Abstract. XML Schema defines identity constraints, including key and foreign key constraints. We consider the problem of navigating from a keyed element to its referring "children" and, conversely, from an element containing a foreign key to the "parent" it references. Our contributions are as follows. We extend an expressive fragment of XPath, called \textit{XPath}'\textsuperscript{′}, with navigation axes according to foreign keys, resulting in a fragment called \textit{XPath}'\textsubscript{fk}. We show a polynomial time algorithm for the evaluation of \textit{XPath}'\textsubscript{fk} queries. We prove that \textit{XPath}'\textsubscript{fk} is strictly more expressive than \textit{XPath}'\textsuperscript{′}. Finally, we show that under certain conditions, if keys are restricted to have only one field then \textit{XPath}'\textsubscript{fk} is equivalent to \textit{XPath}'\textsuperscript{′}.

1 Introduction

XPath 1.0, a W3C recommendation [1], is a query language for XML documents. XPath enables navigation (in an XML document) and selection of nodes, and is used in many XML applications such as XSLT [8]. Navigation from a node to other nodes is done according to several axes, that allow navigation from a node to its parent, ancestors, children, descendants, following and preceding nodes as well as following and preceding siblings (in depth-first order). In this paper we introduce additional axes to XPath 1.0\textsuperscript{1}. For the rest of the paper, we will use simply XPath when referring to XPath 1.0 (unless stated otherwise).

One of the useful mechanisms provided by XML Schema [3] is the ability to define identity constraints, including keys and foreign keys. A key definition \(<\text{key}>\text{tag}\)> appears inside an element definition. This element is called the scope of the key. The key definition imposes constraints on the sub-tree of the scoping element. The key definition includes a selector expression and one or more field expressions. These expressions are XPath expressions that conform to a simple fragment of XPath. They do not contain predicates, and in each path the first location step may be ".//" , but the other steps may only be 'child' steps. Also, for a field expression the path may end with an attribute. The selector expression is evaluated with an instance of the scoping element as a context node, to produce a set of nodes which we call the target node set. For each node in the target node set, every field expression must evaluate (relative

\textsuperscript{1} The new axes are also useful for XPath 2.0.
to the node) to a node set containing exactly one node, of a simple type. Within an instance of the scoping element, there must not exist two distinct nodes of the target node set that have the same sequence of field values.

A keyref definition is very similar to a key definition. It appears inside of the definition of a scoping element and specifies selector and field expressions. It also specifies the key to which it refers. For each node of the target node set of the keyref there must exist a node in the node set of the key that has the same sequence of field values.

For convenience of notation, we sometimes call the instances of a scoping element (of a key or a keyref) ‘scoping nodes’ or ‘scope instances’. We sometimes call the nodes obtained by evaluating a selector expression in the context of some scoping node ‘selector-identified nodes’.

Consider the schema depicted (informally) in Figure 1. A Sale element describes a sale made by a specific sales person (identified by SalesPersonID) to a specific customer (identified by CustNumber) on a specific date. We assume that at most one sale is made on a specific date to a specific customer by a specific sales person. For every such sale there is an invoice. The status of an invoice may be ‘in preparation’, ‘issued’, ‘in progress’, ‘paid’ or ‘deal complete’. Suppose the schema contains a key definition, within the scope of the Sales element, with a selector expression that selects Sale elements, and with fields (SalesPersonID, CustNumber, Date). Suppose further that the schema contains a keyref definition within the same scope, with a selector expression that selects Invoice elements, and with fields (SalesPersonID, CustNumber, Date). Suppose we wish to find all Sale elements whose corresponding Invoice has a ‘paid’ status. There is no easy way (and possibly no way at all!) in XPath to navigate from Sale elements to their corresponding Invoice elements and

\footnote{Had XML Schema ([3]) allowed using the parent axis in the definition of key fields then the Sale.SalesPersonID element would not be needed and the key would be (parent :: SalesPerson/ID, CustNum, Date).}
vice versa. We introduce foreign-key navigation axes to facilitate such navigation. With these axes, we can simply navigate from elements specified in 
//Invoice[Status = "paid"] directly to the Sale elements that those Invoice elements reference. Suppose the keyref is called KR, then we simply write 
//Invoice[Status = "paid"]/KR_Parent. Similarly, to navigate from Sale elements to their associated paid invoices, we can write //Sale/KR_Children[Status = "paid"].

The semantics of foreign key references, as described in [4], is quite complex:

• These references are local to a scoping node of the keyref. Suppose n’ is a selector-identified node of a keyref scoping node n. Then n’ is considered as referencing nodes that are selector-identified nodes of the key (and have the same field values as n), and whose scoping node is a descendant of n.

• In a valid document, every selector-identified node of a keyref references (within a scoping node) exactly one selector-identified node of a key. To ensure this, there is a mechanism that resolves conflicts. Let n be a scoping node of a keyref KR that refers to a key K. There is a table, associated with n, which holds K’s selector-identified nodes that may be referenced by KR’s selector-identified nodes whose scoping node is n. For each such node the table holds the node’s key-sequence (i.e., the values of its fields). In order to construct the table for n, we compute the union of the tables of n’s children. Also, if n is a scoping node of K, we add its selector-identified nodes, and key sequences, to the combined table. Then, if the combined table contains two or more rows with the same key-sequence ks (and different nodes), this is considered a conflict. The conflict is resolved as follows. All nodes with key-sequence ks that were added from the children’s tables are removed.

• Consider, for example, the tree in Figure 2. Suppose that a is a scoping node of a keyref. e is a selector-identified node of a (i.e., e is reachable from a via the selector expression of the keyref). c1, c2 and c3 are scoping nodes of the relevant key. d11 and d12 are selector-identified nodes of c1. d21 and d22 are selector-identified nodes of c2. d3 is a selector-identified node of c3. Selector-identified nodes of a can reference d3 but they cannot reference d11, d12, d21 or d22. This is because when key information “percolates” bottom up, d11 and d21 cancel each other out and similarly d12 and d22 cancel each other out. This means that in a valid document, a selector-identified node of the keyref (whose scoping node is a) cannot have a key-sequence of (1,2). It can have the key-sequence (3,4), but that would mean that it references d3 and not d12 or d22. In our example, e references d3 but does not reference d12 or d22. If we were to change the key-sequence of e from (3,4) to (1,2) then the document would become invalid.

One might question the necessity of foreign-key axes. In many cases the need for such navigation may be eliminated by adding ‘artificial’ id fields (attributes or elements) that uniquely identify the (selected) elements that contain the key

\(^{3}\) And is not apparent at first reading.
fields. In the sales example above, we might add a new SaleID element (or attribute) as a child element (or attribute) of the Sale element, and of the Invoice element. However, this solution is very cumbersome. It is also redundant to add such elements for each key and foreign key defined in the schema. Furthermore, the values of these artificial elements or attributes need to be carefully maintained so that they remain unique and correspond to the actual key and keyref fields to which they relate. Maintenance is particularly difficult if XML data files are independently maintained, and especially in a distributed environment. Hence, the solution of adding new axes seems more practical.

Instead of adding foreign-key axes to XPath 1.0, one might use Xquery. In the example of Figure 1, instead of writing \[ //Invoice\[Status = "paid"\]/KR\]Parent to retrieve the Sale elements of paid invoices, we can use the following Xquery expression.

```xquery
for $s in document("my_doc.xml")//Sale
for $i in document("my_doc.xml")//
  Invoice[Status="paid"
  and SalesPersonID=$s/SalesPersonID
  and CustNum=$s/CustNum
  and Date=$s/Date]
return $s
```

This query is also a legal XPath 2.0 query (XPath 2.0 includes a limited form of the Xquery FLWOR expressions, containing only For clauses and Return).

There are several advantages to using foreign-key navigation axes and not XQuery in such scenarios:

- Foreign-key navigation axes enable information retrieval in a simple navigational language that does not use variables.
• The XQuery expression may become very cumbersome. Accommodating the semantic complexities of foreign key references, described above, is non-trivial. This example is rather simple, but in more complicated examples (i.e., when the key and foreign key are defined in different scoping elements, and the scoping elements may have more than one instance in the document), the XQuery expression becomes more complicated. The expression might get even more complicated because it must identify nodes that are not selector-identified nodes but have the same labels as such nodes, and avoid performing navigation from these nodes.

• The syntax of foreign-key navigation axes is more intuitive, and makes it clear that we are retrieving information according to a key and foreign key defined in the schema and not simply according to the values of arbitrary fields.

• The use of foreign-key navigation axes can help a 'smart' query processor to execute the query more efficiently, using the constraints imposed on the document by the key and foreign key definitions.

Contributions. (1) We define a syntax for axes that navigate according to foreign key constraints defined in XML Schema. (2) We find bounds on the complexity of XPath query evaluation in the presence of these axes. (3) For a substantial fragment of XPath, which we call $XPath'$, we show that foreign-key navigation axes enable writing queries that can not be expressed otherwise. (4) For $XPath'$, we show conditions under which foreign-key navigation axes can be expressed using the standard axes. (5) We examine the expressive power of $XPath'$ with the addition of operators for union and intersection.

Related Work. To the best of our knowledge, there has not been much work on adding new axes to XPath. One example of such work is [19] and [18], where a new 'conditional axis' is introduced. This axis can be used for queries such as "find a descendant $y$ with $\@x=val1$ such that all the nodes on the path to $y$ have $\@x=val2$". Such queries have similar semantics to that of the temporal logic operator 'until’. This 'conditional axis’ adds expressive power to a path language called XCore, which is very similar to Core XPath (defined in [14]).

The expressive power of several XPath fragments is analyzed in [17]. The largest of these fragments, which is similar to Core XPath, is proven to be equivalent to positive existential first order logic in two free variables (one to represent the context node and another to represent the output of the query). The XPath dialect defined in [19] and [18] by introducing conditional axes is proven to be equivalent to full first order logic. It is important to observe that the XPath fragments treated in these papers ([17], [18], [19]) do not include comparison of values - the fragments of [17] do not include an equality operator at all and the dialect defined in [18] uses an equality operator only to check the label of a node and not its string value. Thus these fragments (and the logic languages to which they are equivalent) cannot express navigation according to foreign keys, since the comparison of node values is essential to such navigation and to the basic semantics of keys and foreign keys. It is also important to note that the set of labels in these fragments is finite. Node labels can be used to represent
values (i.e., a node of label "x" and text value "a" would be represented as a node of label "x" having a child node of label "a"), but if there is a finite set of labels than only text nodes whose text is taken out of a finite domain of strings can be represented. As explained in section 2, when the field values come from a finite domain, foreign-key navigation can, in many cases, be expressed via simple XPath queries.

In [15], a formalism for definition of keys (which is independent of XML Schema) is suggested, and decision problems are studied. These problems are satisfiability (is there a document that satisfies the key constraints?) and implication (if a document satisfies a set of constraints, does this imply satisfaction of another constraint?)

2 Foreign-Key Navigation Axes

Foreign-key navigation axes are defined as follows. Suppose the schema contains definitions for a key $K$ and a keyref $KR$ that refers to $K$. For some document $D$, let the set of selector-identified nodes of $K$ be $X$, and let the set of selector-identified nodes of $KR$ be $Y$. Two new axes, $KR_Children$ and $KR_Parent$, are implicitly defined. $KR_Children$ allows navigation from $X$ nodes to the $Y$ nodes that contain foreign key references to them (i.e., that are their “children” according to the relation defined by the key constraint) and $KR_Parent$ allows navigation from $Y$ nodes to the $X$ nodes whose keys they contain. Foreign-key-navigation axes may appear wherever an XPath axis may appear. These axes adhere to the semantics of foreign key references, described in section 1.

It is legal to write $P/KR_Children$ where $P$ is an expression that does not necessarily navigate to the $X$ nodes. When $KR_Children$ is applied to a context set, it navigates from nodes of $X$ that appear in the context set to the corresponding nodes of $Y$. If there are no $X$ nodes in the context set, then the application of the axis returns an empty set of nodes.

Consider the XML Schema $SchemaCD$ depicted in Figure 3. In an XML document that conforms to this schema, the combination of $f_1$ and $f_2$ must be unique for a $C$ node, and for each $D$ node there must exist a $C$ node with the same $f_1$ and $f_2$ values. Figure 5 shows $BaseXML$, a specific XML document that conforms to this schema.

Foreign-key-navigation axes navigate between the elements defined by the selector expression of the key definition and the elements defined by the selector expression of the keyref definition. In this example these are $/R/A/C$ and $/R/B/D$, respectively. For instance, the query $//C[Name = "b"][f1 = "3"]/KR_Children$ selects the $D$ nodes which reference a $C$ node with $Name$ "b" and $f1 = 3$. In $BaseXML$ this is $D1$. The query $//D[f1 = "3"][f2 = "2"]/KR_Parent/Name/text()$ returns "a".

Note that a selector-identified node of a keyref may have more than one scoping node. In such cases it is possible, though uncommon, for a node to reference different nodes (selector-identified nodes of a key) with respect to different

---

4 For simplicity, we disregard namespaces throughout this paper.
Fig. 3. SchemaCD.

scoping nodes (of the keyref). In such cases, our foreign-key navigation axes regard only the references that occur within the scoping node which is highest in the document tree (i.e., the scoping node which is an ancestor of the other scoping nodes). We disregard references that are only valid with respect to the other ('lower' in the tree) scoping nodes. Another option would be to consider references that occur in all scoping nodes of the selector-identified node of the keyref\(^5\).

