XRPC: efficient distributed query processing on heterogeneous XQuery engines
Zhang, Y.

Citation for published version (APA):
Zhang, Y. (2010). XRPC: efficient distributed query processing on heterogeneous XQuery engines

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
6

Correctness Proof of XQuery Decomposition

In this chapter, we formally prove the correctness of the decomposition algorithms presented in Chapter 5. For each algorithm, we prove that executing subexpressions of a query remotely over XRPC will produce results deep-equal to those of the original query, if this is allowed by a certain algorithm. We use a definition of deep-equal query results that also takes into account the freedoms an implementation has in processing some aspects of the language.

Roadmap Sections 6.1 and 6.2 contain auxiliary definitions, properties, lemmas and rules. Sections 6.3 - 6.7 are the main components of this chapter and contain theorems of the correctness of the techniques proposed in Chapter 5.

Since the goal of this chapter is to prove that a decomposed query produces deep-equal result to the original query, we start in Section 6.1 with definitions of different kinds of deep-equal for both sequences and queries. The contents of this section are important for the understanding of the proofs of the theorems in Sections 6.3 - 6.7.

Section 6.2 contains a complete list of judgement rules for all kinds of XQuery expressions. These rules specify how certain static properties of an expression are inferred. These rules are referred to by the proofs in Sections 6.3 - 6.7, because, to determine whether it is correct to decompose a certain expression, we need to know if the expression has the desired properties. This section can be used as a reference.

In Sections 6.3 - 6.5, we prove the correctness of each decomposition algorithm on read-only XCore queries. In Section 6.6, we prove the correctness of the decomposition algorithms on XCore queries containing XQUF expressions. Finally, we prove the correctness of the distributed code motion technique in Section 6.7.

6.1 Preliminaries

Notations. We use $\vec{a}$ to denote sequences $(a_1,...,a_n)$ of length $n$ (denoted as $|\vec{a}|$). We use $\vec{a}[i]$ to explicitly refer to the $i$-th item in a sequence $\vec{a}$. We permit $\vec{a}$ in set contexts to represent the set $\{a_1,...,a_n\}$.

The symbol $x \mapsto E$ indicates that the variable $x$ is mapped to the value of the expression $E$, and $(Env + x \mapsto E)$ means that the environment $Env$ is extended with the variable $x$ bound to (the value of) the expression $E$. We will use $E$ interchangeably to represent an XCore expression and the result sequence of evaluating the expression.
We use $\mathbb{T}$ to denote the set of XML node types: \{attribute, comment, document, element, processing-instruction, text\}.

We abbreviate the XPath step “descendant-or-self” as “dos”. Grammar rules (Table 5.2) concerning XPath step expressions are abbreviated as the following: ST = StepExpr, AS = AxisStep, NT = NodeTest.

6.1.1 Equality Relationships of Sequences

Equality relationships between sequences can be defined at different levels. The most strict equality relationship is the absolute equivalence between two sequences, which we define as the following:

**Definition 6.1.1. Equivalent sequences:** Two sequences $\vec{s}_1$ and $\vec{s}_2$ are equivalent to each other, denoted $\vec{s}_1 \equiv \vec{s}_2$, iff they satisfy the following condition\(^1\):

$$dEq(\vec{s}_1, \vec{s}_2) \land \forall i \in 1..|\vec{s}_1|: \text{dm:node-kind}(\vec{s}_1[i]) \in \mathbb{T} \Rightarrow \vec{s}_1[i] \text{ is } \vec{s}_2[i]$$

However, in most circumstances, such as in XRPC, this strict equality relationship is not required, and deep-equal is sufficient. Using our notations, the XQuery deep-equal semantics for sequences can be expressed as follows:

**Definition 6.1.2. Deep-equal sequences:** Two sequences $\vec{s}_1$ and $\vec{s}_2$ are deep-equal to each other, denoted $dEq(\vec{s}_1, \vec{s}_2)$, iff they satisfy the following conditions:

$$\vec{s}_1 = \vec{s}_2 = (), \text{or}$$

$$|\vec{s}_1| = |\vec{s}_2| \land \forall i \in 1..|\vec{s}_1|:\begin{cases} \vec{s}_1[i] = \text{NaN} \land \vec{s}_2[i] = \text{NaN}, \\
\vec{s}_1[i] = \vec{s}_2[i], \text{if dm:node-kind}(\vec{s}_1[i]) \notin \mathbb{T} \land \text{dm:node-kind}(\vec{s}_2[i]) \notin \mathbb{T} \land \text{fn:deep-equal}(\vec{s}_1[i], \vec{s}_2[i]), \text{if dm:node-kind}(\vec{s}_1[i]) \in \mathbb{T} \end{cases}$$

We use $dEq(E_1, E_2)$ to indicate that the result sequences of evaluating the expressions $E_1$ and $E_2$ are deep-equal to each other. For short, we say $E_1$ and $E_2$ are deep-equal.

In this section, the definitions of equality relationships of sequences are all based on the appearances of the sequences, i.e., two sequences could only possibly be equal (either equivalent or deep-equal), if their literal values appear to be equal. In the next section, we introduce a new kind of equality relationship for sequences, in which the equality relationship of two sequences is based on whether the results of applying a certain set of paths on these sequences are deep-equal.

6.1.2 Equality Relationships of Sequences with Projection

Let $\vec{P}_{rel}$ denote a set of relative projection paths\(^2\) which consists of a set of relative used paths, denoted $\vec{P}_{rel}.\vec{U}$, and a set of relative returned paths, denoted $\vec{P}_{rel}.\vec{R}$. Each path in $\vec{P}_{rel}$ may contain XPath steps on all axes and the special built-in functions root(), id() and idref() (i.e., as defined by SimplePath in Table 5.6). Given a set of relative projection paths $\vec{P}_{rel}$, the results of applying $\vec{P}_{rel}$ on two non-deep-equal sequences $\vec{s}_1$ and $\vec{s}_2$ could be deep-equal. Therefore, we introduce a lower level equality

---

\(^1\)In XDM [71], the dm:node-kind accessor is only defined on the seven kinds of node. Here, we assume that dm:node-kind returns an error if $\vec{s}_1[i]$ is an atomic value.

\(^2\)The projection paths are “relative” as they do not start from a document root, but rather from the node typed items in the sequences (Section 5.6.2).
relationship for sequences, with respect to a certain set of relative projection paths \( \tilde{P}^{rel} \). We first give a formal definition of how the projection of \( \tilde{P}^{rel} \) on a sequence \( \tilde{s} \) is computed at runtime.

**Definition 6.1.3. Runtime XML projection operator \( \mathcal{P} \):** Let \( \tilde{s} \) be an XQuery node sequence, which may contain duplicates or overlapping nodes (i.e., nodes that have an ancestor-descendant relationship), and \( \tilde{P}^{rel} \) a non-empty set of relative projection paths. The runtime XML projection operator \( \mathcal{P} \) creates a set of projected XML fragments \( \tilde{F} \) by projecting \( \tilde{P}^{rel} \) on \( \tilde{s} \). \( \tilde{F} \) is computed as follows:

1. Apply \( \tilde{P}^{rel}, \tilde{U} \) and \( \tilde{P}^{rel}, \tilde{R} \) on \( \tilde{s} \) to produce the set of used nodes \( \tilde{N}^{u} \) and the set of returned nodes \( \tilde{N}^{r} \), respectively; and add \( \tilde{s} \) to \( \tilde{N}^{e} \);
2. Let \( \tilde{D} \) be the set of documents, from which nodes in \( \tilde{N}^{u} \) and \( \tilde{N}^{e} \) originate, sorted by document order;
3. Let \( \tilde{N}^{u}_{i} \) and \( \tilde{N}^{e}_{i} \) be respectively a subset of \( \tilde{N}^{u} \) and \( \tilde{N}^{e} \) which contain all nodes in \( \tilde{N}^{u} \) and \( \tilde{N}^{e} \) that originate from the same document \( \tilde{D}[i] \). Then \( \tilde{F}[i] \) is the projection of \( \tilde{N}^{u}_{i} \cup \tilde{N}^{e}_{i} \) on \( \tilde{D}[i] \), computed by the RuntimeXMLProjection algorithm (Section 5.6.2, Algorithm 1), i.e.:

\[
|\tilde{F}| = |\tilde{D}| \land \forall i \in 1..|\tilde{D}| : \tilde{F}[i] = \text{RuntimeXMLProjection}(\tilde{N}^{u}_{i}, \tilde{N}^{e}_{i}, \tilde{D}[i])
\]

**Definition 6.1.4. By-projection equal sequences:** Let \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \) be two XQuery sequences, and \( \tilde{P}^{rel} \) a set of relative projection paths. If \( \tilde{P}^{rel} \) is not an empty set, projections of \( \tilde{P}^{rel} \) on \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \), respectively, are computed as follows\(^3\): \( \tilde{F}_{1} = \mathcal{P}(\tilde{s}_{1}, \tilde{P}^{rel}) \) and \( \tilde{F}_{2} = \mathcal{P}(\tilde{s}_{2}, \tilde{P}^{rel}) \). We use \( dEq^{\tilde{P}^{rel}}(\tilde{s}_{1}, \tilde{s}_{2}) \) to denote that \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \) are by-projection equal to each other, with respect to \( \tilde{P}^{rel} \). Whether \( dEq^{\tilde{P}^{rel}}(\tilde{s}_{1}, \tilde{s}_{2}) \) holds is determined by the following rules:

1. If the set of relative projection paths is empty (i.e., no projection can be computed) \( s_{1} \) or \( s_{2} \) are by-projection equal, iff they are deep-equal to each other:

\[
\tilde{P}^{rel} = \emptyset : dEq^{\tilde{P}^{rel}}(\tilde{s}_{1}, \tilde{s}_{2}) \Leftrightarrow dEq(\tilde{s}_{1}, \tilde{s}_{2})
\]

2. Otherwise, \( s_{1} \) or \( s_{2} \) are by-projection equal, if their projections \( \tilde{F}_{1} \) and \( \tilde{F}_{2} \) are deep-equal to each other:

\[
\tilde{P}^{rel} \neq \emptyset : dEq^{\tilde{P}^{rel}}(\tilde{s}_{1}, \tilde{s}_{2}) \Leftrightarrow dEq(\tilde{F}_{1}, \tilde{F}_{2})
\]

**Example 6.1.5.** The leftmost column of Figure 6.1 shows two sequences \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \) that each contains one different \( \langle a \rangle \ldots \langle /a \rangle \) node. The right-most column of Figure 6.1 shows the resulting projected fragments \( \tilde{F}_{1} \) and \( \tilde{F}_{2} \), when the set of relative projection paths \( \tilde{P}^{rel} \) is applied on \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \), respectively. Here, \( \tilde{P}^{rel}, \tilde{U} = \{ \ldots /b, /b/c \} \) and \( \tilde{P}^{rel}, \tilde{R} = \{ ./b/i \} \). Since \( \tilde{F}_{1} \) and \( \tilde{F}_{2} \) are deep-equal, we say that \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \) are by-projection equal, with respect to \( \tilde{P}^{rel} \). In Section 6.5, we explain in detail how \( \tilde{P}^{rel} \) is computed.

---

\(^3\)If \( \tilde{s}_{1} \) or \( \tilde{s}_{2} \) contains a literal value, applying \( \tilde{P}^{rel} \) on \( \tilde{s}_{1} \) or \( \tilde{s}_{2} \) results in a runtime error. However, in our compile time analysis, we can omit this check and assume that \( \tilde{s}_{1} \) and \( \tilde{s}_{2} \) contain correct nodes.
6.1. PRELIMINARIES

Figure 6.1: Two non-deep-equal sequences $\vec{s}_1$ and $\vec{s}_2$ that are by-projection deep-equal, with respect to $\vec{P}_{rel}$, where $\vec{P}_{rel}.U = \{./b, ./b/c\}$ and $\vec{P}_{rel}.R = \{./b/i\}$.

Remark In Definition 6.1.4, we require that $\vec{s}_1$ and $\vec{s}_2$ are by-projection equal, if and only if their projected fragments $\vec{F}_1$ and $\vec{F}_2$ are deep-equal, which implies that in the projected fragments, a used node may not contain (unused) descendant nodes. This requirement is more strict than what is exactly necessary, because, for instance, in Figure 6.1, if the ⟨c/⟩ node in the projected fragment $\vec{F}_2$ would have contained some descendants, it will not affect result of query evaluation (that is applied on $\vec{F}_2$ instead of on $\vec{s}_2$). In case of XRPC, it only causes unnecessary bandwidth usage in request/response messages, however, this defeats a major purpose of the by-projection semantics: minimise message sizes by pruning unused nodes. Thus, we regard it to be necessary to use this strict requirement in our definition of by-projection equal sequences, and the runtime XML projection operator $\mathcal{P}$ guarantees that unused descendants of used nodes are pruned.

6.1.3 Equality Relationship of Read-Only Queries

Intuitively, one would say that two read-only queries are deep-equal to each other if their result sequences are deep-equal. Unfortunately, this simple comparison is not general enough to deal with the variations in the XQuery language. In the XQuery specifications\footnote{This includes the following documents: XQuery 1.0 and XPath 2.0 Data Model [71], XQuery 1.0: An XML Query Language [38], XQuery 1.0 and XPath 2.0 Formal Semantics [67], XQuery 1.0 and XPath 2.0 Functions and Operators [124], XSLT 2.0 and XQuery 1.0 Serialization [39], and XQuery Update Facility 1.0 [58].}, certain aspects of language processing are described as “implementation defined” or “implementation dependent”. Implementation defined indicates an aspect that may differ between implementations, but must be specified by the implementer for each particular implementation. Implementation dependent indicates an aspect that may differ between implementations, is not specified by any W3C specification, and is not required to be specified by the implementer for any particular implementation. Since in our proofs, there is no need to differentiate whether a certain aspect of language processing is implementation defined or implementation dependent, we will refer to all such aspects as “XQuery features with implementation freedom”. Because of these freedoms in the language, two queries containing the same expressions (or different executions of the same query) do not necessarily return literally the same results (i.e., XQuery deep-equal), while they both return correct results.
In this section, we first look into some examples of how the implementation freedom on the ordering of different XML documents affects query results. In case of differences due to nodes’ document order, the XRPC rewriting framework guarantees consistent results by taking some extra precautions. For the differences, the basic approach of our rewriting framework is to re-define equality relationships of XQuery expressions to allow such differences.

**Inter-Document Ordering**

Document order is defined in XQuery 1.0 and XPath 2.0 Data Model (XDM) [71]. The relative order of nodes in different XML documents (for short: inter-document ordering) is defined as: “stable but implementation dependent, subject to the following constraint: If any node in a given tree, \( T_1 \), occurs before any node in a different tree, \( T_2 \), then all nodes in \( T_1 \) are before all nodes in \( T_2 \)”, where stable means that “the relative order of two nodes will not change during the processing of a given query”. That is, any ordering (including random order) of two different XML documents is correct, as long as the same ordering is used in one execution of a query. Under this definition, the query:

\[
(doc("a")/a ≪ doc("b")/b) = (doc("a")/a ≪ doc("b")/b)
\]

must always return \( \text{true} \), since the ordering of \( \text{doc(”a”)} \) and \( \text{doc(”b”)} \) must be stable within this query. However, a query like

\[
(doc("a")/a ≪ doc("b")/b) = (doc("d")/d ≪ doc("c")/c)
\]

may return \( \text{true} \) or \( \text{false} \), depending on how inter-document ordering is defined by the executing XQuery engine. In fact, it may even differ in subsequent executions using the same engine. Thus, when judging whether two queries are deep-equal, it is actually sufficient to check if they produce result sequences that are “deep-equal with inter-document freedom”:

Let \( Q \) and \( Q' \) be two XQuery queries containing one or more comparison(s) of inter-document ordering. Let \( \vec{r}_Q \) and \( \vec{r}_Q' \) be the result sequences of \( Q \) and \( Q' \), respectively. We say \( Q \) and \( Q' \) are deep-equal with inter-document freedom to each other, if either \( \text{dEq}(\vec{r}_Q, \vec{r}_Q') \) holds, or the differences between \( \vec{r}_Q \) and \( \vec{r}_Q' \) are only caused by that the executions of \( Q \) and \( Q' \) use different but stable inter-document ordering.

If the query \( Q' \) above is a decomposition of \( Q \), we regard \( Q' \) as a valid decomposition. Guaranteeing a stable ordering is trivial if \( Q \) opens each document only once, as any ordering is correct. Otherwise, a stable ordering is guaranteed by the decomposition algorithms by making sure that a subexpression comparing inter-document ordering is never decomposed, if there is another subexpression in the query that compares document order of the same document(s).

**Deep-Equal Read-Only Queries with Implementation Freedom**

Besides inter-document ordering, the XQuery specifications have defined a number of other implementation defined or implementation dependent XQuery features. A complete list of these features, including those defined by XQUF, is given in Appendix B. Taking into account all XQuery features with implementation freedom, we say that a decomposed query \( Q' \) is “deep-equal with implementation freedom” to the original query \( Q \), if the result sequence of \( Q' \) is deep-equal to one of the result sequences that could be returned by \( Q \).

Recall that XQuery queries are always executed in a dynamic context \( \text{dynEnv} \), which we simplify to a database state \( \text{db} \), i.e., the documents and their contents stored in an XML database. The semantic judgement \( \text{db} \models E \Rightarrow \vec{s}^E \) specifies that in the database state \( \text{db} \), a read-only expression \( E \) evaluates to a sequence \( \vec{s}^E \), which is an instance of the XDM.
Definition 6.1.6. Deep-equal read-only queries (with implementation freedom): Let \( Q \) be an XQuery query and \( \mathcal{S} = \{ \mathcal{S}_1, \ldots, \mathcal{S}_k \} \) the set of all valid result sequences of \( Q \). That is, assume that \( Q \) contains \( k \) XQuery features with implementation freedom, and for each of these features, the number of choices an implementation has is \( \{ m_1, m_2, \ldots, m_k \} \), respectively.

