Eliminating Context State from XPath

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

October 6, 2003

Abstract

XPath is a language for selecting sequences of nodes and computing values from XML document trees; specifically XPath can express *navigation* in an XML document. A peculiarity of the XPath semantics is that every expression is defined in terms of not just the current *context node* in the tree but also the *context position* and *context size* that depend on the sequence from which the current node was extracted. In this paper we show how any expression involving the context position "position()" and context size "last()" can be transformed into an equivalent expression that does not, and we explain to which extent the transformed expression is larger than the original.

1 Introduction

XPath [4] was introduced as part of the W3C "XSLT" transformation language [3] to have a non-XML format for selecting nodes and computing values from an XML document. (For a gentle introduction to XPath see one of the numerous books on XSLT; for a more formal presentation see [12].) Since then XPath has become part of several other standards, in particular the forthcoming XPath Version 2.0 [2] forms the "navigation subset" of the also forthcoming XQuery XML database access language.

In this section we introduce XPath, explain the main contribution of the paper, give some pointers to related work, and present a plan for the main part of the paper.

XPath. In their simplest form XPath expressions look like "directory navigation paths". For example, the XPath

$$\text{/company/personnel/employee} \tag{1}$$

navigates from the root of a document through the top-level "company" element to it's "personnel" child elements and on to it's "employee" child elements. The result of the evaluation of the entire expression is the *sequence* of all the "employee" elements that can be reached in this manner, returned in the order they occurred in the document. At each step in the navigation the selected nodes for that step can be filtered with a test and of special interest to us are the predicates that test the selected node's position in the previous step's selection. So if we ask for

$$\text{/company/personnel/employee[2]} \tag{2}$$

then the result is *all* employee elements that are the *second* employee element among the employee child elements of each personnel element selected by the previous step.

The situation becomes extra interesting when combined with XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed the above XPath is a shorthand for

$$/\text{child::company}/\text{child::personnel}/\text{child::employee}[\text{position}() = 2] \tag{3}$$

where it is made explicit that each *path step* is meant to search the "child" axis containing all children of the previous context node, and that a numeric index is really a shorthand for a *predicate* that tests the position number. If we instead asked for

$$/\text{child::company}/\text{child::personnel}/\text{child::employee}/\text{following-sibling::}*[\text{position}() \leq 2] \tag{4}$$

then the last step selects nodes of any kind that are in the first two sibling positions immediately *after* each employee.

More precisely, the result of an XPath 2.0 expression is a *sequence*: an ordered collection of zero or more items that can be XML document nodes or atomic values. The evaluation of an expression depends on a dynamic context of which the main part for the purpose of this paper is called the *focus*, composed of three components:

- The *context item* is the item currently being processed in a path step. When an expression $p[q]$ is evaluated, each item in the sequence obtained by evaluating p becomes the context item for the evaluation of q.
- The *context position*, returned by the expression "position()", is the position of the context item within the sequence of items currently being processed.
- The *context size*, returned by the expression "last()", is the number of items in the sequence of items currently being processed.

In classic interpreted models an XPath expression is evaluated by traversing the input XML tree according to each sub-expression and updating the focus along the way. This means that the run-time system needs to maintain the focus state at all times in case it is accessed.

Contribution. In this paper we propose to rewrite XPath expressions that contain context-sensitive expressions into other XPath expressions without context references. This is useful when a stateless implementation is desired. Take the position() of (3): the position among the employee children of each personnel parent (node) can be computed relative to the child by counting the preceding siblings. Specifically (3) is equivalent to

$$/\text{child::company}/\text{child::personnel}/\text{child::employee}[\text{count}(\text{preceding-sibling::employee}) + 1 = 2] \tag{5}$$

More precisely, it turns out that the position can be calculated from expressions relative to the current node and the node that generated the innermost sequence. This is because these two nodes define a clean partitioning of the complete collection of nodes. Figure 1 illustrates this (and we formalize it below).

