Compiling XPath into a State-less Forward-only Subset

Pierre Genevès* Kristoffer Rose
IBM T. J. Watson Research Center

March 2004

Abstract

We show how the *context state* of XPath, accessed through the `position()` and `last()` pseudo-functions, can be eliminated in most cases by translating references to the context state with an equivalent context-free expression, and how this enables the use of context state in combination with a subsequent "forward-only" transformation, allowing for execution of (almost) full XPath on any of the emerging "streaming" subsets.

Specifically we show how the "normalization" into a "core" language as proposed in the current W3C "Last Call" draft of the XPath/XQuery Formal Semantics can be extended such that the context state and reverse axes can be eliminated from the core XPath (and potentially XQuery) language.

1 Introduction

XPath [6] is emerging as the dominant notation for describing *selection* of nodes in XML data as well as for performing (basic) computations over the values stored in the nodes. The idea of XPath is to "navigate" XML data in "steps" that each move the "focus" from one node to another. The language for specifying steps is very rich in what kind of node associations one can use to navigate between them in order to make it as easy as possible to reach any focus of interest from any other.

However, when XML is stored or transmitted then the system architecture often imposes limitations on what kinds of navigation are efficient for the XML data as well as what kinds of data can be selected. For example,

- When storing XML data in a single file following the XML standard [5], then only so-called "streaming" access, where the nodes are visited in left-to-right tree order, is truly efficient, and it is hard to represent a reference to a specific node in the data.

- When storing XML data in an indexed table per element name then it is possible to reference individual nodes and efficient to extract all nodes with the same element name but expensive to follow the tree structure.

Since XPath allows any conceivable access policy then current mainstream XPath implementations such as Xalan [1] implement XPath by copying the entire XML data contents into a linked memory structure such as the Document Object Model that then easily supports the full XPath language [16].

Indeed many cite this concern as a reason for not using XPath at all but instead inventing and using a subset that can be efficiently implemented on top of the desired data structure; especially the sequential or streaming case has attracted attention as this is the natural access policy for generic textual XML files [3, 7, 10, 12, 14].

However, work has also been undertaken to attempt to adapt and optimize general XPath to specific data access policies in the form of schema constraints [9] or to streaming [2, 13]. However, these adaptations are rather complicated because of the size of the XPath language.

In this work we propose a way to translate the full XPath language into a minimal subset based on the XPath/XQuery formal semantics [8] *core expression*

*Currently INRIA Rhône-Alpes; this work done while visiting Watson in the summer of 2003.

language. Specifically we propose the following staged approach as our translation:

Normalization. Transform the XPath expression into an equivalent expression in the minimal but fully expressive "XPath core" language specified in the XPath/XQuery formal semantics.

Eliminate context position. All references[1] to the context position (and size) are replaced by an expression computing the context position from the context node.

Eliminate reverse axes. Steps involving a reverse axis are converted to steps using the corresponding forward axis to facilitate streaming.

Leveraging our translation on the XPath 2 normalization translation makes our job much easier and more transparent. The elimination of the context position is important because it frees implementations from following the notion of "iteration over a sequence" operationally since index numbers are just values like any other – in a sense the expressions that come out of the context position elimination stage are as data access policy independent as possible.

The last stage specializes the expression to facilitate streaming and indeed the final translated term is sufficiently simple that it should be implementable by any of the streaming subsets mentioned above; we have based our implementation on the χαος algorithm [2].

The remainder of this paper first illustrates our idea with an example in Section 2 that we shall also use to introduce the parts of the formal semantics core XPath/XQuery language that are not in XPath 1.0 [6]. The following two sections explain the transformations in detail: "statelessness" in Section 3 and "forward-only" in Section 4. Finally we conclude in Section 5.

2 Example

Consider the XPath expression

```
/descendant::employee/ancestor::manager[1]
```

which enumerates all `employee` elements and then collects for each the closest `manager` ancestor element. In this section we will explain the three translation stages for this expression.

Normalization to core XPath. The first translation stage, normalization, translates the expression into the core expression shown in Figure 1, makes the semantics much more explicit by expressing the individual path steps.[2] The full core language and the precise normalization rules are given in the XPath/XQuery formal semantics [8]; here we will just explain the parts we have used that are not part of XPath [6]:[3]

```
ddo(
 let $seq := /descendant::employee
 return
  for $dot in $seq
  return
   let $seq := ddo($dot/ancestor::manager)
   return
    let $last := count($seq)
    return
     for $dot at $rpos in $seq
     return
      let $pos := $last - $rpos + 1
      return
       if $pos eq 1 then $dot else ()
)
```

Figure 1: Normalized sample expression.