Then, the total number of valid result sequences of \( Q \) is bound by the permutation of the sum of \( \{ m_1, m_2, \ldots, m_k \} \):

\[
 n = |\mathcal{S}| = \left( \sum_{i=1}^{k} m_i \right)!
\]

Let \( Q' \) be a decomposed query of \( Q \) and \( \mathcal{S}' \) a result sequence of \( Q' \), then, we say that \( Q \) and \( Q' \) are “deep-equal with implementation freedom” to each other, iff there is one \( \mathcal{S}_i \in \mathcal{S} \), such that \( dEq(\mathcal{S}_i, \mathcal{S}'_i) \). For simplicity, we just say that \( Q \) and \( Q' \) are deep-equal to each other, and denote it as \( dEq(Q, Q') \). Hence:

\[
\begin{align*}
\forall db: \ db \vdash Q & \Rightarrow x_1' \ | \ db \vdash Q \Rightarrow x_2' \ | \ldots \ | db \vdash Q \Rightarrow x_n' \\
\forall db: \ db \vdash Q' & \Rightarrow x_1'' \ | \ db \vdash Q' \Rightarrow x_2'' \ | \ldots \ | db \vdash Q' \Rightarrow x_n'' \\
\exists \mathcal{S}_i \in \{ x_1', \ldots, x_n' \}, \exists \mathcal{S}'' \in \{ x_1'', \ldots, x_n'' \} : dEq(\mathcal{S}_i, \mathcal{S}''_i) \\
\end{align*}
\]

**6.1.4 Equality Relationship of Updating Queries**

The results of updating queries, i.e., XCore queries containing XQUF expressions, are reflected on the documents affected by an updating query, after all updates in the query are made effective. Naturally, one would regard two updating queries distributed over XRPC to be deep-equal if they update the same documents (on the same peers), and the resulting documents (after updates have been made effective) are deep-equal. However, also due to the XQuery features with implementation freedom, this simple comparison is too strict to be useful. For instance, the XQUF specifies that “If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation dependent.”. This implies that even in local executions of XQUF queries, after the execution of the query (assuming an XML document “a.xml” containing one node \(<a/>\)):

\[
\{ \text{insert nodes } ((b/), (c/)) \text{ into doc("a.xml")/a, insert nodes } ((d/), (e/)) \text{ into doc("a.xml")/a} \}
\]

the original document could be changed into \(<a/>(b/)(c/)(d/)(e/)(/a)\) or \(<a/>(d/)(e/)(b/)(c/)(/a)\).

A more general way to compare if two updating queries are deep-equal is to compare their pending update list (PUL), which is an unordered collection of all updates that should be applied after the evaluation of a certain updating query. The PUL contains sufficient information to compare two queries, yet, it provides space to take into account the XQuery features with implementation freedom that have been defined for updating queries.

For updating queries we use a slightly different semantic judgement. A successful execution of an updating query always yields an empty sequence and a Pending Update List (PUL) \( \Delta^5 \). Thus, the semantic judgement \( db \vdash E \Rightarrow (\_, \Delta) \) specifies that in the database state \( db \), evaluating an updating expression \( E \) yields a tuple, which consists of the empty sequence ‘\( () \)’ and a PUL \( \Delta \). Each update primitive \( \delta \) in a \( \Delta \) is a triple \( (N, T, C) \), where:

\(^5\text{A PUL is an unordered list of possibly duplicate update primitives [58], thus, our notation } \overline{a} \text{ for sequences or sets is not applicable here.}\)
• \(N\) is the name of this update primitive. It may be any one of the update primitive functions defined by XQUF [58].

• \(T\) is the node identifier of the target node of this update primitive.

• \(\tilde{C}\) contains the new contents, for instance, \(\tilde{C}\) contains a sequence of to be inserted nodes, if \(N=\text{"upd:insertInto"}\); \(\tilde{C}\) contains a string literal, if \(N=\text{"upd:rename"}\); and \(\tilde{C}\) is \(\emptyset\), if \(N=\text{"upd:delete"}\).

**Definition 6.1.7. Deep-equal update primitives:** Let \(\delta\) and \(\delta'\) be two update primitives. We say that \(\delta\) and \(\delta'\) are deep-equal, denoted \(dEq(\delta, \delta')\), iff \(\delta\) and \(\delta'\) both represent the same update action on the same node with the same deep-equal contents:

\[
\delta.N = \delta'.N \land \delta.T \Leftrightarrow \delta'.T \land dEq(\delta.\tilde{C}, \delta'.\tilde{C}) \Rightarrow dEq(\delta, \delta')
\]

Because the PULs are unordered lists, we cannot check if two PULs are deep-equal by comparing the corresponding update primitives at the same position in the lists. Moreover, two PULs do not necessarily contain the same number of update primitives to have the same effect on the affected documents, since multiple \(\text{upd:delete}\) operations may be applied to the same node during execution of a query. In XQUF, deleting the same node multiple times in a query has the same effect as deleting the node just once, thus, they could be treated as equal. Contrary to deletion, multiple insertions of the same node sequence is not equal to inserting the sequence only once. Finally, a PUL may not contain more than one \(\text{upd:rename}\) operation that has the same target node. The same condition holds for the operations \(\text{upd:replaceNode}\) and \(\text{upd:replaceElementContent}\).

**Definition 6.1.8. Deep-equal pending update lists (with implementation freedom):** Let \(\Delta\) and \(\Delta'\) be two PULs. Let \(\Delta_{\text{del}}, \Delta_{\text{re}}\) and \(\Delta_{\text{ins}}\) represent subgroups of \(\Delta\) that contain certain kinds of update actions, defined as the following:

1. \(\Delta_{\text{del}} = \{\delta | \delta \in \Delta \land \delta.N = \text{"upd:delete"}\}\)

2. \(\Delta_{\text{re}} = \{\delta | \delta \in \Delta \land \delta.N \in \{\text{"upd:rename"}, \text{"upd:replaceNode"}, \text{"upd:replaceValue"}, \text{"upd:replaceElementContent"}\}\}\)

3. \(\Delta_{\text{ins}} = \{\delta | \delta \in \Delta \land \delta.N \in \{\text{"upd:insertBefore"}, \text{"upd:insertAfter"}, \text{"upd:insertInto"}, \text{"upd:insertIntoAsFirst"}, \text{"upd:insertIntoAsLast"}, \text{"upd:insertAttributes"}\}\}\)

Similarly, \(\Delta'_{\text{del}}, \Delta'_{\text{re}}\) and \(\Delta'_{\text{ins}}\) represent the same subgroups of \(\Delta'\). We say that \(\Delta\) and \(\Delta'\) are deep-equal to each other with implementation freedom (for short: \(\Delta\) and \(\Delta'\) are deep-equal), denoted \(dEq(\Delta, \Delta')\), iff \(\Delta\) and \(\Delta'\) satisfy the following conditions:

1. \((\forall \delta_i \in \Delta_{\text{del}}. \exists \delta'_j \in \Delta'_{\text{del}} : dEq(\delta_i, \delta'_j)) \land (\forall \delta'_j \in \Delta'_{\text{del}}. \exists \delta_j \in \Delta_{\text{del}} : dEq(\delta_j, \delta'_j))\)

2. \((\forall \delta_i \in \Delta_{\text{re}}. \exists \delta'_j \in \Delta'_{\text{re}}. \exists \delta'_k \in \Delta'_{\text{re}} : dEq(\delta_i, \delta'_j) \land i \neq k \land \delta'_j.N = \delta'_k.N \land \delta'_j.T \Leftrightarrow \delta'_k.T \Rightarrow (\forall \delta'_j \in \Delta'_{\text{re}}. \exists \delta_j \in \Delta_{\text{re}} : dEq(\delta_j, \delta'_j) \land i \neq k \land \delta_j.N = \delta_k.N \land \delta_j.T \Leftrightarrow \delta_k.T)\)

3. \(|\Delta_{\text{ins}}| = |\Delta'_{\text{ins}}| \land \forall \delta_i \in \text{distinct-primitives}(\Delta_{\text{ins}}). \tilde{\delta}_i = \{\forall \delta_j \in \Delta_{\text{ins}} \land dEq(\delta_i, \delta_j)\} \land |\Delta_{\text{ins}}| \land \forall \delta'_j \in \Delta'_{\text{ins}} : dEq(\delta'_j, \delta_i)\)
In the above definition, condition 1 checks if for each update primitive \( \delta_i \in \Delta_{del} \), \( \Delta_{del} \) contains at least one update primitive \( \delta'_j \) that is deep-equal to \( \delta_i \); and vice versa. Condition 2 checks if for each update primitive \( \delta_i \in \Delta_{ins} \), \( \Delta_{ins} \) contains exactly one update primitive \( \delta'_j \) that is deep-equal to \( \delta_i \); and vice versa. Condition 3 checks if \( \Delta_{ins} \) and \( \Delta_{ins}' \) both contain the same number of deep-equal insertion primitives. It does this by first computing, for each distinct primitive \( \delta_i \) in \( \Delta_{ins} \), a subgroup \( \delta_i \), which consists of all primitives in \( \Delta_{ins} \) that are deep-equal to \( \delta_i \). Then, for each such subgroup \( \delta_i \), the corresponding group \( \delta'_i \) with primitives from \( \Delta_{ins} \) is computed. Finally, \( \delta_i \) and \( \delta'_i \) are compared to see if they are deep-equal. Here, \( \text{distinct-primitives()} \) is defined as:

\[
\text{distinct-primitives}(\Delta) = \{ \Delta^{\text{dist}} | \forall \delta_i \in \Delta, \exists \delta_j^{\text{dist}} \in \Delta^{\text{dist}} : \text{dEq}(\delta_i, \delta_j^{\text{dist}}) \land \\
\text{dEq}(\delta^{\text{dist}}_i, \delta_i) \land i \neq j \land \text{dEq}(\delta'^{\text{dist}}_i, \delta^{\text{dist}}_j) \}\n\]

**Definition 6.1.9. Deep-equal updating queries (with implementation freedom):** Two updating XQuery queries \( Q \) and \( Q' \) are deep-equal with implementation freedom to each other (for short: deep-equal), denoted \( \text{dEq}(Q, Q') \), if, in any database state, the evaluations of \( Q \) and \( Q' \) yield the tuples \((\Delta, \Delta')\) and \((\Delta', \Delta')\), respectively, where \( \Delta \) and \( \Delta' \) are deep-equal:

\[
\forall db : db \vdash Q \Rightarrow ((\Delta, \Delta) \land db \vdash Q' \Rightarrow ((\Delta', \Delta') \land \text{dEq}(\delta, \delta')) \Rightarrow \text{dEq}(Q, Q')
\]

### 6.1.5 Sequence Properties

We define several properties concerning XML node typed items in XQuery sequences. A sequence \( \vec{s} \) is **distinct**, denoted \( \eta(\vec{s}) \), if \( \vec{s} \) does not contain duplicate XML nodes. **Disjunct** is a more strict property: \( \vec{s} \) is disjunct, denoted \( \mu(\vec{s}) \), if none of the XML node typed items in \( \vec{s} \) has an ancestor/descendant relationship with another node typed item in \( \vec{s} \). Finally, \( \vec{s} \) is **ordered**, denoted \( \sigma(\vec{s}) \), if all XML node typed items in \( \vec{s} \) appear in document order. Formally:

**Property 6.1.10. Sequence properties:**

- **Distinct** \( \eta \):
  \[
  \eta(\vec{s}) \Leftrightarrow \forall s_i \in \vec{s}. \exists s_j \in \vec{s}. i \neq j \land \text{type}(s_i) \in \mathbb{T}. \text{type}(s_j) \in \mathbb{T}: s_j = s_i
  \]

- **Disjunct** \( \mu \):
  \[
  \mu(\vec{s}) \Leftrightarrow \forall s_i \in \vec{s}. \exists s_j \in \vec{s}. i \neq j \land \text{type}(s_i) \in \mathbb{T}. \text{type}(s_j) \in \mathbb{T}: s_j \in \{s_i/d-o-s::node()\}
  \]

- **Ordered** \( \sigma \):
  \[
  \sigma(\vec{s}) \Leftrightarrow \forall i, j \in [1..|\vec{s}|]. i < j \land \text{type}(\vec{s}[i]) \in \mathbb{T}. \text{type}(\vec{s}[j]) \in \mathbb{T}: \vec{s}[i] < \vec{s}[j] \Rightarrow \text{true}
  \]

The next three lemmas can be deduced directly from the above property definitions:

**Lemma 6.1.11. Empty sequence property:** The empty sequence is always distinct, disjunct and ordered.

**Lemma 6.1.12. Single item property:** Sequences containing a single item are always distinct, disjunct and ordered.

**Lemma 6.1.13. Disjunct implies distinct:** If a sequence is disjunct, then it is also distinct.

As we will see later, the combination of distinct and ordered properties is needed for determining if the resulting sequence of applying the function \( \text{fs:distinct-doc-order()} \) on a sequence is equivalent with the original sequence. The combination of disjunct and ordered properties is crucial in the conservative algorithm to determine if forward XPath steps on XML nodes from remote peers would return correct results.
6.1.6 XPath Steps and distinct-doc-order

The XQuery 1.0 and XPath 2.0 Formal Semantics [67] defines a function \( \text{fs:distinct-doc-order}() \) (\( \text{ddo}() \) for short), which sorts its input nodes sequence by document order and removes duplicates. It is trivial to see that if the input sequence of nodes \( \vec{e} \) is distinct and ordered, the result of \( \text{ddo}(\vec{e}) \) is equivalent with \( \vec{e} \):

**Lemma 6.1.14.** DDO equivalence: \( \eta(\vec{e}) \land \sigma(\vec{e}) \Rightarrow \vec{e} \equiv \text{ddo}(\vec{e}) \)

XQuery requires that XPath expressions return their resulting nodes sequences in document order with duplicates eliminated. This is ensured in the XQuery Formal Semantics [67] by passing the intermediate result of an XPath expression to the function \( \text{fs:distinct-doc-order}() \) to produce the final result.

**Definition 6.1.15.** Raw XPath result: We use the notation: \( \vec{e}/\text{as}::\text{NT} = \text{ddo}(\vec{e}/\text{as}::\text{NT}) \), where \( \text{as} \) and \( \text{as} \) are the same AxisStep, to differentiate the input (indicated by the lower-cased abbreviation \( \text{as} \)) and the output of \( \text{ddo}() \). We call \( \vec{e}/\text{as}::\text{NT} \) the raw result of applying an XPath step \( \text{as}::\text{NT} \) on the node sequence \( \vec{e} \).

Let \( n = |\vec{e}| \), the semantics of \( \vec{e}/\text{as}::\text{NT} \) is defined as:

\[
\vec{e}/\text{as}::\text{NT} = (\vec{e}[1]/\text{as}::\text{NT},...,\vec{e}[n]/\text{as}::\text{NT}),
\]

which is a literal concatenation of the resulting sequence of applying \( \text{as}::\text{NT} \) on each node \( \vec{e}[i] \in \vec{e} \), in the same order they appear in \( \vec{e} \) (for short: sequence order).

If \( \vec{e} \) is a sequence consisting of only one node, i.e., \( \vec{e} = (e) \), it is directly seen that \( \vec{e}/\text{as}::\text{NT} \) is equivalent to \( \vec{e}/\text{as}::\text{NT} \).

**Lemma 6.1.16.** Raw XPath result on single node: The raw result of applying an XPath step on a single XML node \( e \) is distinct and ordered: \( \eta(e/\text{as}::\text{NT}) \) and \( \sigma(e/\text{as}::\text{NT}) \).

Now we are ready to deduce the properties of the result of a \( \text{FwdAxis} \) step\(^6\):

**Lemma 6.1.17.** Distinct-and-ordered \( \text{FwdAxis} \): Let \( \vec{e} \) be a sequence of XML nodes and \( \vec{e}/\text{as}::\text{NT} \), where \( \text{as} \in \text{FwdAxis} \), the raw result of applying a \( \text{FwdAxis} \) step on each node in \( \vec{e} \). If \( \vec{e} \) is disjunct, then \( \vec{e}/\text{as}::\text{NT} \) is distinct; if \( \vec{e} \) is ordered, then \( \vec{e}/\text{as}::\text{NT} \) is also ordered:

\[
\forall \text{as} \in \text{FwdAxis} : \mu(\vec{e}) \Rightarrow \eta(\vec{e}/\text{as}::\text{NT}), \sigma(\vec{e}) \Rightarrow \sigma(\vec{e}/\text{as}::\text{NT})
\]

**Proof.** The proof can be done by induction: the base case \( |\vec{e}| = 0 \) is trivial; the other case \( |\vec{e}| = 1 \) is proven by Lemma 6.1.16. We assume that the lemma holds when \( |\vec{e}| = n \), and we need to prove that the lemma also holds when \( |\vec{e}| = n + 1 \). Let \( \vec{e} = (\vec{e}_n, e_{n+1}) \) and \( \vec{e}_n = (e_1, ..., e_n) \), we have the following hypotheses:

\[
\begin{align*}
(h0) \quad & \text{as} \in \text{FwdAxis} \\
(h1-a) \quad & \mu(\vec{e}_n) \Rightarrow \eta(\vec{e}_n/\text{as}::\text{NT}) \\
(h1-b) \quad & \sigma(\vec{e}_n) \Rightarrow \sigma(\vec{e}_n/\text{as}::\text{NT}) \\
(h2) \quad & \vec{e}/\text{as}::\text{NT} = (\vec{e}_n/\text{as}::\text{NT}, e_{n+1}/\text{as}::\text{NT}) \\
(h3-a) \quad & \mu(\vec{e})
\end{align*}
\]

\(^6\)Similar work has been done by Hidders et al. in [96] and Fernández et al. in [73] to avoid unnecessary ordering and duplicate elimination operations in XPath expressions. The authors present rules to infer the ordered and distinct properties of the results of XPath steps on all axes, except \( \text{self} \).
(h3-b) \( \eta(\vec{r}) \)
(h3-c) \( \sigma(\vec{r}) \)
to prove that \( \eta(\vec{r}/as::NT) \) and \( \sigma(\vec{r}/as::NT) \) hold, it is equivalent to prove:

(t1) \( \eta((\vec{e}_n/as::NT, e_{n+1}/as::NT)) \)
(t2) \( \sigma((\vec{e}_n/as::NT, e_{n+1}/as::NT)) \)

Since \( e_{n+1} \) is a single node, by Lemma 6.1.16 we have:

(h4-a) \( \eta(e_{n+1}/as::NT) \)
(h4-b) \( \sigma(e_{n+1}/as::NT) \)

With (h1-ab), (h4-ab) and (h2), we only need to prove that the following statements hold:

(t1’) \( \forall e_i \in (\vec{e}_n/as::NT). \forall e_j \in (e_{n+1}/as::NT): \neg(e_i \text{ is } e_j) \)
(t2’) \( \forall e_i \in (\vec{e}_n/as::NT). \forall e_j \in (e_{n+1}/as::NT): e_i \ll e_j \)

We consider all possible values of as:

- as = self:

  \( e_{n+1/self::NT} \) returns \( e_{n+1} \) (or “\( \)”). (t1’) holds, because \( e_{n+1} \) is distinct from \( \vec{e}_n \) (h3-a).