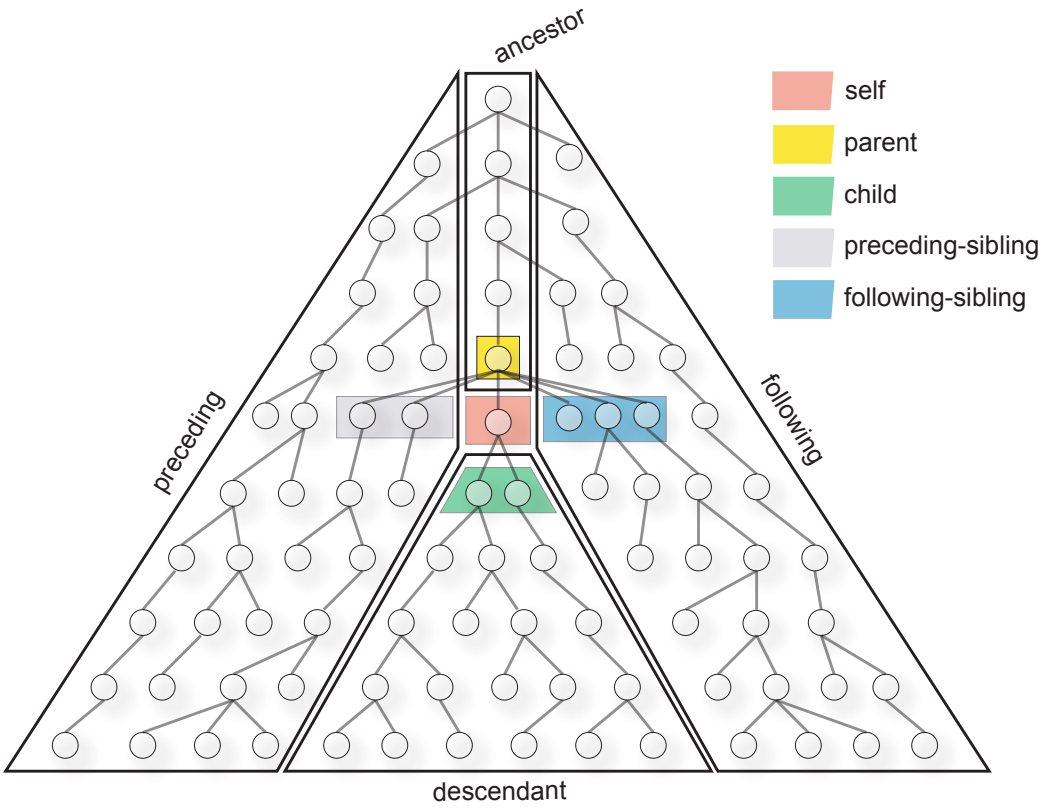

Figure 1: Partitions of document nodes from current node.

Related Work. A growing interest in optimization of path expressions has emerged during the past few years. Several methods have been proposed for rewriting XPath expressions taking integrity constraints or schemas into account [7]. The containment problem for XPath expressions has also been investigated using rewriting techniques [11].

Streaming XPath processing has recently gained considerable interest due to its application in publish-subscribe architectures [9]. A few approaches have been proposed to allow the evaluation of particular XPath queries in a stream-based context. Some consider a navigational approach consisting of a restricted subset of XPath composed of forward axes and simplified predicates [5, 8]. Some approaches have shown that it is not necessary to restrict the use of axes for progressive processing [1, 10]. By proposing rewriting techniques to turn reverse axes into forward ones, they enabled the use of the unrestricted set of XPath axes in a stream-based context. However, none of these approaches explain how the context position and -size are maintained and we believe that the present work could eliminate that burden.

Outline. In Section 2 we present the abstract syntax we consider. The next sections detail the transformation (the top level in Section 3, simple steps in Section 4, actual "position()" and "last()" instances in Section 5, and for complex steps with nested predicates in Section 6). Section 7 explains some optimizations before we briefly discuss the correctness and complexity, in Sections 8 and 9, respectively. Finally Section 10 shows some example translations before we conclude in Section 11.

UnionExpr	u	::=	$u_1 \mid u_2 \mid /p \mid p$
PathExpr	p	::=	$p_1/p_2 \mid s$
StepExpr	s	::=	$v \mid \text{let } v = u_1 \text{ return } u_2 \mid a{::}n\,l \mid u\,l$
Axis	a	::=	child | descendant | self | descendant-or-self
			| following-sibling | following | parent | ancestor
			| preceding-sibling | preceding | ancestor-or-self
NodeTest	n	::=	$name \mid * \mid \text{node}() \mid \text{text}()$
Predicates	l	::=	$[q]\ l \mid \epsilon$
Qualifier	q	::=	$q_1 \text{ or } q_2 \mid \text{not}(q) \mid u \mid u_1 \ll u_2 \mid u_1 \gg u_2 \mid e_1 = e_2$
Expr	e	::=	$e_1 + e_2 \mid e_1 - e_2 \mid k \mid v \mid \text{count}(u) \mid \text{position}() \mid \text{last}()$
Var	v	::=	$\$name$

Figure 2: Syntax of XPath expressions.

2 Abstract Syntax

The considered abstract syntax of patterns and qualifiers is as shown in Figure 2. The syntax is somewhat simplified, focusing on simple node steps (composed of one axis and one node test), complex node steps (involving an expression, with unions for example), and on the role of context-sensitive expressions "position()" and "last()". Note that what we refer to as simple and complex node steps respectively correspond to the AxisStep and FilterStep non-terminals of the XPath 2.0 grammar [2].