Also note that if the domain of field values is finite then basic foreign-key navigation can be performed by traversing all possibilities of field values. For example, suppose we constrain the \(f_1\) values to be only 1 or 3 and the \(f_2\) values to be 2 or 4. Then instead of writing \(//C[Name = "b"]//C[Name = "b"]\) we can write

\[
//D[ (f_1 = "1" \text{ and } f_2 = "2" \text{ and } //C[Name = "b"][f_1 = "1" \text{ and } f_2 = "2"] ) \text{ or } (f_1 = "3" \text{ and } f_2 = "4" \text{ and } //C[Name = "b"][f_1 = "3" \text{ and } f_2 = "4"] ) \text{ or } ... ].
\]

In other words, we explicitly check for every possible combination of field values in a \(D\) node whether there exists a \(//C[Name = "b"]\) node with the same field

\(^5\) And query evaluation would be done in a very similar manner to the one described in section 3.
values. Note, however, that to fully emulate foreign-key navigation axes, we need to accommodate the complex semantics described in section 1.

3 Complexity of query evaluation

We examine the problem of evaluating a query \( Q \) on a document \( D \) that conforms to an XML Schema \( S \). We denote the the number of symbols in the string representation of the document \( D \) (respectively, the query \( Q \) and the schema \( S \)) by \( |D| \) (respectively, \( |Q| \) and \( |S| \)). A simple polynomial-time algorithm for evaluating an XPath query is presented as algorithm 6.3 in [13]. In this section we briefly describe the algorithm and explain how it can be adapted to evaluate queries that may also contain foreign-key navigation axes. We show that for such queries the algorithm still runs in polynomial time. We assume that the queries are evaluated on documents that are valid with respect to a schema (in which keys and foreign keys are defined), otherwise foreign-key navigation is meaningless.

The algorithm uses the concept of context-value tables. A context value table for an expression \( e \) contains a row for every possible context on which \( e \) may be evaluated. A context is a tuple \((x,k,n)\), where \( x \) is a context node, \( k \) is its position in the (ordered) context set and \( n \) is the size of the context set. For every such context the table contains the result of evaluating \( e \) on the context (the result may be a node-set, a number, a string or a boolean value). A query is represented as a parse tree of sub-expressions: the root represents the query and each node has children representing its sub expressions. For example, if the query is descendant::b[position()\(!=\)last()]), the root node has a child node that represents the sub-expression position()\(!=\)last(), and that node has two child nodes that represent the sub-expressions position() and last(). The algorithm traverses the parse tree bottom-up. First, it computes context-value tables for the leaves. Then, it moves up the tree - for every node whose children’s tables have been computed, it computes the node’s table based on those of its children. For example, if we have context value tables for the expressions \( E1 \) and \( E2 \), the table for \( E3=E1/E2 \) is constructed as follows. For each possibility of \((x,k,n)\) to be installed in table \( E3 \), we fetch a row \( ((x,k,n),S) \) from the table of \( E1 \), where \( S \) is a node-set. For every node \( y \) whose position in \( S \) is \( k^y_1 \), we fetch a row \( ((y,k^y_1,S),S^y_1) \) from the table of \( E2 \). The value of the row for \((x,k,n)\) in the table for \( E3 \) is the union of these \( S^y_1 \) node-sets. In order to construct a table for the expression \( E1 \ RelOp \ E2 \), where \( RelOp \in \{<,>,=,\neq,\leq,\geq\} \), for each possible \((x,k,n)\) we fetch a row \( ((x,k,n),S1) \) from the table of \( E1 \) and a row \( ((x,k,n),S2) \) from the table of \( E2 \). The value of the row for \((x,k,n)\) in the table of \( E3 \) is \textit{true} if and only if there is a node \( s1 \) in \( S1 \) and a node \( s2 \) in \( S2 \) such that \( s1 \ RelOp s2 \).

The algorithm continues until it reaches the root of the parse tree. For a query \( Q \) and a document \( D \) it runs in time \( O(|D|^3 \ast |Q|^2) \).

In order for the algorithm to evaluate queries with foreign-key navigation axes, we need to provide a method of constructing context-value tables for
the new foreign-key axes, i.e., for expressions of the form $KR\_Parent::t$ and $KR\_Children::t$, where $KR$ is a foreign key and $t$ is either * or a label. Once we provide such a method, the algorithm works as described above, since the only change is in the computation of tables for the leaves of the query parse tree.

In order to construct the context-value table for $KR\_Parent::t$ (respectively, $KR\_Children::t$), we need to evaluate $KR\_Parent::t$ (respectively, $KR\_Children::t$) for every node of the document. We do this efficiently via lookups in special tables that we construct in a pre-processing stage, which consists of the following stages:

1. We obtain the instances of the scoping element (i.e., the scoping nodes) of each key and foreign key. This is done as follows. We use an algorithm that is similar to a validation algorithm presented in [21]. Given a schema $S$, we construct finite deterministic automata that represent the structural constraints imposed by the schema, and use these automata to locate the scoping nodes through a traversal of the document. The algorithm returns a list of scoping nodes for every key and keyref defined in $S$.

Given a schema $S$, we assign unique names to all anonymous complex types that appear in the schema. Then, for each complex type $T$ we construct a regular expression $R_T$ that represents the structure of $T$. The symbols of $R_T$ are the elements (labels) that appear in $T$. We ignore attributes, since we assume the documents on which we evaluate queries are valid. For each regular expression $R_T$ we construct a Glushkov automaton $A_{R_T}$ (see 2.2.1 in [21]). Aside from the initial state, the states of $A_{R_T}$ are positions in $R_T$. Each such position corresponds to an element $e$ that appears in $T$ ($e$ being the element’s label). With every state we associate the type $T$ of the corresponding element and also the keys $\{K_i\}$ and keyrefs $\{KR_i\}$ that are defined in the element. Note that there may be several elements with the same label $e$ and different types defined inside different types. We consider all simple types as type $T_{\text{simple}}$. After constructing the automata, we create a mapping of every global (i.e., top level) element $e$ of type $T$ to the automaton $A_{R_T}$ that corresponds to the type $T$. We also save sets of the keys and keyrefs defined in the element. Note that there are no two global elements with the same name. Therefore we can simply save a mapping of element names (labels) to automata (and to lists of keys and keyrefs).

Our algorithm is based on the validation algorithm presented in section 3.1 of [21]. The algorithm performs a depth-first traversal of the document tree. At each stage it keeps an automaton for every "open" element (i.e., an element whose sub-tree has not yet been fully traversed). The current states of the running automata are saved in a stack. On encountering the root node, of label $r$, we start running the corresponding automaton (which we find according to the element name). As this automaton encounters the first child (i.e., sub-element) of $r$, it moves from its initial state to a new state $q$. Recall that when constructing the automaton, we associated with $q$ a type $T$. If $T$ is not $T_{\text{simple}}$, we start running the corresponding automaton $A_{R_T}$. We continue traversing the tree. After we finish running an automaton
for some node (i.e., traverse the node’s sub-tree), we make a transition in the automaton of the parent node.

We construct the algorithm output as we go along. For the root node \( r \), we have a set of keys \( S_K \) and a set of keyrefs \( S_{KR} \) that we access according to the label \( r \) (recall that we save these sets after creating the automata). For every key \( k \in S_K \), we add the root node to the list of \( k \)’s scoping nodes. For every keyref \( kr \in S_{KR} \), we add the root node to the list of \( kr \)’s scoping nodes. When, during the traversal of the document, we reach a non-root node \( n \), this is done in a transition to some state \( q \) (possibly a transition from \( q \) to \( q \)) of the automaton run at \( n \)’s parent node. Let \( S_{Kr} \) and \( S_{KR} \) be the sets of keys and keyrefs, respectively, associated with \( q \). Then, for every key \( k \in S_{Kr} \), we add \( n \) to the list of \( k \)’s scoping nodes. For every keyref \( kr \in S_{KR} \), we add \( n \) to the list of \( kr \)’s scoping nodes.

We handle various features of XML Schema:

- Sequences are represented as concatenations in the regular expression. For example a sequence of an element \( e_1 \) with minOccurs=0, maxOccurs=unbounded, and of an element \( e_2 \) with minOccurs=1, maxOccurs=2, is represented as \( e_1^*e_2(e + e_2) \). If the aforementioned sequence appears with maxOccurs=unbounded then the regular expression would be \( (e_1^*e_2(e + e_2))^* \).
- A choice in the schema is represented using ‘+’ in the regular expression.
- If an xs:all appears in the schema, we need to write all possibilities in the regular expression. For example an xs:all between elements \( e_1, e_2 \) and \( e_3 \) is translated into the expression \( e_1e_2e_3 + e_1e_3e_2 + e_2e_1e_3 + e_2e_3e_1 + e_3e_1e_2 + e_3e_2e_1 \).
- If a type contains a reference to a model group then we write the content explicitly and then generate the regular expression.
- If a type contains an \(<xs:element ref="a"/>\), for some global element \( e \), \( e \) appears in the regular expression and \( e \)’s type is associated with the corresponding state of the automaton.
- Substitution groups: Suppose some (global) element \( e_1 \) is the head of a substitution group whose members are the (global) elements \( e_2, ... , e_k \). Whenever a ‘ref’ to \( e_1 \) appears in some type, instead of using \( e_1 \) in the corresponding regular expression, we use \( (e_1 + e_2 + e_3) \).
- If a type \( T' \) derives from a type \( T \), in order to construct the regular expression \( R_{T'} \) we write the content of \( T' \) explicitly and then translate it into a regular expression.
- If an element \( e \) appears in the document with an xs:complexType specification, xs:complexType=\( T \), we use the automaton \( A_{R_T} \) instead of the automaton to be used according to the description of the algorithm above. Note that xs:complexType is the only mechanism through which an element appears in the document with a different type than the type specified in the schema (specifically, with a type derived from the one specified in the schema).
- If a type contains an xs:any element, it means that any well formed XML element may appear. Let \( l_1, ..., l_n \) be the labels of elements that appear in the schema. Then xs:any is represented in the regular expression by \( (l_1 +\)
...+I_n+\gamma), where \gamma represents an ‘unknown’ symbol. In the corresponding automaton there are \(n+1\) states that correspond to the positions of the \texttt{xs:any} representation. With each of these states we associate the type \texttt{T\_any} and a NULL automaton. In run time, if we get to one of these states then we do not go down the current branch, since anything may appear there. In a pre-processing stage, we replace any label of the document that does not appear in the schema with \(\gamma\).

Now we discuss the time complexity of the algorithm. The construction of a Glushkov automaton is quadratic in the size of the regular expression. Therefore the first stage of our algorithm, in which we translate a schema into automata, can be done in time \(O(|S|^4)\). According to [21], running the algorithm for a document \(D\) takes time \(O(|D| \log |S|)\) \(^6\). Therefore the complexity of obtaining the scoping nodes is \(O(|S|^4) + O(|D| \log |S|)\). For a fixed schema, the complexity is \(O(|D|)\).

2. For each key \(K\) we create a table \(\text{tbl}_K\), in which we include a row for each selector-identified node of \(K\) (i.e., a node reachable via evaluating \(K\)’s selector expression in the context of some scoping node). In the row we save the node, its key-sequence (i.e., the values of its fields) and its scoping nodes (there may be more than one). The table is created as follows. For each scoping node of \(K\) (obtained in the previous stage), we evaluate the selector expression. For every resulting node \(n\), if \(n\) already appears in the table then we add the current scoping node to the set of scoping nodes saved in the row for \(n\). Otherwise, we evaluate the field expressions in the context of \(n\) and add a new row to the table. For each keyref \(KR\) we create a similar table \(\text{tbl}_{KR}\).

We now consider the time complexity of this construction. There may be \(O(|D|)\) instances of the scoping element. For each such instance we evaluate the selector expression \(K_{Sel}\). Since \(K_{Sel}\) is in Core XPath ((defined in [14])), its evaluation time is \(O(|D| * |K_{sel}|)\) (see [13]), which is also \(O(|D| * |S|)\). For every selector-identified node, we evaluate the field expressions, which are also in Core XPath. It costs \(O(|D| * |S|)\) to evaluate all field expressions for a single selector-identified node (because the sum of expression sizes is \(O(|S|)\)). So we get \((\text{Number of scoping nodes}) \times (\text{Evaluation of the selector expression for a scoping node}) \times (\text{Evaluation of the field expressions for a selector-identified node}) = O(|D| * |D| * |S|) = O(|D|^3 * |S|^2)\). Since the schema has \(O(|S|)\) keys, the complexity is \(O(|D|^3 * |S|^3)\).

3. For every key \(K\) and node \(n\) we create a table \(n.K_{Inf}\). In this table we keep selector-identified nodes of \(K\), whose scoping node is either \(n\) itself or a descendant of \(n\) (which means these selector-identified nodes are descendants of \(n\), and possibly \(n\) itself). The table holds rows of the form (node-id, key-sequence). A node \(n'\) appears in the table only if it is valid for some node (that is a selector-identified node of some keyref that refers to \(K\)) to reference it. These tables are similar to the ones described in section 3.11.5

\(^6\) The algorithm presented in [21] handles DTDs and a basic subset of XML Schema while we handle also more advanced features of XML Schema. However, our algorithm’s execution is very similar.
of [4]. The table is built in a bottom-up traversal of the document tree. For each node $n$ we build its table for a key $K$ in the following manner. First we compute the union of its children’s tables. Then we add the selector-identified nodes whose scoping node is $n$ to the table. If some key-sequence appears in more than one row of the table (with different nodes) then we omit the occurrences that came from the children’s tables (If there is an occurrence of the key-sequence in a selector-identified node whose scope is $n$, we leave it in the table). This omission is intended to prevent ambiguity of foreign-key references. It is consistent with the semantics describes in [4]. We obtain the selector-identified nodes whose scope is $n$ by fetching rows from $tbl_K$ where $n$ is listed as a scoping node. This construction takes time $O(|D|^4)$. Getting the selector-identified nodes for a node $n$ costs $O(|D|)$ (going over $tbl_K$). The higher complexity stems from the complexity of performing the union operations. If we union two tables at a time, we perform $O(|D|^4)$ union operations and each union operation costs $O(|D|^{3/2})$, because when creating the union of two tables, for each row of one table we need to go over the other table to check if the key-sequence appears there. Each table has $O(|D|)$ rows, and comparing two key-sequences takes time $O(|D|)$ (since we compare key-sequences that appear in the document and therefore their lengths is $O(|D|)$).

