressed in an arbitrary higher-order language, so long as its execution takes no longer than the extraction of the next input instance from the input stream.

In this paper a type algebra is used to express the first-phase query. This builds on previous work, which shows how traditional programming language types may be given a semantics within XML, therefore allowing their projection onto XML resources. A side-effect of this definition of projection is that instances of a type may be extracted in a form available for computation within a traditional domain. The use of type projection in this context also allows Q'' to be statically typed according to the type filter used for Q', which itself may be deduced by inference over Q''. A mechanism for translating a type into a network of event driven automata, which has the effect of gathering all data captured by that type from a semistructured input stream, is described. Although at an early stage of investigation, initial results suggest this approach provides a credible alternative to stream-based querying in at least some application domains.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Languages—*Query Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data Types and Structures*

Keywords

Type projection, stream processing, query typing, language integration, semistructured data

1. INTRODUCTION

The requirement for efficient, stream-based XML querying is by now established. There are several applications of such systems, notably in *selective dissemination of information* (SDI) and publish / subscribe systems [1] and for very large data environments and data integration [8]. Another motivation is the efficient transformation of streaming XML data, in applications such as XSLT processing and continuous data streams [9]. Such applications require processing to occur in real time as data becomes available, rather than after the end of the input stream is reached as in most XML query models.

SAX[1] provides a fully general XML stream-processing abstraction, giving the programmer an event-based interface based on callbacks. This interface provides full real-time streaming functionality; however the nature of the interface makes programming challenging for many applications, as conversion from the event stream to the logical structure of the underlying data is entirely the burden of the programmer.

Given this difficulty, a number of authors have investigated translation from a higher-level expressive form into an event-based model, expressed most commonly as a deterministic finite state automata (DFA) network [2, 7, 9, 11, 12]. Partial translations from both XPath and XQuery have been discussed, and while there are still open issues, significant progress has been made. One of the major outstanding issues however is the identification of the level of expression which can be sensibly handled in this model. Both XPath and XQuery contain expressions which can not usefully be translated into a DFA model without compromising computational thresholds underlying the purpose of the transla-

[1]`http://www.saxproject.org`

tions; the same of course is true for any single query language with sufficient expressive power to handle any reasonable range of queries.

We propose a significant departure from this methodology which we show works well for certain classes of application. The query is coded as a function in a Turing-complete programming language, with the assumption that the execution of this function will occur within acceptable bounds if its input can be isolated and passed to it as a process separated from the parsing of the input stream. This function could represent the query in its entirety, but in this domain the whole query is more likely to be represented by the repeated application of the function to instances of its input as they are extracted from the XML stream.

The function is typed according to its input, and this type is used to generate a deterministic state automata based input filter which extracts corresponding values from the XML input stream. These values are then passed, as they occur, to the query. In this way fully general queries can proceed in parallel with the parsing of the input. This builds on our previous work [5, 14, 15, 6, 10], in which we show how traditional programming language types may be given a semantics within XML, therefore allowing their extraction from XML resources.

The method will only work well if the query function is short-lived, and its type represents a significantly small subset of the XML stream; however we believe there exist many instances of this pattern. The main properties of the method are as follows:

- only the type-based extraction requires to be translated into the event-based paradigm
- only the second part of the query requires to be expressed, as the filter can be automatically generated from it
- the two phases of the execution can proceed in parallel

Compared with single-phase deterministic automata translations from XPath or XQuery, the method seems likely to work relatively well for complex queries over core regular data within a loosely structured stream, while it is likely to work relatively badly for simple queries over data that is inherently unstructured. The tradeoffs with respect to simplicity of expression and efficiency are complex and require further investigation, but in this paper we have at least shown credibility in these domains with respect to some classes of application.