We have extended XPath with a "let ... return ..." borrowed from XQuery [6]. Moreover, we do not consider "attribute" and "namespace" axes as position in the document is less important for attribute and namespace nodes. ϵ stands for an empty predicate list.

3 Translation of Expressions

Our goal is to transform an XPath expression into another XPath expression without any "position()" or "last()" expressions. The translation is performed using a one-pass recursive traversal that propagates information to rewrite "position()" and "last()" accordingly.

We give the formal translation functions in Figures 3, 4, and 8. There are six primary translation functions (U, P, S, L, Q, and E) named after the non-terminal of the considered abstract syntax they translate. U and P initiate the traversal of the expression, whereas S translates a node step and L, Q, and E translate simple node steps composed of an axis and a node test. λ, ρ, and ξ are three additional translation functions which perform the same task as L, Q, and E for complex node steps involving an expression.

We write $U[\![u]\!]$ for the translation of an XPath expression u and $P[\![p]\!]$ for the translation of a path p. Note that in the way dynamic context is defined in XPath 2.0, no information needs to be propagated between two paths.

We write $S[\![s]\!]$ for the translation of a step s. The "let" expression borrowed from XQuery is a key component in our translations: "let" expressions allow us to extend the current naming capability of XPath by making it possible for qualifiers to refer to the node which was selected by the previous step and used to generate the sequence that is about to be filtered. For this purpose, the two translations $S[\![a{::}n\,l]\!]$ and $S[\![u\,l]\!]$ introduce a "let" expression, which names the context

$$
\begin{array}{rcl}
U & : & \mathit{UnionExpr} \to \mathit{UnionExpr} \\
U[\![u_1 \mid u_2]\!] & = & U[\![u_1]\!] \mid U[\![u_2]\!] \\
U[\![/p]\!] & = & /P[\![p]\!] \\
U[\![p]\!] & = & P[\![p]\!] \\
 & & \\
P & : & \mathit{PathExpr} \to \mathit{PathExpr} \\
P[\![p_1/p_2]\!] & = & P[\![p_1]\!]/P[\![p_2]\!] \\
P[\![s]\!] & = & S[\![s]\!] \\
 & & \\
S & : & \mathit{StepExpr} \to \mathit{StepExpr} \\
S[\![v]\!] & = & v \\
S[\![\text{let } v = u_1 \text{ return } u_2]\!] & = & \text{let } v = U[\![u_1]\!] \text{ return } U[\![u_2]\!] \\
S[\![a{::}n\ l]\!] & = & \text{let } v = \text{self::node() return } a{::}n\ L^a_{n,v}[\![l]\!]\epsilon \\
S[\![u\ l]\!] & = & \text{let } v = \text{self::node() return } U[\![u]\!]\ \lambda^{\epsilon}_{U[\![u]\!],v}[\![l]\!] \\
 & & \\
L & : & \mathit{Axis} \to (\mathit{NodeTest} \times \mathit{Var}) \to \mathit{Predicates} \to \mathit{Predicates} \to \mathit{Predicates} \\
L^a_{n,v}[\![[q]\ l]\!](\theta) & = & [\ Q^a_{n,\theta,v}[\![q]\!]\ ]\ L^a_{n,v}[\![l]\!](\theta[\ Q^a_{n,\theta,v}[\![q]\!]\ ]) \\
L^a_{n,v}[\![\epsilon]\!](\theta) & = & \epsilon \\
 & & \\
Q & : & \mathit{Axis} \to (\mathit{NodeTest} \times \mathit{Predicates} \times \mathit{Var}) \to \mathit{Qualifier} \to \mathit{Qualifier} \\
Q^a_{n,l,v}[\![q_1 \text{ or } q_2]\!] & = & Q^a_{n,l,v}[\![q_1]\!] \text{ or } Q^a_{n,l,v}[\![q_2]\!] \\
Q^a_{n,l,v}[\![\text{not}(q)]\!] & = & \text{not}(Q^a_{n,l,v}[\![q]\!]) \\
Q^a_{n,l,v}[\![u]\!] & = & U[\![u]\!] \\
Q^a_{n,l,v}[\![u_1 \ll u_2]\!] & = & U[\![u_1]\!] \ll U[\![u_2]\!] \\
Q^a_{n,l,v}[\![u_1 \gg u_2]\!] & = & U[\![u_1]\!] \gg U[\![u_2]\!] \\
Q^a_{n,l,v}[\![e_1 = e_2]\!] & = & E^a_{n,l,v}[\![e_1]\!] = E^a_{n,l,v}[\![e_2]\!]
\end{array}
$$

Figure 3: Translation of expressions

node, and propagate the variable name, in order for the translations of the qualifiers to use it. "v" stands for an XPath variable name which was not already used in the expression before the translation process.