Therefore the construction of the $K.Info$ tables for all keys in the schema takes time $O(|D|^4 \times |S|)$

Figure 4 shows an example of the tables created during the pre-processing stage. In this example $a$ and $a'$ are scoping nodes of a foreign key $KR$ and $c_1$, $e_2$, $e_1'$ and $e_2'$ are their selector-identified nodes. $c_3$, $c_1$, $c_2$, $c_1'$ and $c_2'$ are the scoping nodes of a key $K$ and $d_3$, $d_{11}$, ..., $d_{22'}$ are their selector-identified nodes. $c_3$ is a scoping node that has other scoping nodes as children. Some of its selector-identified nodes are also selector-identified nodes of its child scoping nodes. Note that the key selector-identified nodes with key-sequence (1,2) cannot be referenced since this key-sequence appears both in $c_1$ and $c_2$ and also both in $c_1'$ and $c_2'$. This is why this sequence does not appear in $a.K.Info$ and $a'.K.Info$.

Following the pre-processing stage, we proceed to constructing the context-value tables. In order to do this, we need to evaluate the foreign-key navigation axes. For a context node $x$, we evaluate $KR.Children$ as follows. We fetch a row $r$ from $tbl_K$ according to the node-id of $x$. $r$ contains a key-sequence $ks$ and a set of scoping nodes (instances of $K$’s scoping element) $KSN$. According to $ks$, we fetch a set of $O(|D|)$ rows $R$ from $tbl_KR$. Each row $r_1$ in $R$ contains a node-id $r_1.nodeId$, a key-sequence $r_1.ks$ (which is identical to $ks$) and a set $r_1.KRSN$ of scoping nodes (of the foreign key). If there is more than one node in $r_1.KRSN$ then one of them is the ancestor of all the others (since the scoping nodes are all ancestors of the node with id $r_1.nodeId$). Let $n_{r_1}$ be this ancestor node. Now we check whether $n_{r_1}.K.Info$ contains a row for the node $x$. If so, then the node whose id is $r_1.nodeId$ is included in the result set (the result of evaluating $KR.Children$ for $x$).

Now we consider the time complexity of evaluating $KR.Children$ for a context
node $x$. Searching $tbl_{KR}$ according to the key-sequence takes time $O(|D|^2)$, since there are $O(|D|)$ rows in the table and for each row we compare key-sequences of $O(|D|)$ symbols (these are key-sequences that appear in the document and therefore their length is at most the length of the document). Then we $n_{r1}$ for $O(|D|)$ rows $r1$, which takes $O(|D|^2)$ time. So the evaluation of $KR_{Children}$ for a single context node takes $O(|D|^2)$, and the construction of the context-value table for $KR_{Children}$ (of $O(|D|^3)$ rows) takes $O(|D|^5)$.

For a context node $x$, we evaluate $KR_{Parent}$ as follows. We fetch a row $r$ from $tbl_{KR}$ according to the node-id. $r$ contains a key-sequence $ks$ and a set of scoping nodes (instances of $KR$'s scoping element) $KRSN$. If there is more than one node in $KRSN$ then one of them is the ancestor of all the others (since the scoping nodes are all ancestors of $x$). Let $n$ be this ancestor node. Now we look in $n.K_{Info}$ and fetch rows from it according to the key-sequence $ks$. The result set for $x$ consists of the nodes whose node-ids appear in these rows.

Now we consider the time complexity of evaluating $KR_{Parent}$ for a context node $x$. Finding the ancestor node $n$ in a set $KRSN$ of $O(|D|)$ nodes can be done in time $O(|D|)$ in the following manner. Given $O(|D|)$ 'candidate' nodes, we start advancing towards the root from each 'candidate'. In each iteration we make one 'parent' step from each candidate. If from candidate $c1$ we reach another candidate $c2$ then $c1$ is no longer a candidate (since $c2$ is its ancestor).
If we reach the root from candidate \( c \) then \( c \) is the 'winner', since there is no other candidate between it and the root. Once we find \( n \), it costs \( O(|D|^2) \) to search \( n.K_N \) according to the key-sequence. Therefore it costs \( O(|D|^2) \) to evaluate \( K.R.Parent \) for a single context node. Thus, the construction of the context-value table (of \( O(|D|^3) \) rows) takes \( O(|D|^5) \).

**Theorem 1.** In the presence of foreign-key navigation axes, the complexity of evaluating a query \( Q \) on a document \( D \) that conforms to an XML Schema \( S \) is \( O(|D|^5 * |Q|^2) + O(|D|^5 * |S|^3) + O(|D|^4 * |S|) + O(|S|^4) \). For a fixed schema this complexity is \( O(|D|^5 * |Q|^2) \).

**Proof.** When evaluating a query \( Q \), we construct at most \( O(|Q|) \) context-value tables for the foreign-key navigation axes, which costs \( O(|D|^5 * |Q|) \). To this we add the complexity of the pre-processing stage and the complexity of the algorithm for query evaluation presented in [13].

Therefore, query evaluation can be done in polynomial time, as efficiently as in [13].

### 4 Expressive power

**Definition 1.** A query is a function that maps XML data trees to node sets. If \( t \) is an XML data tree and \( Q \) is a query, then \( Q(t) \) is a set of nodes extracted from \( t \).

**Definition 2.** The expressible set of an XPath fragment is the set of queries that can be expressed via expressions in that fragment. A fragment \( f \) is said to have a larger expressive power than another fragment \( g \) if the expressible set of \( f \) properly contains the expressible set of \( g \).

In section 5 we show that for a large class of schemas in which only single-field keys are allowed, one can, in many cases, rewrite expressions containing foreign-key navigation axes into equivalent expressions that use ordinary XPath. We conjecture that for multi-field keys, augmenting XPath with foreign-key navigation axes increases its expressive power. We prove this for XPath', a substantial fragment of XPath.

**Definition 3.** XPath' is the fragment of XPath defined via the following grammar:

- \( locpath ::= '/'locpath | locpath '/'locpath | locpath | locstep \)
- \( locstep ::= axis::'ntst'['bexpr']...['bexpr']\)
- \( bexpr ::= bexpr 'and' bexpr | bexpr 'or' bexpr | 'not('bexpr')' | locpath | comparison \)
- \( comparison ::= compoperand '=' compoperand \)
- \( compoperand ::= locpath | value \)
- \( value ::= number | string \)
- \( axis ::= 'self' | 'child' | 'parent' | 'descendant' | 'descendant-or-self' | 'ancestor' | 'ancestor-or-self' \)
In other words, the following restrictions are imposed on full XPath:

- Use of the 'following', 'following-sibling', 'preceding' and 'preceding-sibling' axes is not allowed.
- Expressions contain no functions except for the Boolean function not(), and no arithmetic operations are allowed.
- Inequality ($\neq$) is not allowed (whereas equality ($=$) is allowed). The $<$, $\leq$, $>$ and $\geq$ operators are also not allowed.

Our XPath$'$ fragment is quite similar to the Core XPath fragment, defined in [14]. However, there are two important differences.

- Core XPath includes all XPath axes, whereas XPath$'$ does not include the 'following', 'following-sibling', 'preceding' and 'preceding-sibling' axes.
- Core XPath is only navigational, in the sense that it does not include comparisons, whereas XPath$'$ includes comparisons via the '=' operator. This is an important issue, since comparisons have an existential semantics.

**Definition 4.** XPath$'_{fk}$ is an extension of XPath$'$ that includes foreign-key navigation axes. It is defined via a very similar grammar. The difference is in the definition of the 'axis' grammar element:

```
axis ::= 'self' | 'child' | 'parent' | 'descendant' | 'descendant-or-self' | 'ancestor' | 'ancestor-or-self' | 'FK_Children' | 'FK_Parent', for every possible string 'FK'.
```

Note that the axes FK_Children and FK_Parent have meaning only for a query that is evaluated on a document that conforms to a schema in which a keyref named 'FK' is defined (otherwise, these axes always return an empty set).

**Theorem 2.** The expressive power (see definition 2) of XPath$'_{fk}$ is greater than that of XPath$'$.

Overview of the proof: We show a query that can be expressed in XPath$'_{fk}$ but not in XPath$'$. Given an XML data tree that conforms to SchemaCD (see Figure 3), the query selects the $D$ nodes that reference $C$ nodes whose Name is "a". In XPath$'_{fk}$, this is written as //C[Name = "a"]//KR_Children. Intuitively, this cannot be done using an XPath$'$ expression, because in order to obtain the correct $D$ nodes, the expression must check, for a $D$ node $d$, whether there exists a $C$ node $c$ whose Name is "a" and that has both the same $f_1$ value and the same $f_2$ value as $d$. Since there are no variables in XPath 1.0, the expression cannot ensure that the $f_1$ value of $d$ and the $f_2$ value of $d$ are compared to the $f_1$ and $f_2$ values of the same $C$ node.

We prove the result using two claims:

1. We restrict the discussion to expressions that do not use constant values (i.e., the expression can compare values from the document to other values from the document but not to constant values). We show that such an expression cannot perform the required task (i.e., obtain the correct $D$ nodes) on a specific conforming document, BaseXML, depicted in Figure 5. Intuitively,
if an expression can perform the required task for all documents that conform to SchemaCD then it should be able to do so without using constants, since there are infinitely many values that may appear in the conforming documents (and so a fixed set of constants in the expression will not help).

2. Based on our proof for BaseXML, we define a set of similar documents and show that there is no expression that works correctly for all documents in the set, even if constants are allowed. We define a set of XML data trees \( T_0, T_1, \ldots \) with the same structure as BaseXML but with different values in each tree. The value of an \( f_1 \) or \( f_2 \) node in \( T_i \) is \( 10^4 \ast \) (the value of the corresponding node in BaseXML). \( T_0 \) is BaseXML. For example, the value of \( D_1.f_1 \) in \( T_1 \) is \( 3 \ast 10^4 = 30000 \). For each document in this set, there is no expression that works correctly without using constants. Therefore, if there is an expression that works correctly for some document in the set, it does so by using the values that appear in the document (i.e., these values appear as constants in the expression). Since there is no upper bound on the values that appear in the documents of the set, there is no one expression that works correctly for all these documents.

We prove claim (1) above in the following manner. We denote the fragment of XPath that disallows constants by XPath′_NoConst. We prove that every predicate in XPath′_NoConst gives the same result for the nodes \( D_1 \) and \( D_2 \) of the document BaseXML. Therefore, there is no XPath′_NoConst expression that selects the \( D \) nodes of BaseXML that reference \( C \) nodes whose Name is "a". If there was such an expression \( e \), it would select \( D_2 \) but not \( D_1 \). This means \( e \) could be used to construct a predicate that distinguishes between \( D_1 \) and \( D_2 \), which is a contradiction. Given a query \( e \) that, regardless of the context, evaluates to a node set that contains \( D_2 \) but not \( D_1 \), a predicate that distinguishes between \( D_1 \) and \( D_2 \) can be written as \([self = e]/D\).

Intuitively, a predicate that uses no constants can only use comparisons of \( f_1 \) and \( f_2 \) values of \( D \) nodes to \( f_1 \) and \( f_2 \) values of \( C \) nodes in order to distinguish between \( D_1 \) and \( D_2 \). However, BaseXML is constructed in a way that "confuses" predicates and prevents them from distinguishing between different \( D \) nodes. The following properties hold for BaseXML:

- \( \text{distinct_values}(///C/f_1) = \text{distinct_values}(///C[Name = "a"]/f_1) = \text{distinct_values}(///C[Name = "b"]/f_1) = \text{distinct_values}(///D/f_1) = \{1, 3\} \)
- \( \text{distinct_values}(///C/f_2) = \text{distinct_values}(///C[Name = "a"]/f_2) = \text{distinct_values}(///C[Name = "b"]/f_2) = \text{distinct_values}(///D/f_2) = \{2, 4\} \)

Let \( p \) be a location path, \( comp \) a comparison (of the form \( p_1 = p_2 \) where \( p_1 \) and \( p_2 \) are location paths) and \( n \) a node. We denote the set of nodes obtained from evaluating \( p \) in the context of \( n \) by \( p(n) \) and the Boolean result of evaluating \( comp \) in the context of \( n \) by \( comp(n) \). In order to prove that an XPath′_NoConst predicate cannot distinguish between \( D_1 \) and \( D_2 \), we prove that (a) For every location path \( p, p(D_1) = \emptyset \) if and only if \( p(D_2) = \emptyset \); and (b) For every comparison \( comp, comp(D_1) = comp(D_2) \). Therefore, the predicate gives the same result for \( D_1 \) and for \( D_2 \). This proves claim (1) and completes the proof. \( \square \)
A detailed proof of theorem 2 is presented in section B.