In more generality, not considered further in this paper, the same basic two-part query framework may be used entirely within the XML standards domain, by expressing the filter by a projection defined over XML Schema. Instances of XML would then be generated by the first phase, allowing the secondary part to be expressed in any XML query language. One particularly interesting aspect of this is that, if the entire data stream is known to be valid with respect to a given schema, then the soundness of the filter may be statically assessed, and furthermore user-level tools for generating it from the XML Schema stream description could be envisaged.

2. A MOTIVATING EXAMPLE

For motivation, an example streamed application is coded in Java using various alternative implementation techniques, namely SAX, XPath, and TypEx, the system based on the approach described. SAX only creates temporary data structures to report parsing events which need to be transferred manually into the application specific data model for processing. The XPath code uses an XPath expression to define a superset of the required data, in conjunction with DOM code to perform the required query; the TypEx code uses a Java class definition to define a superset of the required data, in conjunction with a method of that class to query it. In the cases of XPath and TypEx the initial extraction may be performed incrementally and processed in parallel with the code that uses the extracted values, whereas in the SAX application the entire processing must be performed within the parsing callbacks.

The example is a news ticker application, which extracts news items from an arbitrary XML stream. Schema information of the expected stream is incomplete and restricted to the relevant data. The application programmer knows that there are *item* elements which contain at least two direct child elements named *title* and *description*. Both these items contain only textual content and can occur only once within any *item*. This data may be embedded at any point in the source and additional content may appear beneath *item* elements. The application is to display the content of these two fields as they are detected within an unbounded stream of XML.

2.1 Parsing Event Based Model

An example of an approach in which the query is directly contained in the application code is the event-based abstraction model used by *SAX*. Mappings between parsing events and the data model used are spread over various callback methods, making it hard to understand and thus maintain. The mixture of selection and computation also increases the coupling between user-specified computation and parsing process and is thus undesirable.

Listing 1: The SAX handler for the news ticker

```
public class NewsHandler extends DefaultHandler {
  private StringBuffer _title , _description;
  private Stack elements = new Stack();
  public void characters(char[] ch,
      int start, int length) {
    if (elements.search("item") == 2) {
        if (elements.search("title") == 1)
          _title .append(ch, start, length);
        if (elements.search("description") == 1)
          _description.append(ch, start, length);
    }
  }
  public void startElement(String namespaceURI,
      String localName, String qName, Attributes atts) {
    elements.push(qName);
    if (qName.equals("item")) {
          _description = new StringBuffer();
          _title  = new StringBuffer();
        }
20    }
      public void endElement(String namespaceURI,
          String localName, String qName) {
        elements.pop();
        if (qName.equals("item")
25        System.out.println( _title +"\n"+_description);
      }
      public static void main(String[] args) {
        SAXParser parser =
          SAXParserFactory.newInstance().newSAXParser();
30      parser.parse(new URL(args[0]).openStream(),
          new NewsHandler());
      }
    }
```

Listing 1 shows that the required state information is maintained by using a stack containing all opened tags (line 15). String buffers are created once an opening *item* tag is found. Upon occurrence of character data the top two elements of the stack are checked. If they fit the structural constraints, the content is appended to the relevant buffers. The content of these buffers is printed when the end of a news item is detected. Note that this implementation does not verify order, multiplicity or even existence of required fields, but just checks that identified fields suffice the structural constraints. Other constraints would need to be checked using either a more complicated handler or a partial schema validation process, which is currently not part of any of the relevant standards.

2.2 XPath/DOM Based Model

The combination of *XPath* [16] and *DOM* allows a different approach. Programs specify the desired set of nodes using a path expression, and operate over the results using tree traversal code. XPath expressions are navigation expressions over tree structured data which are sent to an execution engine in a similar fashion as embedded SQL statements. Selection is mechanically performed by the system and returns a collection of trees, requiring tree traversal code in the user specified computation. Typically a superset of the data required is returned because XPath returns the entire subtrees rooted at the selected nodes, regardless of the data requirements of the subsequent computation. However, as there is no strong coupling between the two phases, it cannot be guaranteed that the returned data actually satisfies the computational needs.