  Similarly, (t2’) holds, because \( \forall e_i \in \vec{e}_n: e_i \ll e_{n+1} \) (h3-b).

- as \( \in \{ \text{child, descendant, descendant-or-self, attribute} \} \)

  First, let us consider the case as=descendant: \( e_{n+1/descendant::NT} \) selects all descendants of \( e_{n+1} \) that satisfy the condition NT. Assume there exists a \( e_s \in e_{n+1/descendant::NT} \) that is also a descendant of a node \( e_y \) in \( \vec{e}_n \), which implies that \( e_s \) and \( e_{n+1} \) have the ancestor-descendant relationship. However, this conflicts with the hypothesis (h3-a) that \( e_{n+1} \) is disjunct with \( \vec{e}_n \). Hence, the assumption does not hold, which proves the statement:

  (t1’) \( \forall e_i \in (\vec{e}_n/descendant::NT). \forall e_j \in (e_{n+1/descendant::NT}): \neg(e_i \text{ is } e_j) \)

  From the XQuery definition of document order [38], we have:

  (h5) \( \forall e_j \in (e_{n+1/descendant::NT}): e_{n+1} \ll e_j \)

  with (h3-b) we have: \( \vec{e}_n \ll e_{n+1} \), which implies

  (h6) \( \forall e_i \in (\vec{e}_n/descendant::NT): e_i \ll e_{n+1} \).

  From (h5) and (h6), we can deduce:

  (t2’) \( \forall e_i \in (\vec{e}_n/descendant::NT). \forall e_j \in (e_{n+1/descendant::NT}): e_i \ll e_j \)

  The other three cases are proven in a similar way.

\[ \square \]

6.2 Static Properties Analysis

In Section 5.6, we have described a new runtime XML projection technique, which extends the basic compile-time XML projection technique [125] with new inference rules to handle a larger subset of expressions defined in the XQuery Core grammar. The compile-time projection technique [125] defines an inference rule for each expression in its grammar to calculate a set of projection paths, based on the subexpressions. The projection paths are an over-estimation of the set of nodes that will be touched by an expression, which are divided into a set of returned paths \( \vec{r} \) (specify nodes that are returned by the expression), and a set of used paths \( \vec{u} \) (specify nodes that are used to compute the result of the expression, but are not part of the result).

To compute the distinct, disjunct and ordered properties for an expression, we extend the main judgement rule of the path analysis with the triple \( \langle \eta, \mu, \sigma \rangle \). Thus, the judgement:
\[ Env \vdash E \Rightarrow \bar{r}, \bar{u}, (\eta, \mu, \sigma) \] holds iff, under the environment \( Env \), the expression \( E \) returns the set of paths \( \bar{r} \), uses the set of paths \( \bar{u} \), and evaluating \( E \) produces a sequence that has the properties \( (\eta, \mu, \sigma) \). Each path in \( \bar{r} \) and \( \bar{u} \) is a SimplePath as defined in Table 5.6. If the result sequence of an expression does not have a certain property, the property is replaced by the symbol \( \varnothing \) in the judgement.

In the remainder of this section, we redefine inference rules for those expressions whose results have at least one of the properties \( (\eta, \mu, \sigma) \). We omit redefining rules for expressions for which the judgement \( Env \vdash E \Rightarrow \bar{r}, \bar{u}, (\varnothing, \varnothing, \varnothing) \) always holds. In the inference rules, we use \( \bot \) in premises to denote that the value of a property is not significant.

### 6.2.1 Literal Values

\[
\begin{align*}
\text{LITERAL} & : \\
Env \vdash \text{Literal} \Rightarrow (), (), (\eta, \mu, \sigma)
\end{align*}
\]

Literal values do not access or return any paths. By Lemma 6.1.12, a literal value is always distinct, disjunct and ordered.

#### 6.2.2 Variables

\[
\begin{align*}
\text{VAR} & : \\
Env \vdash E_1 \Rightarrow \bar{r}_1, \bar{u}_1, (\eta_1, \mu_1, \sigma_1) \\
Env \vdash \text{$sx$} = E_1 \Rightarrow r_1, \bar{u}_1, (\eta_1, \mu_1, \sigma_1)
\end{align*}
\]

If a variable \( $sx$ \) is bound to the expression \( E \), accessing the variable uses and returns the same set of paths as \( E \).

The distinct, disjunct and ordered properties of a variable \( $sx$ \) is determined by these properties of the expression \( E \) to which the variable is bound.

#### 6.2.3 Sequences

\[
\begin{align*}
\text{EMPTYSEQ} & : \\
Env \vdash () \Rightarrow (), (), (\eta, \mu, \sigma)
\end{align*}
\]

The empty sequence does not use or return any paths, and it is always distinct, disjunct and ordered (Lemma 6.1.11).

\[
\begin{align*}
\text{SEQ} & : \\
Env \vdash E_1 \Rightarrow \bar{r}_1, \bar{u}_1, (\perp, \perp, \perp) \\
Env \vdash E_2 \Rightarrow \bar{r}_2, \bar{u}_2, (\perp, \perp, \perp) \\
Env \vdash (E_1, E_2) \Rightarrow r_1 \cup r_2, \bar{u}_1 \cup \bar{u}_2, \bar{v}(\varnothing, \varnothing, \varnothing)
\end{align*}
\]

The sequence expression concatenates two sequences into one sequence, without eliminating duplicate nodes or changing the order in which the items appear in the resulting sequence. The returned and used paths of a sequence expression is the union of the returned and used paths of its subexpressions. As it cannot be statically determined, this rule deduces that the result of a (non-empty) sequence expression is not distinct, disjunct or ordered.

#### 6.2.4 for Expressions

\[
\begin{align*}
\text{FOR} & : \\
Env \vdash E_1 \Rightarrow \bar{r}_1, \bar{u}_1, (\perp, \perp, \perp) \\
Env' = Env + (sx \mapsto E_1) \\
Env' \vdash E_2 \Rightarrow \bar{r}_2, \bar{u}_2, (\perp, \perp, \perp) \\
Env \vdash \text{for $sx$ in $E_1$ return $E_2$} \Rightarrow r_1 \cup r_2, \bar{u}_1 \cup \bar{u}_2, \bar{v}(\varnothing, \varnothing, \varnothing)
\end{align*}
\]

A for expression binds new variables in the environment, hence, the environment is first extended with the new variable and passed to the evaluation of \( E_2 \). A for expression returns the returned paths of its return clause. All other paths are used to calculate the result of a for expression.
Thus, all paths needed for a value comparison are used paths.

6.2. STATIC PROPERTIES ANALYSIS

for their document order, hence, this rule deduces that the result sequence of a for expression is not distinct, disjunct or ordered.

6.2.5 let Expressions

\[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1; (\perp, \perp, \perp) \]
\[ \text{Env}' = \text{Env} + \{x \mapsto E_1\} \]
\[ \text{Env}' \vdash E_2 \Rightarrow r_2, u_2; (\eta_2, \mu_2, \sigma_2) \] (LET)

A let expression binds a new variable in the environment, hence, the environment is first extended with the new variable and passed to the evaluation of \( E_2 \). A let expression returns the returned paths of its return clause. All other paths are used to calculate the result of a let expression.

A let expression has the same distinct, disjunct and ordered properties as its return expression \( E_2 \).

6.2.6 Conditionals

\[ \text{Env} \vdash E_0 \Rightarrow r_0, u_0; (\perp, \perp, \perp) \]
\[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1; (\eta_1, \mu_1, \sigma_1) \]
\[ \text{Env} \vdash E_2 \Rightarrow r_2, u_2; (\eta_2, \mu_2, \sigma_2) \] (If)

An if expression returns either the expression in the then branch or the expression in the else branch. Thus, the returned paths of an if expression is the union of the returned paths of these two expressions. All other paths are used to calculate the result of the if expression.

At compile time, we can only conclude that the result of an if expression is distinct, disjunct and ordered, iff it can be statically determined that both its then and else branches are distinct, disjunct and ordered.

6.2.7 Typeswitch

\[ \text{Env} \vdash E_n \Rightarrow r_n, u_n; (\perp, \perp, \perp) \]
\[ \text{Env} \vdash E_0 \Rightarrow r_0, u_0; (\eta_0, \mu_0, \sigma_0) \]
\[ \vdots \]
\[ \text{Env} \vdash \text{typeswitch} (E_0) \text{case } x \text{ as SequenceType1 return } E_1 \ldots \text{default } x \text{ return } E_n \]
\[ \Rightarrow r_1 \cup \ldots \cup r_n, r_0 \cup u_0 \cup \ldots \cup u_n, (\eta_1 \& \ldots \& \eta_n, \mu_1 \& \ldots \& \mu_n, \sigma_1 \& \ldots \& \sigma_n) \] (TPSWITCH)

The inference rule for typeswitch is very similar to the one for the conditionals, except that typeswitch needs to handle multiple branches.

If the return expression of all case clauses and the default clause are distinct, disjunct and ordered, the result of the typeswitch also has these properties.

6.2.8 Value and Node Comparisons

\[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1; (\perp, \perp, \perp) \]
\[ \text{Env} \vdash E_2 \Rightarrow r_2, u_2; (\perp, \perp, \perp) \] (COMP)

The symbol \( \circ \) represents the value and node comparison operators =, !=, <, <=, >, >=, is, \( \ll \) and \( \gg \). Value and node comparisons never return nodes, but a literal boolean value. Thus, all paths needed for a value comparison are used paths.
By Lemma 6.1.12, a single literal value is always distinct, disjunct and ordered, thus the result of a value comparison also has these properties, regardless of whether its subexpressions \( E_1 \) and \( E_2 \) have these properties or not.

### 6.2.9 Order Expressions

\[
\begin{align*}
\text{Env} &\vdash E_0 \Rightarrow \vec{r}_0, \vec{u}_0, (\perp, \perp, \perp) \\
\text{Env} &\vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\vdots \\
\text{Env} &\vdash E_n \Rightarrow \vec{r}_n, \vec{u}_n, (\perp, \perp, \perp) \\
\text{Env} &\vdash (E_0 \text{ order by } E_1 \text{ ascending|descending} \ldots E_n \text{ ascending|descending}) \\
&\Rightarrow \vec{r}_0, \vec{u}_0 \cup \vec{r}_1 \cup \ldots \cup \vec{r}_n \cup \vec{u}_1 \cup \ldots \cup \vec{u}_n, (\eta_0, \mu_0, \varnothing)
\end{align*}
\]

An order by expression returns the expression \( E_0 \) reordered by the OrderSpec expressions \( E_1, \ldots, E_n \). Thus, the returned paths of \( E_0 \) are also the returned paths of the order by expression, and all other paths are propagated as the used paths of the order by expression.

The distinct and disjunct properties of an order by expression are determined by its input expression \( E_0 \). However, the result of an order by expression is regarded as never ordered by document order, because it can not be determined at compile time how the result sequence will be ordered.

### 6.2.10 Node Set Expressions

\[
\begin{align*}
\text{Env} &\vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\text{Env} &\vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, (\perp, \perp, \perp) \\
\text{Env} &\vdash E_1 \text{ union } E_2 \Rightarrow \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, (\eta, \varnothing, \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{Env} &\vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \mu_1, \perp) \\
\text{Env} &\vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, (\perp, \perp, \perp) \\
\text{Env} &\vdash E_1 \text{ intersect } E_2 \Rightarrow \vec{r}_1 \cap \vec{r}_2, \vec{u}_1 \cap \vec{u}_2, (\eta, \mu_1, \sigma)
\end{align*}
\]

The symbol \( \square \) represents the node set operators intersect and except. For all three kinds of node set expressions, it holds that (i) the returned and used paths are respectively the union of the returned and used paths of their subexpressions, and (ii) their results are always distinct and ordered as required by the XQuery language.

However, situations are different regarding the disjunct property. The result of a union expression is never disjunct, because it combines nodes from two sequences. Even if the two subexpressions \( E_1 \) and \( E_2 \) are disjunct themselves, statically, it can not be determined if all nodes in \( E_2 \) are disjunct with all nodes in \( E_1 \). The operators intersect and except only return nodes from their first subexpression \( E_1 \), hence, their disjunctness depends on that of \( E_1 \).

### 6.2.11 Constructors

\[
\begin{align*}
\text{Env} &\vdash E_0 \Rightarrow \vec{r}_0, \vec{u}_0, (\perp, \perp, \perp) \\
\text{Env} &\vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\text{Env} &\vdash \text{element|attribute } \{E_0\}\{E_1\} \Rightarrow \text{doc}(v_1:v_i), \vec{r}_0 \cup \vec{u}_0 \cup \vec{r}_1 \cup \vec{u}_1, (\eta, \mu, \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{Env} &\vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\text{Env} &\vdash \text{document|text } \{E_1\} \Rightarrow \text{doc}(v_1:v_i), \vec{r}_1 \cup (\vec{r}_1/\text{descendant::}) \cup \vec{u}_1, (\eta, \mu, \sigma)
\end{align*}
\]

Constructors make a deep copy of their operands. The newly constructed element is annotated with a synthetic (unique) URI \( \text{doc}(v_1:v_i) \) to denote the \( d \)-graph vertex \( v_i \), from which
the element originates (see Section 5.4). Evaluation of the name expression $E_0$ always yields a single literal string value. Constructors do not return any nodes from the original XML nodes or documents. However, as the whole subtree of nodes in the content expression $E_2$ are copied, we add all descendants of those nodes to the set of used paths.

As constructors always return a single fresh node, their result is always distinct, disjunct and ordered.

### 6.2.12 XPath Expressions

\[
\begin{align*}
\text{Env} \vdash E_0 &\Rightarrow \tilde{r}_0, \tilde{u}_0, \langle 1, 1, 1 \rangle & \text{(STEPa)} \\
\text{Env} \vdash E_0/\text{attribute::NT} &\Rightarrow \tilde{r}_0/\text{attribute::NT}, \tilde{r}_0 \cup \tilde{u}_0, \langle \eta, \mu, \sigma \rangle \\
\text{Env} \vdash E_0 &\Rightarrow \tilde{r}_0, \tilde{u}_0, \langle 1, \mu_0, 1 \rangle & \text{(STEPb)} \\
\text{Env} \vdash E_0/\text{AS::NT} &\Rightarrow \tilde{r}_0/\text{AS::NT}, \tilde{r}_0 \cup \tilde{u}_0, \langle \eta, \mu_0, \sigma \rangle \\
\text{Env} \vdash E_0 &\Rightarrow \tilde{r}_0, \tilde{u}_0, \langle 1, 1, 1 \rangle & \text{(STEPc)} \\
\text{Env} \vdash E_0/\text{AS::NT} &\Rightarrow \tilde{r}_0/\text{AS::NT}, \tilde{r}_0 \cup \tilde{u}_0, \langle \eta, \sigma, \sigma \rangle & \text{(STEPd)}
\end{align*}
\]

An XPath step applies a `StepExpr` on each returned path of its subexpression $E_0$. The result of an XPath step is always distinct and ordered, as required by the XQuery language. To compute the disjunct property, however, the path steps need to be treated differently.

- **STEPa**: attribute nodes of distinct nodes never overlap, thus, the result of an attribute step is always disjunct.

- **STEPb**: if $E_0$ is disjunct, the result of applying a `self`, `child`, `preceding-sibling` or `following-sibling` step on $E_0$ is also disjunct. The case $E_0/\text{self::NT}$ is trivial. The child nodes of a single node are disjunct, and all nodes in $E_0$ are disjunct, so $E_0/\text{child::NT}$ is disjunct as well.

- **STEPc**: for the remaining axis steps that include `ancestor`, `ancestor-or-self`, `parent`, `preceding`, `preceding-sibling`, `following`, `following-sibling`, `descendant` and `descendant-or-self`, it can not be statically determined if their results are disjunct, regardless of if the result of $E_0$ is disjunct or not.

