Commutativity Analysis for XML Updates

GIORGIO GHELLI
Università di Pisa
and
KRISTOFFER ROSE and JÉRÔME SIMÉON
IBM Research

An effective approach to support XML updates is to use XQuery extended with update operations. This approach results in very expressive languages which are convenient for users but are difficult to optimize or reason about. A crucial question underlying many static analysis problems for such languages, from optimization to view maintenance, is whether two expressions commute. Unfortunately, commutativity is undecidable for most existing XML update languages. In this paper, we propose a conservative analysis for an expressive XML update language that can be used to determine commutativity. The approach relies on a form of path analysis that computes upper bounds for the nodes that are accessed or modified in a given expression. Our main result is a commutativity theorem that can be used to identify commuting expressions. We illustrate how the technique applies to concrete examples of query optimization in the presence of updates.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Query languages; H.2.4 [**Database Management**]: Query processing

General Terms: Algorithms, Performance

Additional Key Words and Phrases: XML, Updates, XQuery, Commutativity, Optimization

1. INTRODUCTION

Most of the proposed XML update languages [Chamberlin et al. 2007; Lehti 2001; Tatarinov et al. 2001; Benedikt et al. 2005a; Ghelli et al. 2006] rely on extending a full-fledged query language such as XQuery [Boag et al. 2007] with update primitives. To simplify specification and reasoning, some of the first proposals [Chamberlin et al. 2007; Lehti 2001; Benedikt et al. 2005a] have opted for a so-called *snapshot semantics*, which delays update application until the end of the query. However, this leads to counter-intuitive results for some queries, limits the expressiveness in a way that is not always acceptable for applications, and introduces a semantic distinction between queries and subqueries. For that reason, more recent proposals [Ghelli et al. 2006; Carey et al. 2006; Hidders et al. 2006] give the ability to apply updates in the course of query evaluation. Such languages are usually

Giorgio Ghelli, Dipartimento di Informatica, Università di Pisa, Via Buonarroti 2, I-56127 Pisa, Italy; *ghelli@di.unipi.it*. Kristoffer Rose and Jérôme Siméon, IBM Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, U.S.A.; *krisrose@us.ibm.com*, *simeon@us.ibm.com*.

specified using a semantics with a strict evaluation order. For example, consider the following query, which first inserts a set of elements, then accesses those elements using a path expression.

for $x **in** $doc/country **return do insert** <new/> **into** $x,
count($doc/country/new)

Such an example cannot be written in a language based on a snapshot semantics, as the count function always returns zero.[1] However, it can be written in the XQuery! [Ghelli et al. 2006] or the XQueryP [Carey et al. 2006] proposals, which both support immediate update application and explicit left-to-right evaluation order. Still, such semantics severely restricts the optimizer's ability for rewritings, unless the optimizer is able to decide that some pairs of expressions *commute*, *i.e.*, they return the same value and have the same effect independently of their evaluation order.

Deciding *commutativity*, or more generally whether an update and a query *interfere*, has numerous applications, including optimizations based on rewritings, detecting when an update needs to be propagated through a view (usually specified as a query), deciding whether sub-expressions of a given query can be executed in parallel, *etc.* Unfortunately, commutativity is undecidable for XQuery extended with updates. In this paper, we propose a conservative approach to detect whether two query/update expressions commute. Our technique relies on an extension of the path analysis proposed by Marian and Siméon [2003] that infers upper bounds for the nodes accessed and modified by a given expression. Such upper bounds are specified as simple path expressions for which disjointness is decidable [Benedikt et al. 2005; Miklau and Suciu 2004; Hammerschmidt et al. 2005].

1.1 Problem Statement

In the rest of the paper, we focus on a simple XQuery extension with insert and delete operations, whose syntax and semantics is essentially that of the XQuery! language proposed by Ghelli et al. [2006]. To simplify exposition, we use a language in which updates are always applied immediately, *i.e.*, without XQuery! fine-grained control over the snapshot semantics. This language is powerful enough to exhibit the main problems related to commutativity analysis, yet simple enough to allow a complete formal specification and treatment within the space of this paper. To illustrate our purpose, let us consider a few queries and updates in that language, shown in Figure 1. Some of the examples obviously commute, for instance (U1) deletes nodes that are unrelated to the nodes accessed by (Q1) or (Q2). This can be inferred easily by looking at the paths in the query used to access the corresponding nodes. On the contrary, (U2) does not commute with (Q1) since the query accesses some of the nodes being inserted. Deciding whether the set of nodes accessed or modified are disjoint is hard for any non-trivial update language. For instance, deciding whether (U3) and (Q2) interfere requires some analysis of the predicates, which can be arbitrarily complex in XQuery.

In this paper, we propose a commutativity analysis that relies on a form of abstract interpretation that approximates the set of nodes processed by a given ex-

[1] More precisely, the number of `new` elements in the original document

(Q1) count($doc/country/new)

(Q2) $doc/country[population > 20]

(Q3) **for** $x **in** $doc//country
return ($x//name)

(Q4) **for** $x **in** $doc/country
return $x/new/../very-new

(U1) **do delete** $doc/wines/california

(U2) **for** $x **in** $doc/country
return do insert <new/> **into** $x

(U3) **for** $x **in**
$doc/country[population < 24]
return do delete $x/city

Fig. 1. Some sample queries.

pression. To be useful, this analysis must satisfy the following properties. Firstly, since we are looking to check *disjointness*, we must infer an upper bound for the corresponding nodes. Secondly, the analysis must be precise enough to be useful in practical applications. Finally, the language used to describe those upper bounds must be such that disjointness is decidable [Hammerschmidt et al. 2005]. In the context of XML updates, *paths* are a natural choice for the approximation of the nodes being accessed or updated, provided we can show that they satisfy the precision and decidability requirements.

1.2 Approach

The path analysis itself is a relatively intuitive extension of the analysis from Marian and Siméon [2003] to handle update operations. This is done by computing not only the set of paths for the nodes being *accessed* by the query, as in Marian and Siméon [2003], but also the set of paths for the nodes being *updated* by the query. The following table illustrates the result of our path analysis for query (Q3) and update (U3).

query	accessed paths	updated paths
(U3)	$doc/country $doc/country/population//node() $doc/country/city	$doc/country/city//∗
(Q3)	$doc//country $doc//country//name	none

A specific challenge in extending the work of Marian and Siméon [2003] in the context of updates is to find a precise definition for which nodes are being accessed and updated. For instance, one may argue that (U3) only modifies nodes reached by the path country/city, since, in our language, **do delete** only detaches nodes from their parents [Chamberlin et al. 2007; Ghelli et al. 2006]. However, one would then miss the fact that (U3) interferes with (Q3) because the city nodes may have a country or a name descendant, which is made unreachable by the deletion. In this particular case, our analysis considers that all the descendants below the path country/city are affected by the deletion in update (U3) .

A second issue is the volatile nature of the store being analyzed. A natural interpretation for the paths resulting from the analysis is to denote the set of nodes returned by that path. However, such an interpretation would change during the execution of an update statement. We define a way to associate a meaning to a path

that is *stable* in the face of a changing data model instance. To address that issue, we formalize XQuery execution as a manipulation of *store histories*. This approach is better adapted to path-based analysis in the presence of immediate updates, compared to more traditional techniques [Tofte and Talpin 1997; Benedikt et al. 2005a], which partition the store into regions to which the different locations are immutably associated.

Finally, the analysis requires an unusual interpretation for path expressions. Traditionally, the semantics of a path expression is defined as the set of nodes that are returned by that path. However, that interpretation is not sufficient in the presence of reverse axes. Consider for instance query (Q4). If the returned expression $\$x$/new/../very-new were just associated to the nodes denoted by country/new/../very-new, then the interference with (U2) would not be observed, since the path country/new//$*$ updated by (U2) refers to a disjoint set of nodes. Hence, the analysis must also consider the nodes *traversed* by the evaluation of $\$x$/new/../very-new, corresponding to the path country|country/new|country/new/.., whose second component intersects with country/new//$*$.

1.3 Commutativity and Optimization

Optimization based on rewritings is one of the main applications and motivations for our work. In practice, commutativity must often be checked in order for rewritings to be sound. For instance, consider the following query.

for $\$x$ **in** $\$doc$/wines **return** (**do delete** $\$x$/price, count($\$doc$/country))

Since the count expression does not depend on the $\$x$ variable used in the loop, one may want to hoist that expression as follows.

let $\$y$:= count($\$doc$/country) **return**
for $\$x$ **in** $\$doc$/wines **return** (**do delete** $\$x$/price, $\$y$)

This is an important optimization, which avoids recomputing the number of countries many times. However, this rewriting is only sound if the deletion inside the loop does not apply to the nodes being accessed in the count expression. In other words, the rewriting is sound only because the subqueries count($\$doc$/country) and **do delete** $\$x$/price commute.

More generally, consider any complex FLWOR[2] expression with some side-effects. Almost invariably, the code is designed so that the side-effects do not affect the queried data. If the optimizer can indeed *prove* that the different subexpressions commute, it can explore plans with different evaluation orders, and can also consider pipelined execution. Without commutativity analysis the search space for the optimizer is severely limited. Also, observe that commutativity must in general be checked on subexpressions, for which some variables may be free. We show how the path analysis and commutativity theorem can be performed on such subexpressions, by using a notion of environment which binds free variables to paths. Even though our path analysis only provides an approximation for the nodes being used or accessed, we believe it is a good starting point that covers a lot of common cases found in practice.

[2]FLWOR stands for the key for-let-where-order by-return query construct of XQuery.

1.4 Contributions

The main contributions of the paper are as follows:

—We develop static analysis techniques to infer paths which approximate the nodes that are *accessed* and *updated* by a given expression in an XML update language;
—We present a formal definition of when such an analysis is sound, based on a notion of *store history equivalence* and a notion of *stability*; this formal definition provides a guide for the definition of the inference rules;
—We provide a detailed soundness proof for the proposed path analysis;
—We present a commutativity theorem, that provides a sufficient condition for the commutativity of two expressions, based on the given path analysis;
—We provide several examples of the use of the analysis for query optimization.

Part of this work was previously published in Ghelli et al. [2007]; this article contains a revised formal framework which greatly simplifies the soundness proof for commutativity, and extended proof material that provides additional insights into the approach (Section 5). We also extend Ghelli et al. [2007] with several concrete examples that illustrate the use of the proposed path analysis techniques for query optimization (Section 7). We believe the study of commutativity properties to be crucial in order to support optimization in real XML updates or XQuery Scripting [Carey et al. 2006] compilers. To the best of our knowledge, this is the first attempt at such a study. In the absence of any prior work in the area, we focused on a simple incarnation of the general commutativity analysis problem. Notably, the following aspects are not treated in the paper, and are left for future work: aspects related to XML Schema validation, snapshot semantics, and namespaces handling.

1.5 Related Work

Numerous update languages have been proposed in the last few years [Chamberlin et al. 2007; Lehti 2001; Tatarinov et al. 2001; Benedikt et al. 2005a; Ghelli et al. 2006]. Some of the most recent proposals [Ghelli et al. 2006; Carey et al. 2006] are very expressive, notably providing the ability to observe the effect of updates during query evaluation. Although the language proposed by Carey et al. [2006] limits the locations where updates occur, this has little impact on our static analysis, which also works for a language where updates can occur anywhere in the query such as XQuery! [Ghelli et al. 2006]. Very little work has been done so far on optimization or static analysis for such XML update languages, a significant exception being the work of Benedikt et al. [2005a; 2005b]. However, they focus on analysis techniques for a language based on snapshot semantics, while we consider a much more expressive language. A notion of path analysis was proposed by Marian and Siméon [2003], which we extend here by considering side effects.

Independence between updates and queries has been studied for the relational model by Elkan [1990] and Levy and Sagiv [1993]. The problem becomes more difficult in the XML context because of the expressivity of XML languages. In the relational case, the focus has been on trying to identify fragments of Datalog for which the problem is decidable, usually by reducing the problem to deciding reachability. Instead, we propose a conservative approach using a technique based

on paths analysis which works for arbitrary XML updates and queries. Finally, commutativity properties for tree operations are important in the context of transactions for tree models [Dekeyser et al. 2003; 2004; Lanin and Shasha 1986], but these papers rely on dynamic knowledge while we are interested in static commutativity properties, hence the technical tools involved are quite different.

1.6 Organization

The rest of the paper is organized as follows. Section 2 presents the XML data model and store. Section 3 presents the update language syntax and semantics. Section 4 presents the path analysis followed by the soundness theorem in Section 5 and the commutativity theorem in Section 6. In Section 7 we illustrate the use of the analysis for query optimization purposes. Finally, we conclude in Section 8.

2. AN XML STORE FOR UPDATES

In this section we define a notion of XML *store* that can support XML queries and updates. The proposed formalization is consistent with the XPath 2.0 and XQuery 1.0 Data Model [Fernández et al. 2007]. In addition, we define a notion of *store history* that we use to keep track of the evolution of the data model during the evaluation of update expressions. For simplicity, we only treat element and text nodes. We also ignore sibling order, since it has little impact on the approach and on the analysis precision.

Notations. We use $\vec{a}$ for vectors $(a_1, \ldots, a_n)$ of length n and $\vec{a}, \vec{b}$ for the concatenated vector $(a_1, \ldots, a_n, b_1, \ldots, b_m)$, and we permit $\vec{a}$ in set contexts to mean the set $\{a_1, \ldots, a_n\}$. We use $\perp$ to denote undefined values, specifically functions have the "value" $\perp$ where undefined. We use the notation $x \mapsto v$ to indicate that x is mapped to v, and $f + f'$ to denote the extension of f with f', where the second overrides the first, if needed; this is notably used in the notation for environments used during path analysis. For instance, $(dEnv + \$x \mapsto \vec{v}_1)$ denotes the environment *dEnv*, extended with variable $\$x$ bound to value $\vec{v}_1$.

2.1 The Store

In our model, we assume an infinite set of nodes $\mathcal{N}$ which are pre-arranged into an infinite set of trees; a store is built by selecting a finite subset of $\mathcal{N}$. Formally, we assume the existence of disjoint sets of *node ids* $\mathcal{N}$ (denoted by n and m), *node kinds* $\mathcal{K} = \{element, text\}$, *names* $\mathcal{Q}$ (denoted by q, f, and x), and *strings* $\mathcal{S}$ used to specify the content of text nodes. All these sets, apart from $\mathcal{K}$, are infinite. We assume a binary relation $\mathcal{E} \subseteq (\mathcal{N} \times \mathcal{N})$ that structures $\mathcal{N}$ as a forest, *i.e.*, the transitive closure $\mathcal{E}^+$ of $\mathcal{E}$ is irreflexive and $\{(n, m), (n', m)\} \subseteq \mathcal{E} \Rightarrow n = n'$. Moreover, for each n, the set $\{\, m \mid (n, m) \in \mathcal{E} \,\}$ is infinite. A "root" is a node that has no parent in $\mathcal{E}$. We also assume a function $\mathcal{R}$ mapping roots to node *locations*; locations are defined as follows (the notation $\ell \in \mathcal{L} ::= production$ is used to indicate that $\mathcal{L}$ is the set of all terms produced by the non-terminal ℓ)

$$\ell \in \mathcal{L} ::= uri \mid cl$$

A "uri-location" *uri* specifies that the tree rooted at n_r can be reached with **doc**(*uri*). A "code-location" *cl* specifies that the tree rooted in n_r has been created

by the execution of a piece of code identified by *cl*. Every path used in our analysis starts from a location. For each location $\ell \in \mathcal{L}$, the set $\{ n \mid \mathcal{R}(n) = \ell \}$ is infinite. Hence, locations add one more level of tree-structure on top of the trees of $\mathcal{N}$. We are now ready to define our notion of store.

Definition 1 (store). A *store* σ is a triple (N, D, F) where $N \subset \mathcal{N}$ contains the finite set of the document nodes, $D \subseteq N$ contains the set of nodes that have been "deleted", *i.e.*, that have been detached from their parent, and the node description F maps each node n to a node descriptor $[\mathsf{kind} : k, \mathsf{name} : q, \mathsf{value} : v]$, where $F(n).\mathsf{kind} \in \mathcal{K}$ is the kind of n, $F(n).\mathsf{name} \in \mathcal{Q}$ is its name (or is undefined) and $F(n).\mathsf{value} \in \mathcal{S}$ is its string content (or is undefined).

We use N_σ, D_σ, and F_σ to denote the components of a particular store σ. When two nodes in N_σ are related by $\mathcal{E}$, and the second has not been "deleted", we say that m is a parent of n and n is a child of m in σ, and write $(m, n) \in E_\sigma$, or $E_\sigma(m, n)$. Formally we define $E_\sigma = (\mathcal{E} \cap (N_\sigma \times (N_\sigma \setminus D_\sigma)))$. As usual, E_σ^+ is the transitive closure of E_σ, and E_σ^* is the transitive and reflexive closure. Finally, R_σ denotes the restriction of $\mathcal{R}$ to $N_\sigma \setminus D_\sigma$.

Further constraints, such as the fact that text nodes have no children, can be easily added, but we ignore such details for simplicity.

Example 2. Consider the following document, assumed to be associated to the URL `http://example.com/1.xml`.

```
<root><a><b/></a><a>c</a></root>
```

Assuming that n_1, n_2, n_3, n_4, n_5 are such that $\mathcal{E} \supseteq \{(n_1, n_2), (n_1, n_4), (n_2, n_3), (n_4, n_5)\}$ and that $\mathcal{R}(n_1) =$ `http://example.com/1.xml`, the document can be encoded through the following store σ_1, where $kind_{\sigma_1}$ represents the function that maps each n to $F_{\sigma_1}(n).\mathsf{kind}$, and similarly for $name_{\sigma_1}$ and $value_{\sigma_1}$.

$$
\begin{aligned}
N_{\sigma_1} &= \{n_1, n_2, n_3, n_4, n_5\} \\
D_{\sigma_1} &= \{\} \\
kind_{\sigma_1} &= \{n_1 \mapsto \mathit{element}, n_2 \mapsto \mathit{element}, n_3 \mapsto \mathit{element}, n_4 \mapsto \mathit{element}, n_5 \mapsto \mathit{text}\} \\
name_{\sigma_1} &= \{n_1 \mapsto \mathsf{root}, n_2 \mapsto \mathsf{a}, n_3 \mapsto \mathsf{b}, n_4 \mapsto \mathsf{a}\} \\
value_{\sigma_1} &= \{n_5 \mapsto \text{“}c\text{”}\}
\end{aligned}
$$

According to our definition, the edges and the root mapping of σ_1 are:

$$
\begin{aligned}
E_{\sigma_1} &= \{(n_1, n_2), (n_1, n_4), (n_2, n_3), (n_4, n_5)\} \\
R_{\sigma_1} &= n_1 \mapsto \texttt{http://example.com/1.xml}
\end{aligned}
$$

If the `<b/>` element (node n_3) were later deleted, we would get a store σ_2 whose D and E are listed below, while N, R, and F are equal to those for σ_1.

$$
\begin{aligned}
D_{\sigma_2} &= \{n_3\} \\
E_{\sigma_2} &= \{(n_1, n_2), (n_1, n_4), (n_4, n_5)\}
\end{aligned}
$$

Remark 3. Extending this notion of store to the complete XPath 2.0 and XQuery 1.0 Data Model [Fernández et al. 2007] should not pose any fundamental difficulties. The missing aspects include: (1) namespaces [Bray et al. 1999], which introduce

two levels of naming, (2) the generalization of "textual content" to lists of "atomic" as defined by XML Schema [Biron and Malhotra 2001], (3) attribute, comment, processing instruction, and document node kinds, and (4) document order, which requires a "next-sibling" relation in the store. Also note that some subtleties of the semantics of updates are not fully covered here, such as merging of adjacent text nodes after an update or potential revalidation. Those issues are largely orthogonal to our purpose here.