Listing 2: The same application using XPath

```
    public class BBCNewsXPath {
      public static void main(String[] args){
        DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
5       DocumentBuilder builder =
          factory.newDocumentBuilder();
        Node doc = builder.parse(
          new InputSource( new URL (
            args [0]). openStream()));
10      NodeList results =
          XPathAPI.selectNodeList(doc,
            "//item[description and child::text()]"
            +"[title and child::text()]"
            +"[count(title)=1]"
15          +"[count(description)=1]");
        for (int i=0; i<results.getLength(); i++) {
          Node result = results.item(i);
          Element aItem = (Element) result;
          NodeList titles = aItem
20          .getElementsByTagName("title");
          String title =
            ((Element) titles.item(0))
            .getFirstChild().getNodeValue();
          NodeList descriptions = aItem
25          .getElementsByTagName("description");
          String desc =
            ((Element) descriptions.item(0))
            .getFirstChild().getNodeValue();
          System.out.println( title + "\n" + desc);
30      }
      }
    }
```

The program shown in Listing 2 uses the single XPath statement stretching from lines 12–15 to declare the navigational steps required to select the relevant data. The XPath execution engine returns a **NodeList** containing the selected nodes (line 10). The loop starting in line 16 iterates through this list and extracts the data from the relevant text nodes using the DOM API, and in particular the method *getElementsByTagName* which selects nodes based on their name. Structural checks upon the input format have been added to the XPath query in Listing 2, which allows them to be omitted from the result processing code which otherwise would contain explicit structural checking. The XPath implementation used is not streaming, but this does not affect the purpose of the example.

2.3 Type Projection Based Model

The approach suggested by this paper has been implemented in the *TypEx* system. The extraction phase identifies data that may be relevant for the query by means of a filter type and binds this data to an instance of this type for use in the second phase. Since the computation is specified in terms of the filter type, the returned instances of this type will always contain a superset of the information required.

Listing 3: The filter class *item*

```
    public class item {
      String title , description;
      public String toString() {
        return title+"\n"+description;
5     }
    }
```

In our example application, the class used to store and print news items is also used as the filter type (Listing 3). This defines the data extraction in terms of the host programming language and acts as a data model for the following computation, ensuring a match between the two phases and a seamless integration with the language. The actual com-

Listing 4: The TypEx Newsticker

```
public class BBCNews implements Observer {
  Extractor stories ;
  public BBCNews() {
    stories = new Extractor(item.class);
    stories .addObserver(this);
  }
  public void parse(InputStream in) {
    stories .parse(in );
  }
  public void update(Observable o, Object i) {
    System.out.println((item) i );
  }
  public static void main(String[] args){
    BBCNews p = new BBCNews();
    p.parse(new URL(args[0]).openStream());
  }
}
```

putation, i.e. the printing of the content is also defined by this type. Listing 4 shows the usage of this filter. It creates an **Extractor** parameterised by the type (line 4), which extracts **item** objects during parsing and passes them to the observer for display.

3. TYPE BASED EXTRACTION

In this section we outline the translation of a core type language into a network of co-operating, deterministic automata. The resulting network is capable of extracting values of those types.

The type language includes both record and list constructors, but does not allow anonymous lists to occur as they have no semantic projection in XML. In addition, lists cannot occur at the top level, as this would be incongruous with the purpose of the mechanism, which is to release typed data as soon as possible. A grammar for the type language is given below.

$$
\begin{array}{lll}
\textit{typedef} & ::= & \text{label}: \textit{type} \\
\textit{type} & ::= & \textit{record} \mid \textit{scalar} \\
\textit{record} & ::= & \{\text{label}_1\text{: } \textit{field}_1, \ldots, \text{label}_n\text{: } \textit{field}_n\} \\
\textit{field} & ::= & \textit{scalar} \mid \textit{record} \mid \textbf{list}\ [\ \textit{type}\ ] \\
\textit{scalar} & ::= & \textbf{int} \mid \textbf{string}
\end{array}
$$