### 6.2.13 Built-in Function Calls

\[
\begin{align*}
\forall i \in 1..k: \text{Env} \vdash E_i &\Rightarrow \tilde{r}_i, \tilde{u}_i, \langle \eta_i, \mu_i, \sigma_i \rangle \\
\mathcal{R}(\mathcal{F}(E_1, ..., E_k)) &\Rightarrow \tilde{r}_f, \tilde{u}_f, \langle \eta_f, \mu_f, \sigma_f \rangle & \text{(Blin)} \\
\text{Env} \vdash \mathcal{F}(E_1, ..., E_k) &\Rightarrow \tilde{r}_f, \bigcup_{i=1}^k (\tilde{r}_i \cup \tilde{u}_i), \langle \eta_f, \mu_f, \sigma_f \rangle
\end{align*}
\]

For each built-in function, we assume that there is a corresponding helper rule $\mathcal{R}$ which specifies how the returned paths of the function results depend on the returned paths of the parameters and if the result of the function is distinct, disjunct or ordered. The helper rules for all built-in functions defined in [124] are listed in Appendix C.
6.2.14 Transform Expressions

\[ \text{Env} \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, (\perp, \perp, \perp) \]

\[ \text{Env} \vdash E_n \Rightarrow \vec{r}_n, \vec{u}_n, (\perp, \perp, \perp) \]

\[ \text{Env} \vdash E_m \Rightarrow \vec{r}_m, \vec{u}_m, (\perp, \perp, \perp) \]

\[ \text{Env} \vdash E_{n} \Rightarrow \vec{r}_n, \vec{u}_n, (\eta, \mu, \sigma) \]

\[ \text{Env} \vdash \text{copy } S_x := E_1, \ldots, S_{x_n} := E_{n_m} \text{ modify } E_{n} \text{ return } E_{r} \]

\[ \Rightarrow \vec{r}_i, (\bigcup_{i=1}^{n} (\vec{r}_i \cup \vec{u}_i \cup \vec{r}_i/\text{descendant::}*)) \cup \vec{r}_m \cup \vec{u}_m \cup \vec{u}_n, (\eta, \mu, \sigma) \]

A TransformExpr expression returns the value of the expression in its return clause, thus the returned paths of a TransformExpr are the returned paths of \( E_{r} \), all other paths are propagated as used paths. Since deep-copies of the results of \( E_1, \ldots, E_n \) are made, it is necessary to add their descendants into used paths.

A TransformExpr makes deep copies of the subexpressions in its copy clause, and each newly created node gets a new node identity. The subexpression in the modify clause must be an updating expression (or a vacuous expression [58]), which does not return any value. Thus, the \( \eta, \mu, \sigma \) properties of a TransformExpr is determined by the \( \eta, \mu, \sigma \) properties of the return expression \( E_{r} \).

6.3 Correctness Proof of the Conservative Decomposition Algorithm

The semantics of marshalling and unmarshalling function parameters (or results) in XRPC under the pass-by-value semantics is defined by the functions \( s2n() \) and \( n2s() \) (see Section 3.4). The effect of marshalling \( (s2n()) \) a sequence and then unmarshalling it \( (n2s()) \) is that a deep-copy of the sequence is created.

Definition 6.3.1. By-value copy operator \( C^{v} \): Let \( \vec{s} \) be an XQuery sequence that may contain duplicates or overlapping nodes, we use \( C^{v}(\vec{s}) \) to indicate a by-value copy of \( \vec{s} \). The semantics of the by-value copy operator \( C^{v} \) is defined as: \( C^{v}(\vec{s}) = n2s(s2n(\vec{s})) \).

Under the pass-by-value semantics, the semantics of executing an expression on a remote peer is to first replace each parameter of the expression with a by-value copy, then execute the expression (using the by-value copies on the local peer), and finally return a by-value copy of the result.

Property 6.3.2. \( C^{v} \) properties:

\[ \text{dEq}(\vec{s}, C^{v}(\vec{s})) \Rightarrow \eta(\vec{s}) \Rightarrow \eta(C^{v}(\vec{s})) \]

\[ \mu(\vec{s}) \Rightarrow \mu(C^{v}(\vec{s})) \]

\[ \sigma(\vec{s}) \Rightarrow \sigma(C^{v}(\vec{s})) \]

Below we introduce a mapping function to denote the relationship between two corresponding nodes in the subtrees rooted at node typed items in \( \vec{s} \) and \( C^{v}(\vec{s}) \), respectively.

Definition 6.3.3. By-value mapping function \( m_{v} \): Let \( \vec{e} \) be a sequence of XML nodes and \( C^{v}(\vec{e}) \) its by-value copy. The by-value mapping function \( m_{v} \) maps each node \( e_{i} \in \{ \vec{e}/d-o-s::\text{node()} \} \) to exactly one node \( e'_{i} \in \{ C^{v}(\vec{e})/d-o-s::\text{node()} \} \), and vice versa. Formally, the result of \( m_{v} \) is defined as the following:

\[ \forall i \in 1..|\vec{e}| \forall j \in 1..|C^{v}(\vec{e})/d-o-s::\text{node()}| : m_{v}(\vec{e}[i]/d-o-s::\text{node()}[j]) \text{ is } C^{v}(\vec{e})[i]/d-o-s::\text{node()}[j] \]
For each \( e_i \), \( m_v(e_i) \) is called the by-value-mapping of \( e_i \) in \( C^v(\vec{e}) \). The reverse function \( m_v^{-1} \) maps an \( \vec{e}' \) back to its corresponding \( e_i \), such that, \( e_i \equiv m_v^{-1}(m_v(e_i)) \).

**Lemma 6.3.4.** Mapped raw results of \( \text{FwdAxis} \) steps on \( \vec{e} \) and \( C^v(\vec{e}) \): Let \( \vec{e} \) be a sequence of XML nodes and \( C^v(\vec{e}) \) its by-value copy. The raw results of applying multiple consecutive \( \text{FwdAxis} \) steps on \( \vec{e} \) and \( C^v(\vec{e}) \) are the by-value-mapping of each other, denoted: \( C^v(\vec{e})|\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \equiv m_v(\vec{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \), where \( \forall \text{as}_i \in \text{FwdAxis} \).

**Proof.** If \( \vec{e} = (e) \) and \( n = 1 \), it is trivial to see that the following holds:

1. \( C^v(e)/\text{as}_1::\text{NT}_1 = m_v(e/\text{as}_1::\text{NT}_1) \)

If \( |\vec{e}| > 1 \) and \( n = 1 \), with Definition 6.1.15, we know that the raw results of applying \( \text{as}_1::\text{NT}_1 \) on \( \vec{e} \) is a literal concatenation of the intermediate raw result \( \vec{e}_1[i]/\text{as}_1::\text{NT}_1 \) in sequence order. The same holds for \( C^v(\vec{e}) \). Thus, we have:

2. \( C^v(\vec{e})/\text{as}_1::\text{NT}_1 = m_v(\vec{e}/\text{as}_1::\text{NT}_1) \)

If \( |\vec{e}| > 1 \) and \( n > 1 \), the raw results are again a concatenation of each intermediate raw result (i.e., one step on one node) in sequence order, which implies:

3. \( C^v(\vec{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n = m_v(\vec{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \)

**Definition 6.3.5.** Node relationship function \( R \): Let \( e_i \) and \( e_r \) be two XML nodes, the relationship function \( R \) takes \( e_i \) and \( e_r \) as its input and returns the relationship between \( e_i \) and \( e_r \), as the following:

\[
R(e_i, e_r) = \begin{cases} 
"\ll", & \text{if } e_i \ll e_r; \\
"\gg\gg", & \text{if } e_i \gg e_r.
\end{cases}
\]

Note that \( R \) is exactly what is needed by \( \text{ddo()} \) to process its input sequence. The following lemma can be directly deduced from the definition of by-value copy:

**Lemma 6.3.6.** By-value node relationships: Let \( e \) be a single XML node and \( C^v(e) \) its by-value copy. Then, we have:

\[
\forall u, w \in \{e/d-o-s::\text{node}()\}. \forall u', w' \in \{C^v(e)/d-o-s::\text{node}()\}. u' = m_v(u) \land w' = m_v(w) \Rightarrow R(u, w) = R(u', w')
\]

That is, the relationship between any two nodes \( u \) and \( w \) in the subtree rooted at \( e \) is the same as the relationship between their corresponding nodes \( u' \) and \( w' \) in \( C^v(e) \).

**Lemma 6.3.7.** By-value deep-equal \( \text{ddo()} \): Let \( e \) be a single XML node and \( C^v(e) \) its by-value copy. Let \( \vec{a} \) be a sequence containing XML nodes in the subtree rooted at \( e \), i.e., \( \forall a_i \in \{e/d-o-s::\text{node}()\} \), and \( \vec{b} \) the by-value mapping of \( \vec{a} \) in \( C^v(e) \), then: \( \text{dEq} (\text{ddo}(\vec{a}), \text{ddo}(\vec{b})) \).

**Proof.** With \( \vec{b} = m_v(\vec{a}) \), we have:

- (h1) \( \forall i \in 1..|\vec{a}| : \vec{b}[i] = m_v(\vec{a}[i]) \)

With (h1) and Lemma 6.3.6, we have:

- (h2) \( \forall i, j \in 1..|\vec{a}| : R(\vec{a}[i], \vec{a}[j]) = R(\vec{b}[i], \vec{b}[j]) \)

That is, the relationship between any two nodes in \( \vec{a} \) is the same as the corresponding nodes in \( \vec{b} \). This directly leads to \( \text{dEq} (\text{ddo}(\vec{a}), \text{ddo}(\vec{b})) \).

**Lemma 6.3.8.** By-value deep-equal \( \text{FwdAxis} \): Let \( \vec{e} \) be a sequence of XML nodes and \( C^v(\vec{e}) \) its by-value copy. If \( \vec{e} \) is disjunct and ordered, the result sequences of applying any number of consecutive \( \text{FwdAxis} \) steps on \( \vec{e} \) and \( C^v(\vec{e}) \) are deep-equal to each other. Formally:

\[
\mu(\vec{e}), \sigma(\vec{e}) \Rightarrow \text{dEq} (\vec{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n, C^v(\vec{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \land \forall i \in 1..n : \text{as}_i \in \text{FwdAxis}
\]
Proof. We prove this lemma in two steps. First, we show that the lemma holds if \( \vec{e} \) contains a single XML node. Then, we generalise the proof to the case that \( \vec{e} \) contains multiple XML nodes. Note that computing the final results of applying XPath steps on a node sequence can be done by first computing the raw results (Definition 6.1.15), and then applying \( \text{ddo}() \) on the raw results to eliminate duplicates and sort nodes (Section 6.1.6). Lemma 6.3.4 has already proven that \( \vec{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \) and \( C^v(\vec{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \) are by-value-mapping of each other. Thus, the crucial issue in this proof is to show that \( \text{ddo}() \) works correctly on \( C^v(\vec{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \).

Case 1: \( \vec{e} = (e) \)

With Lemma 6.3.4, we have:

\[
(\text{h1-1}) \quad C^v(e)/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n = \text{as}_v(e/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

For short, we use \( \vec{u} \) and \( \vec{w} \) to denote \( e/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \) and \( C^v(e)/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \), respectively. Since all \( \text{as}_j \) are \( \text{FwdAxis} \) steps, we have:

\[
(\text{h1-2}) \quad \forall x \in 1..[\vec{u}]: \vec{u}[x] \in \{e/d-o-s::\text{node}()\} \quad \text{and} \quad \vec{w}[x] \in \{C^v(e)/d-o-s::\text{node}()\}
\]

With (h1-1) and (h1-2) Lemma 6.1.14, we have:

\[
(\text{h1-3}) \quad \text{Eq}(\text{ddo}(\vec{u}), \text{ddo}(\vec{w}))
\]

With (h1-3) and Definition 6.1.15, we have:

\[
(\text{t1}) \quad \text{Eq}(e/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n, C^v(e)/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

Thus, the lemma holds, when \( \vec{e} = (e) \).

Case 2: \( \vec{e} = (e_1,...,e_x) \)

With (t1), we have:

\[
(\text{h2-1}) \quad \forall x \in 1..[\vec{e}]: \text{Eq}(\vec{e}[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n, C^v(\vec{e})[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

With Property 6.3.2, we have:

\[
(\text{h2-2}) \quad \mu(\vec{e}) \Rightarrow \mu(C^v(\vec{e})), \sigma(\vec{e}) \Rightarrow \sigma(C^v(\vec{e}))
\]

Which implies that the intermediate results of applying \( \text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n \) on each node in \( \vec{e} \) (or \( C^v(\vec{e}) \)) in sequence order, are distinct and ordered, i.e., \( \forall i, j \in 1..[\vec{e}] \wedge i < j \):

\[
(\text{h2-3}) \quad \forall u \in \{\vec{e}[i]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n\}, \forall w \in \{\vec{e}[j]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n\}:
\]

\[
R(u, w) = "\ll"
\]

\[
(\text{h2-4}) \quad \forall u' \in \{C^v(\vec{e})[i]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n\}, \forall w' \in \{C^v(\vec{e})[j]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n\}:
\]

\[
R(u', w') = "\ll"
\]

With (h2-3), (h2-4) and Lemma 6.1.14 we have:

\[
(\text{h2-5}) \quad \vec{e}[1]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n,...,\vec{e}[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \equiv \text{ddo}(\vec{e}[1]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n,...,\vec{e}[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

\[
(\text{h2-6}) \quad C^v(\vec{e})[1]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n,...,C^v(\vec{e})[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \equiv \text{ddo}(C^v(\vec{e})[1]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n,...,C^v(\vec{e})[x]/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

Which implies:

\[
(\text{t2}) \quad \text{Eq}(\vec{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n, C^v(\vec{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)
\]

Thus, the lemma also holds, when \( \vec{e} = (e_1,...,e_x) \).

Finally, the correctness of the conservative decomposition algorithm is defined as follows:

**Theorem 6.3.9. Conservative Decomposition Correctness:** Let \( Q \) be a normal read-only XCore query (i.e., without any XRPC expressions) and \( G \) the corresponding d-graph. \( \Gamma(G) \subset G \) is the non-empty set of decomposition points validated by the by-value insertion conditions. Let \( G' \) be the d-graph derived by doing an XRPCExpr insertion above each vertex in \( \Gamma(G) \)
This can be illustrated more clearly by an example, as shown in Figure 6.2. Then, \( dEq(Q, Q') \) holds, under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).

**Proof.** We prove this theorem by contradiction: assume \( \neg dEq(Q, Q') \). Then there must exist one vertex \( v_x \in G \), which depends on by-value copies of remote sequences\(^7\), and its corresponding vertex in \( G' \) is \( v'_x \) such that \( v_x \) and \( v'_x \) are not deep-equal, even if each vertex \( v_z \), on which \( v_x \) depends, is deep-equal to its corresponding vertex \( v'_z \) in \( G' \). Formally, \( \exists v_x \in G \) and \( \exists v'_x \in G' \), such that:

\[
\begin{align*}
(Cnd1) & \quad \neg dEq(v_x, v'_x) \land \forall v_z \in \{v_1 \ldots v_{11}\} . \forall v'_z \in \{v'_1 \ldots v'_{11}\} . dEq(v_z, v'_z) \\
(Cnd2) & \quad (v_x \in I'(G) \land \exists v_0 \in V(G) \setminus V(G_x) \land v_x \leadsto v_0) \lor (\exists v_0 \in I'(G) \land v_x \leadsto v_0)
\end{align*}
\]

This can be illustrated more clearly by an example, as shown in Figure 6.2.

The figure shows a d-graph \( G \) with \( I'(G) = \{v_4, v_6\} \) at its left side. The varref-edge \( (v_8, v_3) \) determines that the vertices \( v_1 \) and \( v_8 \) represent a Var and a VarRef grammar rule, respectively. At the right side of the figure, it is the corresponding decomposed d-graph \( G' \), in which the effect of remote executions of \( v_4 \) and \( v_6 \) is indicated using by-value copy operators. As \( v_4 \) does not depend on any copied values, the effect of executing \( v_4 \) remotely is equivalent to executing \( v_4 \) locally, and return a by-value copy of the result. The effect of executing \( v_6 \) remotely is equivalent to executing \( v_6 \) locally on a by-value copy of its parameter \( v_8 \) and return a by-value copy of the result. Whether \( dEq(v_6, v'_6) \) holds is the subject of this proof, as \( v'_6 \) depends on a copied parameter.

Thus, the set of vertices that could possibly return non-deep-equal results includes \( \{v_1, v_2, v_3, v_6, v_7, v_8\} \). We need to check (i) if remote executions of \( v_6 \) would produce non-deep-equal results, because it depends on a by-value copied parameter; and (ii) if the vertices \( v_1, v_2, v_3, v_7 \) and \( v_8 \) return non-deep-equal results, because they depend on by-value copies of remote results. Note that, when checking, for instance, if \( \neg dEq(v_4, v'_4) \) holds, we have the hypothesis: \( \forall i \in 3, 4, \ldots, 7, 11 : dEq(v_i, v'_i) \). At this point, it is safe to assume, e.g., that \( dEq(v_3, v'_3) \) holds, even if \( v_3 \) depends on \( v_4 \), because, otherwise, \( v_3 \) is already a \( v_x \).

In the remainder of this proof, we examine each kind expression in the XCore grammar (Table 5.2) to see if it could be a \( v_x \), i.e., expressions that produces non-deep-equal results, if any of their subexpressions are replaced by a deep-equal copy (either as parameters for remote executions, or as results of remote executions).

---

\(^7\)Either \( v_x \) is in \( I'(G) \) and uses those sequences as its parameters; or those sequences are results of remote executions and are used by \( v_x \).
6.3.9.1 Easy cases

The easy cases include Literal, variables, ExprSeq, IfExpr, Typeswitch, OrderExpr, value comparisons, and constructors. The results of these expressions are not determined by XML node identities or structural properties. Hence, it is easy to see that for an expression in this group, replacing any of its subexpressions by a deep-equal copy would not alter the result of this expression:

- Literals and variables are trivial cases.
- An ExprSeq simply concatenates the resulting sequences of its subexpressions, thus, replacing any of the subexpressions of the ExprSeq with a deep-equal copy will produce deep-equal result.
- The boolean value of the condition of an IfExpr determines the branch to be returned, and it could be replaced by a deep-equal copy without altering the decision. Since an IfExpr returns either its then or its else branch, the IfExpr would return deep-equal result if any of its branches is replaced with a deep-equal copy. Similar reasoning applies to Typeswitch and OrderExpr.
- Value comparisons compare the literal values of their operands.
- Constructors always produce fresh nodes by making a deep-copy of their operands.

Hence, $v_x$ can not be an expression listed above.