4 Translation of Simple Node Steps

In XPath 2.0, simple node steps always return a sequence of nodes ordered in document order. Context positions are assigned to the items in this sequence in document order for forward axes and in reverse document order for backward axes.

The translations of "position()" and "last()" expressions only depend on the last axis, the last node test and predicates used. We write $L^a_{n,v}[\![l]\!]$ for the translation of a predicate list l contained within a simple node step composed of an axis a and a node test n. v is the variable name previously inserted with the potentially introduced "let" expression. Predicates of a node step are translated in

$$
\begin{array}{rcl}
E & : & \mathit{Axis} \to (\mathit{NodeTest} \times \mathit{Predicates} \times \mathit{Var}) \to \mathit{Expr} \to \mathit{Expr} \\
E^a_{n,l,v}[\![e_1 + e_2]\!] & = & E^a_{n,l,v}[\![e_1]\!] + E^a_{n,l,v}[\![e_2]\!] \\
E^a_{n,l,v}[\![e_1 - e_2]\!] & = & E^a_{n,l,v}[\![e_1]\!] - E^a_{n,l,v}[\![e_2]\!] \\
E^a_{n,l,v}[\![k]\!] & = & k \\
E^a_{n,l,v}[\![v_1]\!] & = & v_1 \\
E^a_{n,l,v}[\![\mathrm{count}(u)]\!] & = & \mathrm{count}(U[\![u]\!]) \\
\\
E^{\mathrm{child}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}(\text{preceding-sibling::}n\,l) + 1 \\
\\
E^{\mathrm{descendant}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}((\text{preceding::}n\,l \mid \text{ancestor::}n\,l)[\text{self::node}() \gg v]) + 1 \\
\\
E^{\mathrm{self}}_{n,l,v}[\![\mathrm{position}()]\!] & = & 1 \\
\\
E^{\text{descendant-or-self}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}((\text{preceding::}n\,l \mid \text{ancestor::}n\,l)[\text{self::node}() \gg v] \mid v/\text{self::}n\,l) + 1 \\
\\
E^{\text{following-sibling}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}(\text{preceding-sibling::}n\,l[\text{self::node}() \gg v]) + 1 \\
\\
E^{\mathrm{following}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}((\text{preceding::}n\,l \mid \text{ancestor::}n\,l)[\text{self::node}() \gg v]) \\
 & & -\ \ \mathrm{count}(v/\text{descendant::}n\,l) + 1 \\
\\
E^{\mathrm{parent}}_{n,l,v}[\![\mathrm{position}()]\!] & = & 1 \\
\\
E^{\mathrm{ancestor}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}(v/\text{ancestor::}n\,l) - \mathrm{count}(\text{ancestor::}n\,l) \\
\\
E^{\text{preceding-sibling}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}(v/\text{preceding-sibling::}n\,l) \\
 & & -\ \mathrm{count}(\text{preceding-sibling::}n\,l) \\
\\
E^{\mathrm{preceding}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}((\text{descendant::}n\,l \mid \text{following::}n\,l)[\text{self::node}() \ll v]) \\
 & & -\ \mathrm{count}(v/\text{ancestor::}n\,l) + 1 \\
\\
E^{\text{ancestor-or-self}}_{n,l,v}[\![\mathrm{position}()]\!] & = & \mathrm{count}(v/\text{ancestor-or-self::}n\,l)\ -\mathrm{count}(\text{ancestor::}n\,l) \\
\\
E^a_{n,l,v}[\![\mathrm{last}()]\!] & = & \mathrm{count}(v/a\text{::}n\,l)
\end{array}
$$

Figure 4: Elimination of context-references from simple node steps

the order they appear in the expression. Because the translation of a predicate can make use of the previous translated predicates, the l parameter holds the list of all previous translated predicates of the node step.

The axis, node test, translated predicates and the variable name introduced by the "let" expression are passed to Q that essentially propagates them to the expressions within predicates.

5 Elimination of Context References

We write $E^a_{n,l,v}[\![e]\!]$ for the translation of an expression e, shown separately in Figure 4. Position references are translated depending on the axis and the filtered node test of a step. The translations take into account that context positions are assigned in reverse document order for backward axes. Translations were built considering the tree partioning defined by the XPath axes (Figure 1). We illustrate below the four main cases, from which it is possible to derive the others. In each case, the node that generated the innermost sequence is called "p".

Position reference after an "ancestor" axis. The principle of the translation include counting the ancestors of the context item inside a predicate. This is done using a recursive call with another "ancestor" axis (Figure 5). If we name the result i, then $i + 1$ is the index of the selected node in the sequence, in document order. In order to reverse the order, i is subtracted from the translation of the context size (defined in Figure 8).