A type expression is translated into a set of named component types as shown by the following example:

$$p\text{: } \{a\text{: } \textbf{int},\ b\text{: } \textbf{list}\,[\{d\text{: } \textbf{int}\}],\ c\text{: } \{d\text{: } \textbf{int}\}\}$$

is transformed into

$p\ :\ T_1$
T_1: $\{a\text{: } T_2,\ b\text{: } \textbf{list}[T_3],\ c\text{: } T_3\}$
T_2: **int**
T_3: $\{d\text{: } T_2\}$

Each type T_i in this representation is mapped to an automaton, whose job is to extract an instance of that type from the input stream and return it as a value. The topology produced by this example is shown in Figure 1. The size of the automata network equals the size of the corresponding type graph plus one for the additional **sink** machine.

The **root** and **sink** machines always occur as single instances; the others are generated according to the individual type components. Each internal connection is a two-way link, passing input events in one direction and results in the other. Only one machine at a time actively processes events; a machine become active when it receives an *init()* event, and remains so until either a *return()* or *fail()* event is passed back to its initiator. Non-active machines pass events on to the currently active machine. Events are propagated synchronously in that they are not passed to the active machine until the previous event has been processed; this avoids synchronicity problems with the return events being passed back up the chain.

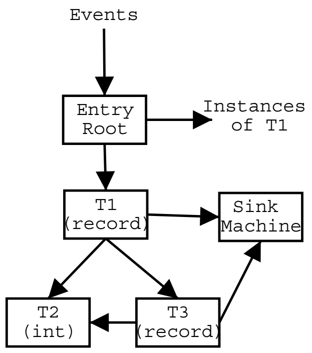

Figure 1: The example automata network

The following messages are defined: *open(l)*, *close(l)*, *text(v)*, *init(l)*, *return(v)*, and *fail()*. A parameter l stands for a label corresponding to an XML tag label, and a parameter v stands for a value. *open(l)*, *close(l)* and *text(v)* are events corresponding to a simplified input stream, and correspond to the textual form of the XML being processed. *init(l)*, as described, causes a machine to become active; its parameter is a label which, when subsequently received within a *close(l)* message, will cause it to pass control back to its initiator. This is either by means of a *return(v)* message, in the case it has been able to extract a value corresponding to its type from the input stream, and by means of a *fail()* message otherwise. There are four different classes of machines: **entry**, **sink**, **scalar** and **record**, as defined by the following behaviours:

- **Entry**: discard every event until an *open(l)* occurs, where l is the label corresponding to the name of the top-level *typedef*. At this point, *init(l)* is passed on to the machine corresponding to the type of the top-level *typedef*, which then becomes the active machine. Further events are passed to this machine until either a *return(v)* or *fail()* is received from it. On a *return(v)*, the parameter v is passed on to the network's receiver; on *fail()*, no further action occurs.

- **Sink**: the purpose of this machine is to discard an XML subtree, which cannot be of interest to the extraction. On receipt of *init(l)*, it becomes the active machine: its only required function is to discard events until the corresponding *close(l)* event is received, ensuring that any contained subtrees using the same label l are also discarded. No value is returned.

- **Scalar**: on receipt of *init(l)*, a scalar machine requires the first event it receives to be a *text(v)* message, and its second to be a *close(l)* message. Depending on the particular scalar type, the string parameter v is examined to ensure it is structurally compatible, and if so is coerced and returned. Any other combination of events causes the machine to pass a *fail()* event back to its initiator.
- **Record**: a record machine starts with internal state variables corresponding to each of the field names occurring in its progenitor type description. For each one, the value is initialised to *undefined* if the field is a scalar or record type, and to an empty list if it is a list type. These variables are used to build up the state corresponding to the record value it attempts to extract. It may receive *text(v)*, *open(l)* and *close(l)* events from its initiator:
  - *text(v)*: the event is discarded
  - *open(l)*: the behaviour depends on whether there is an internal state variable corresponding to l, and if so what its value is. If there is no internal variable, an *init(l)* message is sent to the **sink** machine. If there is a variable which is a list, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a value, it is appended to the list, otherwise no action is taken. If there is a variable whose value is defined, this results in termination with *fail()*. Finally, if there is a variable whose value is not yet defined, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a result, it is assigned to the internal variable, otherwise no action is taken.