`let $v :=` *Expr*$_1$ `return` *Expr*$_2$. Computes the first expression, *Expr*$_1$, and then computes the second expression, *Expr*$_2$ with the variable `$v` bound to the value computed for the first expression.

`for $dot at $pos in $seq return` *Expr*. Iterates over the sequence `$seq` (which must be given as a variable) and concatenates all the result of computing the expression, *Expr*, with `$dot` bound to each of the members of the sequence value, in order, and `$pos` bound to the index number of each member in the sequence (corresponding to the value of the `position()` pseudo-function in XPath).

[1] Actually only all *local* references to the context state can be eliminated; we explain this below.

[2] Here and in the remainder of the paper we have allowed some simplifications compared to the even more verbose expression produced by the formally specified normalization rules.

[3] The "last call" draft of XPath 2.0 [4] includes some of these.

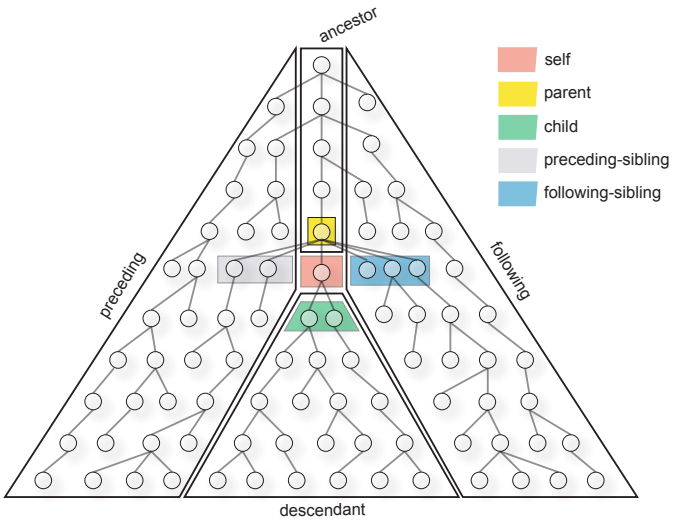

Figure 2: Axis partitions for context node.

`if` *Expr*$_1$ `then` *Expr*$_2$ `else` *Expr*$_3$. Computes *Expr*$_1$ and, depending on the truth value, returns the result of computing either *Expr*$_2$ or *Expr*$_3$.

<<. Binary test of whether two nodes are in left-to-right document order.

`intersect`. Binary node operations that construct the intersection of the operand node sequences. The result is in document order without duplicates.

`()`. The empty sequence, similar to `/..` in XPath 1.

`ddo(`*Expr*`)`. `ddo(`*Expr*`)` is equivalent to `(`*Expr* `intersect` *Expr*`)`, *i.e.*, it orders the parameter node sequence in document order without duplicates.

Like the original XPath expression, the normalized expression involves one use of a reverse axis, `ancestor`, and one use of the context position, present as the inconspicuous `at $rpos` in the last `for` expression that binds the name `$rpos` to the current position of the node being processed by the `for` expression.

Notice that `position()` and `last()` do not appear explicitly in normalized expressions: instead the generic variable names `$pos` and `$last` are by convention always bound to the corresponding numbers – indeed we can use the same variable names in each loop corresponding to an XPath path step because XPath 1 only allows access to one context sequence and node and thus the same variables can be reused.

```
ddo(
 let $seq := /descendant::employee
 return
  for $dot in $seq
  return
   let $seq := $dot/ancestor::manager
   return
    let $last := count($seq)
    return
     for $dot in $seq
     return
      let $rpos :=
       count($dot/ancestor-or-self::manager)
      return
       let $pos := $last - $rpos + 1
       return
        if $pos eq 1 then $dot else ()
)
```

Figure 3: Stateless version of sample.