2.2 Access and Updates

For convenience, we define here some of the usual accessors, plus a *node-test* auxiliary function, as follows.

$$\begin{aligned}
\mathit{children}(\sigma, n) &\stackrel{\text{def}}{=} \{\, n' \mid E_\sigma(n, n') \,\} \\
\mathit{desc}(\sigma, n) &\stackrel{\text{def}}{=} \{\, n' \mid E^+_\sigma(n, n') \,\} \\
\mathit{parent}(\sigma, n) &\stackrel{\text{def}}{=} \{\, n' \mid E_\sigma(n', n) \,\} \\
\mathit{ancestor}(\sigma, n) &\stackrel{\text{def}}{=} \{\, n' \mid E^+_\sigma(n', n) \,\} \\
\mathit{node\text{-}test}(\sigma, q, n) &\stackrel{\text{def}}{\Leftrightarrow} F_\sigma(n).\mathsf{name} = q \\
\mathit{node\text{-}test}(\sigma, *, n) &\stackrel{\text{def}}{\Leftrightarrow} \exists q\colon F_\sigma(n).\mathsf{name} = q \\
\mathit{node\text{-}test}(\sigma, \mathsf{text}(), n) &\stackrel{\text{def}}{\Leftrightarrow} F_\sigma(n).\mathsf{kind} = \mathit{text} \\
\mathit{node\text{-}test}(\sigma, \mathsf{node}(), n) &\qquad \text{is always true}
\end{aligned}$$

We now focus on operations that modify the store. Most existing XML update languages [Chamberlin et al. 2007; Lehti 2001; Tatarinov et al. 2001; Benedikt et al. 2005a; Ghelli et al. 2006] rely on a few kinds of atomic updates, such as node insertion, deletion, replacement and renaming. Here we focus on simple insert and delete operations. Replace adds no new issues and can be easily added; however, as we will see later on in Remark 22, renaming actually raises some problems with our path analysis. Some proposals also support operations for *moving* nodes from one part to another in the tree, which differs from inserting a copy in the new location and deleting the original, because *moving* preserves the node identity. Most proposals do not include moving, because it makes it impossible to encode parent-child relationship through node identities, and because it has bad interactions with optimization. We can only confirm this belief. Our store model is incompatible with moving, but we could easily adopt one which supports moving, as we did in Ghelli et al. [2007]. Nevertheless, a moving operation would severely weaken our ability to infer commutativity, as discussed later (Remark 22).

According to the "snapshot" semantics used by XQuery Updates [Chamberlin et al. 2007] and XQuery! [Ghelli et al. 2006], an update operation generates an atomic update request, which is applied to the store when the snapshot scope is closed. We ignore this issue and apply every update immediately (but comment on it in Remark 34). However, we still record the update details in "atomic update records", essential to define our notion of *store history*. Atomic update records represent update operations with enough detail to allow the update to be re-executed on a store, using the "update application" operation *apply* defined below.

Definition 4 (atomic update records). Atomic update records are tuples with the following structure, where $\vec{n}$ is a set of nodes and F maps $\vec{n}$ to node descriptors as in Definition 1.

$$\delta \in \mathsf{Update} ::= \mathsf{insert}(\vec{n}, F) \mid \mathsf{delete}(\vec{n})$$

Definition 5 (update application). The operation $\mathit{apply}(\sigma, \vec{\delta})$ applies a list of updates to a store, producing a new store, as follows.

$$\mathit{apply}((N_\sigma, D_\sigma, F_\sigma),\ (\mathsf{insert}(\vec{n}, F),\ \vec{\delta})) \stackrel{\text{def}}{=} \mathit{apply}((N_\sigma \cup \vec{n}, D_\sigma, F_\sigma + F),\ \vec{\delta})$$
$$\mathit{apply}((N_\sigma, D_\sigma, F_\sigma),\ (\mathsf{delete}(\vec{n}),\ \vec{\delta})) \stackrel{\text{def}}{=} \mathit{apply}((N_\sigma, D_\sigma \cup \vec{n}, F_\sigma),\ \vec{\delta})$$
$$\mathit{apply}(\sigma,\ ()) \stackrel{\text{def}}{=} \sigma$$

Definition 6. We use $\mathit{inserted}(\vec{\delta})$ to denote the set of nodes inserted by $\vec{\delta}$, and $F(\vec{\delta})$ to indicate their node description.

$$\mathit{inserted}(\vec{\delta}) \stackrel{\text{def}}{=} \bigcup_{\mathsf{insert}(\vec{n}, F) \in \vec{\delta}} \vec{n} \qquad F(()) \stackrel{\text{def}}{=} ()$$
$$\mathit{deleted}(\vec{\delta}) \stackrel{\text{def}}{=} \bigcup_{\mathsf{delete}(\vec{n}) \in \vec{\delta}} \vec{n} \qquad F(\mathsf{delete}(), \vec{\delta}) \stackrel{\text{def}}{=} F(\vec{\delta})$$
$$F(\mathsf{insert}(\vec{n}, F'), \vec{\delta}) \stackrel{\text{def}}{=} F' + F(\vec{\delta})$$

2.3 Store History

Finally, we introduce the notion of store history, denoted by η, as pairs $(\sigma, \vec{\delta})$. In our semantics each expression, instead of modifying its input store, extends the input history with new updates. With this tool we will be able, for example, to discuss commutativity of two expressions $\mathrm{EX}_1, \mathrm{EX}_2$ by analyzing the histories $(\sigma, (\vec{\delta_1}, \vec{\delta_2}))$ and $(\sigma, (\vec{\delta'_2}, \vec{\delta'_1}))$ produced by their evaluations in different orders, by proving that $\vec{\delta_1} = \vec{\delta'_1}$ and $\vec{\delta_2} = \vec{\delta'_2}$, and by finally invoking a permutation property (see Property 10) to conclude that $\mathit{apply}(\sigma, (\vec{\delta_1}, \vec{\delta_2})) = \mathit{apply}(\sigma, (\vec{\delta'_2}, \vec{\delta'_1}))$.

A store history $(\sigma, \vec{\delta})$ can be mapped to a plain store either by $\mathit{apply}(\sigma, \vec{\delta})$, as in Definition 5, or by applying $\mathit{no\text{-}delete}(\vec{\delta})$ only, which is $\vec{\delta}$ without any deletion. The latter mapping, denoted $\mathit{merge}(\sigma, \vec{\delta})$, yields a store that over-approximates $\mathit{apply}(\sigma, \vec{\delta})$, but has the advantage of always growing when $\vec{\delta}$ grows. This always-growing mapping will be crucial in order to define an approximation of the store content that is stable in presence of updates.

Definition 7 (store history application and merging).

$$\mathit{apply}((\sigma, \vec{\delta})) \stackrel{\text{def}}{=} \mathit{apply}(\sigma, \vec{\delta})$$
$$\mathit{merge}((\sigma, \vec{\delta})) \stackrel{\text{def}}{=} \mathit{apply}(\sigma, \mathit{no\text{-}delete}(\vec{\delta}))$$

where

$$\mathit{no\text{-}delete}(()) \stackrel{\text{def}}{=} ()$$
$$\mathit{no\text{-}delete}(\mathsf{insert}(\vec{n}, F), \vec{\delta}) \stackrel{\text{def}}{=} \mathsf{insert}(\vec{n}, F), \mathit{no\text{-}delete}(\vec{\delta})$$
$$\mathit{no\text{-}delete}(\mathsf{delete}(\vec{n}), \vec{\delta}) \stackrel{\text{def}}{=} \mathit{no\text{-}delete}(\vec{\delta})$$

By abuse of notation we shall:

—implicitly promote a store σ to the corresponding history $(\sigma, ())$;
—extend accessors to store histories using the convention that, for any function f defined on stores, $f(\eta) \stackrel{\text{def}}{=} f(\mathit{apply}(\eta))$, in particular, we use this convention for N_η, F_η, E_η; and
—when $\eta = (\sigma, \vec{\delta})$, we will write $\eta, \vec{\delta}'$ to denote $(\sigma, (\vec{\delta}, \vec{\delta}'))$.

Finally, we define history difference $\eta \setminus \eta'$ as follows: $(\sigma, (\vec{\delta}, \vec{\delta}')) \setminus (\sigma, \vec{\delta}) \stackrel{\text{def}}{=} \vec{\delta}'$.

We introduce here a couple of properties which will be useful later on. We first observe that the effect of an update is fully described by $\mathit{inserted}(\vec{\delta})$, $\mathit{deleted}(\vec{\delta})$ and $F(\vec{\delta})$, as follows.

Property 8 (Total effect).

$$\mathit{apply}(\sigma, \vec{\delta}) = \mathit{apply}(\sigma, \mathsf{insert}(\mathit{inserted}(\vec{\delta}), F(\vec{\delta})), \mathsf{delete}(\mathit{deleted}(\vec{\delta})))$$
$$\mathit{merge}(\sigma, \vec{\delta}) = \mathit{apply}(\sigma, \mathsf{insert}(\mathit{inserted}(\vec{\delta}), F(\vec{\delta})))$$

We say that a history is *non-rewriting* if no node n is inserted twice; when a history is non-rewriting, the order of the update records is irrelevant. As we will see, the evaluation of any expression in our language always produces a non-rewriting history.

Definition 9 ($\vec{\delta}$ is non-rewriting). A history $(\sigma, \vec{\delta})$ is *non-rewriting* iff for any $\vec{\delta}_1, \vec{m}, F, \vec{\delta}_2$ with $\vec{\delta}_1, \mathsf{insert}(\vec{m}, F), \vec{\delta}_2 = \vec{\delta}$, the set $\vec{m}$ is disjoint from $N_{\mathit{apply}(\sigma, (\vec{\delta}_1, \vec{\delta}_2))}$.

Property 10 (Permutation). If $\vec{\delta}_1$ is a permutation of $\vec{\delta}_2$, and, for any $\mathsf{insert}(\vec{m}, F)$ and $\mathsf{insert}(\vec{m}', F')$ in δ_1 with $\vec{m} \cap \vec{m}' \neq \emptyset$, the relative order of the two insertions is the same in $\vec{\delta}_1$ and $\vec{\delta}_2$, equalities (1) and (2) below hold. In particular, if either $(\sigma, \vec{\delta}_1)$ or $(\sigma, \vec{\delta}_2)$ is non-rewriting, then (1) and (2) hold.

$$\mathit{apply}(\sigma, \vec{\delta}_1) = \mathit{apply}(\sigma, \vec{\delta}_2) \tag{1}$$
$$\mathit{merge}(\sigma, \vec{\delta}_1) = \mathit{merge}(\sigma, \vec{\delta}_2) \tag{2}$$

Remark 11. We have here adopted a very liberal notion of update application, where every update sequence is well-formed, hence the same node may be inserted or deleted many times, in any order, regardless of its previous presence or absence in the store. On the contrary, the language semantics we define in Section 3 is totally standard, since it only inserts fresh nodes, and only deletes nodes that are present, hence we are not going to exploit application liberality in Section 3. However, the lack of well-formedness restrictions will be extremely handy to lighten the amount of bookkeeping that we need in the definitions and proofs of Sections 4 and 5.

3. UPDATE LANGUAGE

We consider XQueryU, a simple language based on XQuery with updates [Boag et al. 2007], and characterized by the fact that the evaluation order is fixed and each update operation is applied immediately. It is not difficult to extend our analysis to languages with snapshot semantics, but the machinery becomes heavier, while

we are trying here to present the simplest incarnation of our approach. We briefly come back to this issue at the beginning of Section 4.

3.1 Syntax

Our syntax just includes the core constructs of XQuery with minimal update features, however, we do not restrict XML navigation or construction in any essential way except for not supporting navigation through "ID" references.

Definition 12 (syntax of XQueryU). The syntax of the language that we shall study:

$$
\begin{aligned}
\text{EX} \in \mathit{Expression} ::= {} & \$x \mid \text{EX}/\text{ST} \mid () \mid (\text{EX},\text{EX}) \mid \mathit{lit} \\
& \mid \textbf{let } \$x := \text{EX } \textbf{return } \text{EX} \mid \textbf{for } \$x \textbf{ in } \text{EX } \textbf{return } \text{EX} \\
& \mid \textbf{if } (\text{EX}) \textbf{ then } \text{EX } \textbf{else } \text{EX} \mid f(\text{EX},\ldots,\text{EX}) \mid \textbf{data}(\text{EX}) \\
& \mid \textbf{element}_{cl}\ q\ \{\ \} \mid \textbf{text}_{cl}\ \{\ \text{EX}\ \} \mid \textbf{doc}(\mathit{uri}) \\
& \mid \textbf{do delete } \text{EX} \mid \textbf{do insert } \text{EX } \textbf{into } \text{EX} \\
\text{ST} \in \mathit{Step} ::= {} & \text{AX}::\text{NT} \\
\text{AX} \in \mathit{Axis} ::= {} & \textbf{child} \mid \textbf{descendant} \mid \textbf{parent} \mid \textbf{ancestor} \\
\text{NT} \in \mathit{NodeTest} ::= {} & \mathsf{text}() \mid \mathsf{node}() \mid q \mid *
\end{aligned}
$$

where $\$x$ are variables, q and f are names, *lit* are literal constants. Element creation is written **element** q { } by the programmer, and the *cl* code-location (introduced in Section 2) is inserted by the compiler, using a different *cl* for each different element constructor. Code-locations are used for path inference, as exemplified in the next section.

We use the usual abbreviations: EX/.. for EX/**parent**::∗, EX/NT for EX/**child**::NT, and EX//NT for EX/**descendant**::NT.[3]

Remark 13. We only include enough in the language to model our simplified store. We have left out several complex features, notably recursive function declarations and general FLWOR query expressions. In addition, we have simplified element creation to not take any child arguments because of the equivalence

$$
\begin{aligned}
& \textbf{element } q\ \{\text{EX}_1,\ldots,\text{EX}_n\} \\
& \quad \equiv\ \textbf{let } \$n := \textbf{element } q\ \{\} \textbf{ return } (\textbf{do insert } (\text{EX}_1,\ldots,\text{EX}_n) \textbf{ into } \$n,\ \$n)
\end{aligned}
$$

This allows for a more uniform handling of the language's effects over the store, and simplies the forthcoming formal treatment.

3.2 Semantics

We define *values*, ranged over by $\vec{v}$ and $\vec{w}$, to be sequences of nodes and constants. The main semantic judgment "$dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{v}$" specifies that the evaluation of an expression EX, with respect to a store history η_0 and to a dynamic environment *dEnv* (that associates a value to each variable free in EX), produces a value $\vec{v}$, and extends the history η_0 to $\eta_1 = \eta_0, \vec{\delta}$. The judgment defines a *relation*, because the choice of newly created nodes is non-deterministic, and because our store is not ordered. In an implementation, we would not manipulate the history η_0 but the

[3]For our use it does not matter that EX//NT usually means EX/**descendant-or-self**::node()/**child**::NT.

store $apply(\eta_0)$, since the value of every expression only depends on that. However, store histories allow us to isolate the store effect of each single expression, both in our definition of soundness and in our proof of commutativity.

We interpret steps using $[\![\text{ST}]\!]_\sigma$, that associates a step to a set of pairs, and using $[\![\text{ST}]\!]^n_\sigma$, that associates a step to all the nodes reached from n. The result is unordered, for simplicity, and because order is not tracked by the analysis we study in this paper.

Definition 14 (step semantics).

$$[\![\textbf{child}::\text{NT}]\!]_\sigma \stackrel{\text{def}}{=} \{\, (n_1, n_2) \mid n_2 \in \mathit{children}(\sigma, n_1) \wedge \mathit{node\text{-}test}(\sigma, \text{NT}, n_2) \,\}$$

$$[\![\textbf{descendant}::\text{NT}]\!]_\sigma \stackrel{\text{def}}{=} \{\, (n_1, n_2) \mid n_2 \in \mathit{desc}(\sigma, n_1) \wedge \mathit{node\text{-}test}(\sigma, \text{NT}, n_2) \,\}$$

$$[\![\textbf{parent}::\text{NT}]\!]_\sigma \stackrel{\text{def}}{=} \{\, (n_1, n_2) \mid n_2 \in \mathit{parent}(\sigma, n_1) \wedge \mathit{node\text{-}test}(\sigma, \text{NT}, n_2) \,\}$$

$$[\![\textbf{ancestor}::\text{NT}]\!]_\sigma \stackrel{\text{def}}{=} \{\, (n_1, n_2) \mid n_2 \in \mathit{ancestor}(\sigma, n_1) \wedge \mathit{node\text{-}test}(\sigma, \text{NT}, n_2) \,\}$$

$$[\![\text{ST}]\!]^n_\sigma \stackrel{\text{def}}{=} \{\, n' \mid (n, n') \in [\![\text{ST}]\!]_\sigma \,\}$$

We use the meta-variable conventions to constrain rule applicability, so if a judgment in the premise uses n in the result position, as in:

$$\mathit{dEnv} \vdash \eta_0; \text{EX} \Mapsto \eta_1; n,$$

the judgment can only be applied if EX evaluates to a value which is a single node.