    This behaviour corresponds to a structural subtyping approach to extracting the corresponding data values in the XML whilst ignoring order fields and any extra fields that are not pertinent to the query.
  - *close(l)*: if all internal state fields are defined, a record value based on them is constructed and returned to the calling machine in a *return(v)* message; otherwise, a *fail()* message is passed back.

4. IMPLEMENTATION

The automata networks within *TypEx* form the middle layer of the system architecture. Below this layer an event handler translates parsing events generated by an underlying XML parser into automata events. Above the automata layer is the transformation layer, which generates networks from filter specifications and transforms extracted data graphs into instances of the specified type. The top layer is the programmer visible API, which allows the specification of an input source and associated filter types. It allows multiple listeners per filter and multiple filters per data stream, affording a degree of parallelism in the query process.

Each automaton is implemented as a Java class. Automata networks are generated using reflection to examine the field types and names of filter types. Reflection is also used during data instantiation process.

5. EXPERIMENTAL RESULTS

To determine the viability of our approach, we have processed a more complex example query than the one discussed in Section 2. The data set under consideration has been generated using the XMark benchmark [13] scalable data generation tool and describes auction site details containing items for sale, persons (bidders and sellers) and some more information not relevant for our purposes. In particular, we have queried large XML files (up to 11GB in size) for people (with attributes such as name, address, etc.). We compare programs based on SAX, DOM XPath and TypEx both in terms of the complexity of the code and their runtime performance. The number of lines have been taken as a simple measure of the complexity of the code (Table 1).

System	Lines of Code		Comment
	Select	Extract	
SAX	150		Selection and extraction
DOM	43		cannot be separated.
XPath	1	39	Uses DOM extraction code.
TypEx	6	17	

Table 1: Length of Query

System	1 MB	10 MB	15 MB	30 MB	100 MB
DOM	1.73	147.0	326.0	1296	N/A
XPath	0.38	4.6	7.7	45	N/A
SAX	0.19	1.1	1.5	3	13
TypEx	0.43	3.6	5.6	10	21

Table 2: Length of Query Execution (s)

Table 1 supports the observation gained from the news ticker example and illustrates the conciseness possible using the TypEx approach. The SAX program, contains a mix of high-level application code and low-level parsing callbacks and results in an order of magnitude increase of code. DOM and XPath approaches lie between these two extremes and are almost identical because of their common tree traversal code.

The SAX query execution time scales linearly with the size of the input data and its memory usage depends only the level of element nesting. DOM is unsuitable for streaming due to its inherent whole document approach. As the input size increases, the time taken to complete the query increases more than linearly, while the memory usage increases linearly. With an 100 MB source file, the query fails to complete due to an *OutOfMemory* error using a heap size of 512 MB. We were unable to obtain a complete implementation of XPath to operate upon streaming data. The implementation we used for this experiment, Xalan/J, is backed by a incrementally built tree structure, and thus scales similarly to a DOM implementation in space. It does not however, provide results in an incremental fashion and thus does not allow a direct comparison with either SAX or TypEx. TypEx, while slower to execute than the equivalent SAX program also scales linearly with the size of the input data. The memory requirements are determined by the size of the extracted type, i.e. are independent of the size of the document. We have used it to successfully query inputs of up to 11 GB in size.

6. RELATED WORK

To the best of our knowledge, this is the first attempt of using types as filters in a two stage stream query process. The requirements for stream processing have been identified in [3] and elsewhere.