Eliminating context position state. The second rewriting step is meant to rewrite the core XPath expressions to eliminate the implicitly updated context state. Here we can exploit the symmetry of the XPath axes:

the position can be calculated from expressions relative to the current node and the node that generated the context sequence. This is because these two nodes define a clean partitioning of the complete collection of nodes. Figure 2 illustrates this (and we formalize it below). Specifically we can observe that for the index of a node in the sequence

`ddo(`$dot`/ ancestor::`*NodeTest*`)`

can be computed by

`count(ancestor-or-self::`*NodeTest*`)`

when the context node is one of the nodes in the sequence. Notice that we have expressed the equivalence *with* the `ddo` document ordering function generated by the normalization transformation, so for the reverse axes we are really computing the position from the end of the sequence (from which the proper position is then indeed derived in the generated code by the subsequent calculation of `$pos`). However, *after* the translation this reordering serves no purpose as the index binding it was used for is now independent of the actual order.

Applying this to our sample expression gives the core expression in Figure 3, where we compute the reverse position variable `$rpos` explicitly, avoiding `at`.

Reverse axis elimimination. The third and final transformation is about making sure that only forward axes are used. Again it is based on the axis symmetries, in fact very closely based on equivalences like those of the "looking forward" analysis [13] extended to handle variable binding.

For our example the reverse `ancestor` axis, now occuring twice, is converted to the forward converse `descendant` axis by observing that the nodes generated by the context

`$dot / ancestor::manager`

can be generated by a search

```
let $s := /descendant-or-self::manager return
  for $d in $s return
    if $dot intersect $d / descendant::node()
    then $d else ()
```

```
ddo(
 let $managers := /descendant-or-self::manager
 return
  let $seq := /descendant::employee
  return
   for $dot in $seq
   return
    let $seq :=
     for $m in $managers
     return
      if $d/descendant-or-self::node() intersect $dot
      then $d else ()
    return
     let $last := count($seq)
     return
      for $dot in $seq
      return
       let $rpos := count(
        for $m in $managers
        return
         if $d/descendant-or-self::node() intersect $dot
         then $d else ()
       )
       return
        let $pos := $last - $rpos + 1
        return
         if $pos eq 1 then $dot else ()
)
```

Figure 4: Stateless & forward version of sample.

Applied to our example that becomes the expression shown in Figure 4: instead of using the ancestor axis we search all *potential* ancestors and for each test that it has the context node as a descendant.

Notice that both ancestor expressions are separately expanded and that there cannot be sharing between the two because they operate from different context nodes (except for the top-level search expression that is shared): the reversal transformation does some *code duplication* that we will discuss in the conclusion.

3 Eliminating context position.

To eliminate state we must modify the XPath expression into another expression without any use of the `at` binder in `for` constructions. This is achieved by

- Keeping track for every node sequence `let` variable what the defining step is.
- For every occurrence of `at` replace it with an explicit `let` binding to a computation of the index.

$S[\![\cdot]\!] : Expr \to (Var \to Step) \to Var \to Expr$

$S[\![\texttt{let \$}Var_{\texttt{seq}} \texttt{ := /}ForwardAxis\texttt{::}NodeTest \texttt{ return } Expr]\!]\ \rho\ Var$
$= \texttt{let \$}Var_{\texttt{seq}} \texttt{ := }ForwardAxis\texttt{::}NodeTest \texttt{ return } S[\![Expr]\!](\rho[Var_{\text{seq}} \mapsto ForwardAxis\texttt{::}NodeTest])Var$

$S[\![\texttt{let \$}Var_{\texttt{seq}} \texttt{ := \$}Var_{\texttt{c}} \texttt{ / }ForwardAxis\texttt{::}NodeTest \texttt{ return } Expr]\!]\ \rho\ Var$
$= \texttt{let \$}Var_{\texttt{seq}} \texttt{ := \$}Var_{\texttt{c}} \texttt{ / }ForwardAxis\texttt{::}NodeTest \texttt{ return } S[\![Expr]\!](\rho[Var_{\text{seq}} \mapsto ForwardAxis\texttt{::}NodeTest])Var$

$S[\![\texttt{let \$}Var_{\texttt{seq}} \texttt{ := ddo(/ }ReverseAxis\texttt{::}NodeTest\texttt{) return } Expr]\!]\ \rho\ Var$
$= \texttt{()}$

$S[\![\texttt{let \$}Var_{\texttt{seq}} \texttt{ := ddo(\$}Var_{\texttt{c}} \texttt{ / }ReverseAxis\texttt{::}NodeTest\texttt{) return } Expr]\!]\ \rho\ Var$
$= \texttt{let \$}Var_{\texttt{seq}} \texttt{ := \$}Var_{\texttt{c}} \texttt{ / }ReverseAxis\texttt{::}NodeTest \texttt{ return } S[\![Expr]\!](\rho[Var_{\text{seq}} \mapsto ReverseAxis\texttt{::}NodeTest])Var$