5 Foreign-key navigation axes with single-field keys

In section 4 we showed that foreign-key navigation axes cannot be expressed in XPath. Now we analyze the case where the key and foreign key have only one field. Intuitively, foreign-key navigation is "easier" in such cases. We show that in many cases expressions that use these axes (i.e., expressions in XPath\textsuperscript{fk}) can be rewritten into XPath\textsuperscript{'} expressions (that do not).

Let $K$ be a single-field key, defined in some schema $S$. Let $K.Sel$ be $K$’s selector expression and let $f.exp$ be $K$’s (only) field expression. Let $KR$ be a foreign key that references $K$. Let $KR.Sel$ be $KR$’s selector expression and let $g.exp$ be $KR$’s (only) field expression. For XPath$\textsuperscript{fk}$ expressions that use the foreign-key navigation axes $KR.Parent$ and $KR.Children$, we show equivalent expressions that do not use these axes, under the following assumptions:

1. $K$ and $KR$ are defined in the same scoping element, and this element is not recursive (i.e., a scoping node cannot appear as a child of another scoping node).
2. There is a XPath' expression $ScopeExp$ that returns the instances of this scoping element (i.e., the scoping nodes) when operating in the context of the root.

3. There is an expression $K_{Sel_{exact}^{-1}}$, such that if $n'$ is a selector-identified node of a scoping node $n$ of the key $K$, $K_{Sel_{exact}^{-1}}(n') = \{n\}$. In other words, $K_{Sel_{exact}^{-1}}$ is guaranteed to return only the correct scoping node.

4. There is an expression $KR_{Sel_{exact}^{-1}}$ such that if $n'$ is a selector-identified node of a scoping node $n$ of the keyref $KR$, $KR_{Sel_{exact}^{-1}}(n') = \{n\}$.

**Definition 5.** The reverse expression of an XPath expression $e$, where $e$ does not start with ‘/’, is an expression $e^{-1}$ such that for every two nodes $n$ and $n'$, $n' \in e(n)$ if and only if $n \in e^{-1}(n')$. The reverse expression of an XPath expression $e$, where $e$ does not start with ‘/’, is an expression $(e)^{-1}$ such that for every node $n'$, if $n' \in e(root)$ then $(e)^{-1}(n') = \{\text{root}\}$, otherwise $(e)^{-1}(n') = \emptyset$, where root is the root node of the document. Note that $(e)^{-1}$ is equivalent to $e^{-1}[\text{not}(\text{parent} :: *)].$

Note that $(e)^{-1}$ can be used to check whether a node is reachable from the root via $e$, since $(e)^{-1}(n')$ is nonempty if and only if $n' \in e(root)$.

Every XPath' expression $e$ has a reverse expression. Let $e$ be the expression $axis_1 :: ntest_1[bexp_1]/...$

$\quad /axis_{n-1} :: ntest_{n-1}[bexp_{n-1}]/axis_n :: ntest_n[bexp_n]$. Then $e^{-1}$ is the expression $self :: ntest_n[bexp_n]$

$\quad /axis_{n-1} :: ntest_{n-1}[bexp_{n-1}]/.../axis_{2}^{-1} :: ntest_1[bexp_1]/axis_{1}^{-1} :: *$. \(^7\)

Note that this would not be correct if XPath' allowed functions such as last() and position(), whose result depends on the context set. If a predicate that uses these functions appears in an expression, we cannot use the predicate in the reverse expression, since its context set would be different.

Examples: If $e = l_1/l_2[l_3]/following :: l_4[l_5]$ then $e^{-1} = self :: l_4[l_5]/preceding :: l_2[l_3]/parent :: l_1/parent :: *$. If $e = \text{descendant} :: l_1/\text{preceding} :: l_2/l_3/\text{parent} :: l_1/\text{ancestor} :: l_3/\text{descendant} :: l_4$ then $e^{-1} = self :: l_2/\text{ancestor} :: l_1/\text{ancestor} :: * | self :: l_4/\text{ancestor} :: l_3/\text{ancestor} :: *$. Here, the $l_i$'s are labels.

**Theorem 3.** Let $P$ and $Q$ be some XPath' expressions. Then the following hold:

- Let $e_1$ be the expression $/P/KR_Children/Q$. Then $e_1$ is equivalent to the expression $e_2$, where $e_2 = ScopeExp/KRSel[gExp = KRSel_{exact}^{-1}/KSel[(/P)^{-1}]/fExp]/Q$.

- Let $e_3$ be the expression $/P/KR_Parent/Q$. $e_3$ is equivalent to the expression $e_4$, where $e_4 = ScopeExp/KRSel[fExp = KSel_{exact}^{-1}/KRSel[(/P)^{-1}]/gExp]/Q$.

\(^7\) $axis_i^{-1}$ is the opposite of $axis_i$. For example, $\text{child}^{-1}$ is $\text{parent}$.  

18
Note that $e_2$ and $e_4$ may contain the union operator $|$ in non-top level positions. However, as explained in section A, there is an equivalent XPath' expression that uses the union operator only at the top level.

**Proof.** We may use ScopeExp in $e_2$ and $e_4$ due to assumption 2. We may use $KR_{Sel}^{-1}$ due to assumption 3. We may use $KR_{Sel}^{-1}$ due to assumption 4. We explain the equivalence of the expressions.

- Equivalence of $e_1$ and $e_2$. The expression $/P/KR.Children$ navigates to selector-identified nodes of $KR$, that reference selector-identified nodes of $K$ which are reachable via $/P$. Therefore, we need to navigate to ScopeExp/$KR.Sel$ and then use a predicate to choose the correct $KR$ selector-identified nodes. We need to compare $g_{exp}$ to the $f_{exp}$ of the appropriate $K$ selector-identified nodes. The 'appropriate $K$ selector-identified nodes' are those that appear in the same scoping node as the current $KR$ selector-identified node (this is due to assumption 1), and are reachable via $/P$. To obtain these nodes, we navigate to the scoping node via $KR_{Sel}^{-1}$ exact and then execute the selector expression $K.Sel$. We use $(/P)^{-1}$ to make sure that we obtain only $K$ selector-identified nodes that are reachable via $/P$, i.e., reachable from the root via $P$. Therefore, the expression $/P/KR.Children$ is equivalent to the expression $ScopeExp/KR.Sel[g_{exp} = KR.Sel^{-1} /K.Sel](/P)^{-1}/f_{exp}]$. Adding $/Q$, of course, maintains the equivalence.

- Equivalence of $e_3$ and $e_4$ can be argued in a similar manner. Note: The translation can be done in a similar manner when the foreign-key navigation axes are used within predicates, not necessarily in the top level of the expression. For example the expression $/P_1[P_2/KR.Children/Q]$ is equivalent to the expression $/P_1[ScopeExp/KR.Sel[g_{exp} = KR.Sel^{-1} /K.Sel](/P_1)/P_2)] /f_{exp}] /Q$. However, if $P_2$ starts with '/', i.e., $P_2$ is of the form $/P_3$ (which means $P_2$ always operates in the context of the root) then $/P_1/P_2$ is not the intended expression (we get $/P_1//P_3$). In such a case when we write $(/P_1/P_2)^{-1}$ we actually mean $(/P_3)^{-1}$ (since executing $P_2$ means executing $P_3$ at the root, no matter which expression is executed beforehand). This means that in this case $P_1$ does not appear in the predicate, which might seem as a problem if $/P_1$ evaluates to an empty set. However, this is not a problem because if $/P_1$ evaluates to an empty set then the predicate does not matter.

### 6 Conclusions

We considered enriching XPath with XML-Schema oriented axes. Specifically, we considered axes that correspond to key and keyref constraints. We defined foreign-key navigation axes and showed that for XPath', a substantial fragment of XPath, foreign-key axes add expressive power. We showed how queries using foreign-key navigation axes can be evaluated efficiently. In addition, we showed that foreign-key navigation with single-field keys is simpler, and in many common cases equivalent XPath' expressions (with no foreign-key navigation axes)
can be written. In section A we introduce a generalization of foreign-key navigation axes and show that the generalized operator also adds expressive power to XPath', but fails to capture the full semantics of foreign-key navigation axes. Furthermore, we examine adding union and intersection operators to the XPath' fragment and show that this does not increase the expressive power.

References

12. Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjunction, DTDs and variables. ICDT 2003, p. 315-329.
A Extensions

A.1 A Generalized Navigation Operator

The axes introduced in this paper enable ‘static’ navigation according to key and foreign key constraints as defined in an XML schema. They enable ‘jumping’ from a node which is one of the selector-identified nodes of a key to selector-identified nodes of a keyref and vice versa. A natural generalization of this kind of navigation is a ‘dynamic navigation’ operator:

\[
dyn_{\text{nav}}([f_1, f_2, \ldots, f_k], \text{SelectorPath}, [g_1, g_2, \ldots, g_k]),
\]

where \(\text{SelectorPath}\), \(f_i\) and \(g_i\) (1 \(\leq\) \(i\) \(\leq\) \(k\)) are XPath expressions. Suppose the operator is executed in some context set \(S_1\). Each \(f_i\), 1 \(\leq\) \(i\) \(\leq\) \(k\), is executed with respect to \(S_1\) and must navigate to a single element with a simple content, or to an attribute, for each element of \(S_1\). \(\text{SelectorPath}\) is evaluated relative to the root, resulting in some context set \(S_2\). Each \(g_i\), 1 \(\leq\) \(i\) \(\leq\) \(k\), is evaluated with respect to \(S_2\) and must navigate to a single element with a simple content, or to an attribute, when executed on an element of \(S_2\). The operator returns the elements of \(S_2\) for which there is an element of \(S_1\) such that for all \(i = 1..k\), the value of the element selected by \(g_i\) is equal to the value of the element selected by \(f_i\).

At first glance, the dynamic navigation operator seems closely related to foreign-key navigation axes. It seems that one can use this operator to simulate the foreign-key navigation axes. Suppose we have an XML schema with definitions for a key \(K\) and a keyref \(KR\). The instances of \(K\)’s scoping element are reachable via the XPath expression \(K\text{Scope}\) and the instances of \(KR\)’s scoping element are reachable via the XPath expression \(KR\text{Scope}\). \(K\)’s selector expression is \(K\text{Selector}\) and its field expressions are \(f_1, \ldots, f_k\). \(KR\)’s selector expression is \(KR\text{Selector}\) and its field expressions are \(g_1, \ldots, g_k\). Suppose there is only one scoping node (an instance of the scoping element) for \(K\) and one scoping node for \(KR\). Then, when executed in the context of \(K\)’s selector-identified nodes (i.e., \(K\text{Scope}/K\text{Selector}\)), the expression \(\text{dyn}_{\text{nav}}([f_1, \ldots, f_k], K\text{Scope}/K\text{Selector}, [g_1, \ldots, g_k])\) is equivalent to \(KR\text{Children}\). When executed in the context of \(KR\)’s selector-identified nodes (i.e., \(KR\text{Scope}/KR\text{Selector}\)), the expression \(\text{dyn}_{\text{nav}}([g_1, \ldots, g_k], K\text{Scope}/K\text{Selector}, [f_1, \ldots, f_k])\) is equivalent to \(KR\text{Parent}\). However, in most circumstances this equivalence does not hold, since the dynamic navigation operator is not aware of the semantic complexities of foreign key references (described in section 1). Also, when the \(KR\text{Children}\) axis is applied to a node which is not a \(K\) selector-identified node, or the \(KR\text{Parent}\) axis is applied to a node which is not a \(KR\) selector-identified node, an empty set must be returned. A dynamic navigation operator, that has no knowledge of the key and foreign key definitions, does not check this.
The query that, given an XML document that conforms to SchemaCD, returns the D nodes that reference a C node whose Name is "a" (see section 4), can be written as C[Name = "a"]/dyn-nav([f1, f2], D, [f1, f2]). Since this query cannot be expressed in XPath′ (as explained in section 4), the dynamic-navigation operator adds expressive power to XPath′, although it cannot capture the semantics of foreign-key navigation. Also, this operator may be very useful, as it allows navigation according to relations that are not defined at all by the schema’s foreign-key constraints.

A.2 The union operator

We suggest incorporating the union operator inline as a sub-expression of other expressions. For example: //a/b/(e1 | e2)/c/d, where a, b, c and d are some QNames (see [3]) and e1 and e2 are XPath expressions. If S is a set of nodes, S1 is the context set after executing e1 from context set S and S2 is the context set after executing e2 from context set S, then the context set after executing e1 | e2 from context set S is S1 ∪ S2. It is easy to see that in XPath′ this can be rewritten as //a/b/e1/c/d | //a/b/e2/c/d. Clearly, this variation does not add expressive power, though it makes for more succinct expressions.8

A.3 An intersection operator

We define an intersection operator ∩. Suppose that e1 and e2 are XPath′ expressions, S is a set of nodes, S1 is the context set after executing e1 from context set S, and S2 is the context set after executing e2 from context set S. Then, the context set after executing e1 ∩ e2, from context set S, is S1 ∩ S2. The following lemma shows that adding an intersection operator to XPath′ does not increase its expressive power. Note that [17] proves a similar claim (closure under intersection), but it is done for a different XPath fragment, which is not equivalent to XPath′ since it does not include comparisons (string values of nodes are disregarded in [17]).

Lemma 1. The intersection of two XPath′ (see def. 3) expressions is equivalent to the union of some XPath′ expressions that do not use intersection.