6.3.9.2 ForExpr and LetExpr

Assume $v_x$ is a ForExpr, according to condition (Cnd1), the following statement must be true:

\[
\frac{dEq(E_0, E'_0)}{\neg dEq(\text{for } x \in E_0 \text{ return } E_1, \text{ for } x \in E'_0 \text{ return } E'_1)}
\]  

(t_for)

Directly, the value of $E_0$ only determines the number of iterations of a for-loop, and it is trivial to see that $E'_0$ leads to the same number of iterations. The result of a for expression is mainly determined by the result of its return clause $E_1$, which possibly has dependency on $E_0$ (indicated in the second premise by passing $E_0$ as a parameter to $E_1$). As we have pointed out earlier, at this point, it is safe to assume that $dEq(E_1(E_0), E'_1(E'_0))$ holds. Thus, replacing $E_1$ by $E'_1$ implies that each iteration would produce a deep-equal sequence, and clearly, the concatenation of all those sequences is again deep-equal.

Hence, $v_x$ can not be a ForExpr, and similarly, $v_x$ could not be a LetExpr.

6.3.9.3 NodeCmp and NodeSetExpr

The by-value insertion condition ii (Section 5.4.1) states that there must not exist a valid decomposition point on which a NodeCmp (is, ≪, ≫) or a NodeSetExpr (union, intersect, except) depends. This prevents NodeCmp and NodeSetExpr expressions from using by-value copied results of remote executions. Condition ii also states that a decomposed NodeCmp or NodeSetExpr expression may not depends on any sequences outside the decomposed subgraph, preventing these expressions from using copied parameters. This contradicts with (Cnd2) above, which state that a $v_x$ must depend on at least one (copied) remote sequence.

Hence, $v_x$ can not be a node comparison or a node set expression.
6.3.9.4 StepExpr

The by-value insertion condition i excludes RevAxis and HorAxis from depending on a (by-value) copied sequence, which contradicts with \((Cnd2)\) above. Thus, assume \(v_x\) is a StepExpr, according to \((Cnd1)\) above, the following statement must be true:

\[
\frac{dEq(E_1, E'_1)}{\neg dEq(E_1/AS::NT, E'_1/AS::NT)} \quad (t'_\text{Step})
\]

With Lemma 6.3.8 we have: \(dEq(E_1/AS::NT, E'_1/AS::NT)\) iff \(AS \in FwdAxis\) and \(E_1\) is disjunct and ordered. Thus, the statement \(t'_\text{Step}\) is true if we can find an \(E_1\) which is non-disjunct or unordered.

According to the static property analysis rules (Section 6.2), these expressions return non-disjunct results (regardless of the properties of their subexpressions): ExprSeq, ForExpr, union, parent, ancestor, ancestor-or-self, descendant, descendant-or-self, preceding, preceding-sibling, following and following-sibling, and order by expressions return unordered results. However, the by-value insertion condition iv forbids a FwdAxis step to depend on copies of any of these expressions. This implies that \(E_1\) is always disjunct and ordered, thus, the statement \(t'_\text{Step}\) is not true.

Hence, \(v_x\) can not be a StepExpr.

6.3.9.5 Function calls

Assume \(v_x\) is a FunCall to a built-in function \(f(r_1, ..., r_k)\)\(^8\), according to \((Cnd1)\) above, the following statement must be true:

\[
\frac{\forall i \in 1..k : dEq(p_i, p'_i), f \in \text{built-in}}{\neg dEq(f(p_1, ..., p_k), f(p'_1, ..., p'_k))} \quad (t'_\text{BltIn})
\]

A built-in function \(f\) would return non-deep-equal result if it needs to access values outside the subtrees of its parameters. Such built-in functions include fn:id(), fn:idref(), fn:root() and fn:lang(). However, the by-value insertion condition iv prevents any of these functions to depend on copied parameters.

Hence, \(v_x\) can not be a FunCall to a built-in function.

6.3.9.6 TransformExpr

Note that a TransformExpr is a read-only expression [58] and it is allowed to be decomposed by all three decomposition algorithms. Thus, we analyse here if \(v_x\) can be a TransformExpr.

Assume \(v_x\) is a TransformExpr, according to condition \((Cnd1)\), the following statement must be true:

\[
\frac{\forall i \in 1..c : dEq(E_i, E'_i), dEq(E_r, E'_r)}{\neg dEq(\text{copy } s_1 := E_1, ..., s_c := E_c \text{ modify } E_m \text{ return } E_r, \text{copy } s_1 := E'_1, ..., s_c := E'_c \text{ modify } E_m \text{ return } E'_r})} \quad (t'_\text{Trnsf})
\]

A TransformExpr makes deep copies of its source expressions \(E_1, ..., E_c\), which is equivalent to replacing these expressions with their by-value copies. The expression \(E_m\) in the modify clause must be an UpdExpr [58], which is not allowed to be decomposed by the XQUF insertion conditions (Section 5.7.1). The result of a TransformExpr is determined by the value of

\(^8\)Our XCore grammar only allows calls to built-in functions (Section 5.3).
the return expression $E_r$. Clearly, replacing $E_r$ with a by-value copy of it will not cause the TransformExpr to return non-deep-equal result. Thus, the statement $t'_{\text{expr}}$ is not true. Hence, $v_x$ can not be a TransformExpr.

In summary, we were not able to find an $v_x$, which is not deep-equal to its corresponding vertex $v'_i$ in $G'$, while all vertices, on which $v_x$ depends, are deep-equal to their corresponding vertices in $G'$. Thus, the assumption $dEq(Q, Q')$ does not hold, which proves the correctness of the theorem. \qed

6.4 Correctness Proof of the By-Fragment Decomposition Algorithm

Definition 6.4.1. Canonical subsequence: The canonical subsequence of a sequence $\bar{s}$, denoted $\zeta(\bar{s})$, consists of a single occurrence of all node-typed items in $\bar{s}$ that are not a descendant of another node-typed item in $\bar{s}$, sorted by their document order. Formally, nodes in $\zeta(\bar{s})$ satisfy the following conditions:

- exist: $\forall s_j \in \zeta(\bar{s}) : \exists s_i \in \bar{s} \land s_i \neq s_j$
- unique: $\forall s_i \in \bar{s} : \exists s_j \in \{\zeta(\bar{s})/d-o-s::node()\} \land \text{exactly-one}(s_j) \land s_i \neq s_j$
- disjunct: $\forall i, j \in 1..|\zeta(\bar{s})| : i \neq j \Rightarrow \zeta(\bar{s})[i] \notin \{\zeta(\bar{s})[j]/d-o-s::node()\}$
- ordered: $\forall k \in 2..|\zeta(\bar{s})| : \zeta(\bar{s})[k - 1] \ll \zeta(\bar{s})[k]$

Definition 6.4.2. By-fragment copy operator $C^f$: Let $\bar{s}$ be an XQuery sequence that may contain duplicates or overlapping nodes. A by-fragment-copy, denoted $C^f(\bar{s})$, of $\bar{s}$ is a pair $(\bar{S}, \bar{F})$, where

- $\bar{F}$ is a set of fresh XML fragments, created by making a by-value copy (i.e., a deep-copy) of the canonical subsequence of $\bar{s}$, i.e., $\bar{F} = C^v(\zeta(\bar{s}))$
- $\bar{S}$ is the return sequence of the by-fragment copy operator $C^f$. It is a one-to-one mapping of the items in $\bar{s}$ constructed according to the rules below:

  $\forall i \in 1..|\bar{s}|, \begin{cases} \bar{S}[i] = \bar{s}[i], & \text{if } \text{type}(\bar{s}[i]) \notin \bar{F}; \\ \bar{S}[i] = \bar{F}[j]/d-o-s::node()[k], & \text{where } (j, k) \text{ is } \zeta(\bar{s})[j]/d-o-s::node()[k]. \end{cases}$

  That is, if $\bar{s}[i]$ is a literal value, $\bar{S}[i]$ gets the value of $\bar{s}[i]$; otherwise, $\bar{S}[i]$ is a reference to the node in $\bar{F}$ that corresponds to $\bar{s}[i]$.

Under the pass-by-fragment semantics, the semantics of executing an expression on a remote peer is equal to first replacing each parameter of the expression with a by-fragment copy, then executing the expression (using the by-fragment copies on the local peer), and finally returning a by-fragment copy of the result. We use the notations $C^f(\bar{s}), \bar{S}$ and $C^f(\bar{s}), \bar{F}$ to refer to the sets $\bar{S}$ and $\bar{F}$ that belong to $C^f(\bar{s})$. However, since only $\bar{S}$ is the return value of $C^f$, we use the shorthand $C^f(\bar{s})$ to refer to $C^f(\bar{s}), \bar{S}$, if there is no ambiguity.

Definition 6.4.3. By-fragment mapping function $m_f$: Let $\bar{e}$ be a sequence of XML nodes and $C^f(\bar{e})$ its by-fragment copy. The by-fragment mapping function $m_f$ maps each node $e_x$ in $\{\bar{e}/d-o-s::node()\}$ to exactly one node $e'_x$ in $\{C^f(\bar{e}).\bar{F}/d-o-s::node()\}$. The reverse
function \( m_f^{-1} \) maps each node \( e_x' \) in \( \{ C_f(\vec{e}), \vec{\gamma} / d-o-s::node() \} \) to exactly one node \( e_x \) in \( \{ \zeta(\vec{e}) / d-o-s::node() \} \). Formally, the results of \( m_f \) and \( m_f^{-1} \) are defined as the following:

\[
\forall e_x \in \{ \zeta(\vec{e}) / d-o-s::node() \} : e_x \text{ is } \zeta(\vec{e})[i] / d-o-s::node()[j] \implies m_f(e_x) \text{ is } C_f(\vec{e}).\vec{\gamma}[i] / d-o-s::node()[j]
\]

\[
\forall e_x' \in \{ C_f(\vec{e}) / d-o-s::node() \} : e_x' \text{ is } C_f(\vec{e}).\vec{\gamma}[i] / d-o-s::node()[j] \implies m_f^{-1}(e_x') \text{ is } \zeta(\vec{e})[i] / d-o-s::node()[j]
\]

We call \( m_f(e_x) \) by-fragment-mapping of \( e_x \) in \( C_f(\vec{e}) \), and vice versa.

**Lemma 6.4.4.** Mapped raw results of \( \text{FwdAxis} \) steps on \( \vec{e} \) and \( C_f(\vec{e}) \): Let \( \vec{e} \) be a sequence of XML nodes and \( C_f(\vec{e}) \) its by-fragment copy. The raw results of applying multiple consecutive \( \text{FwdAxis} \) steps on \( \vec{e} \) and \( C_f(\vec{e}) \) are the by-fragment-mapping of each other, i.e.:

\[
C_f(\vec{e})/a_s_1::NT_1/.../a_s_n::NT_n = m_f(\vec{e}/a_s_1::NT_1/.../a_s_n::NT_n) \land \forall a_s \in \text{FwdAxis}
\]

**Proof.** The proof is similar to the proof of Lemma 6.3.4. \( \square \)

**Lemma 6.4.5.** By-fragment node relationships: Let \( \vec{e} \) be a sequence of XML nodes and \( C_f(\vec{e}) \) its by-fragment copy. Then, the relationship between any two nodes \( u \) and \( w \) in the subtrees rooted at the nodes in \( \vec{e} \) is the same as the relationship between their corresponding nodes \( u' \) and \( w' \) in \( C_f(\vec{e}) \). Formally:

\[
\forall u, w \in \{ \vec{e} / d-o-s::node() \}, \forall u', w' \in \{ C_f(\vec{e}) / d-o-s::node() \} : u' = m_f(u) \land w' = m_f(w) \implies R(u, w) = R(u', w')
\]

**Proof.** Since \( u \) and \( w \) are both nodes in the subtrees rooted at nodes in \( \vec{e} \), with Definition 6.4.1, we can assume:

1. \( u \) is \( \zeta(\vec{e})[i] / d-o-s::node()[k] \), \( w \) is \( \zeta(\vec{e})[j] / d-o-s::node()[l] \)
2. \( i, j \in 1..|\zeta(\vec{e})| \) and \( k, l \in 1..|\zeta(\vec{e})[i] / d-o-s::node()| \).

With Definition 6.4.3 and \( u' = m_f(u), w' = m_f(w) \), we have:

1. \( u' = C_f(\vec{e}).\vec{\gamma}[i] / d-o-s::node()[k] \), \( w' = C_f(\vec{e}).\vec{\gamma}[j] / d-o-s::node()[l] \)

There are two possibilities: \( i = j \) (i.e., \( u, w \) belong to the same subtree in \( \zeta(\vec{e}) \)) or \( i \neq j \) (i.e., \( u, w \) belong to different subtrees in \( \zeta(\vec{e}) \)). Below we consider each case.

**Case 1: \( i = j \)**

With Definition 6.4.2 we have:

1. \( dE(C_f(\vec{e}).\vec{\gamma}[i], \zeta(\vec{e})[i]) \)

With (h1-3) we have:

1. \( R(u, w) = R(u', w') \)

Thus, the lemma holds when \( u \) and \( w \) belong to the same subtree in \( \zeta(\vec{e}) \).

**Case 2: \( i \neq j \)**

With Definition 6.4.2 we have:

1. \( dE(C_f(\vec{e}).\vec{\gamma}[i], \zeta(\vec{e})[i]) \)

With Definition 6.4.1 we have:

1. \( \mu(\zeta(\vec{e})), \sigma(\zeta(\vec{e})) \)

With (h4,5) we have:

1. \( i < j \implies \zeta(\vec{e})[i] \ll \zeta(\vec{e})[j] \land C_f(\vec{e}).\vec{\gamma}[i] \ll C_f(\vec{e}).\vec{\gamma}[j] \)
2. \( i > j \implies \zeta(\vec{e})[i] \gg \zeta(\vec{e})[j] \land C_f(\vec{e}).\vec{\gamma}[i] \gg C_f(\vec{e}).\vec{\gamma}[j] \)

With (h1,2) and (h6,7), we have:

1. \( R(u, w) = R(u', w') \)

Thus, the lemma also holds when \( u \) and \( w \) belong to different subtrees in \( \zeta(\vec{e}) \). \( \square \)
Lemma 6.4.6. By-fragment deep-equal \( \text{ddo}() \): Let \( \overline{e} \) be a sequence of XML nodes, and \( C'(\overline{e}) \) its by-fragment copy. Let \( \overline{a} \) be a sequence containing XML nodes in the subtree rooted at \( \overline{e} \), i.e., \( \forall a \in \{ \overline{e}/d-o-s::\text{node}() \} \), and \( \overline{b} \) the by-fragment mapping of \( \overline{a} \) in \( C'(\overline{e}) \), then \( \text{dEq}(\text{ddo}(\overline{a}), \text{ddo}(\overline{b})) \).

Proof. With Lemma 6.4.5, this lemma can be proven using a similar reasoning as that of the proof of Lemma 6.3.6.

Lemma 6.4.7. By-fragment deep-equal \( \text{FwdAxis} \): Let \( \overline{e} \) be a single sequence of XML nodes and \( C'(\overline{e}) \) its by-fragment copy. The results of applying any number of consecutive \( \text{FwdAxis} \) steps on \( \overline{e} \) and \( C'(\overline{e}) \) are deep equal to each other. Formally:

\[
\text{dEq}(\overline{e}/\text{AS}_1::\text{NT}_1/.../\text{AS}_n::\text{NT}_n, C'(\overline{e})/\text{AS}_1::\text{NT}_1/.../\text{AS}_n::\text{NT}_n) \land \forall i \in 1..n : \text{AS}_i \in \text{FwdAxis}
\]

Proof. With Lemma 6.4.4, we have:

(1) \( C'(\overline{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \equiv m_f(\overline{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)

With (h1) and Lemma 6.4.6, we have:

(2) \( \text{dEq}(\text{ddo}(\overline{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n), \text{ddo}(C'(\overline{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n))

With Definition 6.1.15, we have:

(3) \( \overline{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \equiv \text{ddo}(\overline{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)

(4) \( C'(\overline{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n) \equiv \text{ddo}(C'(\overline{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)

With (h2), (h3) and (h4), we have:

(1) \( \text{dEq}(\text{ddo}(\overline{e}/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n), \text{ddo}(C'(\overline{e})/\text{as}_1::\text{NT}_1/.../\text{as}_n::\text{NT}_n)) \)

The correctness of the by-fragment decomposition algorithm is proven as follows:

Theorem 6.4.8. By-Fragment Decomposition Correctness: Let \( Q \) be a normal read-only XCore query (i.e., without any XRPC expressions) and \( G \) the corresponding d-graph. \( I'(G) \subseteq G \) is the (non-empty) set of decomposition points validated by the by-fragment insertion conditions. Let \( G' \) be the d-graph derived by doing an XRPC expr insertion above each vertex in \( I'(G) \) (Section 5.3.2), and \( Q' \) be the corresponding query of \( G' \). Then, \( \text{dEq}(Q, Q') \) holds, under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).

Proof. We prove this theorem using a similar strategy as the proof of the correctness of the conservative decomposition (Theorem 6.3.9). Assume \( \neg \text{dEq}(Q, Q') \), then we need to find a vertex \( v_x \in G \) which depends on by-fragment copies of remote sequences, with its corresponding vertex \( v'_x \in G' \), such that \( v_x \) and \( v'_x \) are not deep-equal, even if each vertex \( v_x \), on which \( v_x \) depends, is deep-equal to its corresponding vertex \( v'_x \) in \( G' \). Formally, \( \exists v_x \in G \land \exists v'_x \in G' \), such that:

\[
\begin{align*}
(Cnd1) & \neg \text{dEq}(v_x, v'_x) \land \forall v \in \{ v_x \sim v_x \} \land \forall v' \in \{ v'_x \sim v'_x \} : \text{dEq}(v_x, v'_x) \\
(Cnd2) & (v_x \in I'(G) \land \exists v_x \in V(G) \setminus \text{V}(G_{v_x}) \land v_x \sim v_x) \lor (\exists v_x \in I'(G) \land v_x \sim v_x)
\end{align*}
\]

Compared to the conservative algorithm, the by-fragment decomposition algorithm allows, in addition, the following kind of expressions to be decomposed:

- NodeCmp and NodeSetExpr expressions may depend on copied subexpressions, iff the NodeCmp and NodeSetExpr expressions do not depend on two different applications of fn:doc() with the same URI (condition II). This change is considered in 6.4.8.1 and 6.4.8.2.
• FwdAxis steps\(^9\) may depend on copied subexpressions containing OrderExprs or axis steps including ancestor, ancestor-or-self, preceding, following, descendant and descendant-or-self. FwdAxis steps may also depend on copied subexpressions containing ForExprs, ExprSeqs or NodeSetExprs, iff such subexpressions do not depend on two different applications of \(\text{fn:doc()}\) with the same URI (condition III). This change is considered in 6.4.8.3.