$S[\![\texttt{for \$}Var_2 \texttt{ at \$}Var_{\texttt{pos2}} \texttt{ in \$}Var_{\texttt{seq2}} \texttt{ return } Expr]\!]\ \rho\ Var$
$= \texttt{for \$}Var_2 \texttt{ in \$}Var_{\texttt{seq2}} \texttt{ return let \$}Var_{\texttt{pos2}} \texttt{ := }Expr_2 \texttt{ return } S[\![Expr]\!]\rho\ Var_2$
where $Expr_2$ is given by the following table:

$\rho(Var_{\text{seq2}})$	$Expr_2$
`self::`*NodeTest*	1
`child::`*NodeTest*	`count(preceding-sibling::`*NodeTest*`) + 1`
`parent::`*NodeTest*	1
`ancestor::`*NodeTest*	`count(ancestor-or-self::`*NodeTest*`)`
`ancestor-or-self::`*NodeTest*	`count(ancestor-or-self::`*NodeTest*`)`

other *Axis*`::`*NodeTest*	`count(for $`Var_3 `in $`$Var_{\texttt{seq2}}$ `return`
	`if $`Var_3 `<< $`Var_2 `then $`Var_3 `else ()) + 1`
	where $\texttt{\$}Var_3$ is a fresh variable

Figure 5: Transformation to Forward Stateless form.

$F[\![\cdot]\!] : Expr \to Expr$

$F[\![\texttt{\$}Var \texttt{ / }ReverseAxis\texttt{::}NodeTest]\!]$
$= \texttt{let \$}Var_{\texttt{s}} \texttt{ := /descendant-or-self::}NodeTest \texttt{ return}$
$\quad \texttt{for \$}Var_{\texttt{d}} \texttt{ in \$}Var_{\texttt{s}} \texttt{ return}$
$\quad\quad \texttt{if \$}Var \texttt{ intersect \$}Var_{\texttt{d}} \texttt{ / }ForwardAxis\texttt{::node() then \$}Var_{\texttt{d}} \texttt{ else ()}$
where $Var_{\texttt{s}}$ and $Var_{\texttt{d}}$ should be chosen fresh and the *ForwardAxis* is determined from the *ReverseAxis* by this table:

ReverseAxis	*ForwardAxis*
parent	child
ancestor	descendant
ancestor-or-self	descendant-or-self
preceding-sibling	following-sibling
preceding	following

Figure 6: Converting reverse steps to forward steps.

As explained in the introduction, the basic idea is that the index can be recomputed for every node by counting the nodes in the context sequence that occur before the context node except in a the (few but common) cases where the XPath axis symmetries provide a more direct way to compute the count.

Figure 5 formally specifies the translation S as a "derivor", written $S[\![Expr]\!]\,\rho\,Var$, where

- The *Expr* parameter is the one that is rewritten (and since it is source language syntax we surround it with the special "syntax" braces $[\![]\!]$).
- The additional parameter ρ maps all node sequence variables that are in scope to the axis and node-test used. It supports two operations:
  - $\rho[Var \mapsto Axis::NodeTest]$ returns a new environment which is like the ρ environment except it includes a description that the *Var* variable is a node sequence constructed using the *Axis*::*NodeTest* path step.
  - $\rho(Var)$ denotes the most recent pair *Axis*::*NodeTest* added to the ρ environment for `$`*Var*.
- The *Var* index contains a variable bound to the context node.

Note that the derivor is only specified formally for the interesting cases – for all other expression forms it is just distributed over the subexpressions, *e.g.*,

$$S[\![Expr_1 + Expr_2]\!]\,\rho\,Var = S[\![Expr_1]\!]\rho Var \;+\; S[\![Expr_2]\!]\rho Var$$

Finally, we observe the inherent limitation in this approach: *The global context state is not eliminated.* Thus instances of `position()` and `last()` used at the top level of XPath queries should not be translated but merely return the global context's position and size, respectively.

4 Eliminating Reverse Axes

The elimination of reverse axes also proceeds based on the symmetries illustrated in Figure 2: The nodes in the sequence constructed by the reverse axis are instead obtained by *searching* for ways to reach the context node from any node, using the symmetric forward axis. The search succeeds for a candidate reverse axis node if the intersection of the converse forward axis finds the context node from the candidate node.