Proof. Given two XPath′ expressions e1 and e2, we present an algorithm for creating an expression which is equivalent to their intersection and is a union of XPath′ expressions. Note that if an intersection appears inside a predicate then this translation will create an expression with union inside a predicate, which is not allowed in standard XPath. However, such an expression can be rewritten so as to include union only at the top level. First, according to [20], every XPath expression can be rewritten so as not to include reverse axes (preceding,

8 The equivalence holds because XPath′ does not include functions such as position(), and therefore it does not matter if we navigate to /c once after applying e1 and once after applying e2 and then merge the results, or if we navigate to /c after merging the results of e1 and of e2.
For simplicity we assume only one predicate. The extension to more predicates is not to include it.\(^9\) Also, \(\mathit{/P/descendant} - \mathit{or} - \mathit{self} :: x\) can be written as \(\mathit{/P/descendant} :: x\ | \ \mathit{/P/self} :: x\), and once \(e_1\) is rewritten as \(e_{11} \ | \ e_{22}\), its intersection with \(e_2\) is simply \((e_{11} \cap e_2) \ | (e_{12} \cap e_2)\). Therefore, without loss of generality, the only axes included in \(e_1\) and \(e_2\) are 'child' and 'descendant'.

\(e_1\) and \(e_2\) each describe a set of paths. The intersection expression must describe the common paths of these sets. This will be done by traversing both expressions and applying the constraints they impose. We'll denote the number of location steps in \(e_1\) (respectively, \(e_2\)) by \(e_1.length\) (respectively, \(e_2.length\)) and the location steps that \(e_1\) (respectively, \(e_2\)) is comprised of by \(e_1.step[i], i = 1..e_1.length\) (respectively, \(e_2.step[j], j = 1..e_2.length\)), where each step is comprised of an axis (\(child\) or \(descendant\)), a node-test and an optional predicate.\(^{10}\) At each stage of the algorithm, \(position_1\) and \(position_2\) will hold the positions we reached so far in \(e_1\) and \(e_2\). If \(position_1 = i\) then this means we are before \(e_1.step[i]\) and if \(position_2 = j\) then this means we are before \(e_2.step[j]\).

The intersection algorithm is recursive. At each stage it tries all possibilities of combining the current steps of \(e_1\) and \(e_2\). It finds a set of possible expressions and returns an expression which is the union of these expressions. The recursive function receives as its arguments the two expressions, the current positions in them and the current intersection expression. It adds a step to the current intersection expression based on the current steps of the two expressions. We denote the addition of this step by \(\text{addStep} (expression, step)\). This function returns the new expression created after adding the location step. An empty expression is denoted by \(\text{EMPTY}\).

We denote the \textit{unification} of two location steps by \(\text{unify} (step_1, step_2)\). The unification is a location step that satisfies the constraints imposed by both steps. Suppose that \(step_1 = \text{axis}_1 :: \text{nodeTest}_1[pred_1]\) (if there is no predicate then \(pred_1 = \text{true}\)) and \(step_2 = \text{axis}_2 :: \text{nodeTest}_2[pred_2]\) (if there is no predicate then \(pred_2 = \text{true}\)). We'll denote the unified step by \(\text{axis} :: \text{nodeTest}[pred]\). If \(\text{axis}_1\) or \(\text{axis}_2\) are \(\text{child}\) then so is \(\text{axis}\). Otherwise (both are \(\text{descendant}\)) \(\text{axis}\) is \(\text{descendant}\). If \(\text{nodeTest}_1\) is \(\ast\) then \(\text{nodeTest} = \text{nodeTest}_2\). If \(\text{nodeTest}_2\) is \(\ast\) then \(\text{nodeTest} = \text{nodeTest}_1\). If neither \(\text{nodeTest}_1\) nor \(\text{nodeTest}_2\) is \(\ast\) then they must be the same, and we get \(\text{nodeTest} = \text{nodeTest}_1 = \text{nodeTest}_2\). Otherwise there is no unified expression. \(pred = pred_1 \ and\ pred_2\), since we must enforce the constraints of both steps. The algorithm is shown in Figure 6.

Some simple examples:

- \(e_1 = \mathit{/child} :: x/child :: a, e_2 = \mathit{child} :: \ast/\mathit{descendant} :: b\) : We first unify \(\mathit{child} :: x\) and \(\mathit{child} :: \ast\) into \(\mathit{child} :: x\) and add it to \(\text{INTER}\). Then we are at the last steps of both \(e_1\) and \(e_2\), which means we are at the last stage of the algorithm, so we try to unify \(\mathit{child} :: a\) and \(\mathit{descendant} :: b\). We can’t, and therefore there is no intersection.

\(^9\) We omit the details.
\(^{10}\) For simplicity we assume only one predicate. The extension to more predicates is straightforward.
findIntersection(e1, e2) {
  A := recursiveFunc(e1, e2, 1, 1, EMPTY)
  k := A |
  return A1 | A2 | ... | Ak
}

recursiveFunc(e1, e2, position1, position2, currentIntersection) {
  If (position1 = e1.length + 1) or (position2 = e2.length + 1) then
    return ∅
  If (position1 = e1.length) and (position2 = e2.length) then
    return [addStep(currentIntersection, unify(e1.step[position1], e2.step[position2]))]
  If the axis of e1.step[position1] and the axis of e2.step[position2] are both child
  then
    if there is a unification to e1.step[position1] and e2.step[position2] then
      return recursiveFunc(e1, e2, position1 + 1, position2 + 1,
        addStep(currentIntersection, unify(e1.step[position1], e2.step[position2]))
    else return ∅.
}

(We reach this stage if the axis of at least one of the two steps is descendant.)
return S1 ∪ S2 ∪ S3, where S1, S2 and S3 are defined as follows:
S1 holds the expressions we get if we proceed by unifying the current steps of
e1 and e2.
If there is a unification to e1.step[position1] and e2.step[position2] then
  S1 := recursiveFunc(e1, e2, position1 + 1, position2 + 1,
    addStep(currentIntersection, unify(e1.step[position1], e2.step[position2]))
else S1 := ∅.
S2 holds the expressions we get if we proceed by adding e1.step[position1] and
deferring e2.step[position2]:
If the axis of e2.step[position2] is descendant then
  S2 := recursiveFunc(e1, e2, position1 + 1, position2,
    addStep(currentIntersection, e1.step[position1]))
else S2 := ∅.
S3 holds the expressions we get if we proceed by adding e2.step[position2] and
deferring e1.step[position1] (similar definition to S2).
}

Fig. 6. The intersection algorithm.

- e1 = descendant : a/descendant :: c, e2 = descendant : b/descendant :: c
  : We get descendant : a/descendant :: b/descendant :: c | descendant :: b/descendant :: a/descendant :: c.

B Expressive Power: A detailed proof of theorem 2

In the following definitions and lemmas, up to and including lemma 10, all execu-
cutions of expressions are with respect to the XML data tree BaseXML (Fig-
ure 5) and all expressions are in XPath'\_NoConst. In the following lemmas we
prove that an XPath'\_NoConst expression cannot ”distinguish” between the
tree nodes D1 and D2.

Definition 6. A set S is D-symmetric if the following conditions hold:
1. If S contains some D node then it contains all D nodes.
2. If S contains some D.f1 node then it contains all D.f1 nodes.
3. If S contains some D.f2 node then it contains all D.f2 nodes.
4. If $S$ contains some $D.f1.text()$ node then it contains all $D.f1.text()$ nodes.
5. If $S$ contains some $D.f2.text()$ node then it contains all $D.f2.text()$ nodes.

Examples:

– $\{C1\}$ is D-symmetric because it does not contain any $D$ related nodes.
– $\{C1, D1.f1, D2.f1, D3.f1, D4.f1\}$ is D-symmetric because it contains all $D.f1$ nodes.
– $\{D1\}$ is not D-symmetric because it contains $D1$ but not $D2$, $D3$ and $D4$.

Lemma 2. If $P$ and $Q$ are D-symmetric sets then $P \cup Q$ is D-symmetric.

Proof. Suppose that $R = P \cup Q$. $R$ contains some $D$ node $\Rightarrow$ $P$ or $Q$ contain some $D$ node $\Rightarrow$ $P$ or $Q$ contain all $D$ nodes $\Rightarrow$ $R$ contains all $D$ nodes. Similarly for $D.f1$, $D.f2$, $D.f1.text()$ and $D.f2.text()$. \hfill $\Box$

Next, we define the concept of C-symmetry. Intuitively, a set of nodes is C-symmetric if it maintains symmetry between the $C$ nodes of the same Name value, i.e., for every Name value (i.e., $a$ and $b$), if it contains a component of one $C$ node with that Name value (i.e., the node itself or one of its descendants) then it must contain the corresponding components of all the other $C$ nodes with the same Name value.

Definition 7. A set $S$ is called C-symmetric if for all $n \in \{a, b\}$, the following conditions are satisfied:

1. If $S$ contains a $C$ node with Name $= n$ then $S$ contains all $C$ nodes with Name $= n$.
2. If $S$ contains a $C.f1$ node of a $C$ node with Name $= n$ then $S$ contains all $C.f1$ nodes of $C$ nodes with Name $= n$.
3. If $S$ contains a $C.f2$ node of a $C$ node with Name $= n$ then $S$ contains all $C.f2$ nodes of $C$ nodes with Name $= n$.
4. If $S$ contains a $C.Name$ node of value $n$ then $S$ contains all $C.Name$ nodes of value $n$.
5. If $S$ contains a $C.f1.text()$ node of a $C$ node with Name $= n$ then $S$ contains all $C.f1.text()$ nodes of $C$ nodes with Name $= n$.
6. If $S$ contains a $C.f2.text()$ node of a $C$ node with Name $= n$ then $S$ contains all $C.f2.text()$ nodes of $C$ nodes with Name $= n$.
7. If $S$ contains a $C.Name.text()$ node of value $n$ then $S$ contains all $C.Name.text()$ nodes of value $n$.

Examples:

– $\{D1\}$ is C-symmetric because it does not contain any $C$ related nodes.
– $\{D1, C1.f1, C4.f1\}$ is C-symmetric because it contains both $C1.f1$ and $C4.f1$.
– $\{C3\}$ is not C-symmetric because it contains $C3$ but not $C2$ (that has the same Name as $C3$).
Lemma 3. If P and Q are C-symmetric sets then $P \cup Q$ is C-symmetric. □

Next, we define the concept of D-identicalness. Intuitively, two sets of nodes $S_1$ and $S_2$ are D-identical if their intersection is D-symmetric, and there are two $D$ nodes $D_i$ and $D_j$ such that the non-identical parts of the sets consist of $D_i$-related nodes ($D_i$, its child nodes ($f_1$ and $f_2$) or its grandchild text nodes) in $S_1$ and corresponding $D_j$-related nodes in $S_2$. In other words, the sets may differ only with respect to $D_i$ and $D_j$. A special case of this is when $S_1$ and $S_2$ are identical, so that their intersection equals both sets, and therefore the sets are required to be D-symmetric.

Define $W_{ij} = \{(P,Q) \mid P \text{ contains some of the } D_i\text{-related nodes } D_i,f_1, \ldots, D_i,f_1.text(),D_i,f_2,D_i,f_2.text() \text{ and } Q \text{ contains the } D_j\text{-related nodes that correspond to the nodes in } P\}$. For example, if $P$ contains $D_i,f_1.text()$ then $Q$ contains $D_j,f_1.text()$. Some of the members of $W_{ij}$ are $\{(D_i),\{D_j\}\}$; $\{(D_i,D_i,f_1),\{D_j,D_j,f_1\}\}$; $\{(D_i,D_i,f_2),\{D_j,D_j,f_2\}\}$; $\{(D_i,D_i,f_1,D_i,f_2),\{D_j,D_j,f_1,D_j,f_2\}\}$ etc.

Definition 8. Two sets of nodes $S_1$ and $S_2$ are called D-identical if one of the following holds:

1. $S_1$ and $S_2$ are identical and $S_1$ is D-symmetric.
2. There are two distinct $D$ nodes $D_i$ and $D_j$ and two sets of nodes $T_1$ and $T_2$ such that $S_1$ contains $T_1$, $S_2$ contains $T_2$, $S_1 \setminus T_1$ is identical to $S_2 \setminus T_2$ and $S_1 \setminus T_1$ is D-symmetric. $T_1$ and $T_2$ may contain only nodes related to $D_i$ and $D_j$, respectively, i.e., $(T_1,T_2) \in W_{ij}$.

$(D_i,D_j)$ are called the diff-base of $(S_1,S_2)$.

Examples:

- $\{D_1\},\{D_2\}$ are D-identical, with diff-base=$(D_1,D_2)$.
- $\{D_1,D_2,D_3,D_4,D_2.f_1,D_2.f_2.text()\},\{D_1,D_2,D_3,D_4,D_3.f_1,D_3.f_2.text()\}$ are D-identical, with diff-base=$(D_2,D_3)$.
- $\{D_1,D_1.f_1\},\{D_1,D_2,f_1\}$ are not D-identical because $\{D_1\}$ is not D-symmetric.
- $\{D_1\},\{D_1\}$ are not D-identical because they are not D-symmetric.

The $D_i$ and $D_j$ nodes in this definition are some two $D$ nodes that appear in $Base\text{XML}$ (see Figure 5).