In Theorem 6.3.9, it has already been proven that \(v_x\) can not be one of the expressions which are allowed by the conservative algorithm to depend on (by-value) copied subexpressions. In this proof, we only consider if \(v_x\) can be one of the expressions which are additionally allowed by the by-fragment decomposition algorithm to depend on (by-fragment) copied subexpressions. Note that the analysis below is two-sided, i.e., a subexpression could be copied either because it is used as a parameter of a decomposed expression, or because the subexpression itself is a decomposed expression whose result is used further in the query.

6.4.8.1 NodeCmp

Assume \(v_x\) is a NodeCmp, according to (Cnd1) above, the following statement must be true:

\[
\frac{d\text{Eq}(E_0, E_0') \quad d\text{Eq}(E_1, E_1') \quad \neg \text{hasMatchingDoc}(E_0, E_1)}{\neg d\text{Eq}(E_0 \bowtie E_1, \ E_0' \bowtie E_1')} (t_{\text{Node}})
\]

where the symbol \(\bowtie\) represents the node comparison operators: is, \(<\) and \(\gg\).

The premise \(\neg \text{hasMatchingDoc}(E_0, E_1)\) implies that \(E_0\) and \(E_1\) contain nodes from different documents. Thus, the query “\(E_0\) is \(E_1\)” will always return false. The decomposed query “\(E_0' \bowtie E_1'\)” also always returns false, since \(E_0'\) and \(E_1'\) refer to nodes in different fragments. The result of “\(E_0 \bowtie E_1\)” is implementation dependent. It is either true or false. The query “\(E_0' \bowtie E_1'\)” also returns either true or false, as it depends on two distinct trees \(C'(\xi(E_0))\) and \(C'(\xi(E_1))\). Recall that the definition of deep-equal queries (Definition 6.1.6) takes into account the XQuery features with implementation freedom, and regards “\(E_0' \bowtie E_1'\)” to be deep-equal to “\(E_0 \bowtie E_1\)” if “\(E_0' \bowtie E_1'\)” returns a value that also could be returned by “\(E_0 \bowtie E_1\)”. Thus, we have \(d\text{Eq}(E_0 \bowtie E_1, E_0' \bowtie E_1')\). Similar reasoning holds for “\(E_0 \gg E_1\)”.

Hence, \(v_x\) can not be a NodeCmp whose subexpressions depend on calls to \(\text{fn:doc()}\) with different URIs.

6.4.8.2 NodeSetExpr

Assume \(v_x\) is a NodeSetExpr, according to (Cnd1) above, the following statement must hold:

\[
\frac{d\text{Eq}(E_0, E_0') \quad d\text{Eq}(E_1, E_1') \quad \neg \text{hasMatchingDoc}(E_0, E_1)}{\neg d\text{Eq}(E_0 \bowtie E_1, \ E_0' \bowtie E_1')} (t_{\text{Node}})
\]

where the symbol \(\bowtie\) represents the node set operators: union, intersect or except.

The premise \(\neg \text{hasMatchingDoc}(E_0, E_1)\) implies that \(E_0\) and \(E_1\) contain nodes from different documents. The expression “\(E_0\ union E_1\)” returns either \((\text{ddo}(E_0), \text{ddo}(E_1))\) or \((\text{ddo}(E_1), \text{ddo}(E_0))\). Similarly, the decomposed expression “\(E_0'\ union E_1'\)” returns either \((\text{ddo}(E_0'), \text{ddo}(E_1'))\) or \((\text{ddo}(E_1'), \text{ddo}(E_0'))\). By Lemma 6.4.6, we have \(d\text{Eq}(\text{ddo}(E_0), \text{ddo}(E_0'))\) and \(d\text{Eq}(\text{ddo}(E_1), \text{ddo}(E_1'))\), which implies:

\(^9\)XPath steps on other axes are excluded by the by-fragment insertion condition I.
CHAPTER 6. CORRECTNESS PROOF OF XQUERY DECOMPOSITION

\[ dEq((\text{ddo}(E_0), \text{ddo}(E_1)), (\text{ddo}(E'_0), \text{ddo}(E'_1))) \]

\[ dEq((\text{ddo}(E_1), \text{ddo}(E_0)), (\text{ddo}(E'_1), \text{ddo}(E'_0))) \]

With Definition 6.1.6, we have \( dEq(E_0 \cup E_1, E'_0 \cup E'_1) \). Thus, the statement \( \neg dEq(E_0 \sqcup E_1, E'_0 \sqcup E'_1) \) does not hold in this case.

If \( \sqcap \) represents intersect or except, with \( \neg \text{hasMatchingDoc}(E_0, E_1) \), the expression “\( E_0 \) intersect \( E_1 \)” returns the empty sequence, and the expression “\( E_0 \) except \( E_1 \)” returns \( \text{ddo}(E_0) \). Similarly, the decomposed expression “\( E'_0 \) intersect \( E'_1 \)” returns the empty sequence, and “\( E'_0 \) except \( E'_1 \)” returns \( \text{ddo}(E'_0) \). By Lemma 6.4.6, we have that \( dEq(\text{ddo}(E_0), \text{ddo}(E'_0)) \). Thus, the statement \( \neg dEq(E_0 \sqcap E_1, E'_0 \sqcap E'_1) \) does not hold in these cases.

Hence, \( v_x \) can not be a \( \text{NodeSetExpr} \), whose subexpressions depend on calls to \( \text{fn:doc()} \) with different URIs.

6.4.8.3 FwdAxis steps

Assume \( v_x \) is a FwdAxis, according to \( \text{(Cnd1)} \) above, at least one of the following statements must be true:

\[ \frac{dEq(E_1, E'_1)}{E_1 \in \{\text{AxisStep}, \text{OrderExpr}\} \quad \text{AS} \in \text{FwdAxis}} \quad (t'_{\text{Step1}}) \]

\[ \frac{\exists E_i : E_1 \sim E_i \wedge E_i \in \{\text{ForExpr}, \text{ExprSeq}, \text{NodeSetExpr}\}}{-dEq(E_1/\text{AS}::\text{NT}, E'_1/\text{AS}::\text{NT})} \]

\[ \frac{dEq(E_1, E'_1)}{E_1 \in \{\text{ForExpr}, \text{ExprSeq}, \text{NodeSetExpr}\} \quad \text{AS} \in \text{FwdAxis}} \quad (t'_{\text{Step2}}) \]

\[ \frac{-\text{hasMatchingDoc}(E_1, E_1)}{-dEq(E_1/\text{AS}::\text{NT}, E'_1/\text{AS}::\text{NT})} \]

In Lemma 6.4.7, it is proven that the results of applying a FwdAxis on a single sequence \( \vec{v} \) (i.e., \( E_1 \) does not depend on any subexpressions \( E_i \) that combine two sequences in their results; this case is covered by the statement \( t'_{\text{Step2}} \)) and its by-fragment copy \( C^f(\vec{v}) \) are deep equal to each other. Note that Lemma 6.4.7 holds for any XML node sequence, regardless of their distinct, disjunct and ordered properties. Thus, the statement \( t'_{\text{Step1}} \) is not true.

If the result of \( E_1 \) is a combination of two single subsequences \( (e_0, e_1) \), then the result of \( E'_1 \) is a combination of the by-fragment copies of the two subsequences \( (e'_0, e'_1) \). The statement \( t'_{\text{Step2}} \) would be true, if we are not able to eliminate duplicate nodes that appear in both \( e'_0 \) and \( e'_1 \), or sort \( e'_0 \) and \( e'_1 \) in the same document order as \( e_0 \) and \( e_1 \). As described earlier (Section 5.2, Problem 4), this could only happen, if \( e_0 \) and \( e_1 \) contain nodes from the same document on a peer. There are three kinds of expressions that return combined subsequences: ForExpr, ExprSeq and NodeSetExpr. For these expressions, the problem with “mixed-call” is guarded by the predicate hasMatchingDoc() in the by-fragment insertion condition III, which simply forbids a FwdAxis to depends on a \( E_1 \) that contain multiple \( \text{fn:doc()} \) calls to access the same document. Thus, the statement \( t'_{\text{Step2}} \) does not hold.

Hence, \( v_x \) can not be a FwdAxis that depends on an OrderExpr or an AxisStep, or on a ForExpr, an ExprSeq or a NodeSetExpr which do not access the same XML document with multiple \( \text{fn:doc()} \).
In summary, we were not able to find a vertex \( v_x \) which is not deep-equal to its corresponding vertex \( v'_x \) in \( G' \), while all vertices on which \( v_x \) depends are deep-equal to their corresponding vertices in \( G' \). Thus, the assumption \( \neg dEq(Q, Q') \) does not hold, which proves the correctness of the theorem. \( \square \)

6.5 Correctness Proof of the By-Projection Decomposition Algorithm

When the by-projection decomposition algorithm is used, the set of projection paths \( \tilde{P}_v \) are calculated for each vertex \( v \) in a d-graph which consists of a set of used paths \( \tilde{P}_v, \tilde{U} \) and a set of returned paths \( \tilde{P}_v, \tilde{R} \) (Section 5.6). Each path in \( \tilde{P}_v \) is a ProjectionPath (Table 5.6) that can contain XPath steps on any axes and any of the special built-in functions \( \text{root}() \), \( \text{id}() \) and \( \text{idref}() \).

The concept relative projection paths (i.e., path suffixes) is always defined between two vertices. Assume \( v_i \) and \( v_j \) are vertices in a d-graph with \( v_i \overset{\sim}{\rightarrow} v_j \). We use \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}} \) to denote the set of relative projection paths between \( v_i \) and \( v_j \) which consists of a set of relative used paths \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{U} \) and a set of relative returned paths \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{R} \), computed using the function \( \text{allSuffixes}() \) (Section 5.6.2):

\[
\tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{U} = \text{allSuffixes}(\tilde{P}_v, \tilde{R}, \tilde{P}_v, \tilde{U})
\]
\[
\tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{R} = \text{allSuffixes}(\tilde{P}_v, \tilde{R}, \tilde{P}_v, \tilde{R})
\]

However, beside the path \( v_i \overset{\sim}{\rightarrow} v_j \), \( v_i \) could also depend on \( v_j \) via other paths, say \( v_i \overset{\sim}{\rightarrow} v_k \sim v_j \). This happens if \( v_j \) is a variable declaration (or a subexpression of a variable declaration) and is referred to by \( v_k \). For instance, as shown in the left part of Figure 6.4, \( v_1 \) depends on \( v_4 \) both via the path \( v_1 \overset{\sim}{\rightarrow} v_2 \overset{\sim}{\rightarrow} v_4 \) and via the path \( v_1 \overset{\sim}{\rightarrow} v_6 \overset{\sim}{\rightarrow} v_4 \). Thus, we use \( \tilde{P}_{v_k \rightarrow v_j}^{\text{rel}} \) to denote the set of relative projection path that \( v_i \) will apply on \( v_j \) via its subexpression \( v_k \). \( \tilde{P}_{v_k \rightarrow v_j}^{\text{rel}} \) is computed using the function \( \text{allSuffixesVia}() \) (Section 5.6.2):

\[
\tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{U} = \text{allSuffixesVia}(\tilde{P}_v, \tilde{R}, \tilde{P}_v, \tilde{U})
\]
\[
\tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{R} = \text{allSuffixesVia}(\tilde{P}_v, \tilde{R}, \tilde{P}_v, \tilde{R})
\]

The paths \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{U} \) and \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{R} \) overestimate the set of nodes, with respect to \( v_j \), that will be used and respectively returned by \( v_i \). Intuitively, if \( v_j \) represents an expression executed on a peer different than the peer, on which \( v_i \) is executed, shipping the nodes determined by \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{U} \) and \( \tilde{P}_{v_i \rightarrow v_j}^{\text{rel}}, \tilde{R} \) together with the result of \( v_j \) could enable correct evaluation of \( v_i \).

In other words, evaluating \( v_i \) on those shipped nodes produces deep-equal results to those of evaluating \( v_j \) in the original query. The main task of this section is to give a formal proof for this statement. But first, let us use an example to make the differences and relationships between projection paths and relative projection paths more clear.

**Example 6.5.1.** Figure 6.3 shows an abstract d-graph in which three vertices are shown explicitly. The root vertex of this d-graph is \( v_i \). The vertex \( v_k \) depends on \( v_j \) via a varref varref edge (indicated using a dotted arrow). Thus, starting from \( v_i \), there are two paths with which we can reach \( v_j \), i.e., \( v_i \overset{\sim}{\rightarrow} v_j \) and \( v_i \overset{\sim}{\rightarrow} v_k \overset{\sim}{\rightarrow} v_j \). The exact projection paths of all three vertices are also shown in Figure 6.3.

\[\text{[Note: This includes both cases: (i) } v_j \text{ is a parameter of the valid decomposition point } v_i, \text{ or (ii) } v_j \text{ is a valid decomposition point, whose result is used by } v_i.\]
CHAPTER 6. CORRECTNESS PROOF OF XQUERY DECOMPOSITION

Figure 6.3: A d-graph with projection paths

It can be seen that \( v_k \) applies two XPath steps on (the result of) \( v_j \): a \( \text{child::c} \) step as a used path and a \( \text{child::d} \) step as a returned path. Then, \( v_i \) applies two XPath steps on \( v_k \): a \( \text{child::e} \) step as a used path and a \( \text{child::f} \) step as a returned path. This implies that \( v_i \), via \( v_k \), the following XPath steps on \( v_j \): a used path \( \text{child::d/child::e} \) and a returned path \( \text{child::d/child::f} \). Since we also have \( v_i \sim v_j \), \( v_i \) can also directly (i.e., not via \( v_k \)) apply XPath steps on \( v_j \), which is, in this example, the \( \text{child::g} \) step. Thus, we can compute the following relative projection paths among these three vertices:

\[
\begin{align*}
\vec{P}_{v_k \rightarrow v_j} \cdot \vec{U} &= \text{allSuffixes}(\vec{P}_{v_j}, \vec{R}_{v_j}, \vec{P}_{v_k} \cdot \vec{U}) \quad = \{\text{child::c}\} \\
\vec{P}_{v_k \rightarrow v_j} \cdot \vec{R} &= \text{allSuffixes}(\vec{P}_{v_j}, \vec{R}_{v_j}, \vec{P}_{v_k} \cdot \vec{R}) \quad = \{\text{child::d}\} \\
\vec{P}_{v_i \rightarrow v_k} \cdot \vec{U} &= \text{allSuffixes}(\vec{P}_{v_k}, \vec{R}_{v_k}, \vec{P}_{v_i} \cdot \vec{U}) \quad = \{\text{child::e}\} \\
\vec{P}_{v_i \rightarrow v_k} \cdot \vec{R} &= \text{allSuffixes}(\vec{P}_{v_k}, \vec{R}_{v_k}, \vec{P}_{v_i} \cdot \vec{R}) \quad = \{\text{child::f}\} \\
\vec{P}_{v_i \rightarrow v_j} \cdot \vec{U} &= \text{allSuffixes}(\vec{P}_{v_j}, \vec{R}_{v_j}, \vec{P}_{v_i} \cdot \vec{U}) \quad = \{\text{child::c,child::d,child::e}\} \\
\vec{P}_{v_i \rightarrow v_j} \cdot \vec{R} &= \text{allSuffixes}(\vec{P}_{v_j}, \vec{R}_{v_j}, \vec{P}_{v_i} \cdot \vec{R}) \quad = \{\text{child::d/child::e}\} \\
\vec{P}_{v_i \rightarrow v_k} \cdot \vec{U} &= \text{allSuffixes}(\vec{P}_{v_k}, \vec{R}_{v_k}, \vec{P}_{v_i} \cdot \vec{U}) \quad = \{\text{child::c,child::d,child::e}\} \\
\vec{P}_{v_i \rightarrow v_k} \cdot \vec{R} &= \text{allSuffixes}(\vec{P}_{v_k}, \vec{R}_{v_k}, \vec{P}_{v_i} \cdot \vec{R}) \quad = \{\text{child::d/child::f}\} \\
\end{align*}
\]

Note the difference between \( \vec{P}_{v_k \rightarrow v_j} \cdot \vec{R} \) and \( \vec{P}_{v_k \rightarrow v_j} \cdot \vec{R} \), which indicates that the step \( \text{child::g} \) is not applied on the result of \( v_k \). Thus, if \( v_k \) would be decomposed (\( v_j \) is then a parameter of this remote expression), we do not need to project the \( g \) child nodes of \( v_j \) for the request message.

From the above example, the following property for the projection paths can be deduced\(^\text{11}\):

**Property 6.5.2. Increasing projection paths:** Starting from leaf vertices\(^\text{12}\), an expression always propagates all projection paths of all its subexpression(s). Thus, an existing projection path is never removed, only a new projection path is added, when there is an XPath step.

The definition of the by-projection-copy operator \( C^p \) is based on the concept of relative projection paths.