We can now present the semantic rules. The rule for **do delete** EX is simple: EX is evaluated, produces a new history η_1 and a sequence of nodes $\vec{n}$, and $\mathsf{delete}(\vec{n})$ is then added to η_1. Adding $\vec{\delta}$ to the history η_1 is equivalent to applying $\vec{\delta}$ to $apply(\eta_1)$, since our rules always access $apply(\eta)$ rather than η itself.

$$\frac{\mathit{dEnv} \vdash \eta_0; \text{EX} \Mapsto \eta_1; \vec{n} \qquad \eta_2 = \eta_1, \mathsf{delete}(\vec{n})}{\mathit{dEnv} \vdash \eta_0; \textbf{do delete}\ \text{EX} \Mapsto \eta_2; ()} \qquad (\textsc{Delete})$$

do insert performs a deep copy of its arguments and creates a new element. The rule for **do insert** depends on the auxiliary function *prepare-deep-copy* mapping $(\sigma, n, \vec{m})$ to $(\vec{m}_c, F_c)$. For each node $m \in \vec{m}$, $\mathit{prepare\text{-}deep\text{-}copy}(\sigma, n, \vec{m})$ chooses a fresh node $f(m) \in (\mathcal{N} \setminus N_\sigma)$ having n as its parent. Then *prepare-deep-copy* visits $\mathit{desc}(\sigma, \vec{m})$ going top-down, and, for each $m_d \in \mathit{desc}(\sigma, \vec{m})$ chooses a node $f(m_d)$ such that, if the parent of m_d is m_p, then the parent of $f(m_d)$ is $f(m_p)$; since m_p is the parent of a node in $\mathit{desc}(\sigma, \vec{m})$, then m_p has already been visited and mapped to a fresh node. The chosen nodes are collected into $\vec{m}_c$ and, for $m_c \in \vec{m}_c$, we define $F_c(m_c) = F_\sigma(f^{-1}(m_c))$ (by construction, f is invertible over $\vec{m}_c$). Notice how the rule only depends on $apply(\eta_2)$, not on the internal structure of η_2. The function $\mathit{prepare\text{-}deep\text{-}copy}(\sigma, n, \vec{m})$ is non-deterministic, since it freely chooses the fresh nodes $\vec{m}_c$; we formalize this non-determinism by stating that $\mathit{prepare\text{-}deep\text{-}copy}(\sigma, n, \vec{m})$ returns the set of all $(\vec{m}'_c, F'_c)$ pairs that respect the description above, and the evaluation of **do insert** EX_1 **into** EX_2 picks one of those pairs.

$$\frac{\begin{array}{c} dEnv \vdash \eta_0; \text{EX}_1 \Rightarrow \eta_1; \vec{m} \\ dEnv \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; n \\ (\vec{m}_c, F_c) \in \mathit{prepare\text{-}deep\text{-}copy}(\mathit{apply}(\eta_2), n, \vec{m}) \\ \eta_3 = \eta_2, \mathsf{insert}(\vec{m}_c, F_c) \end{array}}{dEnv \vdash \eta_0; \textbf{do insert } \text{EX}_1 \textbf{ into } \text{EX}_2 \Rightarrow \eta_3; ()} \quad \text{(Insert)}$$

Element creation just creates a new empty element, whose content will be typically initialized by an insert operation. The new element is linked to cl, only for the purposes of our analysis. The function $\mathit{fresh}(N_\sigma, cl)$ returns all root nodes in $\mathcal{N} \setminus N_\sigma$ such that $\mathcal{R}(n) = cl$.

$$\frac{\begin{array}{c} n \in \mathit{fresh}(N_{\mathit{apply}(\eta_0)}, cl) \\ \eta_1 = \eta_0, \mathsf{insert}(n, (n \mapsto [\mathsf{kind} : \mathit{element}, \mathsf{name} : q, \mathsf{value} : \bot])) \end{array}}{dEnv \vdash \eta_0; \textbf{element}_{cl}\ q\ \{\ \} \Rightarrow \eta_1; n} \quad \text{(Elt)}$$

Text creation is similar but inserts the text value.

$$\frac{\begin{array}{c} dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \mathit{lit} \\ n \in \mathit{fresh}(N_{\mathit{apply}(\eta_0)}, cl) \\ \eta_2 = \eta_1, \mathsf{insert}(n, (n \mapsto [\mathsf{kind} : \mathit{text}, \mathsf{name} : \bot, \mathsf{value} : \mathit{lit}])) \end{array}}{dEnv \vdash \eta_0; \textbf{text}_{cl}\ \{\ \text{EX}\ \} \Rightarrow \eta_2; n} \quad \text{(Text)}$$

(Insert), (Elt) and (Text) are the only rules that use the insert() primitive, and they only insert fresh nodes, hence the following property holds.

Property 15 (non-rewriting). For any σ, $dEnv$, EX, $\vec{v}$, η, $\vec{\delta}$ such that $dEnv \vdash \eta; \text{EX} \Rightarrow (\eta, \vec{\delta}); \vec{v}$, if η is non-rewriting, then $\eta, \vec{\delta}$ is non-rewriting.

The other language construct rules are standard, apart from the fact that an evaluation order is always specified through the way the store history is passed around, *e.g.*, the (Comma) rule specifies that EX_2 is evaluated in the store modified by, *i.e.*, after, EX_1.

$$\frac{dEnv \vdash \eta_0; \text{EX}_1 \Rightarrow \eta_1; \vec{v}_1 \qquad dEnv \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; \vec{v}_2}{dEnv \vdash \eta_0; (\text{EX}_1, \text{EX}_2) \Rightarrow \eta_2; \vec{v}_1, \vec{v}_2} \quad \text{(Comma)}$$

$$\frac{}{dEnv \vdash \eta_0; () \Rightarrow \eta_0; ()} \quad \text{(Empty)}$$

$$\frac{dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{n}_1 \qquad \vec{n}_2 = \bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{\mathit{apply}(\eta_1)}}{dEnv \vdash \eta_0; \text{EX/ST} \Rightarrow \eta_1; \vec{n}_2} \quad \text{(Step)}$$

$$\frac{}{dEnv \vdash \eta_0; \mathit{lit} \Rightarrow \eta_0; \mathit{lit}} \quad \text{(Literal)}$$

$$\frac{dEnv \vdash \eta_0; \text{EX}_1 \Rightarrow \eta_1; \vec{v}_1 \qquad (dEnv + \$x \mapsto \vec{v}_1) \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; \vec{v}_2}{dEnv \vdash \eta_0; \textbf{let } \$x := \text{EX}_1 \textbf{ return } \text{EX}_2 \Rightarrow \eta_2; \vec{v}_2} \quad \text{(Let)}$$

$$\frac{\begin{array}{c} dEnv \vdash \eta_0; \text{EX}_1 \Rightarrow \eta_1; \vec{v} \\ \forall i \in 1..|\vec{v}|\colon (dEnv + \$x \mapsto v_i) \vdash \eta_i; \text{EX}_2 \Rightarrow \eta_{i+1}; \vec{v}'_i \end{array}}{dEnv \vdash \eta_0; \textbf{for } \$x \textbf{ in } \text{EX}_1 \textbf{ return } \text{EX}_2 \Rightarrow \eta_{|\vec{v}|+1}; (\vec{v}'_1, \ldots, \vec{v}'_{|\vec{v}|})} \quad (\textsc{For})$$

$$\frac{dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{v} \quad \vec{v} \neq () \qquad dEnv \vdash \eta_1; \text{EX}_1 \Rightarrow \eta_2; \vec{v}_1}{dEnv \vdash \eta_0; \textbf{if } (\text{EX}) \textbf{ then } \text{EX}_1 \textbf{ else } \text{EX}_2 \Rightarrow \eta_2; \vec{v}_1} \quad (\textsc{IfT})$$

$$\frac{dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; () \qquad dEnv \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; \vec{v}_2}{dEnv \vdash \eta_0; \textbf{if } (\text{EX}) \textbf{ then } \text{EX}_1 \textbf{ else } \text{EX}_2 \Rightarrow \eta_2; \vec{v}_2} \quad (\textsc{IfF})$$

For the built-in function rule, the helper judgment $f(\vec{v}_1; \ldots; \vec{v}_k) \Rightarrow \vec{v}$ is assumed defined for every built-in function f. For the equality function written $\text{EX}_1 = \text{EX}_2$, for example, we have that $(\vec{v}_1 = \vec{v}_2) \Rightarrow \vec{v}$ defines $\vec{v}$ as the literal value 1 for true if the two sequences contain identical elements and () for false otherwise.

$$\frac{\forall i \in 1..k\colon dEnv \vdash \eta_{i-1}; \text{EX}_i \Rightarrow \eta_i; \vec{v}_i \qquad f(\vec{v}_1; \ldots; \vec{v}_k) \Rightarrow \vec{v}}{dEnv \vdash \eta_0; f(\text{EX}_1, \ldots, \text{EX}_k) \Rightarrow \eta_k; \vec{v}} \quad (\textsc{Func})$$

As specified, built-in functions cannot access the store, hence they are restricted to simple functions like equality, arithmetic operators, string operators, *etc.*, which is all we shall need in our examples. In the next section, we discuss the possible addition of built-in functions that access the store (Remark 36).

The **data** accessor merely extracts the literal value from a text node.

$$\frac{dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; n \qquad lit = F_{apply(\eta_1)}(n).\mathsf{value}}{dEnv \vdash \eta_0; \textbf{data}(\text{EX}) \Rightarrow \eta_1; lit} \quad (\textsc{Data})$$

Finally, **doc**(*uri*) returns the nodes that are mapped to *uri* and have not been deleted yet. Similarly to XQuery, we assume that these nodes were already in the store when the computation began, but the definition of a **put**(*uri*) function to create a new document could be easily added, and would be defined almost exactly as the *element* operation.

$$\frac{\vec{n} = \{\, n \mid R_{apply(\eta_0)}(n) = uri \,\}}{dEnv \vdash \eta_0; \textbf{doc}(uri) \Rightarrow \eta_0; \vec{n}} \quad (\textsc{Doc})$$

4. PATH ANALYSIS

In this section we present the path analysis, which computes upper bounds for the nodes accessed and updated by a given expression in XQueryU.

4.1 Paths

We first define the notion of path used by the analysis, which is not the same as paths in the target language. For example, they are always rooted at a given location, and the steps need not coincide: if we added order to the store, we could add a following-sibling axis to the language, but we could approximate it with

parent::∗/**child**::NT in the analysis. The fragment was chosen such that important operations, notably intersection emptiness, can be checked using known algorithms [Benedikt et al. 2005; Miklau and Suciu 2004; Hammerschmidt et al. 2005]. For the XPath fragment we consider here, [Hammerschmidt et al. 2005] describes a polynomial time algorithm based on tree automata.

Definition 16 (static paths). *Static paths*, or simply *paths*, are defined as follows.

$$p ::= () \mid \ell \mid p|p \mid p/\text{ST}$$

where ℓ ranges over locations, and ST is from the language grammar (Definition 12).

The / operator binds tighter than |, so that $uri_1|uri_2/b$ means $uri_1|(uri_2/b)$ rather than $(uri_1|uri_2)/b$; we will use parenthesis either to force a different precedence, or just for clarity. As in the language, ST ranges over steps AX::NT.

Since static paths are rooted, their extensional semantics $[\![p]\!]_\sigma$ maps them to sets of nodes, rather than sets of pairs.

Definition 17 (extensional semantics of paths). For a path p and store σ, $[\![p]\!]_\sigma$ denotes the set of nodes selected from the store by the path, according to the traditional semantics [Wadler 1999], except that order is ignored, and R_σ is used to interpret the locations ℓ; this definition is based on the step semantics of Definition 14.

$$\begin{aligned}
[\![()]\!]_\sigma &\stackrel{\text{def}}{=} \emptyset \\
[\![\ell]\!]_\sigma &\stackrel{\text{def}}{=} \{\, n \mid R_\sigma(n) = \ell \,\} \\
[\![p|p']\!]_\sigma &\stackrel{\text{def}}{=} [\![p]\!]_\sigma \cup [\![p']\!]_\sigma \\
[\![p/\text{ST}]\!]_\sigma &\stackrel{\text{def}}{=} \bigcup_{n \in [\![p]\!]_\sigma} [\![\text{ST}]\!]^n_\sigma
\end{aligned}$$

The following concepts are derived from the extensional semantics.

Definition 18 (extensional relations between paths).

Inclusion. A path p_1 is included in p_2, denoted $p_1 \subseteq p_2$, iff $\forall\sigma\colon [\![p_1]\!]_\sigma \subseteq [\![p_2]\!]_\sigma$.

Disjointness. Two paths p_1, p_2 are disjoint, denoted $p_1 \# p_2$, iff $\forall\sigma\colon [\![p_1]\!]_\sigma \cap [\![p_2]\!]_\sigma = \emptyset$.

While equivalence and disjointness only depend on the extensional semantics of a path, we also need to deal with the following notions, which are *intensional*, meaning that they discriminate paths that are extensionally equivalent. Intuitively, the prefixes of a path p, denoted as $\mathit{pref}(p)$, are the stepping stones that p jumps through before reaching its destination (which is also inside $\mathit{pref}(p)$). The non-interference relation $u \#^\cup p$ specifies that u does not end in any of these stepping stones.

Definition 19 (intensional notions: $\mathit{pref}(p)$, $[\![p]\!]^\cup_\sigma$, $u \#^\cup p$).

$$\begin{aligned}
\mathit{pref}(()) &\stackrel{\text{def}}{=} \emptyset \\
\mathit{pref}(\ell) &\stackrel{\text{def}}{=} \{\ell\} \\
\mathit{pref}(p/\text{ST}) &\stackrel{\text{def}}{=} \{p/\text{ST}\} \cup \mathit{pref}(p)
\end{aligned}$$

$$pref(p|p') \stackrel{\text{def}}{=} pref(p) \cup pref(p')$$

$$[\![p]\!]^{\cup}_{\sigma} = \bigcup_{p' \in pref(p)} [\![p']\!]_{\sigma}$$

$$u \#^{\cup} p \Leftrightarrow \forall p' \in pref(p) \colon u \# p'$$

4.2 The Meaning of the Analysis

Definition 20 (path analysis). Given an expression EX and a path environment *pEnv*, which is a mapping from variables to paths, our path-analysis judgment

$$pEnv \vdash \text{EX} \Rightarrow r; \langle a, u \rangle$$

associates three paths to the expression: r is an upper approximation of the nodes that are *returned* by the evaluation of EX, a of the nodes that are *accessed*, and u of the nodes that are *updated*.

The paths a and u will be used to check commutativity of expressions; the analysis of an expression that updates, or accesses, the results of a subexpression EX′, uses the path r associated to EX′ to compute u and a.

Clearly, the set of nodes being "returned" by an expression must correspond to the result of the query. However, as was observed by Marian and Siméon [2003], there are several reasonable interpretations to the set of nodes being "accessed" by a query. For example, a path $x//a may be considered as accessing only the descendants with an a name, or the descendants with an a name along with all their ancestors. Furthermore, as we have seen in the introduction, there are some interactions between the interpretation chosen for the "accessed" nodes and the interpretation chosen for the "updated" nodes. Our specific choice is based on the following requirements:

—the definition should be as natural as possible,

—it should allow for an easy computation of a static approximation, and

—it should satisfy the following property: if what is accessed by EX_1 is disjoint from what is updated by EX_2, and vice-versa, then the two expressions commute.

In the following paragraphs we present our interpretation, which will guide the definition of the inference rules and is one of the basic technical contributions of this work.

The meaning of r seems the easiest to describe: it is an upper approximation of the result, hence an analysis is sound if $() \vdash \text{EX} \Rightarrow r; \langle a, u \rangle$ and $() \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{v}$ imply that $\vec{v} \subseteq [\![r]\!]_{apply(\eta_1)}$. Unfortunately, this is too simplistic. Consider the following example:

let $x := **doc**('u1')/a **return** (**do delete**($x), $x/b)

Our rules bind a path u1/a to $x, and finally deduce a returned path $p = \text{u1/a/b}$ for the expression above. However, after **do delete**($x), the value of $x/b is not in $[\![p]\!]_{apply(\eta)}$ anymore; the best we can say is that it was in $[\![p]\!]_{apply(\eta')}$ for some past η', which we express by saying that it is in $[\![p]\!]_{merge(\eta)}$. This is just an instance of a general "stability" problem: we infer something about a specific store history, but

we need the same property to hold for the store in some future. We solve this problem by accepting that our analysis only satisfies $\vec{v} \subseteq [\![r]\!]_{merge(\eta_1)}$, which is weaker than $\vec{v} \subseteq [\![r]\!]_{apply(\eta_1)}$ but is stable; we also generalize the notion to environments.

Definition 21 (approximation). A path p approximates a value $\vec{v}$ in the store history η, denoted $p \supseteq_\eta \vec{v}$, iff $\vec{v} \subseteq [\![p]\!]_{merge(\eta)}$. We generalize this to say that a path environment *pEnv* approximates a dynamic environment *dEnv* in a store history η, denoted $pEnv \supseteq_\eta dEnv$, iff

$$(\$x \mapsto \vec{v}) \in dEnv \quad \Rightarrow \quad \exists p.\ (\$x \mapsto p) \in pEnv \ \text{ and } \ p \supseteq_\eta \vec{v}$$

Thanks to this "merge" interpretation, a path also includes all nodes that were reached by the path in some past version of the current history.

Remark 22 *(on move and rename).* This approach works for insertions, deletions, and for a replace operation, although we did not present it here. It is not obvious, however, whether it could be extended to other update operations, notably to *move* nodes without changing their identity, or to *rename* nodes. The use of *rename* interferes directly with the path analysis, since paths use names to identify nodes, hence renaming would invalidate some of the paths collected by the analysis. The case of *move* would also be difficult, for two reasons. Firstly, our current store model links the parent-child relationship to the node identity; this problem could be easily solved, by adopting a different store model. Secondly, consider an operation that moves a node n from one parent to another, represented by a deletion followed by an insertion of the *same* node. As a consequence, *merge*(η) may contain nodes with two parents. Hence, one could not deduce, for example, that $(\mathsf{a/d}) \# (\mathsf{b/c/d})$, because $\$x/\mathsf{a/d}$ and $\$x/\mathsf{b/c/d}$ may denote the same node in *merge*(η). This reflects the fact that the two paths, if evaluated at different times, may actually return the same node, because its parent was moved from $\$x/\mathsf{a}$ to $\$x/\mathsf{b/c}$ in the meanwhile.

We move now to the formal interpretation of a and u. The intuition here is that commutativity (to be presented as Theorem 44) is captured by the following idea: assume that EX_1 transforms η_0 into $(\eta_0, \vec{\delta})$ and only modifies nodes reachable through a path u, while EX_2 only depends on nodes reachable through a. Assume now that $u \#^\cup a$; because EX_1 only modifies nodes in u, the histories η_0 and $(\eta_0, \vec{\delta})$ are "the same" with respect to an expression that only accesses a, hence we may evaluate EX_2 either before or after EX_1.

This is formalized by defining a notion of history equivalence with respect to a path $\eta \sim_p \eta'$, and by proving that the a and u that are inferred for an expression EX and the result of evaluation are related by the following soundness properties: *equal results* and *immutability*.

Assume that $pEnv \vdash \text{EX} \Mapsto r; \langle a, u\rangle$, $dEnv \vdash \eta_0; \text{EX} \Mapsto (\eta_0, \vec{\delta}); \vec{v}$, and $pEnv \supseteq_\eta dEnv$. We define the following properties:

Equal results from a-equivalent stores (first version):.
$\eta_0' \sim_a \eta_0$ implies $dEnv \vdash \eta_0'; \text{EX} \Mapsto (\eta_0', \vec{\delta}); \vec{v}$ *i.e.*, the same $\vec{v}$ and $\vec{\delta}$ are produced when starting from η_0 or from η_0'.

Immutability out of u. For any path c, $u \#^\cup c$ implies $\eta_0 \sim_c (\eta_0, \vec{\delta})$ *i.e.*, for any c, if $u \#^\cup c$, then the store after EX is equivalent, *wrt.* the path c, to the store

before EX.

It is worth noting that the two properties depend, respectively, on a and u only, while r is used during the analysis to compute both u and a.

To help explain our notion of path equivalence, let us consider the path analysis rule for step traversal (introduced formally later in this section).

$$\frac{pEnv \vdash \text{EX} \Rrightarrow r; \langle a, u\rangle}{pEnv \vdash \text{EX/ST} \Rrightarrow r/\text{ST}; \langle (r/\text{ST})|a, u\rangle} \quad (\textsc{Step})$$

By *equal results*, the rule says that if $\eta_0' \sim_{(r/\text{ST})|a} \eta_0$ then the evaluation of EX/ST gives the same result in both η_0 and η_0'. We expect to prove this by induction, hence we need to prove that, during the parallel evaluation of EX/ST starting from η_0 and η_0':

(1) The two evaluations of EX start from two stores that are equivalent with respect to a, so that they produce the same set of nodes; for this aim, we need that $\eta_0' \sim_{(r/\text{ST})|a} \eta_0$ implies $\eta_0' \sim_a \eta_0$. By induction, we can now prove that $dEnv \vdash \eta_0; \text{EX} \Rrightarrow (\eta_0, \vec{\delta}); \vec{v}$, implies $dEnv \vdash \eta_0'; \text{EX} \Rrightarrow (\eta_0', \vec{\delta}); \vec{v}$, with the same $\vec{v}$ and $\vec{\delta}$.