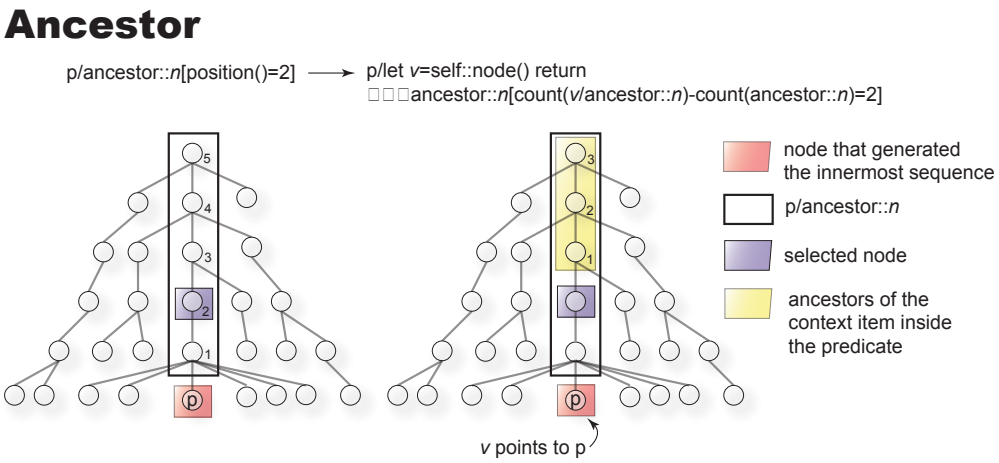

Figure 5: Position reference after an "ancestor" axis.

Position reference after a "descendant" axis The idea behind the translation is to count all the elements which are between "p" and the selected node, in document order (Figure 6).

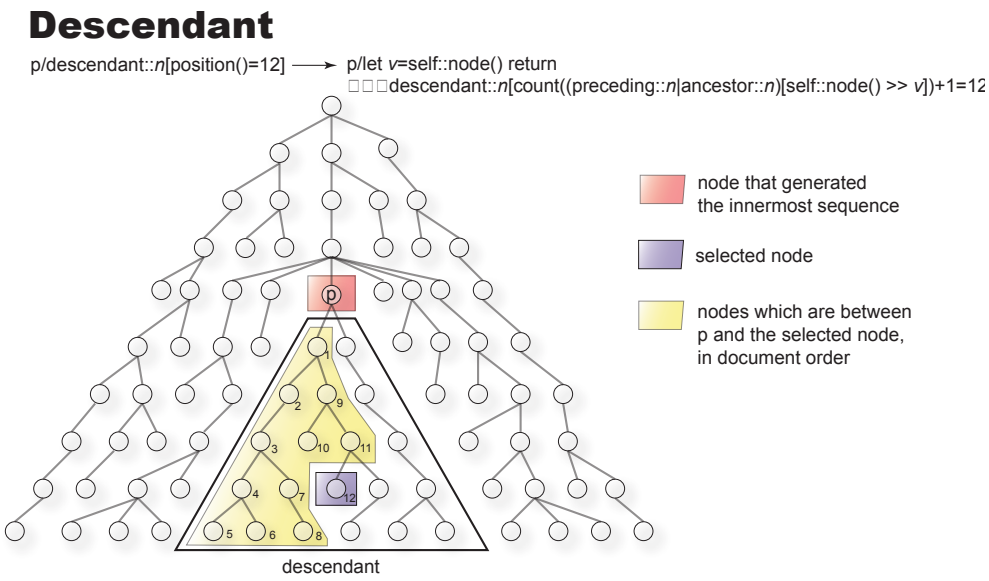

Figure 6: Position reference after a "descendant" axis.

Position reference after a "preceding" axis The principle is also to count all the elements which are between the selected node and "p", in document order. (Figure 7).

Position reference after a "following" axis This case is the symmetric to the preceding one.

Other context position references. The other cases are derived or simplified versions of these four main cases. For example, the idea behind the "child" case is similar to the "descendant" one: "child" simply additionally takes into account that all the nodes between "p" and the selected node, in document order, are child of "p". Note that this case does not even require us to introduce a variable, since "p" can be reached using the "parent" axis. (Some simplifications are given in Section 7.)