\(^\text{11}\)The property for the projection paths can also be deduced by examining the static properties analysis rules in Section 6.2

\(^\text{12}\)By ignoring the varref edges, a d-graph has a tree shape. The leaf vertices are then those vertices without any outgoing edges.
Definition 6.5.3. **By-projection-copy operator** $C^p$: The by-projection-copy operator takes as its input a pair $(\vec{s}, \vec{p}_{rel})$, where:

- $\vec{s}$ is an XQuery sequence that may contain duplicates or overlapping nodes,
- $\vec{p}_{rel}$ is the set of relative projection paths that will be applied on $\vec{s}$;

and produces as its output a pair $(\vec{S}, \vec{F})$, where:

- $(\vec{S}, \vec{F}) = C^f(\vec{s})$, if $\vec{p}_{rel} = \emptyset$; otherwise
- $\vec{F}$ is the set of projected XML fragments which is the projection of $\vec{p}_{rel}$ on $\vec{s}$, i.e., $\vec{F} = \mathcal{P}(\vec{s}, \vec{p}_{rel})$; $\vec{S}$ is the return sequence of the by-projection-copy operator $C^p$. It is a one-to-one mapping of the items in $\vec{s}$ constructed as follows:

$$
\forall i \in 1..|\vec{s}|: \vec{S}[i] \text{ is } \vec{F}[j]/d-o-s::\text{node}()[k], \text{ where } \{j,k|\vec{s}[i]\} \text{ is } \vec{D}[j]/d-o-s::\text{node}()[k].
$$

That is, $\vec{S}[i]$ is a reference to the $k$-th node in the projected document $\vec{D}[j]$ which corresponds to the node referred by $\vec{s}[i]$ in the original document.

We use $C^p((\vec{s}, \vec{p}_{rel}))$ to denote a by-projection copy of $\vec{s}$ with relative projection paths $\vec{p}_{rel}$. The outputs of $C^p$ are referred to as $C^p((\vec{s}, \vec{p}_{rel})).\vec{S}$ and $C^p((\vec{s}, \vec{p}_{rel})).\vec{F}$. If there is no ambiguity, $C^p((\vec{s}, \vec{p}_{rel})).\vec{S}$ is abbreviated as $C^p(\vec{s})$. Since all node typed items in the return sequence $C^p((\vec{s}, \vec{p}_{rel})).\vec{S}$ refer to the projected XML fragments $\vec{F}$ of $\vec{p}_{rel}$ on $\vec{s}$, it is trivial to see that the results of applying $\vec{p}_{rel}$ on both $\vec{s}$ and $C^p((\vec{s}, \vec{p}_{rel})).\vec{S}$ are deep-equal. Hence, we define the following property for the by-projection operator $C^p$:

**Property 6.5.4.** $C^p$ properties: $\text{dEq} \vec{p}_{rel}(\vec{s}, C^p((\vec{s}, \vec{p}_{rel})).\vec{S})$

Under the pass-by-projection semantics, the semantics of executing an expression $v_i$ on a remote peer is to first replace each of its parameter $v_{param_k}$ with a by-projection copy, with $\vec{p}_{rel}^{v_{root} \leadsto v_i \leadsto v_{param_k}}$ (called: the relative projection paths of the parameter $v_{param_k}$), where $v_{root}$ is the root vertex of the d-graph containing $v_i$. Then the expression $v_i$ is executed on the local peer using the by-projection copies of its parameters, and finally a by-projection copy of the result is returned, with $\vec{p}_{rel}^{v_{root} \leadsto v_i}$ (called: the relative projection paths of the expression $v_i$). Thus, when computing the projection for a remote expression, we always compare its returned paths with the projection paths of the root vertex $v_{root}$ (either directly, or via a third vertex).

**Example 6.5.5.** The left part of Figure 6.4 shows a d-graph $G$, in which each vertex is annotated with a set of projection paths. The right part of Figure 6.4 shows what the decomposed d-graph $G'$ looks like (the paths annotations are omitted), expressed using by-projection copies, if, for instance, $v_4$ and $v_6$ are pushed to remote peers. The vertex $v_4$ has no parameter, so only its result is projected using the relative projection paths $\vec{p}_{rel}^{v_1 \leadsto v_4}$ of $v_4$, where:

$$
\vec{p}_{rel}^{v_1 \leadsto v_4}, \vec{U} = \text{allSuffixes}(\vec{p}_{rel}^{v_4, \vec{R}}, \vec{p}_{rel}^{v_1, \vec{U}})
\vec{p}_{rel}^{v_1 \leadsto v_4}, \vec{R} = \text{allSuffixes}(\vec{p}_{rel}^{v_4, \vec{R}}, \vec{p}_{rel}^{v_1, \vec{R}})
$$

For vertex $v_6$, its parameter $v_3$ is first projected using the relative paths $\vec{p}_{rel}^{v_1 \leadsto v_6 \leadsto v_3}$, where:

$$
\vec{p}_{rel}^{v_1 \leadsto v_6 \leadsto v_3}, \vec{U} = \text{allSuffixesVia}(\vec{p}_{rel}^{v_1, \vec{R}}, \vec{p}_{rel}^{v_6, \vec{U}}, \vec{p}_{rel}^{v_1, \vec{U}})
\vec{p}_{rel}^{v_1 \leadsto v_6 \leadsto v_3}, \vec{R} = \text{allSuffixesVia}(\vec{p}_{rel}^{v_1, \vec{R}}, \vec{p}_{rel}^{v_6, \vec{R}}, \vec{p}_{rel}^{v_1, \vec{R}})
$$

Then, the result of $v_6$ is projected using the relative paths $\vec{p}_{rel}^{v_1 \leadsto v_6}$.
The correctness of the by-projection decomposition algorithm is proven as follows:

**Theorem 6.5.6. By-Projection Decomposition Correctness:** Let \( Q \) be a normal read-only XCore query (i.e., without any XRPC expressions) and \( G \) the corresponding d-graph. \( I^p(G) \subseteq G \) is the (non-empty) set of decomposition points validated by the by-projection insertion conditions. Let \( G' \) be the d-graph derived by doing an XRPCExpr insertion above each vertex in \( I^p(G) \) (Section 5.3.2), and let \( Q' \) be the corresponding query of \( G' \). Then, \( dEq(Q, Q') \) holds under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).

**Proof.** We prove this theorem by contradiction, similar to the proof of the correctness theorem of the conservative decomposition (Theorem 6.3.9). Since the relative projection paths of parameters of remote expressions are computed slightly differently than the relative projection paths of remote expressions, we split this proof into two parts. In Part I, we temporarily ignore this difference and assume that the relative projection paths of each vertex \( v \) in a d-graph are computed in the same way, i.e., \( \vec{p}^{rel}_{v^\rightarrow v} \). Under this assumption, we search for a vertex \( v \), which could invalidate the statement \( dEq(Q, Q') \). If we are not able to find such a \( v \) in Part I, we check in Part II if projecting parameters of a remote expression, using relative projection paths computed via the remote expression, could invalidate the statement \( dEq(Q, Q') \).

**Part I**

Assume \( \neg dEq(Q, Q') \). There must exist one vertex \( v_x \in G \) which depends on by-projection copies of remote sequences\(^{14}\) with its corresponding vertex \( v'_x \in G' \), such that \( v_x \) and \( v'_x \) are not by-projection equal with respect to \( \vec{p}^{rel}_{v^\rightarrow v} \), even if each vertex \( v \) on which \( v_x \) depends is by-projection equal with respect to \( \vec{p}^{rel}_{v^\rightarrow v} \) to its corresponding vertex \( v'_x \) in \( G' \). Formally, \( \exists v_x \in G \) and \( \exists v'_x \in G' \), such that:

\[
\begin{align*}
(Cnd1) \quad & \neg dEq^{rel}_{v^\rightarrow v} (v_x, v'_x) \land \forall v \in \{v \mid v \sim v'_x\}, \forall v' \in \{v' \mid v' \sim v'_x\} : dEq^{rel}_{v^\rightarrow v} (v, v') \\
(Cnd2) \quad & v_x \in I^p(G) \land \exists v^\rightarrow v \in V(G) \land v \sim v^\rightarrow v \land (\exists v \in I^p(G) \land v \sim v) \\
\end{align*}
\]

In the remainder of Part I, we examine each kind of expression\(^{15}\) in the XCore grammar.

\(^{13}\)Since XPath steps can only be applied on XML nodes, we assume that \( \vec{v} \) in this case only contain XML nodes.

\(^{14}\)Either \( v_x \) is in \( I^p(G) \) and uses those sequences as its parameters; or those sequences are results of remote execution and are used by \( v_x \).

\(^{15}\)But with focus on expressions whose result sequence can contain XML node-typed items, since it is trivial to see that copying a literal value (under any of our three semantics) does not alter query result.
(Table 5.2) to see if it could be a \( v_x \) that satisfies \((Cnd1)\) and \((Cnd2)\) above. For brevity, we call two corresponding vertices from \( G \) and \( G' \) to be by-projection equal without explicitly mentioning the relative projection paths with respect to which they are deep-equal, since all projection paths are computed in the same way in Part I. Similar with what we have explained in the proof of the correctness theorem of the conservative decomposition (Theorem 6.3.9), when checking if a vertex in \( G \) could be a \( v_x \) as defined above, it is safe to assume that all vertices on which this vertex depends are by-projection deep-equal to their corresponding vertices in \( G' \) (i.e., no \( v_x \) has been found among these vertices). Because, otherwise, we have already proven the assumption \(-dEq(Q, Q')\).

6.5.6.1 Easy cases

Empty sequences, Literal values and variables are easy cases. Literals are not projected, but rather copied literally. It is trivial to see that Literals can always be replaced by a copy of them without altering the query result. A variable merely represents the value of the expression, to which the variable is bound. Thus, it is also trivial to see that if this expression is by-projection deep-equal to its corresponding expression in \( Q' \), the variable is also by-projection equal to its corresponding variable in \( Q' \).

6.5.6.2 LetExpr, IfExpr, Typeswitch, OrderExpr, Constructor and TransformExpr

If \( v_x \) is a LetExpr, according to condition \((Cnd1)\), the following statement must be true:

\[
\begin{align*}
-dEq_{\text{root} \rightarrow E_0}(E_0, E_0') & \quad dEq_{\text{root} \rightarrow E_1}(E_1, E_1') \\
\end{align*}
\]

\[\overset{\text{(i)}\ast}{=}\]

A LetExpr expression merely returns its return clause \( E_1 \). Since it does not apply any XPath steps on \( E_1 \), it is clear that \( \hat{P}_{\text{root} \rightarrow E_0}^{\text{rel}} = \hat{P}_{\text{root} \rightarrow E_1}^{\text{rel}} \). With the premise \( dEq_{\text{root} \rightarrow E_1}(E_1, E_1') \), we thus have \( dEq_{\text{root} \rightarrow E_0}(E_0, E_0') \).

Similar reasoning applies to the expressions IfExpr, Typeswitch, OrderExpr, Constructor and TransformExpr since they do not apply any XPath steps on their subexpressions or test node identities or structural properties of the nodes returned by their subexpressions. Hence, \( v_x \) can not be any one of these kinds of expressions.

6.5.6.3 ExprSeq and ForExpr

If \( v_x \) is a non-empty ExprSeq, according to condition \((Cnd1)\), the following statement must be true:

\[
\begin{align*}
-dEq_{\text{root} \rightarrow E_0}(E_0, E_0') & \quad dEq_{\text{root} \rightarrow E_1}(E_1, E_1') \\
\end{align*}
\]

\[\overset{\text{(ii)}\ast}{=}\]

A ExprSeq concatenates its two subexpressions into one expression. There are two cases.

If \( E_0 \) and \( E_1 \) contain nodes from the same XML documents, according to the by-projection insertion condition \((b)\), there is no AxisStep, NodeCmp or NodeSetExpr that depends on \( E_{\text{seq}} \). Hence, all three sets of relative projection paths \( \hat{P}_{\text{root} \rightarrow E_{\text{seq}}}^{\text{rel}}, \hat{P}_{\text{root} \rightarrow E_0}^{\text{rel}} \) and \( \hat{P}_{\text{root} \rightarrow E_1}^{\text{rel}} \) are

\[\overset{\text{(iii)}\ast}{=}\]

16If any of these kinds of expressions depend on \( E_{\text{seq}} \) in this case, we are not able to eliminate duplicates required by these expressions. Thus, the by-projection insertion condition \((b)\) forbids decomposing \((i)\) \( E_{\text{seq}} \), \((ii)\) all vertices on which \( E_{\text{seq}} \) depends, and \((iii)\) all vertices depending on \( E_{\text{seq}} \) which could be reached from this AxisStep.
empty. With the third and fourth premises and Definition 6.1.4, we have: $dEq(E_0, E'_0)$ and $dEq(E_1, E'_1)$. With the first and second premises we have: $dEq(E_{seq}, E'_{seq})$. Since $\bar{p}^rel_{v_{root} \rightarrow E_{seq}} = \emptyset$, with Definition 6.1.4 we have: $dEq(E_{seq}, E'_{seq}) \Rightarrow dEq(\bar{p}^rel_{v_{root} \rightarrow E_{seq}} (E_{seq}, E'_{seq}))$.

If $E_0$ and $E_1$ only contain nodes from different XML documents, nodes in $E_0$ are disjoint with nodes in $E_1$. XPath steps on $E_{seq}$ are allowed\(^{17}\), thus the relative projection paths might not be empty (if they are empty, the reasoning is the same as that of the first case). Based on the static properties analysis rule SEQ (Section 6.2.3), it can be deduced that: $\bar{p}^rel_{v_{root} \rightarrow E_{seq}} = \bar{p}^rel_{v_{root} \rightarrow E_0} \cup \bar{p}^rel_{v_{root} \rightarrow E_1}$. The result of applying $\bar{p}^rel_{v_{root} \rightarrow E_{seq}}$ on $E_{seq}$ is equivalent to concatenating the results of applying $\bar{p}^rel_{v_{root} \rightarrow E_0}$ on $E_0$ and applying $\bar{p}^rel_{v_{root} \rightarrow E_1}$ on $E_1$. The same holds for $E'_{seq}$, $E'_0$ and $E'_1$. With the third and fourth premises, we have $dEq(\bar{p}^rel_{v_{root} \rightarrow E_{seq}} (E_{seq}, E'_{seq}))$.

If $v_x$ is a ForExpr, according to condition (Cnd1), the following statement must be true:

\[
\begin{align*}
\neg dEq(\bar{p}^rel_{v_{root} \rightarrow E_{for}} (E_{for}, E'_{for})) & \Rightarrow \neg dEq(\bar{p}^rel_{v_{root} \rightarrow E_{for}} (E_{for}, E'_{for}), \text{where} \ E_{for} = \text{for } x \in E_0 \text{ return } E_1, \text{ and } E'_{for} = \text{for } x \in E'_0 \text{ return } E'_1)
\end{align*}
\]

The ForExpr expressions have similar behaviour to the ExprSeq expressions, i.e., they concatenate multiple subsequences into one sequence as their results. Using similar reasoning as above, we can deduce that the statement ($t'_{for}$) is not true.

Hence, $v_x$ can not be an ExprSeq or a ForExpr.

6.5.6.4 CompExpr

Assume $v_x$ is a CompExpr, according to condition (Cnd1), the following statement must be true:

\[
\begin{align*}
\neg dEq(\bar{p}^rel_{v_{root} \rightarrow E_{cmp}} (E_{cmp}, E'_{cmp})) & \Rightarrow \neg dEq(\bar{p}^rel_{v_{root} \rightarrow E_{cmp}} (E_{cmp}, E'_{cmp}), \text{where} \ E_{cmp} = E_0 \circ E_1, \text{ and } E'_{cmp} = E'_0 \circ E'_1)
\end{align*}
\]

The symbol $\circ$ represents a value or a node comparison operator: $=, 
\neq, \leq, >, \geq, \text{is}, \text{is} \& \text{and} \text{implies}$. A CompExpr does not apply any XPath steps on its subexpressions, and it returns a boolean value, on which no XPath steps can be applied, thus: $\bar{p}^rel_{v_{root} \rightarrow E_0} = \emptyset$, $\bar{p}^rel_{v_{root} \rightarrow E_1} = \emptyset$ and $\bar{p}^rel_{v_{root} \rightarrow E_{cmp}} = \emptyset$.

If $E_{cmp}$ is a value comparison expression, with $\bar{p}^rel_{v_{root} \rightarrow E_0} = \emptyset$, $\bar{p}^rel_{v_{root} \rightarrow E_1} = \emptyset$ and Definition 6.1.4 we have $dEq(E_0, E'_0)$ and $dEq(E_1, E'_1)$. It is then clear that $dEq(\bar{p}^rel_{v_{root} \rightarrow E_{cmp}} (E_{cmp}, E'_{cmp})$ holds.

If $E_{cmp}$ is a node comparison expression, according to the by-projection insertion condition (a), $E_0$ and $E_1$ only contain XML nodes from different XML documents\(^{18}\). If $\circ$ represents the is operator, both $E_{cmp}$ and $E'_{cmp}$ return false. If $\circ$ represents the $\leq$ or the $\geq$ operator, both $E_{cmp}$ and $E'_{cmp}$ return either true or false. Thus, $E_{cmp}$ and $E'_{cmp}$ are deep-equal, under the

\[\text{NodeCmp} \text{and NodeSetExpr are also allowed. The requirements that these expressions must eliminate duplicates and order nodes in their results can be treated as if they apply a self step on } E_{seq}.
\]

\[\text{If } E_0 \text{ and } E_1 \text{ contain XML nodes from the XML documents, the by-projection insertion condition (a) forbids these two expressions and all their subexpressions to be decomposed. This conflicts with condition (Cnd2), which requires that } E_{cmp} \text{ must depend on at least one remote sequence.}\]
definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).
Then, with \( \mathcal{P}_{\text{root}^e}^\text{cmp} = \emptyset \) and Definition 6.1.4, we have \( dE_{\text{cmp}} \mathcal{P}_{\text{root}^e}^\text{cmp} (E_{\text{cmp}}, E_{\text{cmp}}') \).
Hence, \( \nu_x \) cannot be a \text{CompExpr}.