(2) We now evaluate $[\![\text{ST}]\!]^n_{apply(\eta_0,\vec{\delta})}$ and $[\![\text{ST}]\!]^n_{apply(\eta_0',\vec{\delta})}$, for each $n \in \vec{v}$, and we need to prove that the same nodes are reached in the two cases, hence we need to prove that $\eta_0' \sim_{(r/\text{ST})|a} \eta_0$ implies $\eta_0', \vec{\delta} \sim_{r/\text{ST}} \eta_0, \vec{\delta}$. Hence, we need a notion of equivalence that is stable with respect to possible updates that take place during the evaluation of subexpressions; this is required for the soundness of essentially every operator. For the step operator, we also need to have that, for each $n \in \vec{v}$, $\eta_1 \sim_{r/\text{ST}} \eta_1'$ implies $[\![\text{ST}]\!]^n_{apply(\eta_1)} = [\![\text{ST}]\!]^n_{apply(\eta_1')}$, where $\eta_1 = \eta_0, \vec{\delta}$ and $\eta_1' = \eta_0', \vec{\delta}$. For this reason, our notion of equivalence with respect to a path r/ST, is based on observing the behaviour of each step in the path.

(3) As we have seen that our *approximation* property only guarantees that the $\vec{v}$ resulting from the evaluation of EX satisfies $\vec{v} \subseteq [\![r]\!]_{merge(\eta_1)}$, hence our equivalence must ensure us that $[\![\text{ST}]\!]^n_{apply(\eta_1)} = [\![\text{ST}]\!]^n_{apply(\eta_1')}$ for each $n \in merge(\eta_1)$, not just for the nodes in $apply(\eta_1)$.

As a last detail, $\eta \sim_p \eta'$ will also imply that nodes in $[\![p]\!]$ have the same description F in η and in η'.

We now formalize this solution in two steps: we first define a notion of "immediate equivalence" of η_1 and η_2 with respect to a path p, that specifies that η_1 and η_2 are behaviourally equivalent wrt p, both in their *apply*() and in their *merge*() interpretation. We then make the notion "stable" by quantifying over all possible updates.

Definition 23 (immediate history path-equivalence). Immediate history equivalence of η_1 and η_2 with respect to a path, denoted $\eta_1 \sim_p^{\emptyset} \eta_2$, is defined by induction

and by cases on p as follows:

$$\begin{aligned}
\eta_1 \sim^{\emptyset}_{()} \eta_2 &\stackrel{\text{def}}{\Leftrightarrow} \text{always}\\
\eta_1 \sim^{\emptyset}_{\ell} \eta_2 &\stackrel{\text{def}}{\Leftrightarrow} [\![\ell]\!]_{merge(\eta_1)} = [\![\ell]\!]_{merge(\eta_2)}\\
&\quad \wedge\, \forall n \in [\![\ell]\!]_{merge(\eta_1)} :\ F_{merge(\eta_1)}(n) = F_{merge(\eta_2)}(n)\\
\eta_1 \sim^{\emptyset}_{p|p'} \eta_2 &\stackrel{\text{def}}{\Leftrightarrow} \eta_1 \sim^{\emptyset}_{p} \eta_2 \ \wedge\ \eta_1 \sim^{\emptyset}_{p'} \eta_2\\
\eta_1 \sim^{\emptyset}_{p/\mathrm{ST}} \eta_2 &\stackrel{\text{def}}{\Leftrightarrow} \eta_1 \sim^{\emptyset}_{p} \eta_2 \ \wedge\ \forall n \in (\!|p|\!)_{merge(\eta_1)} \cap [\![p]\!]_{merge(\eta_2)}) :\\
&\quad (\!|\mathrm{ST}|\!)^n_{apply(\eta_1)} = [\![\mathrm{ST}]\!]^n_{apply(\eta_2)} \ \wedge\ [\![\mathrm{ST}]\!]^n_{merge(\eta_1)} = [\![\mathrm{ST}]\!]^n_{merge(\eta_2)}\\
&\quad \wedge\, \forall m \in [\![\mathrm{ST}]\!]^n_{merge(\eta_1)} :\ F_{merge(\eta_1)}(m) = F_{merge(\eta_2)}(m))
\end{aligned}$$

In the last case $\eta_1 \sim^{\emptyset}_{p} \eta_2$, n ranges over $[\![p]\!]_{merge(\eta_1)} \cap [\![p]\!]_{merge(\eta_2)}$. The next lemma shows that this is equivalent to $n \in (\!|p|\!)_{merge(\eta_1)} \cup [\![p]\!]_{merge(\eta_2)})$.

Lemma 24 (equal semantics).

$$\begin{aligned}
\eta_1 \sim^{\emptyset}_{p} \eta_2 \ \Rightarrow\ & (\!|p|\!)_{merge(\eta_1)} = [\![p]\!]_{merge(\eta_2)} \wedge [\![p]\!]_{apply(\eta_1)} = [\![p]\!]_{apply(\eta_2)}\\
& \wedge \forall n \in [\![p]\!]^{\cup}_{merge(\eta_1)} :\ F_{merge(\eta_1)}(n) = F_{merge(\eta_2)}(n))
\end{aligned}$$

Lemma 25 ($\sim^{\emptyset}_{p}$ equivalence). *For any path p, the relation $\sim^{\emptyset}_{p}$ is reflexive, symmetric, and transitive.*

Equivalence is not extensional, but it only depends on the prefixes of p, as formalized by the following lemma.

Lemma 26 ($\sim^{\emptyset}_{p}$ and $pref(p)$). $\eta \sim^{\emptyset}_{p} \eta' \Leftrightarrow \forall p' \in pref(p) : \eta \sim^{\emptyset}_{p'} \eta'$.

Proof. By induction on p and by cases.

Cases () and ℓ are immediate.

Case $p = p_1|p_2$: $\eta \sim^{\emptyset}_{p_1|p_2} \eta' \Leftrightarrow \eta \sim^{\emptyset}_{p_1|p_2} \eta' \ \wedge\ \eta \sim^{\emptyset}_{p_1} \eta' \ \wedge\ \eta \sim^{\emptyset}_{p_2} \eta'$
$\Leftrightarrow$(by induction) $\eta \sim^{\emptyset}_{p_1|p_2} \eta' \wedge (\forall p' \in pref(p_1) : \eta \sim^{\emptyset}_{p'} \eta') \wedge (\forall p' \in pref(p_2) : \eta \sim^{\emptyset}_{p'} \eta')$
$\Leftrightarrow \forall p' \in pref(p_1|p_2) : \eta \sim^{\emptyset}_{p'} \eta'$

Case $p = p_1/ST$: The $\Leftarrow$ direction is trivial, since $(p_1/ST) \in pref(p_1/ST)$. In the other direction, $\eta \sim^{\emptyset}_{p_1/ST} \eta'$ implies $\eta \sim^{\emptyset}_{p_1} \eta'$, hence, by induction, $\forall p' \in pref(p_1) : \eta \sim^{\emptyset}_{p'} \eta'$. □

Corollary 27 ($\sim^{\emptyset}_{p}$ equivalence). *If $pref(p) = pref(p')$ then $\eta \sim^{\emptyset}_{p} \eta' \Leftrightarrow \eta \sim^{\emptyset}_{p'} \eta'$.*

We finally close the definition for all possible updates, to make it "stable".

Definition 28 (History equivalence with respect to a path). Two histories η_1 and η_2 are *equivalent with respect to a path p*, denoted $\eta_1 \sim_p \eta_2$, iff:

$$\forall \vec{\delta}.\ (\eta_1, \vec{\delta}) \sim^{\emptyset}_{p} (\eta_2, \vec{\delta})$$

Lemma 29 ($\sim_{p}$ equivalence). *For any path p, the relation $\sim_p$ is reflexive, symmetric, and transitive.*

We are now ready for the formal definition of soundness.

Definition 30 (soundness). The static analysis $pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$ is *sound* for the semantic evaluation $dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{v}$ iff for any well-formed η_0, η_1, $dEnv$, $pEnv$, EX, $\vec{v}$, r, a, u, such that:

$$pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$$
$$dEnv \vdash \eta_0; \text{EX} \Rightarrow (\eta_0, \vec{\delta}); \vec{v}$$
$$pEnv \supseteq_{\eta_0} dEnv$$

the following properties hold.

Approximation by r. r is an approximation of the result: $r \supseteq_{(\eta_0, \vec{\delta})} \vec{v}$.

Equal results from a-equivalent stores. For any store history η_0', if $\eta_0' \sim_a \eta_0$ and $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta})$, then $dEnv \vdash \eta_0'; \text{EX} \Rightarrow (\eta_0', \vec{\delta}); \vec{v}$.

Immutability out of u. $\forall c\colon u \#^{\cup} c \Rightarrow \eta_0 \sim_c (\eta_0, \vec{\delta})$.

In the *equal results* property we have to add the $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta})$ assumption because $\vec{\delta}$ creates some nodes that are disjoint from N_{η_0}, and they have also to be disjoint from $N_{\eta_0'}$. This assumption is not really restrictive.

Immutability, together with *equal results*, implies that, for any EX_1 that only accesses an a_1 such that $u \#^{\cup} a_1$, the value returned by EX_1 after EX has been evaluated is the same value returned by EX_1 before EX evaluation. *Immutability* is hence the key to prove that an update and a query commute if $u \#^{\cup} a_1$. We must consider non interference $u \#^{\cup} a_1$, rather than simple disjiontness $u \# a_1$, because the disruption of any path in $\mathit{pref}(a_1)$ may affect the result of an expression that accesses a_1.

For example, $\mathsf{delete}(/\mathsf{a}/\mathsf{b})$ updates a path $u = /\mathsf{a}/\mathsf{b}|/\mathsf{a}/\mathsf{b}//*$, and affects the semantics of a path $/\mathsf{a}/\mathsf{b}/..$ (where .. abbreviates **parent**::$*$). However, we have $(/\mathsf{a}/\mathsf{b}|/\mathsf{a}/\mathsf{b}//*) \# (/\mathsf{a}/\mathsf{b}/..)$, because the first path reaches nodes which are one level deeper than the nodes reached by $/\mathsf{a}/\mathsf{b}/...$ On the other side, $(/\mathsf{a}/\mathsf{b}|/\mathsf{a}/\mathsf{b}//*) \#^{\cup} (/\mathsf{a}/\mathsf{b}/..)$ is false, because $/\mathsf{a}/\mathsf{b}$ belongs to $\mathit{pref}(/\mathsf{a}/\mathsf{b}/..)$, and this captures the fact that deleting nodes reached by $/\mathsf{a}/\mathsf{b}$ interferes with the evaluation of $/\mathsf{a}/\mathsf{b}/...$

Remark 31. The *approximation* and *immutability* properties only depend on $[\![r]\!]$ and $[\![u]\!]$ respectively, hence r and u are extensional: if $pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$ is sound, r' is extensionally equivalent to r, and u' is extensionally equivalent to u, then $pEnv \vdash \text{EX} \Rightarrow r'; \langle a, u'\rangle$ is sound as well. However, *equal results* depends on $\sim_a$, hence a is not extensional: even if a' is extensionally equivalent to a, soundness of $pEnv \vdash \text{EX} \Rightarrow r; \langle a', u\rangle$ does not follow from soundness of $pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$.

As a consequence, a path-inference engine is free to substitute, at any time, any path r or u, say $//*/\mathsf{b} \mid //*$, with any equivalent path whose manipulation may be more efficient, say $//*$. The a component can be substituted with an a' only if they define the same equivalence. This happens, for example, when they have the same prefixes (Corollary 27); hence, the a-path $/\mathsf{a}|/\mathsf{a}/\mathsf{b}|/\mathsf{a}/\mathsf{b}/\mathsf{c}$, that results from following a path $/\mathsf{a}/\mathsf{b}/\mathsf{c}$, can be safely substituted by the a-path $/\mathsf{a}/\mathsf{b}/\mathsf{c}$. Lemma 32 below describes another situation when the substitution of a is safe. Note that this remark can be used to help optimize implementations of the path analysis.

The following lemma shows that, in the *accessed* component, a path $\mathsf{p}/*|\mathsf{p}/\mathsf{a}$ can be safely substituted by a simpler path $\mathsf{p}/*$, or we may simplify a path $\mathsf{p}//*|\mathsf{p}/*$ to

$\mathsf{p}//*$. The same substitutions are safe in the *returned* and *updated* components, but in those cases this is simpler to prove, just by observing that such pairs of paths are extensionally equivalent. The second lemma proves that, when a_1 can be simplified to a_2, then $\mathsf{a}_1|\mathsf{a}$ can be simplified to $\mathsf{a}_2|\mathsf{a}$ as well.

Lemma 32. *Let us write* $\text{ST} \subseteq \text{ST}'$ *iff, for any store* σ *and node* n, $[\![\text{ST}]\!]^n_\sigma \subseteq [\![\text{ST}']\!]^n_\sigma$. *If* $\text{ST} \subseteq \text{ST}'$, *then*

$$\eta \sim_{p/\text{ST}|p/\text{ST}'} \eta' \Leftrightarrow \eta \sim_{p/\text{ST}'} \eta'$$

Lemma 33 (Union). *For any* η, η', p, p':

$$\eta \sim_{p|p'} \eta' \Leftrightarrow (\eta \sim_p \eta' \wedge \eta \sim_{p'} \eta')$$

Remark 34 *(snapshot semantics).* Before we proceed, we say a word about dealing with snapshot semantics in the context of this framework. Fundamentally, adding support for delayed update application does not add any significant issue, although it complicates the formal treatment. In essence, the following adjustments need to be made. (i) At the language level, consider atomic updates as contributing to a *pending update list* in the style of the XQuery Update Facility [Chamberlin et al. 2007], and add a new operator that applies the pending update list to the store, in the style of XQuery! [Ghelli et al. 2006]. (ii) The path analysis judgment must be extended in the form of $pEnv \vdash \text{EX} \Rightarrow r; \langle a, p, u\rangle$, where p is the path approximating the nodes that will be modified by the updates in the pending update list. (iii) The path analysis must be fixed so that the atomic updates contribute their updated paths to the pending updates paths, while the apply expression moves the pending updates paths into the updated paths. Hence, the approach does not fundamentally change, apart from this additional book-keeping. A complete description of those changes is beyond the space allowed by this paper.

4.3 Framework Properties

Our approach is compatible with many different equivalence relations, provided that the relation and the rules are mutually compatible, according to definition 30; you find a very different relation defined in the preliminary version of this paper [Ghelli et al. 2007]. While the exact details of our definitions of equivalence, approximation, and non-interference, are crucial for the soundness of the update rules, the soundness of our analysis of the functional constructs, such as *let*, *for*, *comma*, only depends on the six sanity properties listed in Lemma 35 below, and the same holds for the proof of the commutativity theorem. Hence, these "framework" properties specify the high-level requirements for our notions of equivalence, approximation, and non-interference, which fix the boundaries for our quest for the "best" definitions. All these properties hold trivially for the equivalence that we have chosen.

Lemma 35 (framework properties).

$$(p_1 \supseteq_\eta \vec{v}_1) \wedge (p_2 \supseteq_\eta \vec{v}_2) \Rightarrow (p_1|p_2) \supseteq_\eta (\vec{v}_1, \vec{v}_2) \qquad (\supseteq |)$$

$$p \supseteq_\eta \vec{v} \Rightarrow p \supseteq_{\eta,\vec{\delta}} \vec{v} \qquad (\supseteq \text{ Stability})$$

$$\textit{for each } p,\ \sim_p \textit{ is an equivalence relation} \qquad (\sim \text{ Equivalence})$$

$$\eta_0 \sim_p \eta_1 \Rightarrow \eta_0, \vec{\delta} \sim_p \eta_1, \vec{\delta} \qquad (\sim \text{ Stability})$$

$$\eta_0 \sim_{(q_0|q_1)} \eta_1 \Rightarrow \eta_0 \sim_{q_0} \eta_1 \qquad (\sim |)$$

$$p \#^{\cup} (q_0|q_1) \Leftrightarrow p \#^{\cup} q_0 \wedge p \#^{\cup} q_1 \qquad (\#^{\cup} |)$$

4.4 Path Analysis Rules

Our path inference rules are presented in Figure 2 and explained below in two groups: update rules and selection rules.

Update rules. The (DELETE) rule extends the "updated path" u with all the descendants of r because u approximates those paths whose semantics may change after the expression is evaluated, and the semantics of each path that reaches $r|r//*$ is affected by the deletion.

Assume, for example, that $(\$x \twoheadrightarrow \ell) \in pEnv$, $(\$x \twoheadrightarrow n) \in dEnv$, and n is the root of a tree of the form $\langle\mathsf{a}\rangle\langle\mathsf{b}\rangle\langle\mathsf{c}/\rangle\langle\mathsf{b}/\rangle\langle\mathsf{a}/\rangle$. The evaluation of delete $\$x/\mathsf{b}$ would change the semantics of $\$x//\mathsf{c}$, although this path does not explicitly traverse ℓ/b. This is correctly dealt with, since the presence of $\ell/\mathsf{b}|\ell/\mathsf{b}//*$ in u means: every path that is not disjoint from $\ell/\mathsf{b}|\ell/\mathsf{b}//*$ may be affected by this operation, and, by Definition 18, $\ell//\mathsf{c}$ is *not* disjoint from $\ell/\mathsf{b}|\ell/\mathsf{b}//*$.

Similarly, the (INSERT) rule specifies how **do insert** EX$_1$ **into** EX$_2$ may modify the semantics of every path that steps through the descendants of EX$_2$. Moreover, it depends on all the descendants of EX$_1$, since it copies all of them. In principle, we shoud add $r_1|r_1//*$ to the set of accessed paths, since EX$_1$ is also accessed. However, $pref(r_1|r_1//*) = pref(r_1//*)$, hence we exploit the observation in Remark 31, and use $r_1//*$. In Section 5, the soundness of this choice is proved formally.

Selection rules. These rules analyse the querying fragment of our language by extending the rules from Marian and Siméon [2003] for the proper handling of updated paths.

The (COMMA) rule accumulates the accessed paths and the updated nodes from its arguments. Concatenation in the result is mapped to union, since our analysis does not keep the order, or multiplicity, of nodes into account.

(LITERAL) just specifies that a literal expression does not access or modify any path.

The (VAR) rule specifies that variable access does not access the store. The only subtlety regarding that rule is that r is not regarded as "accessed". This is consistent with both the dynamic semantics for variable references, and with the definition of soundness. More precisely, the value of *$x* is the same in two stores η_0 and η_0' independently of any equivalence among them, hence the accessed path can safely be left empty. This rule also implicitly specifies that variable access commutes with any other expression. For example, "*$x*, **delete** *$x*" is equivalent to "**delete** *$x*, *$x*".

The (STEP) rule has already been presented, and specifies that a step depends on r/ST, which means that it depends on r, and it also relies on the semantics of $[\![\text{ST}]\!]^n_{apply(\eta)}$, for each n in $[\![r]\!]_{merge(\eta)}$.

In the (FOR) rule, the variable is bound and then the body is analysed once. Observe that the rule is effectively identical to the subsequent (LET) rule, since our analysis ignores the order and multiplicity of nodes.