Context size references. Owing to the ability to refer to "p" all context size references ("last()") can be eliminated in the same way: by counting the number of items in the corresponding sequence. Note that other translations exist; the two examples in Section 10 in particular show some other ways to proceed.

$$E^{\text{following}}_{n,l,v}[\![\text{last}()]\!] = \text{count}(/\text{descendant::}nl[\text{self::node}() \gg v]) - \text{count}(v/\text{descendant::}nl)$$

$$E^{\text{preceding-sibling}}_{n,l,v}[\![\text{last}()]\!] = \text{count}(\text{parent::node}()/\text{child::}nl[\text{self::node}() \ll v])$$

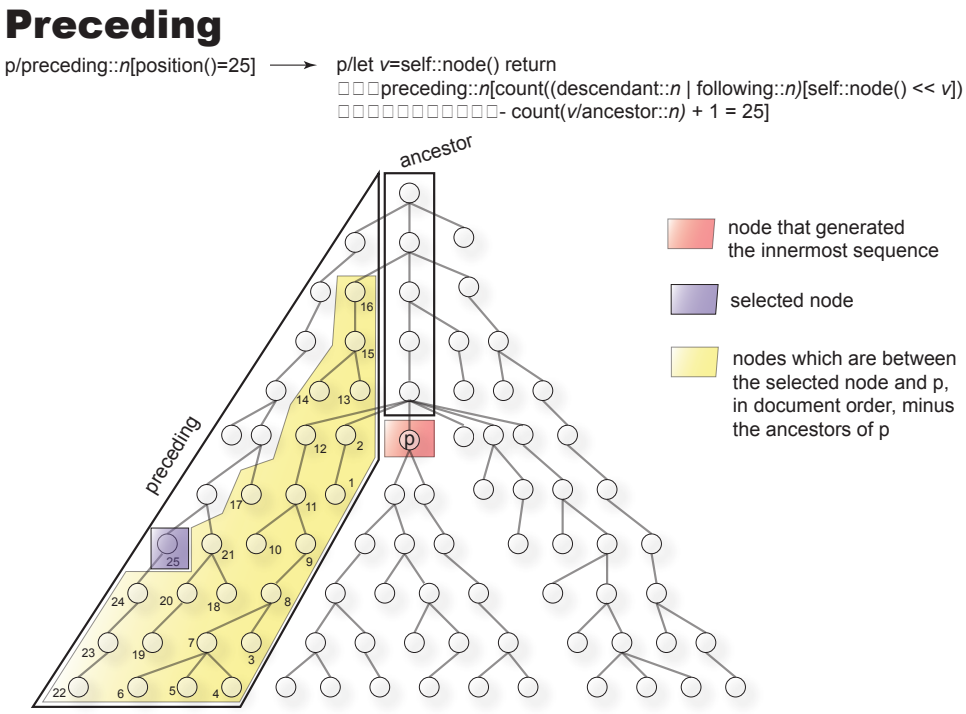

Figure 7: Position reference after a "preceding" axis.

6 Translation of Complex Node Steps

XPath 2.0 defines three operators in order to combine node sequences: *union*, *intersection*, and *except* [2]. We consider the union operator in this paper. Our way of translating complex node steps is based on the fact that the result sequence of this operator is in document order.

The idea behind the translation of context position reference is to count all the elements which are selected by the same sequence and are before the current node in document order. Since intersection and except operators also return their result sequence in document order, our approach can also handle them in the same way.

Translation of context size references rely on the same principle also used to translate them in simple node steps.

Note that XPath 2.0 also defines a way to construct a sequence by using the *comma* operator, which evaluates each of its operands and concatenates the resulting values, in order, into a single result sequence. We do not deal with the comma operator in this paper but leave it for future work. However, we believe that our approach can scale to translate node steps involving comma operators. The basic idea is to evaluate the size of each comma operand by applying the appropriate translation; then, using multiple XPath conditional expressions that refer to these sizes, translations of each operand can be combined to return the corresponding integers.

$$
\begin{array}{rcl}
\lambda & : & \mathit{Predicates} \to (\mathit{UnionExpr} \times \mathit{Var}) \to \mathit{Predicates} \to \mathit{Predicates} \\
\lambda^{\theta}_{u,v}[\![[q]\ l]\!] & = & [\ \rho_v^{(u)\theta}[\![q]\!]\ ]\ \lambda^{(\theta[\ \rho_v^{(u)\theta}[\![q]\!]\ ])}_{u,v}[\![l]\!] \\
\lambda^{\theta}_{u,v}[\![\epsilon]\!] & = & \epsilon
\end{array}
$$