Lemma 4. If $P_1,Q_1$ are D-identical, $P_2,Q_2$ are D-identical and diff-base$(P_1,Q_1)=\text{diff-base}(P_2,Q_2)$, then $P_1 \cup P_2$, $Q_1 \cup Q_2$ are D-identical and have the same diff-base. □

Next, we define the concept of C-identicalness which is similar to D-identicalness, but here the nodes with respect to which the two sets may differ are two arbitrary $C$ nodes, $C_i$ and $C_j$, that have the same Name value. The intersection of the sets must be C-symmetric.

\footnote{If $P$ is D-identical to $Q$ with a diff-base of $(D_i,D_j)$ then $Q$ is D-identical to $P$ with a diff-base of $(D_j,D_i)$.}
Define \( W_{ij} = \{(P,Q) \mid P \text{ contains some of the } C_i\text{-related nodes } C_i.f_1, C_i.f_1.\text{text}(), C_i.f_2, C_j.f_2.\text{text}(), C_i.\text{Name}, C_i.\text{Name.\text{text}()} \text{ and } Q \text{ contains the } C_j\text{-related nodes that correspond to the nodes in } P \} \) For example, if \( P \) contains \( C_i.f_1.\text{text}() \) then \( Q \) contains \( C_j.f_1.\text{text}() \). Some of the members of \( W_{ij} \) are \((\{C_i\}, \{C_j\})\); \((\{C_i, C_i.f_1\}, \{C_j, C_j.f_1\})\); \((\{C_i, C_i.f_2\}, \{C_j, C_j.f_2\})\); \((\{C_i, C_i.f_1, C_i.f_2\}, \{C_j, C_j.f_1, C_j.f_2\})\).

**Definition 9.** Two sets of nodes \( S_1 \) and \( S_2 \) are called \( C \)-identical if one of the following holds.

1. \( S_1 \) and \( S_2 \) are identical and \( S_1 \) is \( C \)-symmetric.
2. There are two distinct \( C \) nodes \( C_i \) and \( C_j \) that have the same \( \text{Name} \) value and two sets of nodes \( T_1 \) and \( T_2 \) such that \( S_1 \) contains \( T_1 \), \( S_2 \) contains \( T_2 \), \( S_1 \setminus T_1 \) is identical to \( S_2 \setminus T_2 \) and \( S_1 \setminus T_1 \) is \( C \)-symmetric. \( T_1 \) and \( T_2 \) may contain only nodes related to \( C_i \) and \( C_j \), respectively, i.e., \((T_1, T_2) \in W_{ij}\). \((C_i, C_j)\) are called the diff-base of \((S_1, S_2)\).

Examples:

- \( \{C_3, C_3.\text{Name}\}, \{C_2, C_2.\text{Name}\} \) are \( C \)-identical, with diff-base=\((C_3, C_2)\).
- \( \{C_1, C_2.\text{f1}\}, \{C_1, C_3.\text{f1}\} \) are not \( C \)-identical, because \( \{C_1\} \) is not \( C \)-symmetric (since it contains only one of the two \( C \) nodes whose \( \text{Name} \) is "b").

**Lemma 5.** If \( P_1, Q_1 \) are \( C \)-identical, \( P_2, Q_2 \) are \( C \)-identical and diff-base\((P_1, Q_1)\) = diff-base\((P_2, Q_2)\), then \( P_1 \cup P_2, Q_1 \cup Q_2 \) are \( C \)-identical and have the same diff-base. \( \square \)

**Definition 10.** We use the following notation for XPath expressions. A navigation step is of the form \( \text{Axis::NodeTest} \), i.e., it is similar to a location-step but without predicates. A navigational-expression is a sequence of navigation steps and predicates. A comparison is an expression of the form \( e_1 = e_2 \) where \( e_1 \) and \( e_2 \) are navigational-expressions or constants. A predicate is written as \( \text{[predExpr]} \), where \( \text{predExpr} \) is an expression (a navigational-expression, a comparison or a combination of such expressions joined by logical connectors) that is evaluated to true or false.

The following sequence of definitions is useful in defining preservation properties of expressions.

---

12 In this definition, the nodes \( C_i \) and \( C_j \) are some two \( C \) nodes that appear in the XML data tree \textit{BaseXML}. Also note that if \( P \) is \( C \)-identical to \( Q \) with a diff-base of \((C_i, C_j)\) then \( Q \) is \( C \)-identical to \( P \) with a diff-base of \((C_j, C_i)\).

13 Our navigational expressions do not include the union operator \(|\). This is because unless it is used at the top level of an expression, i.e., to return nodes, its effect is the same as that of the OR operator (which we do consider). Our concern here, as will become clear later, is with predicates (inside which the navigational expressions appear), and therefore we do not need the \(|\) operator.
Lemma 6. If \( \exp \) is a navigational-expression with no predicates in XPath’\_NoConst then the following claims hold: (1) \( \exp \) preserves C-symmetry , (2) \( \exp \) preserves D-symmetry, (3) \( \exp \) preserves C-identicalness, (4) \( \exp \) preserves D-identicalness.

Proof. 1. (C-symmetry) By induction on the number of navigation steps \( n \).

- \( S \) is the context set before executing the expression and \( S' \) is the context set after it. \( S \) is C-symmetric and we need to prove that so is \( S' \).

Induction base: \( n = 0 \Rightarrow S' = S \) and therefore \( S' \) is C-symmetric.

Induction step: After \( k \) steps we have a context set \( S'' \) which is C-symmetric and we need to prove that after an additional step we get \( S' \) which is C-symmetric.

- Section 1 of the C-symmetry definition: We need to show that if \( S' \) contains some \( C \) node then it also contains all \( C \) nodes with the same Name.

If \( S' \) contains some \( C \) node then there are three possibilities.

(a) The \( C \) node was already in \( S'' \) and it stayed there by using sel.f.

This means all the other \( C \) nodes in \( S'' \) also appear in \( S' \).

(b) We navigated to the \( C \) node from a C.f1, C.f1.text(), C.f2, C.f2.text(), C.Name or C.Name.text() node of \( S' \). Since \( S' \) is C-symmetric it contains the f1 nodes of all \( C \) nodes with the same Name value, and therefore all those \( C \) nodes will appear in \( S' \). Similarly for the f2 and Name nodes and for the text() nodes.
(c) We navigated to the $C$ node from one of its ancestors. Since all $C$ nodes have the same parent, this means $S'$ contains all $C$ nodes.

- Section 2 of the C-symmetry definition: We need to show that if $S'$ contains some $C.f$ node then it contains the $f$ nodes of all $C$ nodes with the same Name. If $S'$ contains some $C.f$ node then there are four possibilities.
  (a) It was already in $S''$ and stayed using self. This means all the other $C.f$ nodes also stayed.
  (b) We navigated to it from its child $text$ node. Since $S'$ is C-symmetric it contains the $C.f.text$ nodes of all $C$ nodes with the same Name value, and therefore the $C.f$ nodes of all those $C$ nodes will appear in $S'$.
  (c) We navigated to it from its parent $C$ node. Since $S''$ is C-symmetric, it contains all other $C$ nodes with the same Name and therefore $S'$ contains all the required $C.f$ nodes.
  (d) We navigated to it from $A$ or $R$. This means that $S'$ contains the $C.f$ nodes of all $C$ nodes in the XML tree.

- Section 3 of the C-symmetry definition: We need to show that if $S'$ contains some $C.f$ node then it contains the $f$ nodes of all $C$ nodes with the same Name. This is analogous to section 2.

- Section 4 of the C-symmetry definition: We need to show that if $S'$ contains some $C.Name$ node then it contains all $C.Name$ nodes with the same value. This is analogous to section 2.

- Section 5 of the C-symmetry definition: We need to show that if $S'$ contains some $C.f.text$ node then it contains all $C.f.text$ nodes of $C$ nodes with the same Name. If $S'$ contains some $C.f.text$ node then there are four possibilities.
  (a) It was already in $S''$ and stayed using self. This means all the other $C.f.text$ nodes also stayed.
  (b) We navigated to it from its parent $C.f$ node. Since $S''$ is C-symmetric, it contains the $C.f$ nodes of all other $C$ nodes with the same Name and therefore $S'$ contains all the required $C.f.text$ nodes.
  (c) We navigated to it from its grandparent $C$ node. Since $S''$ is C-symmetric, it contains all other $C$ nodes with the same Name and therefore $S'$ contains all the required $C.f.text$ nodes.
  (d) We navigated to it from $A$ or $R$. This means that $S'$ contains the $C.f.text$ nodes of all $C$ nodes in the XML tree.

- Section 6 of the C-symmetry definition: We need to show that if $S'$ contains some $C.f2.text$ node then it contains all $C.f2.text$ nodes of $C$ nodes with the same Name. This is very similar to section 5.

- Section 7 of the definition: We need to show that if $S'$ contains some $C.Name.text$ node then it contains all $C.Name.text$ nodes with the same value. This is very similar to section 5.

2. (D-symmetry) Very similar to the proof for C-symmetry.
3. (D-identicalness) By induction on the number of navigation steps $n$.
   $S1$ and $S2$ are D-identical and $S1'$ and $S2'$ are the context sets after executing the expression from $S1$ and $S2$, respectively.
We need to prove that $S_1'$ and $S_2'$ are D-identical, and if they are not identical then they have the same diff-base as $(S_1, S_2)$.

Induction base: $n = 0 \implies S_1' = S_1$ and $S_2' = S_2$ and therefore $S_1'$ and $S_2'$ are D-identical.

Induction step: After $k$ steps we have $S_1''$ and $S_2''$ that are D-identical and we need to prove that after an additional step we get $S_1'$ and $S_2'$ that are D-identical, and the diff-base is also preserved, i.e., they differ in the same $D$ nodes that $S_1$ and $S_2$ differ in (if they are not identical).

We consider the following cases.

(a) If $S_1''$ and $S_2''$ are identical then clearly so are $S_1'$ and $S_2'$. $S_1''$ and $S_2''$ are also D-symmetric. We need to prove that so are $S_1'$ and $S_2'$.

This follows from section 2 of this lemma, in which we proved that every navigation step preserves D-symmetry.

(b) If $S_1''$ and $S_2''$ are not identical, then there are two distinct D-nodes $D_i$ and $D_j$ such that there is a D-symmetric common part of these sets, and in addition to that part $S_1''$ contains some $D_i$-related nodes while $S_2''$ contains the corresponding $D_j$-related nodes.

The navigation from the common, D-symmetric part, leads to identical, D-symmetric sets (since the navigation step preserves D-symmetry).

Therefore, we only need to prove that the navigation from the different parts of $S_1''$ and $S_2''$ preserves D-identicalness.

To do this we will prove that D-identical context sets are created when a navigation step is executed from the different, $D_i$-related and $D_j$-related nodes (of $S_1''$ and $S_2''$, respectively), and also that the diff-base remains $(D_i,D_j)$. This is enough due to lemma 4.

There are several possibilities.

- $S_1''$ contains $D_i$ and $S_2''$ contains $D_j$: If the navigation step is $/f1 suggesting $D_i$, $/f2$ suggesting $D_j$, or $/child :: *$ then we get $\{D_i,f1\},\{D_j,f1\}$ or $\{D_i,f2\},\{D_j,f2\}$ or $\{D_i,f1,D_i,f2\},\{D_j,f1,D_j,f2\}$. In each of these possibilities we get D-identical sets with a diff-base of $(D_i,D_j)$.

  If the navigation step uses the parent or ancestor axes then the sets we get are identical and do not contain any $D$ nodes, so they are D-identical.

  If the ancestor – or – self axis is used then the sets contain $D_i$ and $D_j$, respectively, in addition to a common D-symmetric part, and therefore they are D-identical (and the diff-base is preserved).

- $S_1''$ contains $D_i,f1$ and $S_2''$ contains $D_j,f1$: Navigation from $D_i,f1$ or $D_j,f1$ via the child axis leads to empty sets.

  Navigation to the text() nodes leads to the sets $\{D_i,f1.text()\}$, $\{D_j,f1.text()\}$.

  If the parent axis is used then we get $\{D_i\},\{D_j\}$.

  If the ancestor axis is used then we get sets that contain $D_i$ and $D_j$, respectively, in addition to a common D-symmetric part (that does not contain any $D$ or $f1$ or $f2$ nodes). If ancestor – or – self is used then the sets contain $\{D_i,D_i,f1\}$ and $\{D_j,D_j,f1\}$, respectively in addition to the common part.
In each of these cases the sets are D-identical and the diff-base is preserved.
- \( S_1'' \) contains \( D_i.f.2 \) and \( S_2'' \) contains \( D_j.f.2 \): similar analysis.
- \( S_1'' \) contains \( D_i.f.1.text() \) and \( S_2'' \) contains \( D_j.f.1.text() \): similar analysis.
- \( S_1'' \) contains \( D_i.f.2.text() \) and \( S_2'' \) contains \( D_j.f.2.text() \): similar analysis.

4. \((C\text{-identicalness})\) By induction on the number of navigation steps \( n \).

\( S_1 \) and \( S_2 \) are C-identical and \( S_1' \) and \( S_2' \) are the context sets after executing the expression from \( S_1 \) and \( S_2 \), respectively.

We need to prove that \( S_1' \) and \( S_2' \) are C-identical, and if they are not identical then they have the same diff-base as \((S_1, S_2)\).

Induction base: \( n = 0 \). \( \implies S_1' = S_1 \) and \( S_2' = S_2 \) and therefore \( S_1' \) and \( S_2' \) are C-identical.