Element construction (ELT) returns the code-location cl of the constructor, which

$$\frac{pEnv \vdash \text{EX} \Rightarrow r;\ \langle a, u\rangle}{pEnv \vdash \textbf{do delete}\ \text{EX} \Rightarrow ();\ \langle a, u|(r|r//*)\rangle} \quad \text{(Delete)}$$

$$\frac{pEnv \vdash \text{EX}_1 \Rightarrow r_1;\ \langle a_1, u_1\rangle \quad pEnv \vdash \text{EX}_2 \Rightarrow r_2;\ \langle a_2, u_2\rangle}{pEnv \vdash \textbf{do insert}\ \text{EX}_1\ \textbf{into}\ \text{EX}_2 \Rightarrow ();\ \langle a_1|a_2|(r_1//*), u_1|u_2|(r_2//*)\rangle} \quad \text{(Insert)}$$

$$\frac{pEnv \vdash \text{EX}_1 \Rightarrow r_1;\ \langle a_1, u_1\rangle \quad pEnv \vdash \text{EX}_2 \Rightarrow r_2;\ \langle a_2, u_2\rangle}{pEnv \vdash (\text{EX}_1, \text{EX}_2) \Rightarrow r_1|r_2;\ \langle a_1|a_2, u_1|u_2\rangle} \quad \text{(Comma)}$$

$$\frac{}{pEnv \vdash () \Rightarrow ();\ \langle (), ()\rangle} \quad \text{(Empty)}$$

$$\frac{}{pEnv \vdash lit \Rightarrow ();\ \langle (), ()\rangle} \quad \text{(Literal)}$$

$$\frac{(\$x \twoheadrightarrow r) \in pEnv}{pEnv \vdash \$x \Rightarrow r;\ \langle (), ()\rangle} \quad \text{(Var)}$$

$$\frac{pEnv \vdash \text{EX} \Rightarrow r;\ \langle a, u\rangle}{pEnv \vdash \text{EX}/\text{ST} \Rightarrow r/\text{ST};\ \langle r/\text{ST}|a, u\rangle} \quad \text{(Step)}$$

$$\frac{pEnv \vdash \text{EX}_1 \Rightarrow r_1;\ \langle a_1, u_1\rangle \quad (pEnv + \$x \twoheadrightarrow r_1) \vdash \text{EX}_2 \Rightarrow r_2;\ \langle a_2, u_2\rangle}{pEnv \vdash \textbf{for}\ \$x\ \textbf{in}\ \text{EX}_1\ \textbf{return}\ \text{EX}_2 \Rightarrow r_2;\ \langle a_1|a_2, u_1|u_2\rangle} \quad \text{(For)}$$

$$\frac{pEnv \vdash \text{EX}_1 \Rightarrow r_1;\ \langle a_1, u_1\rangle \quad (pEnv + \$x \twoheadrightarrow r_1) \vdash \text{EX}_2 \Rightarrow r_2;\ \langle a_2, u_2\rangle}{pEnv \vdash \textbf{let}\ \$x := \text{EX}_1\ \textbf{return}\ \text{EX}_2 \Rightarrow r_2;\ \langle a_1|a_2, u_1|u_2\rangle} \quad \text{(Let)}$$

$$\frac{}{pEnv \vdash \textbf{element}_{cl}\ q\ \{\ \} \Rightarrow cl;\ \langle (), cl\rangle} \quad \text{(Elt)}$$

$$\frac{pEnv \vdash \text{EX} \Rightarrow r;\ \langle a, u\rangle}{pEnv \vdash \textbf{text}_{cl}\ \{\ \text{EX}\ \} \Rightarrow cl;\ \langle a, u|cl\rangle} \quad \text{(Text)}$$

$$\frac{pEnv \vdash \text{EX} \Rightarrow r_0;\ \langle a_1, u_1\rangle \quad pEnv \vdash \text{EX}_1 \Rightarrow r_1;\ \langle a_2, u_2\rangle \quad pEnv \vdash \text{EX}_2 \Rightarrow r_2;\ \langle a_3, u_3\rangle}{pEnv \vdash \textbf{if}\ (\text{EX})\ \textbf{then}\ \text{EX}_1\ \textbf{else}\ \text{EX}_2 \Rightarrow r_1|r_2;\ \langle a_1|a_2|a_3, u_1|u_2|u_3\rangle} \quad \text{(If)}$$

$$\frac{\forall i \in 1..k\colon pEnv \vdash \text{EX}_i \Rightarrow r_i;\ \langle a_i, u_i\rangle \quad \underline{f}(r_1, \ldots, r_k) = r}{pEnv \vdash f(\text{EX}_1, \ldots, \text{EX}_k) \Rightarrow r;\ \langle a_1|\ldots|a_k, u_1|\ldots|u_k\rangle} \quad \text{(Func)}$$

$$\frac{pEnv \vdash \text{EX} \Rightarrow r;\ \langle a, u\rangle}{pEnv \vdash \textbf{data}(\text{EX}) \Rightarrow ();\ \langle a|r, u\rangle} \quad \text{(Data)}$$

$$\frac{}{pEnv \vdash \textbf{doc}(uri) \Rightarrow uri;\ \langle uri, ()\rangle} \quad \text{(Doc)}$$

Fig. 2. Path analysis rules.

is a unique tag inserted by the compiler. It is used as a root for paths starting from the new element, as in the following example.

let $\$a_1$:= **element**$_{cl_1}$ a { }
let $\$a_2$:= **element**$_{cl_2}$ a { }
return ($\$a_1$/b, **insert** <b> **into** $\$a_2$)

In this case, cl_1 is associated to $\$a_1$, and cl_2 is associated to $\$a_2$. Paths rooted in cl_i are used by the analysis to infer, for example, that $\$a_1$/b accesses cl_1/b while **insert** <b> **into** $\$a_2$ updates $(cl_2|cl_2//*)$, hence the two commute. Note that the analysis does not keep track of the actual element name being inserted, for simplicity, and because that would add no precision here: every path starting from cl has q as the name of its root element. We discuss possible extensions in Section 7.

(Text) construction is similar but allows for a subexpression to compute the text value.

The conditional (If) rule approximates the result path by merging the result paths of the two branches.

Built-in functions are assumed in the (Func) rule to not directly have side effects, so they only accumulate the effects of their arguments. For each function f, an associated path-function $\underline{f}$ specifies how the returned path of the results depends on the returned paths of the arguments. For example, for the binary equality function, written $\text{EX}_1 = \text{EX}_2$, the associated function is $r_1 \equiv r_2 \mapsto ()$.

Remark 36 *(functions with access to the store).* The approach could be easily extended to built-in functions that access the store. In this case, each function would have an associated path-function $\underline{f}$ that specifies the paths that are returned, accessed, and updated by the function body. For example, a *do-insert*($\$x, \$y$) function that copies $\$y$ below $\$x$, would be associated to a path function $\underline{\textit{do-insert}}(r_1, r_2) \mapsto ((); \langle (r_1//*), (r_2//*) \rangle)$, specifying that it returns nothing, updates $(r_1//*)$, and accesses $(r_2//*)$. The paths that are updated and accessed by the function body would then be added to the paths that are updated and accessed by the function invocation.

Extracting the contents of a text node with (Data) will access that node.

Finally, the (Doc) rule specifies that *doc(uri)* does not update anything, since documents are considered "preloaded" in XQuery. However, it accesses the path *uri*, since its result depends on $R_\sigma(uri)$.

5. SOUNDNESS

In this section, we provide detailed proof of soundness for the path analysis in Section 4.4. The proof is done by induction over the analysis rules. We focus on proving the most representative or difficult rules, specifically, (Comma), (Step), (Element), (Insert), and (Delete), given in the previous section.

5.1 The structure of the proof

Inductive framework. Soundness specifies that

$$pEnv \vdash \text{EX} \Vdash r; \langle a, u \rangle$$
$$dEnv \vdash \eta_0; \text{EX} \Vdash (\eta_0, \vec{\delta}); \vec{v}$$

$$pEnv \supseteq_{\eta_0} dEnv$$

imply some properties, that we call *approximation*, *equal results*, and *immutability*.

The soundness of the analysis judgment, $pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$, is proved by induction over the structure of the expression EX, taking the operational semantics into account. In order to use induction, we apply the following *inversion* property to the $dEnv \vdash \eta_0; \text{EX} \Rightarrow (\eta_0, \vec{\delta}); \vec{v}$ hypothesis.

Inversion is a standard property that specifies that, since any judgement $dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{n}$ can only be proved by the specific rule that matches EX, then, when $dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{n}$ holds, the premises of that specific rule hold as well. We have a special case when $\text{EX} = \textbf{if}\ (\text{EX}_0)\ \textbf{then}\ \text{EX}_1\ \textbf{else}\ \text{EX}_2$, since we have two applicable rules. In this case, either the premises of (IfT) or those of (IfF) hold. Formally, this is specified by cases, as follows.

Proposition 37 (inversion of dynamic semantics).

$$\begin{array}{l}
dEnv \vdash \eta_0; \textbf{do delete}\ \text{EX} \Rightarrow \eta_2; () \\
\Rightarrow\ \exists \vec{n}, \eta_1 \qquad dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{n} \\
\qquad\qquad\quad\ \ \eta_2 = \eta_1, \mathsf{delete}(\vec{n}) \\
\\
dEnv \vdash \eta_0; \textbf{do insert}\ \text{EX}_1\ \textbf{into}\ \text{EX}_2 \Rightarrow \eta_3; () \\
\Rightarrow\ \exists \vec{m}, \eta_1, \eta_2, n, \vec{m}_c, F_c \\
\qquad\qquad\quad\ \ dEnv \vdash \eta_0; \text{EX}_1 \Rightarrow \eta_1; \vec{m} \\
\qquad\qquad\quad\ \ dEnv \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; n \\
\qquad\qquad\quad\ \ (\vec{m}_c, F_{\text{copy}}) \in \mathit{prepare\text{-}deep\text{-}copy}(\mathit{apply}(\eta_2), n, \vec{m}) \\
\qquad\qquad\quad\ \ \eta_3 = \eta_2, \mathsf{insert}(\vec{m}_c, F_c) \\
\ldots \\
dEnv \vdash \eta_0; \textbf{if}\ (\text{EX})\ \textbf{then}\ \text{EX}_1\ \textbf{else}\ \text{EX}_2 \Rightarrow \eta_2; \vec{v} \\
\Rightarrow\ (\exists \eta_1, \vec{v}_0 \neq () \ \ dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; \vec{v}_0, \quad dEnv \vdash \eta_1; \text{EX}_1 \Rightarrow \eta_2; \vec{v}) \\
\quad \vee (\exists \eta_1 \qquad dEnv \vdash \eta_0; \text{EX} \Rightarrow \eta_1; (), \quad dEnv \vdash \eta_1; \text{EX}_2 \Rightarrow \eta_2; \vec{v})
\end{array}$$

Inversion of dynamic semantics allows us to match the premises of the rule used to prove $pEnv \vdash \text{EX} \Rightarrow r; \langle a, u\rangle$ with those of rule used to prove $dEnv \vdash \eta_0; \text{EX} \Rightarrow (\eta_0, \vec{\delta}); \vec{v}$, and to apply induction to each matched pair. This is the starting point of all the different cases.

Expressions with two or more subexpressions. We now sketch how the proof works, using the comma rule as an example. Consider again the comma analysis rule as follows.

$$\frac{pEnv \vdash \text{EX}_1 \Rightarrow r_1; \langle a_1, u_1\rangle \qquad pEnv \vdash \text{EX}_2 \Rightarrow r_2; \langle a_2, u_2\rangle}{pEnv \vdash (\text{EX}_1, \text{EX}_2) \Rightarrow r_1 | r_2; \langle a_1 | a_2, u_1 | u_2\rangle} \quad (\text{Comma})$$

All the rules in Figure 2 with two or more premises combine the effects of the premises in similar ways. For such an expression, which does not access or modify the store, the soundness proof does not make direct use of the definition of equivalence, but only makes use of the framework properties (Lemma 35): the approximation property follows by ($\supseteq$ |) and ($\supseteq$ Stability); *immutability* follows by ($\#^{\cup}$ |) and ($\sim$ Equivalence); equal results follows by ($\sim$ |) and ($\sim$ Stability) (details in Section 5.3). This is an important aspect of our approach, which allows some flexibility in how path equivalence is defined.

Operators that access or modify the store. If the operator accesses the store then the a component of the analysis result contains a specific path, besides those that come from the premises, such as r/ST in the (STEP) rule. In this case, besides the framework properties, the equal results property also depends on a specific lemma that says that, if two stores coincide on the new path, then the operator gets the same result in the two stores, such as Lemmas 38 and Lemma 41 below.

If the operator modifies the store, then a new piece u' appears in the u component, like $r|r//*$ in the (DELETE) rule. In this case, in order to prove *immutability*, we also need a lemma that specifies that $u' \#^{\cup} c$ implies $\eta \sim_c (\eta, \vec{\delta})$, such as Lemmas 39, 40, and 42 below. These lemmas depend on the precise definition of the equivalence relation, not just on its framework properties. This is not surprising: the gist of the definition of our equivalence is deciding what is observed by $\sim_p$. If $\sim_p$ observes a lot of information, *i.e.*, if it is fine grained, then the proof of equal results becomes much easier, but immutability is harder, or may even require a bigger path to be inserted into u. If $\sim_p$ observes little information, *i.e.*, if it is coarse, then the proof of immutability becomes much easier, but equal results is harder, or requires a bigger path to be inserted into a.

To sum up, for every rule, the corresponding case in the soundness proof is constituted by a framework that invokes inversion, uses induction, and combines the results using the framework properties, along the lines of the (COMMA) case (Section 5.3). Moreover, for each action on the store, a specific path is added to the a or the u component, and we have a lemma to prove that this path corresponds to the actual action on the store. These lemmas are the kernel of the proof, and they are collected in the next subsection.

5.2 The Lemmas That Tell The Story

We first present the lemma used for the step operator. The Step Lemma 38 specifies that the semantics of a step ST starting from a node in r is equal in two stores that are equivalent with respect to r/ST, hence justifying the insertion of r/ST into the accessed paths inferred by the (STEP) rule. Actually, the lemma is stronger than this, because it holds not just for the nodes in $[\![r]\!]_{apply(\eta)}$, but also for the bigger set $[\![r]\!]_{merge(\eta)}$. This is necessary because of the stability problem: when we deduce $r \supseteq_\eta n$, this only means that $n \in [\![r]\!]_{merge(\eta)}$, but we are not sure that n is still in $[\![r]\!]_{apply(\eta)}$.

Lemma 38 (step lemma).

$$\eta \sim_{r/\text{ST}} \eta' \wedge n \in [\![r]\!]_{merge(\eta)} \;\Rightarrow\; [\![\text{ST}]\!]^n_{apply(\eta)} = [\![\text{ST}]\!]^n_{apply(\eta')}$$

Proof. $\eta \sim_{r/\text{ST}} \eta'$ implies $\eta \sim^{\emptyset}_{r/\text{ST}} \eta'$, which implies that $n \in [\![r]\!]_{merge(\eta)} \Rightarrow [\![\text{ST}]\!]^n_{apply(\eta)} = [\![\text{ST}]\!]^n_{apply(\eta')}$. □

The next lemma specifies that, if $r \supseteq_\eta \vec{v}$, then $\mathsf{delete}(\vec{v})$ does not affect the nodes that are disjoint from $(r|r//*)$, and is the kernel of the *immutability* proof for **delete**.

Lemma 39 (delete lemma).

$$r \supseteq_\eta \vec{v},\ (r|r//*) \#^{\cup} c \;\Rightarrow\; \eta \sim_c (\eta, \mathit{delete}(\vec{v}))$$

Proof. We must prove the following fact:

$$r \sqsupseteq_\eta \vec{v} \ \wedge \ (r|r//*) \ \#^\cup \ c \ \Rightarrow \ \forall \vec{\delta}.\ (\eta, \vec{\delta}) \sim_c^\emptyset (\eta, \mathsf{delete}(\vec{v}), \vec{\delta})$$

We reason by induction on the size of c and by cases on its shape. All cases are trivial, with the exception of $c = c'/\text{ST}$. In this case we have to prove that $\forall n \in [\![c']\!]_{merge(\eta,\vec{\delta})}$,

$$([\![\text{ST}]\!]^n_{apply(\eta,\vec{\delta})} = [\![\text{ST}]\!]^n_{apply(\eta,\mathsf{delete}(\vec{v}),\vec{\delta})} \text{ and } [\![\text{ST}]\!]^n_{merge(\eta,\vec{\delta})} = [\![\text{ST}]\!]^n_{merge(\eta,\mathsf{delete}(\vec{v}),\vec{\delta})})$$

The second equality follows from $merge(\eta, \vec{\delta}) = merge(\eta, \mathsf{delete}(\vec{v}), \vec{\delta})$, and the inclusion $[\![\text{ST}]\!]^n_{apply(\eta,\vec{\delta})} \supseteq [\![\text{ST}]\!]^n_{apply(\eta,\mathsf{delete}(\vec{v}),\vec{\delta})}$ is also trivial, hence we have just to prove that $m \in [\![\text{ST}]\!]^n_{apply(\eta,\vec{\delta})} \ \Rightarrow \ m \in [\![\text{ST}]\!]^n_{apply(\eta,\mathsf{delete}(\vec{v}),\vec{\delta})}$.

Recall that, by Property 10, $apply(\eta, \mathsf{delete}(\vec{v}), \vec{\delta}) = apply(\eta, \vec{\delta}, \mathsf{delete}(\vec{v}))$.

We use the hypotheses

$$r \sqsupseteq_\eta \vec{v} \tag{a}$$
$$(r|r//*) \ \#^\cup \ c'/\text{ST} \tag{b}$$
$$n \in [\![c']\!]_{merge(\eta,\vec{\delta})} \tag{c}$$
$$m \in [\![\text{ST}]\!]^n_{apply(\eta,\vec{\delta})} \tag{d}$$

and want to prove that $m \in [\![\text{ST}]\!]^n_{apply(\eta,\vec{\delta},\mathsf{delete}(\vec{v}))}$.

We first observe that $E_{apply(\eta)} \subseteq E_{merge(\eta)}$ and (d) imply

$$m \in [\![\text{ST}]\!]^n_{merge(\eta,\vec{\delta})} \tag{e}$$

hence, by (c),

$$m \in [\![c'/\text{ST}]\!]_{merge(\eta,\vec{\delta})} \tag{f}$$

We also observe that, for any m' such that $m' \in [\![c'/\text{ST}]\!]^\cup_{merge(\eta,\vec{\delta})}$, (b) implies that $m' \notin [\![r|r//*]\!]_{merge(\eta,\vec{\delta})}$, *i.e.*, there is no $m'' \in [\![r]\!]_{merge(\eta,\vec{\delta})}$ such that $(m'', m') \in E^*_{merge(\eta,\vec{\delta})}$ hence, by $E_{merge(\eta)} \subseteq E_{merge(\eta,\vec{\delta})}$, there is no $m'' \in [\![r]\!]_{merge(\eta)}$ such that $(m'', m') \in E^*_{merge(\eta,\vec{\delta})}$; by (a), $\vec{v} \subseteq [\![r]\!]_{merge(\eta)}$, hence, (g1) $m' \in [\![c'/\text{ST}]\!]^\cup_{merge(\eta,\vec{\delta})} \ \Rightarrow \ \nexists m'' \in \vec{v}\colon (m'', m') \in E^*_{merge(\eta,\vec{\delta})}$, (g2) by (g1), (f): $\nexists m'' \in \vec{v}\colon (m'', m) \in E^*_{merge(\eta,\vec{\delta})}$, and (g3) by (g1) and $n \in [\![c'/\text{ST}]\!]^\cup_{merge(\eta,\vec{\delta})}$: $\nexists m'' \in \vec{v}\colon (m'', n) \in E^*_{merge(\eta,\vec{\delta})}$.