6.5.6.5 \text{NodeSetExpr}
Assume \( \nu_x \) is a \text{NodeSetExpr}. According to condition (Cnd1), the following statement must be true:

\[
\begin{align*}
\neg dE_{\text{root}^e}^{\text{rel}} (E_0, E_1') & \quad dE_{\text{root}^e}^{\text{rel}} (E_0, E_1) \\
\neg dE_{\text{root}^e}^{\text{rel}} (E_{\text{nset}}, E_{\text{nset}}') & \quad (p_{\text{rel}}^{\text{nset}})
\end{align*}
\]

The symbol \( \Box \) represents a node set operator: \text{union}, \text{intersect} or \text{except}. Like the restriction on node comparison expressions, the by-projection insertion condition \( \text{a} \) enforces that \( E_0 \) and \( E_1 \) only contain XML nodes from different XML documents.
If \( \Box \) represents an \text{intersect}, both \( E_{\text{nset}} \) and \( E_{\text{nset}}' \) return the empty sequence. It is trivial to see that \( dE_{\text{root}^e}^{\text{rel}} (E_{\text{nset}}, E_{\text{nset}}') \) holds.
If \( \Box \) represents a \text{union}, \( E_{\text{nset}} \) returns \( (E_0, E_1) \) or \( (E_1, E_0) \), and \( E_{\text{nset}}' \) returns \( (E_0', E_1') \) or \( (E_1', E_0') \). Thus, we have \( dE_{\text{root}^e} (E_{\text{nset}}, E_{\text{nset}}') \) (with implementation freedom). Based on the static properties analysis rule \text{Union} (Section 6.2.10), it can be deduced that: \( \mathcal{P}_{\text{root}^e}^{\text{rel}} = \mathcal{P}_{\text{root}^e}^{\text{rel}} \cup \mathcal{P}_{\text{root}^e}^{\text{rel}} = \mathcal{P}_{\text{root}^e}^{\text{rel}} \). The result of applying \( \mathcal{P}_{\text{root}^e}^{\text{rel}} \) on \( E_{\text{nset}} \) is equivalent to concatenating the results of applying \( \mathcal{P}_{\text{root}^e}^{\text{rel}} \) on \( E_0 \) and applying \( \mathcal{P}_{\text{root}^e}^{\text{rel}} \) on \( E_1 \). The same holds for \( E_{\text{nset}}' \), \( E_1' \) and \( E_2' \). Thus, we have \( dE_{\text{root}^e}^{\text{rel}} (E_{\text{nset}}, E_{\text{nset}}') \).
If \( \Box \) represents an \text{except}, \( E_{\text{nset}} \) and \( E_{\text{nset}}' \) return \( E_0 \) and \( E_0' \), respectively. With the premise \( dE_{\text{root}^e}^{\text{rel}} (E_0, E_0') \), we have \( dE_{\text{root}^e}^{\text{rel}} (E_{\text{nset}}, E_{\text{nset}}') \).
Hence, \( \nu_x \) cannot be an \text{NodeSetExpr}.

6.5.6.6 \text{StepExpr}
Assume \( \nu_x \) is a \text{StepExpr} (shortened as: \text{ST}), according to (Cnd1) above, the following statement must be true:

\[
\begin{align*}
\neg dE_{\text{root}^e}^{\text{step}} (E_{\text{step}}, E_{\text{step}}') & \quad dE_{\text{root}^e}^{\text{step}} (E_1, E_1') \\
\neg dE_{\text{step}}^\text{rel} (E_{\text{step}}, E_{\text{step}}') & \quad (p_{\text{step}}^{\text{rel}})
\end{align*}
\]

Assume that the returned paths of \( E_1 \) are \( \mathcal{R}_{E_1} \). Assume that \( \nu_x \) is a \text{StepExpr}, \( \mathcal{R}_{E_{\text{step}}} = \{ p_1^{E_{\text{step}}}, \ldots, p_k^{E_{\text{step}}} \} \). Then the returned paths of \( E_{\text{step}} \) are \( \mathcal{P}_{E_{\text{step}}} = \{ p_1^{E_{\text{step}}} / \text{ST}, \ldots, p_k^{E_{\text{step}}} / \text{ST} \} \). Assume that expressions that depend on \( E_{\text{step}} \) apply a number of paths, with each path containing a number of XPath steps, on \( E_{\text{step}} \), either as returned paths or as used paths. Then, the relative projection paths of \( E_{\text{step}} \) are:

\[
\begin{align*}
\mathcal{P}_{\text{root}^e}^{\text{step}} & : \mathcal{U} = \{ \text{ST}_1^{\text{step}} / \text{ST}_1, / \ldots / \text{ST}_m^{\text{step}} \} \\
\mathcal{P}_{\text{root}^e}^{\text{step}} & : \mathcal{U} = \{ \text{ST}_1, / \ldots / \text{ST}_m \}
\end{align*}
\]

\[
\begin{align*}
\mathcal{P}_{\text{root}^e}^{\text{step}} & : \mathcal{R} = \{ \text{ST}_1^{\text{step}} / \text{ST}_1, / \ldots / \text{ST}_m^{\text{step}} \} \\
\mathcal{P}_{\text{root}^e}^{\text{step}} & : \mathcal{R} = \{ \text{ST}_1, / \ldots / \text{ST}_m \}
\end{align*}
\]

\footnote{See the rules \text{STEP}^p, \text{STEP}^w and \text{STEP}^h in Section 6.2.12.}
The relative projection paths of $E_1$ are\(^{20}\):

\[
\begin{align*}
\text{rel}_{\text{vroot} \rightarrow E_1}: \mathcal{U} &= \{ \text{st}/\text{st}_{11}^{m_1}/\text{st}_{12}^{m_2}/\ldots/\text{st}_{1n}^{m_n}, \\
&\phantom{=} \text{st}/\text{st}_{21}^{m_1}/\text{st}_{22}^{m_2}/\ldots/\text{st}_{2n}^{m_n}, \\
&\phantom{=} \ldots, \\
&\phantom{=} \text{st}/\text{st}_{m1}^{m_1}/\text{st}_{m2}^{m_2}/\ldots/\text{st}_{mn}^{m_n} \} \\
\text{rel}_{\text{vroot} \rightarrow E_1}: \mathcal{R} &= \{ \text{st}/\text{st}_{11}'/\text{st}_{12}'/\ldots/\text{st}_{1n}'^{m_n}, \\
&\phantom{=} \text{st}/\text{st}_{21}'^{m_1}/\text{st}_{22}'^{m_2}/\ldots/\text{st}_{2n}'^{m_n}, \\
&\phantom{=} \ldots, \\
&\phantom{=} \text{st}/\text{st}_{m1}'^{m_1}/\text{st}_{m2}'^{m_2}/\ldots/\text{st}_{mn}'^{m_n} \}
\end{align*}
\]

The premise $dEq_{\text{vroot} \rightarrow E_1}(E_1, E_1')$ means that $\forall p_i \in \text{rel}_{\text{vroot} \rightarrow E_1} : dEq(E_1/p_i, E_1'/p_i)$. Substitute $E_1/\text{ST}$ with $E_{\text{step}}$ and $E_1'/\text{ST}$ with $E_{\text{step}}'$, we have $\forall p_j \in \text{rel}_{\text{vroot} \rightarrow E_{\text{step}}} : dEq(E_{\text{step}}/p_j, E_{\text{step}}'/p_j)$.

Hence, $v_x$ can not be a $\text{StepExpr}$.

6.5.6.7 Built-in function calls

Under the by-projection semantics, the built-in functions $\text{fn:root()}, \text{fn:id()}, \text{fn:idref()}$ and $\text{fn:lang()}$ and their parameters can also be decomposed. A common property of these functions is that they need to access nodes outside the subtree(s) of their parameters, something not supported by the by-value and by-fragment semantics. In pass-by-projection, we have extended the projection path to allow the functions $\text{root()}, \text{id()}$ and $\text{idref()}$ to be part of a $\text{SimplePath}$ (Table 5.6), while the function $\text{fn:lang()}$ can be represented by ancestor and attribute steps\(^{21}\).

For all $\text{FunCall}$s to built-in functions, we can use similar reasoning to that for $\text{StepExpr}$ expressions to deduce that $v_x$ can not be a $\text{FunCall}$ to a built-in function.

In summary, we were not able to find a $v_x$ among all vertices in a d-graph $G$ (including the root vertex $v_{\text{root}}$) which is not deep-equal to its corresponding vertex $v_x'$ in $G'$, while all vertices on which $v_x$ depends are deep-equal to their corresponding vertices in $G'$. This implies $dEq_{v_{\text{root}} \rightarrow v_{\text{root}}}(v_{\text{root}}, v_{\text{root}}')$. It is trivial to see that $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}} = \emptyset$, according to Definition 6.1.4, we have $dEq(v_{\text{root}}, v_{\text{root}}')$. Hence, the assumption $\neg dEq(Q, Q')$ of Part I does not hold.

Part II

In this part we drop the assumption used by Part I that the relative projection paths of all vertex $v_i \in G$ are $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$. The relative projection paths for a parameter $v_j$ of a remote expression $v_i$ are actually $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$. Since $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$ is more selective than $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$, we need to check if projecting $v_j$ using $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$ could cause the remote expression $v_i$ to return an incorrect result.

Assume $\neg dEq(Q, Q')$, then there must exist one vertex $v_x \in G$, which is a parameter of a by-projection decomposition point $v_d \in G$, such that projecting $v_x$ using the paths $\text{rel}^{rel}_{v_{\text{root}} \rightarrow v_{\text{root}}}$ will cause the corresponding vertex $v_d'$ of $v_d$ (in the decomposed d-graph $G'$) to be not by-projection equal to $v_d$. Formally, $\exists v_x, v_d \in G : v_d \not\sim v_x \land v_d \in \text{I}^p(G)$ and $\exists v_x', v_d' \in G : v_d' \sim v_x$, such that $dEq_{v_{\text{root}} \rightarrow v_{\text{root}}}(v_x, v_x') \Rightarrow \neg dEq_{v_{\text{root}} \rightarrow v_{\text{root}}}(v_d, v_d')$ holds.

In XQuery, remote expressions can be seen as black boxes with one or more inputs\(^{22}\) and one (possibly empty) output. Let $v_p$ be a parameter of $v_d$. We examine all possible cases to

\(^{20}\)As shown in Example 6.5.1, the relative projection paths of $E_1$ are longer than the relative projection paths of $E_{\text{step}}$, because $E_1$ is further away from $v_{\text{root}}$ than $E_{\text{step}}$.

\(^{21}\)See the rule (BlkIn(\text{mth})) in Section C.14.

\(^{22}\)Remote expressions with zero input have already been handled in Part I.
6.6 Correctness Proof of the XQUF Decomposition Algorithm

**Theorem 6.6.1.** XQUF Decomposition Correctness: Let \( Q_u \) be a normal updating XCore query (i.e., without any XRPC expressions, but containing at least one updating expression which is not a subexpression of a TransformExpr) and \( G \) the corresponding d-graph. \( I(G) \subset G \) is the (non-empty) set of decomposition points validated by one of the decomposition algorithms, and the vertices in \( I(G) \) also satisfy the XQUF insertion conditions (Section 5.7.1).
Let $G'$ be the d-graph derived by doing an $XRPCExpr$ insertion above each vertex in $I(G)$ (Section 5.3.2), and $Q'_u$ the corresponding query of $G'$. Then, $dEq(Q_u, Q'_u)$ holds under the definition of deep-equal updating queries with implementation freedom (Definition 6.1.9).

Proof. Let $\Delta$ and $\Delta'$ be the PULs yielded by $Q_u$ and $Q'_u$, respectively. By Definition 6.1.9, $dEq(Q_u, Q'_u)$ holds if we can prove that $dEq(\Delta, \Delta')$ holds according to the definition of deep-equal pending update lists (Definition 6.1.8). Definition 6.1.8 states that two PULs are deep-equal to each other if they both contain

(i) at least one $upd:delete$ action on the same target nodes,

(ii) exactly one $upd:replaceNode$, $upd:replaceValue$, $upd:replaceElementContent$ or $upd:rename$ action on the same target nodes with deep-equal new contents, and

(iii) the same number of the same kind of insertion action on the same target nodes with deep-equal new contents.

Thus, there are three determining factors for two PULs to be deep-equal: (1) identifiers of the target nodes, (2) the new contents with which the target nodes will be updated, and (3) the number of the same update actions on the same target nodes.

The new contents for the target nodes and the number of times that a certain update action will be executed are computed by the read-only subexpressions that might be decomposed by the particular decomposition algorithm. In Theorem 6.3.9, 6.4.8 and 6.5.6, it has already been proven that a decomposed read-only subexpression produces deep-equal results to those of its corresponding subexpression in the original query. Thus, we only need to prove that $Q_u$ and $Q'_u$ update the same nodes, whose identifiers are computed by the $TargetExpr$ subexpression of an $UpdExpr$.

Case 1: $Q_u$ contains only updates on local documents

For this class of updating queries, the XQUF insertion conditions (Section 5.7.1) forces all $UpdExprs$ and their $TargetExpr$ subexpressions in $Q'_u$ to be executed on the local peer. This also prevents target nodes from being passed as function parameters or results, which would lose the original identities of the target nodes. Thus, it is easy to see that the target nodes of $Q'_u$ are equivalent to those of $Q_u$, which proves $dEq(\Delta, \Delta')$. Hence, $dEq(Q_u, Q'_u)$ holds, in case $Q_u$ only contains updates on local documents.

Case 2: $Q_u$ contains only updates on remote documents In Section 5.7.2, we have extended the semantics of XQUF to allow updating expressions on remote documents. We specified that updates on a remote document are first applied on a local copy of the document (i.e., the remote document is retrieved to the local peer using $fn:doc()$), after which the updated local copy is sent to the peer hosting the remote document using $fn:put()$ to overwrite the existing document.

When computing $Q'_u$, our XQUF rewrites allows those $UpdExprs$ (together with their $TargetExpr$ subexpressions) which only contain updates on remote documents hosted by a single peer $p_i$ to be decomposed and executed on $p_i$\textsuperscript{23}. The constraint on a single peer is necessary to ensure that the target nodes are never passed as function parameters or results.

\textsuperscript{23}As pointed out in Section 5.7.2, the updating functions could also be executed on other peers than the hosting peer, however, it implies the same semantics as executing those functions on the local peer, which makes remote execution not meaningful.
Let $\text{UpdExpr}_{loc}$ denote those $\text{UpdExprs}$ in $Q_u$ that will not be decomposed, and $\text{UpdExpr}_{hi}$ denote the $\text{UpdExprs}$ in $Q_u$ that can be executed on peer $h_i$. Let $\Delta_{loc}$ and $\Delta_{hi}$ denote the partial PULs that evaluating $\text{UpdExpr}_{loc}$ and $\text{UpdExpr}_{hi}$ will yield, respectively. They correspond to $\Delta'_{loc}$ and $\Delta'_{hi}$ in $Q'_u$, i.e., $\Delta'_{loc}$ is the result of evaluating $\text{UpdExpr}_{loc}$ on the local peer, while $\Delta'_{hi}$ is the result of evaluating $\text{UpdExpr}_{hi}$ on peer $h_i$. Hence,

$$\Delta = \Delta_{loc} \cup \left( \bigcup_{i=1}^{n} \Delta_{hi} \right) \quad \text{and} \quad \Delta' = \Delta'_{loc} \cup \left( \bigcup_{i=1}^{n} \Delta'_{hi} \right)$$

where $n$ is the number of remote peers involved in the execution of $Q'_u$. Clearly, $\text{Eq}(\Delta_{loc},\Delta'_{loc})$. For each $\Delta'_{hi}$, it is also easy to see that it identifies the same target nodes, since each $\Delta'_{hi}$ is created by evaluating $\text{UpdExpr}_{hi}$ on the peer hosting the documents. Together, we have $\text{Eq}(\Delta,\Delta')$, which implies that $\text{Eq}(Q_u, Q'_u)$ holds if $Q_u$ only contains updates on remote documents.

**Case 3:** $Q_u$ contains updates on both local and remote documents

For this case, $\text{Eq}(Q_u, Q'_u)$ is proven by combining Case 1 and 2.

### 6.7 Correctness Proof of Distributed Code Motion

In Section 5.4.4, we have described that, in certain cases, subexpressions of a valid decomposition point $r_s$ could be moved out of the subgraph rooted at $r_s$ and be replaced by a new parameter presenting this subexpression. In this section, we prove that applying this so-called distributed code motion technique to the resulting decomposed d-graph of a certain decomposition algorithm will not cause a valid decomposition point $r_s$ to produce non-deep-equal result.

**Theorem 6.7.1. Distributed Code Motion Correctness:** Let $G$ be a d-graph and $G_{r_s}$ is the subgraph of $G$ rooted at the vertex $r_s$, where $r_s$ is a decomposition point validated by one of the decomposition algorithms. Let $\{p_1, ..., p_m\} \in V(G)$ be the set of vertices that are parameters of $r_s$. Let $v_i$ be a vertex in the subgraph rooted at $r_s$ which satisfies all of the following constraints:

- $v_i$ is also a decomposition point validated by the same decomposition algorithm, i.e., $v_i \in V(G_{r_s}) \land v_i \in I(G)$;
- there is exactly one $p_s \in \{p_1, ..., p_m\}$ such that $v_i \overset{p}{\sim} p_s$;
- there is exactly one path $(v_i, v_{i+1}, ..., v_{i+n}, p_s)$ that can reach $p_s$ starting from $v_i$.

Then, moving $v_i$ out of $G_{r_s}$ by introducing a new parameter $p_n$ to represent $v_i$, and replacing $v_i$ with $p_n$ will not cause remote executing of $r_s$ to produce non-deep-equal result to the local execution of $r_s$.

**Proof.** Since $r_s$ and $v_i$ are both valid decomposition points, they could be executed on possibly different remote peers. With Theorem 6.3.9, Theorem 6.4.8 and Theorem 6.5.6, we have that, regardless of on which peer $r_s$ is executed, it produces deep-equal results, possibly using the result of a remote execution of its subexpression $v_i$. Moving $v_i$ out of $G_{r_s}$ and passing its value as a parameter to $r_s$ implies that $v_i$ will be executed on a different peer than the peer, on which $r_s$ will be executed. Therefore, this code motion will cause remote execution of $r_s$ to produce a deep-equal result.