Much of the effort concentrates on implementing the XPath [16] language over streams. A number of independent efforts are proceeding, such as XMLTK [2], SPEX [11], and XSQ [12]. Green et al. [7] describe the implementation of an XPath subset using a lazily generated network of DFA and provide a comparison with other XPath implementations over streams. The work concentrates on the construction of a single automata network representing a large number of XPath queries which are executed simultaneously. Olteanu et al. [11] describe the translation of a subset of XPath expression into equivalent expression using only forward axis, which can then be efficiently processed by their SPEX XPath system using a network of event driven automata. Finally the XSQ system [12] is based upon pushdown transducers with associated buffers and aims at complete XPath support.

A more recent approach by Ludäscher et al. [9] describes the implementation of the computationally complete query language XQuery over streaming data, using transducer networks derived from a query expression. Their XSM system optimises the derived network based on static analysis and available schema information. Optimisation strategies used may prove useful for our approach.

Type projection is introduced in [5] and formalised in [15]. Language integration with Java is described in [14] in more depth, while its use within query languages for XML is also outlined [10, 6].

7. CONCLUSIONS

We have outlined a novel mechanism which allows XML stream queries to be decomposed into an extraction and a computation phase over the extracted data based on a single type description. While the concept requires further investigation, it has a number of advantages for some classes of application. In particular, it may allow the programmer to use a higher-level, and therefore more succinct, query without gravely affecting properties of efficiency. Queries are formulated within the domain of the host language and can thus be statically typed.

The mechanisms have been implemented and first results show some promise. However the investigation is at an early stage and a great deal of work remains to be done.

8. ACKNOWLEDGEMENTS

We would like to thank Fabio Simeoni, David Lievens, Steve Neely and the anonymous referees for making useful suggestions on the presentation of this work. It has been financially supported by EPSRC (GR/M72265) and BBSRC (17/BIO12052.) George Russell and Mathias Neumüller are supported by PhD studentships funded by the University of Strathclyde.

9. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In A. E. Abbadi, et al., editors, *VLDB 2000*, pages 53–64, 2000. Morgan Kaufmann.

[2] I. Avila-Campillo, T. J. Green, et al. XMLTK: An XML toolkit for scalable XML stream processing. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[3] B. Babcock, S. Babu, and other. Models and issues in data stream systems. In L. Popa, editor, *PODS 2002*, pages 1–16, Madison, Wisconsin, USA, 2002. ACM.

[4] P. A. Bernstein, Y. E. Ioannidis, et al., editors. *Proceedings of the 28th International Conference on Very Large Databases*, Hong Kong, China, 2002. Morgan Kaufmann.

[5] R. Connor, D. Lievens, et al. Extracting typed values from XML data. In *OOPSLA Workshop on Objects, XML and Databases*, 2001.

[6] R. Connor, D. Lievens, et al. Projector – a partially typed language for querying XML. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[7] T. J. Green, G. Mikklau, et al. Processing XML streams with deterministic automata. In D. Calvanese et al., editors, *ICDT 2003*, volume 2572 of *LNCS*, pages 173–189. Springer, 2002.

[8] T. Kiesling. Towards a streamed XPath evaluation. Diplomarbeit, Universität München, 2002.

[9] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In Bernstein et al. [4].

[10] P. Manghi, F. Simeoni, et al. Hybrid applications over XML: Integrating the procedural and declarative approaches. In *Fourth International Workshop on Web Information and Data Management (WIDM'02)*, Virginia, USA, Nov 2002.

[11] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *ICDE 2003*, Mar 2003.

[12] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD 2003*, 2003.

[13] A. R. Schmidt, F. Waas, et al. XMark: A benchmark for XML data management. In Bernstein et al. [4], pages 974 – 985.

[14] F. Simeoni, D. Lievens, et al. Language bindings to XML. *IEEE Internet Computing*, 7(1), Jan/Feb 2003.

[15] F. Simeoni, P. Manghi, et al. An approach to high-level language bindings to XML. *Information & Software Technology*, 44(4):217–228, 2002.

[16] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, W3C recommendation 16 November 1999 edition, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.