We consider the different cases:

—ST = **child**::NT: by (d), $(n, m) \in E_{apply(\eta,\vec{\delta})}$; by (g2) and $(m, m) \in E^*_{merge(\eta,\vec{\delta})}$, $m \notin \vec{v}$, hence (n, m) is also in $E_{apply(\eta,\vec{\delta},\mathsf{delete}(\vec{v}))}$.

—ST = **descendant**::NT: by (d), $(n, m) \in E^+_{apply(\eta,\vec{\delta})}$; by (g2), $\nexists m'' \in \vec{v}\colon (m'', m) \in E^*_{merge(\eta,\vec{\delta})}$, hence $\nexists m'' \in \vec{v}\colon (m'', m) \in E^*_{apply(\eta,\vec{\delta})}$, hence (n, m) is also in $E^+_{apply(\eta,\vec{\delta},\mathsf{delete}(\vec{v}))}$.

—ST = **parent**::NT: by (d), $(m, n) \in E_{apply(\eta,\vec{\delta})}$; by (g3) and $(n, n) \in E^*_{merge(\eta,\vec{\delta})}$, $n \notin \vec{v}$, hence $(m, n) \in E_{apply(\eta,\vec{\delta},\mathsf{delete}(\vec{v}))}$.

—ST = **ancestor**::NT: by (d), $(m,n) \in E^+_{apply(\eta,\vec{\delta})}$, by (g3), $\nexists m'' \in \vec{v}$: $(m'',n) \in E^*_{merge(\eta,\vec{\delta})}$, hence $\nexists m'' \in \vec{v}$: $(m'',n) \in E^*_{apply(\eta,\vec{\delta})}$, hence (m,n) is also in $E^+_{apply(\eta,\vec{\delta},\mathsf{delete}(\vec{v}))}$. □

The next lemma specifies that, if $r \supseteq_\eta n_p$, then insertion below n_p does not affect the nodes that are disjoint from $r//*$, and is the kernel of *immutability* for **insert**.

Lemma 40 (insert lemma). *Let* $\sigma = merge(\eta, \mathit{insert}(\vec{m}_c, F_c))$*;*

$$\begin{array}{l} \vec{m}_c \subseteq desc(\sigma, n_p) \ \wedge \ r \supseteq_\eta n_p \ \wedge \ (r//*) \ \#^\cup \ c \\ \quad \Rightarrow \quad \eta \sim_c (\eta, \mathit{insert}(\vec{m}_c, F_c)) \end{array}$$

Proof. The thesis can be rewritten as follows, where $\mathsf{insert}()$ abbreviates $\mathsf{insert}(\vec{m}_c, F_c)$:

$$\forall \vec{\delta}.\ \vec{m}_c \subseteq desc(merge(\eta, \mathsf{insert}()), n_p) \ \wedge \ r \supseteq_\eta n_p \ \wedge \ (r//*) \ \#^\cup \ c \\ \Rightarrow \ (\eta, \vec{\delta}) \sim^{\emptyset}_c (\eta, \mathsf{insert}(), \vec{\delta})$$

We reason by induction on the size of c and by cases on its shape. All cases are trivial, with the exception of $c = c'/\mathrm{ST}$. In this case we have to prove the following fact.
$\forall \vec{\delta},\ \forall n \in [\![c']\!]_{merge(\eta,\vec{\delta})}$:
$([\![\mathrm{ST}]\!]^n_{apply(\eta,\vec{\delta})} = [\![\mathrm{ST}]\!]^n_{apply(\eta,\mathsf{insert}(),\vec{\delta})}$ and $[\![\mathrm{ST}]\!]^n_{merge(\eta,\vec{\delta})} = [\![\mathrm{ST}]\!]^n_{merge(\eta,\mathsf{insert}(),\vec{\delta})})$

We will just prove that $m \in [\![\mathrm{ST}]\!]^n_{apply(\eta,\mathsf{insert}(),\vec{\delta})} \ \Rightarrow \ m \in [\![\mathrm{ST}]\!]^n_{apply(\eta,\vec{\delta})}$ and $m \in [\![\mathrm{ST}]\!]^n_{merge(\eta,\mathsf{insert}(),\vec{\delta})} \ \Rightarrow \ m \in [\![\mathrm{ST}]\!]^n_{merge(\eta,\vec{\delta})}$, since the other direction is immediate; we will actually prove the first, since the proof for the second is equal.

Observe that, by Property 8, $apply(\eta, \mathsf{insert}(\vec{m}_c, F_c), \vec{\delta}) = apply(\eta, \vec{\delta}, \mathsf{insert}(\vec{m}_c, F'))$, for some F'; we are not interested in the F component here, hence we will also abbreviate $\mathsf{insert}(\vec{m}_c, F')$ as $\mathsf{insert}()$.

We have the following hypotheses:

$$m \in \vec{m}_c \Rightarrow (n_p, m) \in E^+_{merge(\eta,\mathsf{insert}())} \tag{a}$$

$$r \supseteq_\eta n_p \tag{b}$$

$$(r//*) \ \#^\cup \ c'/\mathrm{ST} \tag{c}$$

$$n \in [\![c']\!]_{merge(\eta,\vec{\delta})} \tag{d}$$

$$m \in [\![\mathrm{ST}]\!]^n_{apply(\eta,\vec{\delta},\mathsf{insert}())} \tag{e}$$

and want to prove that $m \in [\![\mathrm{ST}]\!]^n_{apply(\eta,\vec{\delta})}$.

We first observe that (d) implies

$$n \in [\![c']\!]_{merge(\eta,\vec{\delta},\mathsf{insert}())} \tag{f}$$

hence

$$m \in [\![c'/\mathrm{ST}]\!]_{merge(\eta,\vec{\delta},\mathsf{insert}())} \tag{g}$$

We also observe that (b) means $n_p \in [\![r]\!]_{merge(\eta)}$, hence $n_p \in [\![r]\!]_{merge(\eta,\vec{\delta})}$, hence, by (a) and $E_{merge(\eta,\mathsf{insert}())} \subseteq E_{merge(\eta,\vec{\delta},\mathsf{insert}())}$ we have (h) $\vec{m}_c \subseteq [\![r//*]\!]_{merge(\eta,\vec{\delta},\mathsf{insert}())}$

Let us define $\vec{m}_{dd} = \{ m' \mid \exists m \in \vec{m}_c.\ (m, m') \in E^*_{merge(\eta,\mathsf{insert}())} \}$; from (a) we deduce (i) $\vec{m}_{dd} \subseteq [\![r//*]\!]_{merge(\eta,\vec{\delta},\mathsf{insert}())}$. (Observe that, if $\eta, \vec{\delta}, \mathsf{insert}()$ were produced by the evaluation of any expression, $\vec{m}_{dd}$ would coincide with $\vec{m}_c$, but, in general, we can only know that $\vec{m}_c \subseteq \vec{m}_{dd}$).

We now observe that, for any n' such that $n' \in [\![c'/\mathrm{ST}]\!]^{\cup}_{merge(\eta,\vec{\delta},\mathsf{insert}())}$, (c) implies that $n' \notin [\![r//*]\!]_{merge(\eta,\vec{\delta},\mathsf{insert}())}$, hence, by (i), (l1) $n' \in [\![c'/\mathrm{ST}]\!]^{\cup}_{merge(\eta,\vec{\delta},\mathsf{insert}())} \Rightarrow n' \notin \vec{m}_{dd}$. Further, (l2) by (l1), (f): $n \notin \vec{m}_{dd}$, and (l3) by (l1), (g): $m \notin \vec{m}_{dd}$.

Recall that $\vec{m}_{dd}$ are the descendants of $\vec{m}_c$ in $merge(\eta, \vec{\delta}, \mathsf{insert}())$, hence include all the descendant of $\vec{m}_c$ in $apply(\eta, \vec{\delta}, \mathsf{insert}())$. We consider the different cases:

—ST = **child**::NT: by (e), $(n, m) \in E_{apply(\eta,\vec{\delta},\mathsf{insert}())}$, hence, by $(l3)$, (n, m) is also in $E_{apply(\eta,\vec{\delta})}$.

—ST = **descendant**::NT: by (e), $(n, m) \in E^+_{apply(\eta,\vec{\delta},\mathsf{insert}())}$, hence, by $(l3)$, (n, m) is also in $E^+_{apply(\eta,\vec{\delta})}$.

—ST = **parent**::NT: by (e), $(m, n) \in E_{apply(\eta,\vec{\delta},\mathsf{insert}())}$, hence, by $(l2)$, (m, n) is also in $E_{apply(\eta,\vec{\delta})}$.

—ST = **ancestor**::NT: by (e), $(m, n) \in E^+_{apply(\eta,\vec{\delta},\mathsf{insert}())}$, hence, by $(l2)$, (m, n) is also in $E^+_{apply(\eta,\vec{\delta})}$. □

The next lemma is used to prove that the subtrees rooted in $\vec{m}$, copied by the **insert** operation, are equal in two histories that are $r//*$-equivalent, provided that $r \supseteq_\eta \vec{m}$.

Lemma 41 (identical trees). *If $r \supseteq_\eta \vec{m}$ and $\eta \sim_{r//*} \eta'$, then, for each $m \in \vec{m}$, m is in $N_{\eta'}$ and the subtrees rooted in m in $apply(\eta)$ and in $apply(\eta')$ are identical, meaning that they are formed by the same nodes, with the same parent-child relation, and the same value for F_η and $F_{\eta'}$.*

Proof. $\eta \sim_{r//*} \eta'$ implies $\eta \sim^{\emptyset}_{r} \eta'$ and $\eta \sim^{\emptyset}_{r//*} \eta'$. By Lemma 24, $\eta \sim^{\emptyset}_{r} \eta'$ and $r \supseteq_\eta \vec{m}$ imply $\vec{m} \subseteq [\![r]\!]_{apply(\eta')} \subseteq N_{merge(\eta')}$. From $\eta \sim^{\emptyset}_{r//*} \eta'$, we deduce that, for any $m \in [\![r]\!]_\eta$, $[\![\textbf{descendant}::*]\!]^m_{apply(\eta)} = [\![\textbf{descendant}::*]\!]^m_{apply(\eta')}$, hence, for any $m \in \vec{m}$, it has the same descendants in $apply(\eta)$ and $apply(\eta')$. For each node m' in $\bigcup_{m\in\vec{m}} [\![\textbf{descendant}::*]\!]^m_{apply(\eta)}$, Lemma 24 states that $F_{apply(\eta)}(m) = F_{apply(\eta')}(m)$. □

Lemma 42 (element lemma).

$$\mathcal{R}(n) = \ell \ \wedge\ \ell \#^{\cup} c \ \Rightarrow\ \eta \sim_c (\eta, \mathit{insert}(n, F))$$

Proof. We must prove the following fact:

$$\mathcal{R}(n) = \ell \ \wedge\ \ell \#^{\cup} c \ \Rightarrow\ \forall \vec{\delta}.\ (\eta, \vec{\delta}) \sim^{\emptyset}_c (\eta, \mathsf{insert}(n, F), \vec{\delta})$$

In a nutshell, this is obvious since equivalence with respect to c is not affected by the addition of a node which is not traversed by c. Formally, we prove it by induction and by cases on c. Case $c = ()$ is immediate. Case $c = \ell'$ is also

immediate because $\ell \#^{\cup} c$ implies that $\ell \neq \ell'$, hence $[\![\ell']\!]_{\eta,\vec{\delta}} = [\![\ell']\!]_{\eta,\mathsf{insert}(n,F),\vec{\delta}}$. Case $c = p|p'$ is immediate by induction. In case $c = p/\text{ST}$ we have to prove $\forall n' \in ([\![p]\!]_{merge(\eta_1)} \cap [\![p]\!]_{merge(\eta_2)})$: $[\![\text{ST}]\!]^{n'}_{apply(\eta_1)} = [\![\text{ST}]\!]^{n'}_{apply(\eta_2)} \wedge [\![\text{ST}]\!]^{n'}_{merge(\eta_1)} = [\![\text{ST}]\!]^{n'}_{merge(\eta_2)}$, where η_1 and η_2 are $(\eta, \vec{\delta})$ and $(\eta, \mathsf{insert}(n, F), \vec{\delta})$. From $\ell \#^{\cup} p/\text{ST}$ and $n' \in [\![p]\!]_{merge(\eta_1)}$, we deduce that n' is not a descendant of n, and this implies $[\![\text{ST}]\!]^{n'}_{apply(\eta_1)} = [\![\text{ST}]\!]^{n'}_{apply(\eta_2)} \wedge [\![\text{ST}]\!]^{n'}_{merge(\eta_1)} = [\![\text{ST}]\!]^{n'}_{merge(\eta_2)}$, since the only difference between η_1 and η_2 is the addition of n to N. □

We are now ready to prove the soundness of each rule (using the comma rule as the template for all the simple combination rules).

5.3 Comma Rule

In what follows, we always use the framework properties of Lemma 35 without explicit reference to the Lemma.

The comma rule deduces

$$pEnv \vdash \text{EX}_1, \text{EX}_2 \Mapsto r_1|r_2; \langle a_1|a_2, u_1|u_2\rangle$$

from

$$pEnv \vdash \text{EX}_1 \Mapsto r_1; \langle a_1, u_1\rangle \tag{1}$$
$$pEnv \vdash \text{EX}_2 \Mapsto r_2; \langle a_2, u_2\rangle \tag{2}$$

We have to prove that the following assumptions,

$$\eta_2 = \eta_0, \vec{\delta}_{12} \tag{3}$$
$$dEnv \vdash \eta_0; \text{EX}_1, \text{EX}_2 \Mapsto \eta_2; \vec{v}_1, \vec{v}_2 \tag{4}$$
$$pEnv \supseteq_{\eta_0} dEnv \tag{5}$$

imply the following facts:

Approximation. $r_1|r_2 \supseteq_{\eta_2} \vec{v}_1, \vec{v}_2$

Immutability. $(u_1|u_2) \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_2$

Equal results.

$$\eta'_0 \sim_{r_1|r_2} \eta_0 \wedge N_{\eta'_0} \# inserted(\vec{\delta}_{12}) \Rightarrow dEnv \vdash \eta'_0; \text{EX}_1, \text{EX}_2 \Mapsto \eta'_0, \vec{\delta}_{12}; \vec{v}_1, \vec{v}_2$$

By the inversion property of the dynamic semantics, (4) implies that $\vec{\delta}_{12}$ can be split into $\vec{\delta}_1$ and $\vec{\delta}_2$ such that:

$$dEnv \vdash \eta_0; \text{EX}_1 \Mapsto \eta_0, \vec{\delta}_1; \vec{v}_1 \tag{6}$$
$$dEnv \vdash \eta_0, \vec{\delta}_1; \text{EX}_2 \Mapsto \eta_0, \vec{\delta}_1, \vec{\delta}_2; \vec{v}_2 \tag{7}$$

By ($\supseteq$ Stability), (5) implies:

$$pEnv \supseteq_{\eta_0, \vec{\delta}_1} dEnv \tag{8}$$

By induction, (1), (6), (5) and (2), (7), (8) imply the following properties, where $\eta_1 = \eta_0, \vec{\delta}_1$:

Approximation induction. $r_1 \supseteq_{\eta_1} \vec{v}_1$, $r_2 \supseteq_{\eta_2} \vec{v}_2$.

Immutability induction. $u_1 \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_0, \vec{\delta}_1$, $u_2 \#^{\cup} c \Rightarrow \eta_0, \vec{\delta}_1 \sim_c \eta_0, \vec{\delta}_1, \vec{\delta}_2$.
Equal Results induction.

$$\eta_0' \sim_{a_1} \eta_0 \wedge N_{\eta_0'} \# \mathit{inserted}(\vec{\delta}_1) \Rightarrow (\mathit{dEnv} \vdash \eta_0'; \text{EX}_1 \Vdash \eta_0', \vec{\delta}_1; \vec{v}_1)$$
$$\eta_0', \vec{\delta}_1 \sim_{a_2} \eta_0, \vec{\delta}_1 \wedge N_{\eta_0, \vec{\delta}_1} \# \mathit{inserted}(\vec{\delta}_2) \Rightarrow (\mathit{dEnv} \vdash \eta_0', \vec{\delta}_1; \text{EX}_2 \Vdash \eta_0', \vec{\delta}_1, \vec{\delta}_2; \vec{v}_2)$$

Now we can prove the three properties.

Comma: Approximation. By induction, $r_1 \supseteq_{\eta_1} \vec{v}_1$ and $r_2 \supseteq_{\eta_2} \vec{v}_2$. By ($\supseteq$ Stability), $r_1 \supseteq_{\eta_2} \vec{v}_1$. By ($\supseteq$ |), $r_1|r_2 \supseteq_{\eta_2} \vec{v}_1|\vec{v}_2$.

Comma: Immutability. We want to prove that $(u_1|u_2) \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_2$. $(u_1|u_2) \#^{\cup} c$, by (#$^{\cup}$ |), implies $u_1 \#^{\cup} c$, hence by induction $\eta_0 \sim_c \eta_0, \vec{\delta}_1$. Similarly, $(u_1|u_2) \#^{\cup} c$, implies $u_2 \#^{\cup} c$, hence, by induction, $\eta_0, \vec{\delta}_1 \sim_c \eta_0, \vec{\delta}_1, \vec{\delta}_2$. By ($\sim$ Equivalence): $\eta_0 \sim_c \eta_0, \vec{\delta}_1, \vec{\delta}_2$.