Again we have only specified the interesting case, namely the translation of an actual reverse step.

5 Conclusion

We have shown how a XPath 1.0 node selection (path expression) can be translated into a form with no explicit use of non-top-level context position (or size) and using only forward axes. However, our approach has certain limitations:

- The context position elimination transformation relies on the property of XPath 1.0 that all sequences that are indexed (with `position()`) must be defined by a single XPath "step" expression with an explicit axis and node test.
- We rely on the efficiency of a few primitive operations that are not in XPath 1.0, namely document order comparison (`<<`) and intersection (`intersect`). These are highly efficient in streamning implementations but may be costly in other contexts.

Thus while the approach is certainly viable for a streaming XPath 1.0 implementation it is not clear how useful the framework is for other combinations of primitives. We would like to investigate in particular how much of the full core XPath 2.0/XQuery [8] that can be implemented in this way.

In future work we will relate the translation to the formal semantics of XPath (and XQuery) [8] as well as the classical semantics of Wadler [15]: In terms of a formal semantics we would like to prove that

$$F[\![S[\![Expr]\!]\;\emptyset\;\texttt{\$root}]\!]$$

evaluates to the same values as *Expr* in any context, that is, in any dynamic environment where an XPath expression evaluates to a result value, the same result value can be obtained from the transformed XPath.

We also plan to report runtimes of using the performance results in connection with streaming by using the various streaming subsets as backends for the transformation.

References

[1] Apache Foundation, *Xalan XSLT implementations*, http://xml.apache.org/xalan-j and http://xml.apache.org/xalan-c.

[2] C. Barton, P. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and Marcus F. Fontoura, *Streaming XPath Processing with Forward and Backward Axes*, ICDE - International Conference on Data Engineering, Bangalore, India, March, 2003.

[3] O. Becker, *Extended SAX Filter Processing with STX*, Extreme Markup Languages, August 4-8, 2003, http://www.idealliance.org/papers/extreme03.

[4] A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon, *XML Path Language (XPath) 2.0*, W3C Working Draft, August 2003, http://www.w3.org/TR/2003/WD-xpath20-20030822.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau (editors), *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, February 2004, http://www.w3.org/TR/2004/REC-xml-20040204.

[6] J. Clark, S. DeRose, *XML Path Language (XPath) Version 1.0*, W3C Recommendation, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[7] A. Desai, *Introduction to Sequential XPath*, Proc. of IDEAlliance XML Conference, 2001, http://www.idealliance.org/papers/xml2001/papers/html/05-01-01.html.

[8] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P. Wadler, *XQuery 1.0 and XPath 2.0 Formal Semantics*, W3C Working Draft, August, 2003, http://www.w3.org/TR/2003/WD-xquery-semantics-20030822/.

[9] M. Fernández, D. Suciu, *Optimizing Regular Path Expressions Using Graph Schemas*, In Proc. of the Fourteenth International Conference on Data Engineering, pages 14-23, Orlando, Florida, Feb. 1998.

[10] A. K. Gupta, D. Suciu, *Stream Processing of XPath Queries with Predicates*, In Proc. of the ACM SIGMOD International Conference on Management of Data, pages 419-430, San Diego, California, 2003.

[11] IBM and BEA, *Service Data Objects*, November 2003, http://www.ibm.com/developerworks/java/library/j-commonj-sdowmt; also JSR 235, http://www.jcp.org/en/jsr/detail?id=235.

[12] L. V. S. Lakshmanan and P. Sailaja, *On Efficient Matching of Streaming XML Documents and Queries*, In Proc. of the Extending Database Technology International Conference, Prague, Czech Republic, March 2002.

[13] D. Olteanu, H. Meuss, T. Furche, F. Bry, *XPath: Looking Forward*, In Proc. of the EDBT Workshop on XML Data Management (XMLDM), 2002.

[14] F. Peng, S. S. Chawathe, *XPath Queries on Streaming Data*, In Proceedings of the ACM SIGMOD International Conference on Management of Data. June 2003. San Diego, California.

[15] P. Wadler, *Two semantics for XPath*, January 2000, http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.

[16] R. Whitmer (editor), *Document Object Model (DOM) Level 3 XPath Specification*, W3C Working Group Note, February 2004, http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226.