$$
\begin{array}{rcl}
\rho & : & \mathit{StepExpr} \to \mathit{Var} \to \mathit{Qualifier} \to \mathit{Qualifier} \\
\rho^s_v[\![q_1 \text{ or } q_2]\!] & = & \rho^s_v[\![q_1]\!] \text{ or } \rho^s_v[\![q_2]\!] \\
\rho^s_v[\![\text{not}(q)]\!] & = & \text{not } (\rho^s_v[\![q]\!]) \\
\rho^s_v[\![u]\!] & = & U[\![u]\!] \\
\rho^s_v[\![u_1 \ll u_2]\!] & = & U[\![u_1]\!] \ll U[\![u_2]\!] \\
\rho^s_v[\![u_1 \gg u_2]\!] & = & U[\![u_1]\!] \gg U[\![u_2]\!] \\
\rho^s_v[\![e_1 = e_2]\!] & = & \xi^s_v[\![e_1]\!] = \xi^s_v[\![e_2]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\xi & : & \mathit{StepExpr} \to \mathit{Var} \to \mathit{Expr} \to \mathit{Expr} \\
\xi^s_v[\![e_1 + e_2]\!] & = & \xi^s_v[\![e_1]\!] + \xi^s_v[\![e_2]\!] \\
\xi^s_v[\![e_1 - e_2]\!] & = & \xi^s_v[\![e_1]\!] - \xi^s_v[\![e_2]\!] \\
\xi^s_v[\![k]\!] & = & k \\
\xi^s_v[\![v_1]\!] & = & v_1 \\
\xi^s_v[\![\text{count}(u)]\!] & = & \text{count}(U[\![u]\!]) \\
\xi^s_v[\![\text{position}()]\!] & = & \text{let } v_1 = \text{self::node() return count}(v/s[\text{self::node()} \ll v_1]) + 1 \\
\xi^s_v[\![\text{last}()]\!] & = & \text{count}(v/s)
\end{array}
$$

Figure 8: Translation of complex node steps

7 Simplifications

We have presented the translations for the general case. However, note that some translations do not need to refer to the node that generated the innermost sequence. In particular, translations of context references after the "child", "self" and "parent" axes do not involve such references. Consequently, optimized versions L', Q' and E' can be built as a replacement for L, Q and E so that no let expression is introduced in the corresponding node step, and v is dropped as a parameter:

$$
\begin{array}{rcl}
S[\![\text{child::}n\ l]\!] & = & \text{child::}n\ L'^{\text{child}}_n[\![l]\!]\emptyset \\
S[\![\text{self::}n\ l]\!] & = & \text{self::}n\ L'^{\text{self}}_n[\![l]\!]\emptyset \\
S[\![\text{parent::}n\ l]\!] & = & \text{parent::}n\ L'^{\text{parent}}_n[\![l]\!]\emptyset
\end{array}
$$

The translations of context-size references can also be simplified for these cases:

$$
\begin{array}{rcl}
E'^{\text{child}}_{n,l}[\![\text{last}()]\!] & = & \text{count(parent::node()/child::}nl) \\
E'^{\text{self}}_{n,l}[\![\text{last}()]\!] & = & 1 \\
E'^{\text{parent}}_{n,l}[\![\text{last}()]\!] & = & 1
\end{array}
$$

Moreover, in XPath host languages such as XSLT, "child::n[position() = 1]" is a common pattern. We suggest a simplification of our translation for this frequently handled case:

$$
Q'^{\text{child}}_{n,l}[\![\text{position}() = 1]\!] \quad = \quad \text{not(preceding-sibling::}nl)
$$

8 Correctness

Given a formal semantics of XPath we can describe the formal correctness of the transformation. Consider, for example, the formal semantic function $\mathcal{S}$ defined by Wadler for XPath 1 [13]: given an XPath u and a context node x (from which the entire document containing it can be reached), $\mathcal{S}[\![u]\!]\,x$ specifies the node sequence selected by u. In this notation our transformation is correct if

$$\forall x : \mathcal{S}[\![u]\!]\,x = \mathcal{S}[\![U[\![u]\!]]\!]\,x \tag{6}$$

If instead we use the notation from the XPath/XQuery formal semantics [6] then we need to prove the inference

$$\frac{\text{dynEnv} \vdash \left[u\right]_{\text{Expr}} \Rightarrow r}{\text{dynEnv} \vdash \left[U[\![u]\!]\right]_{\text{Expr}} \Rightarrow r} \tag{7}$$

that is, in any dynamic environment where an XPath expression u evaluates to a result value r, the same result value can be obtained from the transformed XPath.

To prove either we shall need an induction that "peels off" the compositional layers of each set of rules beyond the scope of this note.

9 Complexity

To understand the complexity of translated expressions we explain the points where the translated expression is nontrivially *larger* than the original. As expressed above, our translations involve four kinds of duplication, i.e., cases where some part of the input expression is copied into different places of the translated expression. These cases are summarized below:

1. Node step duplication in context-size elimination from simple node steps. The axis and filtered node test are both present in the translated node step and in the translation of the context-size reference occuring in the predicates of the node step. For example,

   $$p/\text{descendant::}nl[\text{last}() < 2] \tag{8}$$

   will be rewritten into:

   $$p/\text{let } v = \text{self::node}() \text{ return descendant::}nl[\text{count}(v/\text{descendant::}nl) < 2] \tag{9}$$

   where the simple node step descendant::$n\,l$ is present twice.