Induction step: After \( k \) steps we have \( S_1'' \) and \( S_2'' \) that are C-identical and we need to prove that after an additional step we get \( S_1' \) and \( S_2' \) that are C-identical, and the diff-base is also preserved, i.e., they differ in the same \( C \) nodes that \( S_1 \) and \( S_2 \) differ in (if they are not identical).

We consider the following cases.

(a) If \( S_1'' \) and \( S_2'' \) are identical then clearly so are \( S_1' \) and \( S_2' \). \( S_1'' \) and \( S_2'' \) are also C-symmetric. We need to prove that so are \( S_1' \) and \( S_2' \).

This follows from section 1 of this lemma, in which we proved that every navigation step preserves C-symmetry.

(b) If \( S_1'' \) and \( S_2'' \) are not identical, then there are two distinct C-nodes \( C_i \) and \( C_j \) such that there is a C-symmetric common part of these sets, and in addition to that part \( S_1'' \) contains some \( C_i \)-related nodes while \( S_2'' \) contains the corresponding \( C_j \)-related nodes.

The navigation from the common, C-symmetric part, leads to identical, C-symmetric sets (since the navigation step preserves C-symmetry).

Therefore, we only need to prove that the navigation from the different parts of \( S_1' \) and \( S_2' \) preserves C-identicalness.

To do this we will prove that C-identical context sets are created when a navigation step is executed from the different, \( C_i \)-related and \( C_j \)-related nodes (of \( S_1'' \) and \( S_2'' \), respectively), and also that the diff-base remains \((C_i,C_j)\). This is enough due to lemma 5.

There are several possibilities.

- \( S_1'' \) contains \( C_i \) and \( S_2'' \) contains \( C_j \): If the navigation step is /f1 or /f2 or /Name or /child :: * then we get \{\( C_i,f.1 \),\( C_j,f.1 \)\} or \{\( C_i,f.2 \),\( C_j,f.2 \)\} or \{\( C_i,Name \),\( C_j,Name \)\} or \{\( C_i,f.1,C_i,f.2,C_i,Name \)\}, \{\( C_j,f.1,C_j,f.2,C_j,Name \)\}. In each of these possibilities we get C-identical sets with a diff-base of \((C_i,C_j)\).

If the navigation step uses the parent or ancestor axes then the sets we get are identical and do not contain any \( C \) nodes, so they are C-identical.

If the ancestor – or – self axis is used then the sets contain \( C_i \) and
Without loss of generality, we will consider only the case where \( e \) always

\[ S1'' \text{ contains } C_i.f1 \text{ and } S2'' \text{ contains } C_j.f1: \text{ Navigation from } C_i.f1 \text{ or } C_j.f1 \text{ via the child axis leads to empty sets.} \]

Navigation to the text() nodes leads to the sets \( \{C_i.f1.text()\}, \{C_j.f1.text()\} \).

If the parent axis is used then we get \( \{C_i\},\{C_j\} \).

If the ancestor axis is used then we get sets that contain \( C_i \) and \( C_j \), respectively, in addition to a common C-symmetric part (that does not contain any \( C \) or \( f1 \) or \( f2 \) or Name nodes). If ancestor-or-self is used then the sets contain \( \{C_i,C_i.f1\} \) and \( \{C_j,C_j.f1\} \), respectively in addition to the common part.

In each of these cases the sets are C-identical and the diff-base is preserved.

\[ S1'' \text{ contains } C_i.f2 \text{ and } S2'' \text{ contains } C_j.f2: \text{ similar analysis.} \]

\[ S1'' \text{ contains } C_i.Name \text{ and } S2'' \text{ contains } C_j.Name: \text{ similar analysis.} \]

\[ S1'' \text{ contains } C_i.f1.text() \text{ and } S2'' \text{ contains } C_j.f1.text(): \text{ similar analysis.} \]

\[ S1'' \text{ contains } C_i.f2.text() \text{ and } S2'' \text{ contains } C_j.f2.text(): \text{ similar analysis.} \]

\[ S1'' \text{ contains } C_i.Name.text() \text{ and } S2'' \text{ contains } C_j.Name.text(): \text{ similar analysis.} \]

\( \square \)

The result of executing a navigational-expression \( exp \) on a context node \( n \), i.e., the context set after the execution, will be written as \( exp(n) \). The Boolean result of a comparison \( \text{comp} \) (of the form \( e1 = e2 \)) will be written as \( \text{comp}(n) \). The Boolean result of a predicate expression \( \text{predExpr} \) will be written as \( \text{predExpr}(n) \).

**Lemma 7.** Let \( \text{comp} \) be a comparison of the form \( e1 = e2 \) where \( e1 \) and \( e2 \) are navigational-expressions in XPath\_NoConst that preserve D-identicalness, C-identicalness, D-symmetry and C-symmetry, and possibly \( e1 \) or \( e2 \) is the constant "\( a \)" or the constant "\( b \)". Let \( D_i \) and \( D_j \) be some \( D \) nodes and let \( C_i' \) and \( C_j' \) be \( C \) nodes with the same Name. Then, the following hold:

1. \( \text{comp}(D_i) = \text{comp}(D_j), \text{comp}(D_i.f1) = \text{comp}(D_j.f1), \text{comp}(D_i.f2) = \text{comp}(D_j.f2), \text{comp}(D_i.f1.text()) = \text{comp}(D_j.f1.text()), \text{comp}(D_i.f2.text()) = \text{comp}(D_j.f2.text()) \).

2. \( \text{comp}(C_i') = \text{comp}(C_j'), \text{comp}(C_i'.f1) = \text{comp}(C_j'.f1), \text{comp}(C_i'.f2) = \text{comp}(C_j'.f2), \text{comp}(C_i'.f1.text()) = \text{comp}(C_j'.f1.text()), \text{comp}(C_i'.f2.text()) = \text{comp}(C_j'.f2.text()), \text{comp}(C_i'.Name.text()) = \text{comp}(C_j'.Name.text()) \).

**Proof.** If \( e1 \) and \( e2 \) are both constants then the comparison is always true or always false, regardless of the context. Therefore, for the rest of the proof we will assume that only one of \( e1 \) and \( e2 \) may be a constant.

Without loss of generality, we will consider only the case where \( e2 \) is a constant (which is analogous to the one where \( e1 \) is a constant).
When an expression is a constant, we will consider the "context set" after its execution to be a set containing only the constant (formally this is not a context set, since it does not contain nodes).

Also note that if an expression is not a constant then the context set after its execution cannot contain constants, since navigation from a node leads to other nodes.

1. Let $S_1$ and $S_2$ be the context sets before $\text{comp}$ is evaluated. $S_1 = \{s_1\}$ and $S_2 = \{s_2\}$, and we can have $s_1 = D_i.s_2 = D_j$ or $s_1 = D_i.f1.s_2 = D_j.f1$ or $s_1 = D_i.f2.s_2 = D_j.f2$ or $s_1 = D_i.f1.text().s_2 = D_j.f1.text()$ or $s_1 = D_i.f2.text().s_2 = D_j.f2.text()$.

In any one of those cases $S_1$ and $S_2$ are D-identical and C-symmetric.

Let $S_1'$ and $S_2'$ be the context sets after executing $e_1$ from context sets $S_1$ and $S_2$, respectively. Let $S_1''$ and $S_2''$ be the context sets after executing $e_2$ from context sets $S_1$ and $S_2$, respectively.

Since $e_1$ and $e_2$ preserve D-identicalness, $S_1'$ and $S_2'$ are D-identical, and also $S_1''$ and $S_2''$ are D-identical. The diff-base is $(D_i,D_j)$.

Since $e_1$ and $e_2$ preserve C-symmetry, $S_1'$, $S_2'$, $S_1''$ and $S_2''$ are C-symmetric.

We want to prove that $\text{comp}(s_1) = \text{comp}(s_2)$, i.e., that $\text{comp}(s_1)$ implies $\text{comp}(s_2)$ and $\text{comp}(s_2)$ implies $\text{comp}(s_1)$. It is sufficient to prove that $\text{comp}(s_1)$ implies $\text{comp}(s_2)$ (the other direction is symmetric).

$\text{comp}(s_1) = \text{true} \implies$ there is a node $x$ in $S_1'$ and a node $y$ in $S_1''$ such that the string values of $x$ and $y$ are equal. There are several possibilities.

- $e_2$ is a constant: $x$ is a $\text{C.Name}$ or a $\text{C.Name.text()}$ node. Because of D-identicalness, $x$ also belongs to $S_2'$. Since $e_2$ is a constant, $S_2'' = S_1''$. Therefore, $\text{comp}(s_2) = \text{true}$.

- Neither $x$ nor $y$ is one of $D_i$, $D_i.f1$, $D_i.f2$, $D_i.f1.text()$ and $D_i.f2.text()$: $x$ also belongs to $S_2'$ and $y$ belongs to $S_2''$ (because of the D-identicalness of $S_1'$, $S_2'$ and of $S_1''$, $S_2''$), and therefore we get $\text{comp}(s_2) = \text{true}$.

- $x$ is one of $D_i$, $D_i.f1$, $D_i.f2$, $D_i.f1.text()$ and $D_i.f2.text()$: There is a node $z$ in $S_2'$ such that $z$ is the corresponding "$D_j$ related" node ($D_j$ or $D_j.f1$ or $D_j.f2$ or $D_j.f1.text()$ or $D_j.f2.text()$).

This is ensured by the D-symmetry requirement of the D-identicalness definition. (Because either $S_1'$ and $S_2'$ are identical and D-symmetric, which means they both contain both the $D_i$-related and the $D_j$-related node, or they are not identical and then the identical part is D-symmetric, which means that if $x$ is in the identical part then the $D_j$-related node is also in it and if $x$ is not in the identical part then $S_2'$ contains the $D_j$-related node that corresponds to $x$).

Now we will examine the possibilities for $y$.

- $y$ is identical to $x$: There is a node $w$ in $S_2''$ that is identical to $z$ (for the same reason that $z$ exists in $S_2'$). Therefore, $\text{comp}(s_2) = \text{true}$.

- $y$ is not identical to $x$ but rather a different node with the same string value: We will go over the possibilities for $x$.
  - $x$ cannot be a $D$ node because in our XML tree every $D$ node has a different string value.
* $x$ is $D_i.f1$: There are several possibilities for $y$: (1) $y$ can be
$D_i.f1.text()$. This means $S2''$ contains $D_j.f1.text()$, which has
the same value as $z = D_j.f1$. (2) $y$ can be some other $D.f1$ node.
This means $e2$ must navigate (from the starting $D_i$-related node
$s1$) to $B$ or $R$ and then to the other $D$ node, but since $\{B\}$
and $\{R\}$ are D-symmetric sets, and $e2$ preserves D-symmetry
(after every step), we get that $S1''$ contains all $D.f1$ nodes, in-
cluding $D_i.f1$. Therefore $S2''$, which is D-identical to $S1''$, con-
tains $D_j.f1$. Since $S2'$ (which is D-identical to $S1'$) also contains
$D_j.f1$, $comp(s2) = true$. (3) $y$ can be the $text()$ node of some
other $D.f1$ node, in which case the analysis is very similar. (4) $y$
can be a $C.f1$ node (for every possible $f1$ value there are two
$C$ nodes). Since $S1''$ is C-symmetric, it contains also the $C.f1$
ode of the other $C$ node with the same $Name$. Then we get that $z$
is $D_j.f1$ and $S2''$ (which is D-identical to $S1''$) contains the two
$C.f1$ nodes, one of which has the same value as $D_j.f1$, so that
$comp(s2) = true$. (5) $y$ can be a $C.f1.text()$ node, in which case
the analysis is very similar to the previous case.

* $x$ is $D_i.f2$: Similar reasoning shows that $comp(s2) = true$.

* $x$ is $D_i.f1.text()$ or $D_i.f2.text()$: The analysis is similar.

– $y$ is one of $D_i$, $D_i.f1$, $D_i.f2$, $D_i.f1.text()$, $D_i.f2.text()$: The analysis is
analogous to the case were $x$ is one of these nodes.

2. (The proof for $C$ node comparisons is similar.)

Definition 15. The depth of a predicate is the level of predicate nesting in it,
i.e., if a predicate $p$ does not contain predicates then its depth is 1 and if it
contains predicates and the maximal depth among those predicates is $k$
then the depth of $p$ is $k+1$.

Definition 16. The predicate-depth of an expression is the maximal depth of
predicates appearing in the expression, or 0 if the expression contains no predi-
cates.

Lemma 8. Let $e$ be a navigational-expression in $XPath'_NoConst$. Then the
following conditions are satisfied: (1) $e$ preserves D-identicalness, (2) $e$
_preserves C-identicalness, (3) $e$ preserves D-symmetry, (4) $e$
preserves C-symmetry.

Proof. The proof is by induction on the predicate-depth of $e$.
Basis: depth = 0. There are no predicates. This case is handled in lemma 6.
Induction. Assume the lemma holds for an expression of depth $k$ and consider
depth $k+1$.

1. (D-identicalness) The proof is by induction on the number of steps, $m$, in
$e$, where a step is either a navigation step or a predicate. Let $S1$ and $S2$ be
D-identical sets with a diff-base of $(D_i,D_j)$ (unless they are identical) and
let $S1'$ and $S2'$ be the context sets after executing the expression from $S1$
and $S2$, respectively. We need to prove that $S1'$ and $S2'$ are D-identical and

34
that the diff-base is preserved (if they are not identical).

Basis: zero steps. Here, $S_1' = S_1$, $S_2' = S_2$.

Induction. After $m$ steps, we have $S_1''$ and $S_2''$ that are D-identical. We need to prove for $m + 1$ steps.