Comma: Equal Results. Assume $\eta_0' \sim_{a_1|a_2} \eta_0$. By (|$\sim$), we have that $\eta_0' \sim_{a_1} \eta_0$ and $\eta_0' \sim_{a_2} \eta_0$. The hypothesis $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta}_1, \vec{\delta}_2)$ implies $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta}_1)$ and $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta}_2)$; by property 15, $\vec{\delta}_1, \vec{\delta}_2$ is non-rewriting, hence $N_{\eta_0', \vec{\delta}_1} \# \mathit{inserted}(\vec{\delta}_2)$. Hence, we can apply the induction hypothesis to prove that $\mathit{dEnv} \vdash \eta_0'; \text{EX}_1 \Vdash \eta_0', \vec{\delta}_1; \vec{v}_1$. By ($\sim$ stability), $\eta_0' \sim_{a_2} \eta_0$ implies $\eta_0', \vec{\delta}_1 \sim_{a_2} \eta_0, \vec{\delta}_1$.
Hence, by induction, $\mathit{dEnv} \vdash \eta_0', \vec{\delta}_1; \text{EX}_2 \Vdash \eta_0', \vec{\delta}_1, \vec{\delta}_2; \vec{v}_2$ and, from this,

$$\mathit{dEnv} \vdash \eta_0'; \text{EX}_1, \text{EX}_2 \Vdash \eta_0', \vec{\delta}_1, \vec{\delta}_2; \vec{v}_1, \vec{v}_2$$

5.4 Step rule

The rule deduces:

$$\mathit{pEnv} \vdash \text{EX}/\text{ST} \Vdash r/\text{ST}; \langle (r/\text{ST}) | a, u \rangle$$

from:

$$\mathit{pEnv} \vdash \text{EX} \Vdash r; \langle a, u \rangle \tag{1}$$

We have to prove that the following assumptions:

$$\eta_1 = \eta_0, \vec{\delta}_1 \tag{2}$$
$$\mathit{dEnv} \vdash \eta_0; \text{EX}/\text{ST} \Vdash \eta_1; \vec{n}_2 \tag{3}$$
$$\mathit{pEnv} \supseteq_{\eta_0} \mathit{dEnv} \tag{4}$$

imply the following facts:

Approximation. $r/\text{ST} \supseteq_{\eta_1} \vec{n}_2$.
Immutability. $u \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_1$.
Equal results.

$$\eta_0' \sim_{(r/\text{ST})|a} \eta_0 \wedge N_{\eta_0'} \# \mathit{inserted}(\vec{\delta}_1) \Rightarrow \mathit{dEnv} \vdash \eta_0'; \text{EX}/\text{ST} \Vdash (\eta_0', \vec{\delta}_1); \vec{n}_2$$

By the inversion property of the dynamic semantics, (3) implies that, for some $\vec{n}_1$:

$$\mathit{dEnv} \vdash \eta_0; \text{EX} \Vdash \eta_1; \vec{n}_1 \tag{5}$$

$$\vec{n}_2 = \bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{apply(\eta_1)} \tag{6}$$

By stability, (4) implies:

$$pEnv \supseteq_{\eta_0, \vec{\delta}_1} dEnv \tag{7}$$

By induction, (1), (5), and (4):

Approximation induction. $r \supseteq_{\eta_1} \vec{n}_1$.

Immutability induction. $u \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_1$.

Equal results induction.

$$\eta'_0 \sim_{a_1} \eta_0 \wedge N_{\eta'_0} \# inserted(\vec{\delta}_1) \Rightarrow (dEnv \vdash \eta'_0; \text{EX} \Mapsto \vec{n}_1; (\eta_0, \vec{\delta}_1))$$

Step: Approximation. We must prove that $r/\text{ST} \supseteq_{\eta_1} \vec{n}_2$, *i.e.*, $\bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{apply(\eta_1)} \subseteq [\![r/\text{ST}]\!]_{merge(\eta_1)}$. The result holds by induction, because the semantics of a step in $merge(\eta)$ always includes that of the same step in $apply(\eta)$. Specifically, by induction, $r \supseteq_{\eta_1} \vec{n}_1$, *i.e.*, $\vec{n}_1 \subseteq [\![r]\!]_{merge(\eta_1)}$. Let $m \in \bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{apply(\eta_1)}$, hence, exists $n' \in \vec{n}_1$ such that $m \in [\![\text{ST}]\!]^{n'}_{apply(\eta_1)}$; from this we deduce $m \in [\![\text{ST}]\!]^{n'}_{merge(\eta_1)}$; by $\vec{n}_1 \subseteq [\![r]\!]_{merge(\eta_1)}$, we have that $n' \in [\![r]\!]_{merge(\eta_1)}$, hence $m \in \bigcup_{n \in [\![r]\!]_{merge(\eta_1)}} [\![\text{ST}]\!]^n_{merge(\eta_1)}$, *i.e.* $m \in [\![r/\text{ST}]\!]_{merge(\eta_1)}$.

Step: Immutability. We want to prove: $u \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_1$, which holds by induction.

Step: Equal Results. We want to prove that, for any store history η'_0,
if $\eta'_0 \sim_{(r/\text{ST})|a} \eta_0$ and $N_{\eta'_0} \# inserted(\vec{\delta})$, then $dEnv \vdash \eta'_0; \text{EX/ST} \Mapsto (\eta'_0, \vec{\delta}); \vec{n}_2$.

By induction, $\eta'_0 \sim_a \eta_0$ and $N_{\eta'_0} \# inserted(\vec{\delta})$, then $dEnv \vdash \eta'_0; \text{EX} \Mapsto (\eta'_0, \vec{\delta}); \vec{n}_1$. We want now to prove that

$$\bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{apply(\eta_1)} = \bigcup_{n \in \vec{n}_1} [\![\text{ST}]\!]^n_{apply(\eta'_1)} \tag{*}$$

where $\eta'_1 = (\eta'_0, \vec{\delta})$. By ($\sim$Stability), $\eta_1 \sim_{r/\text{ST}} \eta'_1$. By Lemma 38, for all $n \in [\![r]\!]_{merge(\eta_1)}$, $[\![\text{ST}]\!]^n_{apply(\eta_1)} = [\![\text{ST}]\!]^n_{apply(\eta'_1)}$. By induction, $r \supseteq_{\eta_1} \vec{n}_1$, *i.e.* $\vec{n}_1 \subseteq [\![r]\!]_{merge(\eta_1)}$, hence $\forall n \in \vec{n}_1\, [\![\text{ST}]\!]^n_{apply(\eta_1)} = [\![\text{ST}]\!]^n_{apply(\eta'_1)}$, hence (*) holds.

5.5 Delete Rule

The rule deduces:

$$pEnv \vdash \textbf{do delete}\ \text{EX} \Mapsto (); \langle a, u | (r | r//*) \rangle$$

from

$$pEnv \vdash \text{EX} \Mapsto r; \langle a, u \rangle \tag{1}$$

we have to prove that the following assumptions:

$$\eta_2 = \eta_0, \vec{\delta}_1, \mathsf{delete}(\vec{v}) \tag{2}$$

$$dEnv \vdash \eta_0; \textbf{do delete}\ \text{EX} \Mapsto \eta_0, \vec{\delta}_1, \mathsf{delete}(\vec{v}); () \tag{3}$$

$$pEnv \supseteq_{\eta_0} dEnv \tag{4}$$

imply the following facts:

Approximation:. $() \supseteq_{\eta_2} ()$.
Immutability. $(u|(r|r//*)) \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_2$.
Equal Results. $\eta_0' \sim_a \eta_0 \wedge N_{\eta_0'} \# inserted(\vec{\delta}_1, \mathsf{delete}(\vec{v}))$
$\Rightarrow dEnv \vdash \eta_0'; \textbf{do delete}\ \text{EX} \Mapsto \eta_0', \vec{\delta}_1, \mathsf{delete}(\vec{v}); ()$.

By the inversion property of the dynamic semantics, (3) implies:

$$dEnv \vdash \eta_0; \text{EX} \Mapsto \eta_0, \vec{\delta}_1; \vec{v} \tag{5}$$

By induction, (1), (5), and (4) imply the following properties, where $\eta_1 = \eta_0, \vec{\delta}_1$:

Approximation induction. $r \supseteq_{\eta_1} \vec{v}$.
Immutability induction. $u \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_0, \vec{\delta}_1$.
Equal Results induction. $\eta_0' \sim_a \eta_0 \wedge N_{\eta_0'} \# inserted(\vec{\delta}_1) \Rightarrow dEnv \vdash \eta_0'; \text{EX} \Mapsto \eta_0', \vec{\delta}_1; \vec{v}$.

Now we can prove the three properties. Approximation is trivial. Immutability and equal results are less trivial.

Delete: Immutability. By induction, using Lemma 39: assume $(u|(r|r//*)) \#^{\cup} c$. This implies $u \#^{\cup} c$ and $(r|r//*) \#^{\cup} c$, hence $\eta_0 \sim_c \eta_0, \vec{\delta}_1$ follows by induction and $\eta_0, \vec{\delta}_1 \sim_c \eta_0, \vec{\delta}_1, \mathsf{delete}(\vec{v})$ follows by Lemma 39. Finally, $\eta_0 \sim_c \eta_0, \vec{\delta}_1, \mathsf{delete}(\vec{v})$ follows by transitivity.

Delete: Equal Results. We must prove that $\eta_0' \sim_a \eta_0$ and $N_{\eta_0'} \# inserted(\vec{\delta}_1, \mathsf{delete}(\vec{v}))$ imply $dEnv \vdash \eta_0'; \textbf{do delete}\ \text{EX} \Mapsto \eta_0', \vec{\delta}_1, \mathsf{delete}(\vec{v}); ()$. By induction, $dEnv \vdash \eta_0'; \text{EX} \Mapsto \eta_0', \vec{\delta}_1; \vec{v}$; the thesis follows immediately from the semantics of **delete**.

5.6 Element Rule

The rule deduces:

$$pEnv \vdash \textbf{element}_{cl}\ q\ \{\ \} \Mapsto cl; \langle (), cl \rangle$$

We have to prove that the following assumptions,

$$dEnv \vdash \eta_0; \textbf{element}_{cl}\ q\ \{\ \} \Mapsto \eta_0, \vec{\delta}_1; n \tag{1}$$
$$pEnv \supseteq_{\eta_0} dEnv \tag{2}$$

imply the following facts, where $\eta_1 = \eta_0, \vec{\delta}_1$:

Approximation. $cl \supseteq_{\eta_1} n$.
Immutability. $cl \#^{\cup} c \Rightarrow \eta_0 \sim_c \eta_1$.
Equal Results.

$$\eta_0' \sim_{()} \eta_0 \wedge N_{\eta_0'} \# inserted(\vec{\delta}_1) \Rightarrow dEnv \vdash \eta_0'; \textbf{element}_{cl}\ q\ \{\ \} \Mapsto \eta_0', \vec{\delta}_1; n$$

By the inversion property of the dynamic semantics, (1) implies what follows, where all the new variables are existentially quantified:

$$n \in choose(N_{apply(\eta_0)}, cl) \tag{3}$$

$$\vec{\delta_1} = \mathsf{insert}(n, (n \mapsto [\mathsf{kind} : \mathit{element}, \mathsf{name} : q, \mathsf{value} : \bot])) \quad (4)$$

Now we can prove the three properties.
Approximation follows from $n \in \mathit{choose}(N_{\mathit{apply}(\eta_0)}, \mathit{cl})$.

Immutability follows by Lemma 42.

For equal results, we must prove that

$$\eta_0' \sim_{()} \eta_0 \wedge N_{\eta_0'} \# \mathit{inserted}(\vec{\delta_1}) \Rightarrow \mathit{dEnv} \vdash \eta_0'; \mathbf{element}_{\mathit{cl}}\ q\ \{\ \} \Mapsto \eta_0', \vec{\delta_1'}; m'$$

with $\vec{\delta_1'} = \vec{\delta_1}$ and $m' = n$. From $N_{\eta_0'} \# \mathit{inserted}(\vec{\delta_1})$ we deduce that n is fresh for η_0', hence we can choose $\vec{\delta_1'} = \vec{\delta_1}$, which in turn implies $m' = n$.

5.7 Insert Rule

The rule deduces:

$$\mathit{pEnv} \vdash \mathbf{do\ insert}\ \mathrm{EX}_1\ \mathbf{into}\ \mathrm{EX}_2 \Mapsto (); \langle a_1|a_2|(r_1//*), u_1|u_2|(r_2//*)\rangle$$

from:

$$\mathit{pEnv} \vdash \mathrm{EX}_1 \Mapsto r_1; \langle a_1, u_1\rangle \quad (1)$$
$$\mathit{pEnv} \vdash \mathrm{EX}_2 \Mapsto r_2; \langle a_2, u_2\rangle \quad (2)$$

We have to prove that the following assumptions,

$$\mathit{dEnv} \vdash \eta_0; \mathbf{do\ insert}\ \mathrm{EX}_1\ \mathbf{into}\ \mathrm{EX}_2 \Mapsto \eta_3; () \quad (3)$$
$$\mathit{pEnv} \supseteq_{\eta_0} \mathit{dEnv} \quad (4)$$

imply the following facts:

Approximation. $() \supseteq_{\eta_3} ()$.
Immutability. $(u_1|u_2|(r_2//*))\ \#^{\cup}\ c \Rightarrow \eta_0 \sim_c \eta_3$.
Equal Results.

$$\eta_0' \sim_{a_1|a_2|(r_1//*)} \eta_0 \wedge N_{\eta_0'} \# \mathit{inserted}(\eta_3 \setminus \eta_0) \Rightarrow \mathit{dEnv} \vdash \eta_0'; \mathbf{do\ insert}\ \mathrm{EX}_1\ \mathbf{into}\ \mathrm{EX}_2 \Mapsto \eta_0', (\eta_3 \setminus \eta_0); ()$$

By the inversion property of the dynamic semantics, (3) implies what follows, where $(\vec{m}_c, F_c) \in \mathit{prepare\text{-}deep\text{-}copy}(\mathit{apply}(\eta_2), n, \vec{m})$:

$$\mathit{dEnv} \vdash \eta_0; \mathrm{EX}_1 \Mapsto \eta_0, \vec{\delta_1}; \vec{m} \quad (5)$$
$$\mathit{dEnv} \vdash \eta_0, \vec{\delta_1}; \mathrm{EX}_2 \Mapsto \eta_0, \vec{\delta_1}, \vec{\delta_2}; n \quad (6)$$
$$\eta_3 = \eta_2, \mathsf{insert}(\vec{m}_c, F_c) \quad (7)$$

By stability, (4) implies $\mathit{pEnv} \supseteq_{\eta_0, \vec{\delta_1}} \mathit{dEnv}$. Hence, by induction, (1), (2), (5), and (6) imply the following properties, where $\eta_1 = \eta_0, \vec{\delta_1}$ and where $\eta_2 = \eta_0, \vec{\delta_1}, \vec{\delta_2}$:

Approximation induction. $r_1 \supseteq_{\eta_1} \vec{m}$, $r_2 \supseteq_{\eta_2} n$.
Immutability induction. $u_1\ \#^{\cup}\ c \Rightarrow \eta_0 \sim_c \eta_0, \vec{\delta_1}$, $u_2\ \#^{\cup}\ c \Rightarrow \eta_0, \vec{\delta_1} \sim_c \eta_0, \vec{\delta_1}, \vec{\delta_2}$.
Equal Results induction.

$$\eta_0' \sim_{a_1} \eta_0 \wedge N_{\eta_0'} \# \mathit{inserted}(\vec{\delta_1}) \Rightarrow \mathit{dEnv} \vdash \eta_0'; \mathrm{EX}_1 \Mapsto \eta_0', \vec{\delta_1}; \vec{m}$$

$$\eta'_0, \vec{\delta}_1 \sim_{a_2} \eta_0, \vec{\delta}_1 \wedge N_{\eta_0, \vec{\delta}_1} \# \mathit{inserted}(\vec{\delta}_2) \Rightarrow \mathit{dEnv} \vdash \eta'_0, \vec{\delta}_1; \text{EX}_2 \Mapsto \eta'_0, \vec{\delta}_1, \vec{\delta}_2; n$$

Now we can prove the three properties. Approximation is trivial. Immutability and equal results are less trivial.

Insert: Immutability. Immutability follows by Lemma 40 and by induction. Assume $(u_1|u_2|(r_2//*)) \#^{\cup} c$. This implies $u_1 \#^{\cup} c$ and $u_2 \#^{\cup} c$ and $(r_2//*) \#^{\cup} c$, hence $\eta_0 \sim_c \eta_0, \vec{\delta}_1$ and $\eta_0, \vec{\delta}_1 \sim_c \eta_0, \vec{\delta}_1, \vec{\delta}_2$ follow by induction and

$$(\eta_0, \vec{\delta}_1, \vec{\delta}_2) \sim_c (\eta_0, \vec{\delta}_1, \vec{\delta}_2, \mathsf{insert}(\vec{m}_c, F_c))$$

follows by Lemma 40. Hence: $\eta_0 \sim_c (\eta_0, \vec{\delta}_1, \vec{\delta}_2, \mathsf{insert}(\vec{m}_c, F_c))$ follows by transitivity.

Insert: equal results. We must prove that $\eta'_0 \sim_{a_1|a_2|(r_1//*)} \eta_0 \wedge N_{\eta'_0} \# \mathit{inserted}(\vec{\delta}_1, \vec{\delta}_2, \vec{\delta}_3)$ where $\vec{\delta}_3 = \mathsf{insert}(\vec{m}_c, F_c)$ imply $\mathit{dEnv} \vdash \eta'_0; \textbf{do insert } \text{EX}_1 \textbf{ into } \text{EX}_2 \Mapsto \eta'_0, \vec{\delta}'_1, \vec{\delta}'_2, \vec{\delta}'_3; ()$ with $\vec{\delta}'_1 = \vec{\delta}_1$, $\vec{\delta}'_2 = \vec{\delta}_2$, $\vec{\delta}'_3 = \vec{\delta}_3$. By Property 15 $N_{\eta'_0} \# \mathit{inserted}(\vec{\delta}_1, \vec{\delta}_2, \vec{\delta}_3)$ implies $N_{\eta'_0, \vec{\delta}_1} \# \mathit{inserted}(\vec{\delta}_2, \vec{\delta}_3)$ and $N_{\eta'_0, \vec{\delta}_1, \vec{\delta}_2,} \# \mathit{inserted}(\vec{\delta}_3)$.

The hypothesis $\eta'_0 \sim_{a_1|a_2|(r_1//*)} \eta_0$ implies $\eta'_0 \sim_{a_1} \eta_0$, hence, by induction, $N_{\eta'_0} \# \mathit{inserted}(\vec{\delta}_1, \vec{\delta}_2, \vec{\delta}_3)$ implies $\mathit{dEnv} \vdash \eta'_0; \text{EX}_1 \Mapsto \eta'_0, \vec{\delta}_1; \vec{m}$, which gives us $\vec{\delta}'_1 = \vec{\delta}_1$.

Similarly, we have that $\eta'_0 \sim_{a_2} \eta_0$, hence, by ($\sim$ stability), $\eta'_0, \vec{\delta}_1 \sim_{a_2} \eta_0, \vec{\delta}_1$, hence, by induction, $N_{\eta'_0, \vec{\delta}_1} \# \mathit{inserted}(\vec{\delta}_2, \vec{\delta}_3)$ implies $\mathit{dEnv} \vdash \eta'_0, \vec{\delta}_1; \text{EX}_2 \Mapsto \eta'_0, \vec{\delta}_1, \vec{\delta}_2; n$, which gives us $\vec{\delta}'_2 = \vec{\delta}_2$.

Recall: $\vec{\delta}'_3 = \mathsf{insert}(\vec{m}_c, F_c)$, where $(\vec{m}_c, F_c) \in \mathit{prepare\text{-}deep\text{-}copy}(\mathit{apply}(\eta_2), n, \vec{m})$ and $\vec{m}_c$ can be freely chosen from the infinite supply of nodes in $\mathcal{N}$ that have the correct parent/child relationship and that are not in $\mathit{apply}(\eta'_0, \vec{\delta}_1, \vec{\delta}_2)$.

We know that $N_{\eta'_0, \vec{\delta}_1, \vec{\delta}_2} \# \mathit{inserted}(\vec{\delta}_3)$, hence we can choose in $\vec{\delta}'_3$ the same $\vec{m}_c$ as in $\vec{\delta}_3$. Apart from the choice of the fresh nodes, *prepare-deep-copy* only depends on the content of the subtrees rooted in $\vec{m}$ in the store. By Lemma 41, these trees are identical in $\mathit{apply}(\eta_0, \vec{\delta}_1, \vec{\delta}_2)$ and in $\mathit{apply}(\eta'_0, \vec{\delta}_1, \vec{\delta}_2)$, thanks to the hypothesis $\eta'_0 \sim_{r_1//*} \eta_0$, which implies $\eta'_0, \vec{\delta}_1, \vec{\delta}_2 \sim_{r_1//*} \eta_0, \vec{\delta}_1, \vec{\delta}_2$ by ($\sim$ stability).