2. Node step duplication in context references elimination from complex node steps, for a similar reason: $S[\![u\,l]\!]$ computes $U[\![u]\!]$ which is both used in the translation of the node step and in the translation of its qualifiers.

3. Predicate duplication (for simple and complex node steps). Translations of the predicates that occur between the nodetest and the currently translated predicate are duplicated. For example the expression

   $$\text{child::}n[a][b][\text{position}() = 2] \tag{10}$$

   is rewritten into

   $$\text{child::}n[a][b][\text{count}(\text{preceding-sibling::}n[a][b]) = 2] \tag{11}$$

   where node test n and predicates $[a][b]$ are duplicated.

4. Filtered node test duplication in context position elimination from simple node steps. The filtered node test can appear multiple times in the translation. For example $E^{\text{descendant}}_{n,l,v}$ uses both the node test n and predicate list l twice.

We believe all duplication cases where an expression is duplicated (i.e., cases 1 and 2) can be avoided by taking advantage of the naming capability introduced by the "let" expression. Indeed, we believe that for such cases we can name the duplicated expression, then replace the common sub-expressions by referring to the variable name instead of duplicating. For example, consider the first duplication case and the expression (8). Instead of rewriting it into (9), we can get rid of the duplication by using a let expression and the variable name in place of the common sub-expressions. The expression (8) is then rewritten into:

$$p/\text{let } v = \text{descendant::}n\,l \text{ return } v[\text{count}(v) < 2] \tag{12}$$

Translation functions must consequently be optimized in order to insert "let" expression appropriately. The translation of a simple node step, $S[\![a::n\ l]\!]$, is modified in order to add a second "let" expression, which binds the common step. For the second duplication case, $S[\![u\,l]\!]$ is also modified in order to introduce the appropriate "let" expression. $U[\![u]\!]$ becomes the binded expression instead of self::node(), and only the variable name is passed as a parameter to an optimized translation function λ'.

Note that a step of the form $v\,l$ (*Var Predicates*) must also be considered, at least in the output grammar, to be able to generate such steps in the translated expression. Considering the general construction $v\,l$ in our abstract syntax lead to considering the problem of its general translation. The translation of a step $v\,l$ must take into account that v can hold a sequence with any kind of order (introduced by either reverse axes or the comma operator). If the variable is previously bound in the same expression, it is possible to find the binding and translate the context references by applying the appropriate translation functions relatively to the binded expression.

10 Examples

Consider the XPath expression $//a[b][\text{last}()]$. This expression uses the abbreviated XPath syntax and stands for: /descendant-or-self::node()/child::a[child::b][position() = last()]. It is intended to select the a elements, child of a descendant from the root, which have a child element b and whose position is the last among their siblings. Using our method this expression is translated into the expression

$$//a[b][\text{count}(\text{preceding-sibling::}a[b]) + 1 = \text{count}(../a[b])] \tag{13}$$

which does not contain context-references anymore. The XPath abbreviated syntax ".." stands for parent::node(). (13) is the translated expression using simplifications described in section 7. The optimized translation of (13) is:

$$//\text{let } v = a[b] \text{ return } v[\text{count}(\text{preceding-sibling::}a[b]) + 1 = \text{count}(v)] \tag{14}$$

The more complicated XPath expression $a/(b \mid c)[\text{position}() = 2]$ returns the sequence composed of all the b and c elements, which are in second position in document order, and are child of an element a. This expression contains another expression as a step. It is rewritten into:

a/let $v = (b \mid c)$
return v[let $v_1 =$ self::node()
return count(v[self::node() $\ll v_1$]) + 1 = 2]

11 Conclusion

The result of this paper consists in showing how to eliminate the context state from XPath. For this purpose, we considered an XPath 2.0 fragment extended with XQuery's "let" expression. We then proposed to rewrite an expression into another one that does not contain context-references anymore. The main application of this is to liberate implementations from having to keep track of context information that will never be used or that it is inconvenient to keep track of.

References

[1] 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.

[2] 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.

[3] J. Clark, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

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

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

[6] 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/.

[7] 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.

[8] 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.

[9] Laks 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.

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

[11] J-Y. Vion-Dury, N. Layaïda, *Containment of XPath expressions: an Inference and Rewriting based approach*, Extreme Markup Languages, August 4-8, 2003.

[12] P. Wadler, *A Fornal Semantics of Patterns in XSLT*, March 2000, http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xslt-semantics.pdf

[13] P. Wadler, *Two semantics for XPath*, January 2000, http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xpath-semantics.pdf