If the next step is a navigation step (not a predicate) then D-identicalness is preserved because this is the same case as in lemma 6.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$.

First we will show that for every two $D_i$ nodes $D_r$ and $D_q$, the predicate gives the same result for $D_r$-related node and for the corresponding $D_q$-related node. To show this, we will go over all the possible $D_i$ and $D_j$ related nodes:

- The result of the predicate is the same for $D_r$ and for $D_q$ because the comparisons in the predicate give the same result for $D_r$ and for $D_q$ (by lemma 7), and the navigational-expressions in the predicate preserve D-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(D_r)$ is non-empty iff $e_1(D_q)$ is non-empty.

Note: We can use lemma 7 here because the expressions in the comparison are of depth $\leq k$ and therefore preserve D-identicalness, C-identicalness, D-symmetry and C-symmetry.

- Using a similar argument, we obtain: (1) The result of the predicate is the same for $D_r.f_1$ and for $D_q.f_1$. (2) The result of the predicate is the same for $D_r.f_2$ and for $D_q.f_2$. (3) The result of the predicate is the same for $D_r.f_1.text()$ and for $D_q.f_1.text()$. (4) The result of the predicate is the same for $D_r.f_2.text()$ and for $D_q.f_2.text()$.

Therefore we deduce that: (1) The common, D-symmetric part of $S_1''$ and $S_2''$ becomes a common, D-symmetric part of $S_1'$ and $S_2'$ after the predicate is executed. (2) If $S_1''$ contains some $D_i$-related node and $S_2''$ contains the corresponding $D_j$-related node, the predicate gives the same result for the $D_i$-related node and for the $D_j$-related node and therefore $S_1'$ contains the $D_i$-related node iff $S_2'$ contains the $D_j$ related nodes. Therefore, the predicate preserves D-identicalness.

2. (C-identicalness) We will prove by induction on the number of steps in $e$, where a step is either a navigation step or a predicate.

$S_1$ and $S_2$ are C-identical and $S_1'$ and $S_2'$ are the context sets after executing the expression from $S_1$ and $S_2$, respectively.

We need to prove that $S_1'$ and $S_2'$ are C-identical and the diff-base is preserved.

Induction base: Zero steps $\Rightarrow S_1' = S_1$, $S_2' = S_2$.

Induction step: After $m$ steps we have $S_1''$ and $S_2''$ that are C-identical. We need to prove for $m + 1$ steps.

If the next step is a navigation step (not a predicate) then C-identicalness is preserved because this is the same case as in lemma 6.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$.

We will show that the predicate gives the same result for a $C_i$-related node of
$S_1''$ and for a $C_j$-related node of $S_2''$ and therefore preserves C-identicalness. We will show this for all possible $C_i$ and $C_j$ related nodes.

- $S_1''$ contains $C_i$ and $S_2''$ contains $C_j$ instead: By definition of C-identicalness, $C_i$ and $C_j$ have the same Name value. Therefore, $S_1'$ contains $C_i$ iff $S_2'$ contains $C_j$, because the comparisons in the predicate give the same result for $C_i$ and for $C_j$ (by lemma 7), and the navigational-expressions in the predicate evaluate to the same truth value for $C_i$ and for $C_j$, since they preserve C-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(C_i)$ is non-empty iff $e_1(C_j)$ is non-empty.

- Using a similar argument, we obtain:
  - If $S_1''$ contains $C_i.f1$ and $S_2''$ contains $C_j.f1$ instead, then $S_1'$ contains $C_i.f1$ iff $S_2'$ contains $C_j.f1$.
  - If $S_1''$ contains $C_i.f2$ and $S_2''$ contains $C_j.f2$ instead, then $S_1'$ contains $C_i.f2$ iff $S_2'$ contains $C_j.f2$.
  - If $S_1''$ contains $C_i.Name$ and $S_2''$ contains $C_j.Name$ instead, then $S_1'$ contains $C_i.f1$ iff $S_2'$ contains $C_j.f1$.
  - If $S_1''$ contains $C_i.f1.text()$ and $S_2''$ contains $C_j.f1.text()$ instead, then $S_1'$ contains $C_i.f1.text()$ iff $S_2'$ contains $C_j.f1.text()$.
  - If $S_1''$ contains $C_i.f2.text()$ and $S_2''$ contains $C_j.f2.text()$ instead, then $S_1'$ contains $C_i.f2.text()$ iff $S_2'$ contains $C_j.f2.text()$.
  - If $S_1''$ contains $C_i.Name.text()$ and $S_2''$ contains $C_j.Name.text()$ instead, then $S_1'$ contains $C_i.f2.text()$ iff $S_2'$ contains $C_j.f2.text()$.

Therefore, the predicate preserves C-identicalness.

3. (C-symmetry) We will prove by induction on the number of steps in $e$, where a step is either a navigation step or a predicate.

Induction base: Zero steps $⇒ S' = S$.

Induction step: After $m$ steps we have a C-symmetric context set $S''$. We need to prove for $m + 1$ steps.

If the next step is a navigation step (not a predicate) then C-symmetry is preserved because this is the same case as in lemma 6.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to $k$. The predicate will give the same truth value for all $C$ nodes with the same Name (i.e., will filter all or none of them), because the comparisons will give the same value for these $C$ nodes (by lemma 7) and the navigational-expressions in the predicate evaluate to the same truth value for all $C$ nodes with the same Name since they preserve C-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression $e_1$, $e_1(C)$ is non-empty for one $C$ node iff it is non-empty for all $C$-nodes with the same Name.

Using a similar argument, we obtain:

- The predicate will give the same truth value for all of the $C.f1$ nodes of $C$ nodes with the same Name.
- The predicate and will give the same truth value for all of the $C.f2$ nodes of $C$ nodes with the same Name.
– The predicate will give the same truth value for all \textit{C.Name} nodes with the same value.
– The predicate will give the same truth value for all of the \textit{C.f1.text()} nodes of \textit{C} nodes with the same \textit{Name}.
– The predicate and will give the same truth value for all of the \textit{C.f2.text()} nodes of \textit{C} nodes with the same \textit{Name}.
– The predicate will give the same truth value for all \textit{C.Name.text()} nodes with the same value.

Therefore, the predicate preserves C-symmetry.

4. \textbf{(D-symmetry)} We will prove by induction on the number of steps in \(e\), where a step is either a navigation step or a predicate.

\textbf{Induction base:} Zero steps \(\implies S' = S\).

\textbf{Induction step:} After \(m\) steps we have a D-symmetric context set \(S''\). We need to prove for \(m + 1\) steps.

If the next step is a navigation step (not a predicate) then D-symmetry is preserved because this is the same case as in lemma 6.

Suppose the next step is a predicate. The predicate may contain comparisons and navigational-expressions of predicate-depth smaller than or equal to \(k\). The predicate will give the same truth value for all \(D\) nodes (i.e., will filter all or none of them), because the comparisons will give the same value for all \(D\) nodes (by lemma 7) and the navigational-expressions in the predicate evaluate to the same truth value for all \(D\) nodes since they preserve D-identicalness (hypothesis of the induction on predicate-depth) and therefore for such a navigational-expression \(e_1\), \(e_1(D)\) is non-empty for one \(D\) node iff it is non-empty for all \(D\)-nodes.

Using a similar argument, we obtain:
– The predicate will give the same truth value for all \(D.f1\) nodes.
– The predicate will give the same truth value for all \(D.f2\) nodes.
– The predicate will give the same truth value for all \(D.f1.text()\) nodes.
– The predicate and will give the same truth value for all \(D.f2.text()\) nodes.

Therefore, the predicate preserves D-symmetry.

\[\Box\]

\textbf{Lemma 9.} Let \(\text{comp}\) be a comparison of the form \(e_1 = e_2\), where \(e_1\) and \(e_2\) are navigational-expressions in XPath’\_\_\_NoConst or constants. Then \(\text{comp}(D_1) = \text{comp}(D_2)\) (\(D_1\) and \(D_2\) are two specific nodes in BaseXML (see Figure 5)).

\textbf{Proof.} According to lemma 8, \(e_1\) and \(e_2\) preserve C-identicalness, D-identicalness, C-symmetry and D-symmetry. Therefore, by lemma 7 we obtain \(\text{comp}(D_1) = \text{comp}(D_2)\). \(\Box\)

\textbf{Definition 17.} A predicate \(\text{pred}\) is a \(\text{D12Predicate}\) if \(\text{pred}\) gives a different result for \(D_1\) and for \(D_2\), i.e., if the context set before executing the predicate contains both \(D_1\) and \(D_2\) then the context set after executing the predicate will contain exactly one of the two nodes.
Lemma 10. There exist no $D_{12}$Predicates in $\text{XPath}'_{\text{NoConst}}$.

Proof. Suppose, for the sake of deriving a contradiction, that there exists a $D_{12}$Predicate, of the form $\text{predExpr}$. $\text{predExpr}$ may contain comparisons and navigational-expressions. We claim that $\text{predExpr}(D_1) = \text{predExpr}(D_2)$. In proof, at the "top level" of $\text{predExpr}$ there are comparisons and navigational-expressions, connected with Boolean operators (and, or), and possibly also with the Boolean function not(). According to lemma 9, for every comparison $\text{comp}$, $\text{comp}(D_1) = \text{comp}(D_2)$. According to lemma 8, for every navigational-expression $\text{exp}$, $\text{exp}(D_1)$ and $\text{exp}(D_2)$ are D-identical, and therefore $\text{exp}(D_1)$ is non-empty iff $\text{exp}(D_2)$ is non-empty, which means $\text{exp}$ evaluates to the same truth value when operating in the context of $D_1$ and also in the context of $D_2$. So, $\text{predExpr}$ evaluates to the same truth value when operating in the context of $D_1$ and in the context of $D_2$. The claim is now established, which implies that $\text{pred}$ is not a $D_{12}$Predicate. This is a contradiction and thus $\text{pred}$ can not exist. \hfill \Box

Definition 18. $D_{12}\text{XMLTrees}$ is the set of XML trees $\{T_0, T_1, \ldots \}$ with the same structure as $\text{BaseXML}$ (see Figure 5) but with different values in each tree. The value of an $f_1$ or $f_2$ node in $T_i$ is $10^i$*(the value of the corresponding node in $\text{BaseXML}$). $T_0$ is $\text{BaseXML}$. Note that lemma 10 applies for all $T_i$, since increasing the $f_1$ and $f_2$ values proportionally does not affect our proof.

For example, the value of $D_1.f_1$ in $T_4$ is $3 \times 10^4 = 30000$.

Lemma 11. There is no predicate in $\text{XPath}'$ (i.e., with constants) that, for every XML tree in $D_{12}\text{XMLTrees}$, gives a result for $D_1$ that is different from the result for $D_2$.

Proof. According to lemma 10 there is no such predicate in $\text{XPath}'_{\text{NoConst}}$, because there is no such predicate for $\text{BaseXML}$. So, if there is such a predicate in $\text{XPath}'$, it must use constants other than "a" and "b". In order to derive a contradiction, suppose that such a predicate exists. Since constants can only be used in comparisons with expressions, in order for the predicate to give a different result for $D_1$ and for $D_2$, it must contain some constants (other than "a" and "b") that have the same string value as nodes that appear in the tree (otherwise all comparisons to these constants will evaluate to false). Since the predicate is (syntactically) finite, it contains a finite number of constants. Let us denote the maximal number of characters in a constant that appears in the predicate by $M$. Since the string value of every $f_1$ and $f_2$ node in $T_i$ has $i + 1$ characters and the string values of $D$ and $C$ nodes contain even more characters, for all $i \geq M$, $T_i$ contains no nodes whose string values appear in the predicate (and are not "a" or "b"). Therefore, the predicate will give the same answer for $D_1$ and for $D_2$ in every such tree (since the constants other than "a" and "b" are irrelevant and thus the predicate operates as it would with no constants other than "a" or "b"). Thus we derive a contradiction. \hfill \Box

Lemma 12. There is no expression $e$ in $\text{XPath}'$ such that for every XML data tree in $D_{12}\text{XMLTrees}$, $e$ returns the $D$ nodes that reference $C$ nodes whose Name is "a" (with an initial context set that contains only the root node).
Proof. Suppose, for the sake of deriving a contradiction, that such an expression \(e\) exists. Then, for every tree in \(D12XMLTrees\), \(e\) returns the set \(\{D2, D3\}\). Therefore, for every tree in \(D12XMLTrees\), the following predicate gives a different result for \(D1\) and for \(D2\): \([., = /parent :: B/parent :: R/e]\). This evaluates to \(true\) on \(D2\) but not on \(D1\). This contradicts lemma 11.

Theorem 4. There is no expression \(e\) in \(XPath'\) such that for every XML data tree that conforms to \(SchemaCD\), \(e\) returns exactly the \(D\) nodes that reference \(C\) nodes whose \(Name\) is "a".

Proof. Suppose, for the sake of deriving a contradiction, that such an expression exists. Then, this expression also works for the XML trees of \(D12XMLTrees\), since they all conform to \(SchemaCD\). But this contradicts lemma 12.

And now the proof for theorem 2:

Proof. We show a query that can be written in \(XPath'_{fk}\) but not in \(XPath'\). Given an XML data tree that conforms to \(SchemaCD\), the query selects the \(D\) nodes that reference \(C\) nodes whose \(Name\) is "a". In \(XPath'_{fk}\) this is written as \(C[Name = "a"]//KR,Children\). According to theorem 4, such a query cannot be written in \(XPath'\).