6. COMMUTATIVITY

Our analysis is meant as a tool to prove whether a pair of expressions can be evaluated in any order or, put differently, whether they *commute*. In this section we give a precise definition for that notion, and we show how our analysis provides a sufficient condition for commutativity.

6.1 Main result

Commutativity means that the order of evaluation of expressions in a given environment affects the order of the result, but not the returned items, nor the store.

Definition 43 (commutativity). Two expressions EX_1 and EX_2 *commute* in *pEnv*, written $\text{EX}_1 \overset{pEnv}{\longleftrightarrow} \text{EX}_2$, iff, for all η and *dEnv* such that $pEnv \supseteq_\eta dEnv$,

the following holds:

$$\begin{array}{l} dEnv \vdash \eta; \mathrm{EX}_1 \;\Mapsto\; \eta_1; \vec{v}_1 \\ dEnv \vdash \eta_1; \mathrm{EX}_2 \;\Mapsto\; \eta_2; \vec{v}_2 \\ \qquad \Rightarrow \quad dEnv \vdash \eta; \mathrm{EX}_2 \;\Mapsto\; \eta_1'; \vec{v}_2 \\ \qquad\quad\; dEnv \vdash \eta_1'; \mathrm{EX}_1 \;\Mapsto\; \eta_2'; \vec{v}_1 \\ \qquad\quad\; apply(\eta_2) = apply(\eta_2') \end{array}$$

The proof of the commutativity theorem is far easier than the proof of soundness, and is essentially independent on the actual definition of the equivalence relation. It only relies on soundness plus the "framework properties" collected in Lemma 35.

Theorem 44 (commutativity). *Consider two expressions and their analyses in pEnv:*

$$\begin{array}{l} pEnv \vdash \mathrm{EX}_1 \;\Mapsto\; r_1; \langle a_1, u_1 \rangle \\ pEnv \vdash \mathrm{EX}_2 \;\Mapsto\; r_2; \langle a_2, u_2 \rangle \end{array}$$

If the updates and accesses obtained by the analysis are independent then the expressions commute, in any environment that respects pEnv; formally:

$$u_1 \#^{\cup} a_2, u_2 \#^{\cup} a_1 \quad \Rightarrow \quad \mathrm{EX}_1 \overset{pEnv}{\longleftrightarrow} \mathrm{EX}_2$$

Proof. Here is the hypothesis:

Let
(h1-a) $dEnv \vdash \eta; \mathrm{EX}_1 \;\Mapsto\; \eta, \vec{\delta}_1; \vec{v}_1$
(h1-b) $pEnv \vdash \mathrm{EX}_1 \;\Mapsto\; r_1; \langle a_1, u_1 \rangle$
(h1-c) $pEnv \supseteq_{\eta} dEnv$
(h2-a) $dEnv \vdash \eta, \vec{\delta}_1; \mathrm{EX}_2 \;\Mapsto\; \eta, \vec{\delta}_1, \vec{\delta}_2; \vec{v}_2$
(h2-b) $pEnv \vdash \mathrm{EX}_2 \;\Mapsto\; r_2; \langle a_2, u_2 \rangle$
If
(h3) $u_1 \#^{\cup} a_2, u_2 \#^{\cup} a_1$,
then:
(t1) $dEnv \vdash \eta; \mathrm{EX}_2 \;\Mapsto\; \eta, \vec{\delta}_2; \vec{v}_2$
(t2) $dEnv \vdash \eta, \vec{\delta}_2; \mathrm{EX}_1 \;\Mapsto\; \eta, \vec{\delta}_2, \vec{\delta}_1; \vec{v}_1$
which implies
$dEnv \vdash \eta; \mathrm{EX}_2, \mathrm{EX}_1 \;\Mapsto\; \eta, \vec{\delta}_2, \vec{\delta}_1; \vec{v}_2, \vec{v}_1$
and also, by Properties 10 and 15:
$apply(\eta, \vec{\delta}_1, \vec{\delta}_2) = apply(\eta, \vec{\delta}_2, \vec{\delta}_1)$ and $merge(\eta, \vec{\delta}_1, \vec{\delta}_2) = merge(\eta, \vec{\delta}_2, \vec{\delta}_1)$.

Throughout this proof we use N_i for the set of nodes of store history η_i.
By Property15:
(disj-1) $inserted(\vec{\delta}_2) \# N_\eta$
(disj-2) $inserted(\vec{\delta}_1) \# (N_\eta, \vec{\delta}_2)$
By ($\supseteq$ Stability), (h1-c) implies:
(h2-c) $pEnv \supseteq_{\eta, \vec{\delta}_1} dEnv$
By $u_1 \#^{\cup} a_2$, (h1-abc) and immutability for EX_1 we have that
(h1-imm) $\eta \sim_{a_2} \eta, \vec{\delta}_1$

From (h1-imm), (h2-abc), (disj-1), and *equal results* for EX_2, we have:
(t1) $dEnv \vdash \eta; \text{EX}_2 \Mapsto \eta, \vec{\delta}_2; \vec{v}_2$

Now, we can apply immutability for EX_2 to (t1), (h2-b), (h1-c) and $u_2 \#^{\cup} a_1$, and obtain:
(h2-imm) $\eta \sim_{a_1} \eta, \vec{\delta}_2$

From property (h2-imm), (h1-abc), and (disj-2), we can apply *equal results* for EX_1 in order to transfer the (h1-a) reduction (η to $\eta, \vec{\delta}_1$) into a reduction from $\eta, \vec{\delta}_2$ to a $\eta, \vec{\delta}_2, \vec{\delta}_1$, and finally obtain
(t2) $dEnv \vdash \eta, \vec{\delta}_2; \text{EX}_1 \Mapsto \eta, \vec{\delta}_2, \vec{\delta}_1; \vec{v}_1$. □

7. THE ANALYSIS IN PRACTICE

In this section we illustrate the use and limitations of the path analysis through some examples. Our purpose here is not to provide a full treatment of optimization but to illustrate the role of commutativity in various optimization scenarios.

7.1 Join reordering

Consider the following snippet.

```
declare function getnewprojects() {
  for $n in /projects/project[new]
  return (do delete $n/new, $n)
};
for $p in getnewprojects()
for $t in /tasks/task
where $p/id = $t/proj
return starttask($p,$t)
```

This query extracts all project elements marked new and then searches for all task elements with an id that matches the project, and invokes the starttask external function for each combination of a project and a matching task. As a side effect, the function extracting the new projects deletes the marker flag.

We would like a compiler to generate a join plan for this query, in which the left branch, accessing the projects, performs the deletions. In order for join reordering rewritings to be sound, the optimizer must be able to check that the two branches of a join commute. In our example, that means deciding that the access and updates to the projects and to the tasks do not interfere.

To ascertain that this is safe we will apply the path analysis algorithm to the first two **for** clauses of the query and show that they commute. We assume that the query is first expanded to the core query language described in Section 3, expanding the getnewprojects function body to

```
for $n in
 (for $x in /projects/project return if ($x/new) then $x else ())
return
 (do delete $n/new, $n)
```

Below we have anotated this core query with the result of applying the path analysis algorithm. The annotations are written as triples, $\langle r; a; u \rangle$, consisting of the

returned path r, the accessed path a, and the updated path u, written right before the annotated expression. Since our example assumes only one root (with no new ones introduced by node constructors), and all paths are absolute, we omit the root from the paths.

```
1  ⟨/projects/project;
2   /projects/project/new;
3   /projects/project/new|/projects/project/new//∗⟩
4  (for $n in
5   (⟨/projects/project;/projects/project/new;()⟩
6    for $x in ⟨/projects/project;/projects/project;()⟩ /projects/project return
7     ⟨/projects/project;/projects/project/new;()⟩
8     if (⟨/projects/project/new;/projects/project/new;()⟩ $x/new)
9     then ⟨/projects/project;();()⟩ $x
10    else ⟨();();()⟩ ())
11  return
12   ⟨/projects/project;
13    /projects/project/new;
14    /projects/project/new|/projects/project/new//∗⟩
15  (⟨();/projects/project/new;/projects/project/new|/projects/project/new//∗⟩
16   do delete ⟨/projects/project/new;/projects/project/new;()⟩ $n/new,
17   ⟨/projects/project;();()⟩ $n))
```

The algorithm operates in a bottom-up fashion, *i.e.*, starting on the innermost path expression /projects/project (line 6). Two consecutive applications of the (Step) rule from Figure 2 yields the triple </projects/project;/projects|/projects/project;()>. For better readability, we simplify redundant paths whenever possible, based on Remark 31, hence we rewrite the a component as /projects/project, which is sound because /projects belongs to the prefixes of /projects/project.

This produces the triple </projects/project;/projects/project;()> reported in line 6. The returned path /projects/project is associated to variable $x.

We then analyze the **if**..**then**..**else**. First the condition at line 8 is analyzed, yielding the triple at line 8, through an application of the (Var) and (Step) rules. Lines 9 and 10 are analyzed next using (Var) and (Empty) respectively. The whole if expression is analyzed using the (Cond) rule, yielding the triple at line 7. The for expression on lines 6–10 is analyzed using the (For) rule, yielding the triple on line 5. Here we simplified again the a component, from /projects/project|/projects/project/new to /projects/project/new. The return path /projects/project is bound to variable $n.

We now jump to the next inner-most expression which is $n/new at line 16. We then apply the (Delete) rule to the do delete expression at the same line, which yields the triple at line 15. Note that this includes our first non-empty updated path. Next comes the second argument of the sequence operator, which is the $n variable on line 17. We can then compute the paths for the sequence operator on lines 16–17, which yields the triple on lines 12–14. Finally we can compute the paths for the whole expression, which yields the triple on lines 1–3.

The analysis for the other branch of the join is just

$\langle$/tasks/task;/tasks/task;()$\rangle$ /tasks/task

We can now check whether commutativity holds for the result of the analysis. We need to check the following conditions:

$$(/\text{projects/project/new}|/\text{projects/project/new}//*) \ \#^{\cup} \ (/\text{tasks/task})$$
$$() \ \#^{\cup} \ (/\text{projects/project/new})$$

Application of standard algorithms [Hammerschmidt et al. 2005] allows us to decide that those paths do not interfere, hence the two expressions commute and the optimizer may decide to reorder the join depending on, *e.g.*, relative cardinality of the resulting collections.

Clearly, this is only a simple example. However, many optimizations change the evaluation order of subexpressions and so need to rely on some form of commutativity analysis.

7.2 Limitations of the analysis

Although the previous example shows that the analysis can be quite precise in some cases, some specific limitations are worth pointing out. We believe one of the main limitations at this point is that our definition is based purely on structural distinctions but does not take value predicates into account. For example, let's assume that projects and tasks are distinguished through a value, as illustrated by the following variant of our join query,

```
declare function getnewprojects() {
  for $n in /objects[kind="project"][new]
  return (do delete $n/new, $n)
};
for $p in getnewprojects()
for $t in /objects[kind="task"]
where $p/id = $t/proj
return starttask($p,$t)
```

which results in the commutativity conditions

$$(/\text{object/new}|/\text{object/new}//*) \ \#^{\cup} \ (/\text{objects/kind})$$
$$() \ \#^{\cup} \ (/\text{object/kind}|/\text{object/new})$$

In this case we can, surprisingly, still decide commutativity since the deletion occurs on different children than the one used to select the kind of object. However, if we consider a different side effect, *e.g.*, an insert inside the object, we run into problems, as illustrated by the following query:

```
declare function getnewprojects() {
  for $n in /objects[kind="project"][new]
  return (do insert element1 started {} into $n, $n)
};
for $p in getnewprojects()
for $t in /objects[kind="task")]
where $p/id = $t/proj
return starttask($p,$t)
```

Here, unfortunately, the analysis does not keep track of the kind of element being inserted, and leads to the following path checks:

$$(/\mathsf{object}//*|1)\ \#^{\cup}\ (/\mathsf{objects}/\mathsf{kind})$$
$$()\ \#^{\cup}\ (1//*|/object/kind|/object/new)$$

Here the first two paths are not disjoint since the insert could very well add a new kind element.

Remark 45. One could think of two ways in which the analysis could be extended to cover such cases. The first approach would be to add support for predicates of the form (Ex = Li). A completely different approach, which may have additional benefits, would be to keep track of the 'type' of the inserted nodes (here the fact that the inserted element has name 'started'). This would result in the following paths checks:

$$(/\mathsf{object}/\mathsf{new}|/\mathsf{object}/\mathsf{started}|/\mathsf{object}/\mathsf{started}//*)\ \#^{\cup}\ (/\mathsf{objects}/\mathsf{kind})$$
$$()\ \#^{\cup}\ (/\mathsf{object}/\mathsf{kind}|/\mathsf{object}/\mathsf{new})$$

This would likely be a more complex, but interesting extension. Some design choices would have to be made regarding how deeply we keep track of the type, as well as how to support 'computed elements' in XQuery.

Finally, we would like to point out that we do not cover recursive functions, which are important for some applications. These could be analysed through classical fixed-point based techniques; however, this is beyond the scope of this paper.

Also, the analysis we propose limits itself to the child, descendant, parent and ancestor axes. The other axes can be approximated by a combination of those axes, as in Draper et al. [2006]. This is an intentional choice, which we believe strikes a good compromise between complexity and usefulness.

7.3 Limitations related to path intersection

An important practical limitation arises from the heavy use of the descendant axis in queries. Consider the following example in which we have replaced the child based navigation with descendant based navigation.

```
declare function getnewprojects() {
  for $n in //project[new]
  return (do delete $n/new, $n)
};
for $p in getnewprojects()
for $t in //task
where $p/id = $t/proj
return starttask($p,$t)
```

We cannot statically decide whether the two expressions commute in this case, because a *task* element may occur below a *new* element being deleted; this is consistent with the results of our path analysis, which leads to the following intersection check.

$$(//\mathsf{project}/\mathsf{new}|//\mathsf{project}/\mathsf{new}//*)\ \#^{\cup}\ (//\mathsf{task})$$

```
1  let $state := doc("S")/state return
2    for $request in $state/requests/request return
3    (
4       let $id := $request/id return
5          for $act in $request/act return
6             let $entry := element1 entry {} return
7             (
8                do insert ($id, $act/∗) into $entry,
9                do insert $entry into $state/log,
10               for $valve in $state/valves/valve return
11                  if (data($valve/id) = data($id)) then
12                     if (($entry/open and $valve/close) or ($entry/close and $valve/open)) then
13                     (do delete $valve/∗, do insert $entry/∗ into $valve)
14                     else ()
15                  else ()
16            ),
17       do delete $request
18   )
```

Fig. 3. XQueryU script to open and close valves.

This issue could be solved if the shape of the XML in question is constrained, for example by having schema information available. In this case, we could test for path intersection under a schema. In this case, commutativity would be proved if the schema ensures that *tasks* never occur below *projects*. Observe that the use of schema information to test for path intersection is quite orthogonal to our core issue, that is, the correct inference of accessed and updated paths.

7.4 A complex example

Let us look at a small but realistic example of how a language like XQueryU could be used in practice. Figure 3 shows a script that processes requests for opening and closing valves in some unspecified machine.

The idea of the script is to manipulate an XML document that represents the state of the valve subsystem, including unprocessed requests. Each request is equipped with an "id" (in the real applications presumably a timestamp or such), and we want the application to keep track of these. The script expects the "state document" to be available as **doc**("S") with data as exemplified in Figure 4, where the document (a) corresponds to an initial state where there are three pending requests of open and close actions, all four valves are currently closed, and the log is empty.

When invoked, the script proceeds to create a reference to the state element of the "S" document (line 1), iterate the rest of the script over all the requests (line 2), extract the id from the current request (line 4), iterate over each request action in the request (line 5), create a new entry element (line 6); this is assigned location 1 for future reference, insert a copy of the request id and the content of the action into the just created entry (line 8), insert a copy of the entire just created entry into the log fragment of the state (line 9), walk through the current state of the valves (line 10) and select the one of interest to the current action entry (line 11), replace the state of the valve with the contents of the entry (line 14), and finally,

```
<state>
  <requests>
    <request><id>x</id>
      <act><open>1</open></act>
      <act><open>3</open></act>
    </request>
    <request><id>y</id>
      <act><close>1</close></act>
      <act><close>2</close></act>
    </request>
    <request><id>z</id>
      <act><open>3</open></act>
    </request>
  </requests>
  <valves>
    <valve><close>1</close></valve>
    <valve><close>2</close></valve>
    <valve><close>3</close></valve>
    <valve><close>4</close></valve>
  </valves>
  <log/>
</state>
```

(a)

```
<state>
  <requests/>
  <valves>
    <valve><id>y</id><close>1</close></valve>
    <valve><id>y</id><close>2</close></valve>
    <valve><id>x</id><open>3</open></valve>
    <valve><close>4</close></valve>
  </valves>
  <log>
    <entry><id>x</id><open>1</open></entry>
    <entry><id>x</id><open>3</open></entry>
    <entry><id>y</id><close>1</close></entry>
    <entry><id>y</id><close>2</close></entry>
    <entry><id>z</id><open>3</open></entry>
  </log>
</state>
```

(b)

Fig. 4. Sample valve state documents.

delete each request when processed (line 19).

If the script is run on the data in Figure 4(a) then the result should be the updated document shown in Figure 4(b), where all the requests have been processed, erased, and recorded in the log, and the resulting state has valve number 3 open; in addition each valve has recorded the value of the id of the request that last modified it.

We would like to optimize this script by *caching* the entries corresponding to the requested actions and

—update the valves in a single pass, and

—produce the log fragment with a single update at the end.

(Without updates these would all be standard functional language operations.)

The result of analyzing the script is shown in Figure 5. We can use the result to rewrite the program.

The optimization would go something like this: first a separate loop is created that accumulates all the constructed entries (line 12), including the side effect to populate them, into a common variable, say *$entries*. Such a transformation is safe because the loop condition of the split loop (line 10 of Figure 5) does not overlap with the updated paths of the creation (line 12) or insertion into it (line 15).

With the entry creation separated out we can rearrange the loop by observing that the fragments of the analyzed code used to insert data into the log (lines 19–20) and update the valve list (lines 21–39) commute with each other and thus can be further separated into individual loops because they only *update* parts that are not observed elsewhere, and because individual iterations operate on disjoint 1 node instances. Finally, we can lift the deletion of the request elements up before

```
1  ⟨equal to line 3⟩
2  let ⟨"S"⟩$state := ⟨"S"; "S"; ()⟩doc(⟨(); (); ()⟩"S") return
3     ⟨equal to line 5⟩
4     for ⟨R⟩$request in ⟨R; R; ()⟩($state/requests/request) return
5        ⟨(); R/id//*|R/act/*//*|"S"/log|1//*|V/*|1/*//*; 1|1//*|"S"/log//*|V//*|R|R//*⟩ (lines 7+41)
6        (
7           ⟨equal to line 9⟩
8           let ⟨R/id⟩$id := ⟨R/id; R/id; ()⟩($request/id) return
9              ⟨equal to line 11⟩
10             for ⟨R/act⟩$act in ⟨R/act; R/act; ()⟩($request/act) return
11                ⟨(); R/id//*|R/act/*//*|"S"/log|1//*|V/*|1/*//*; 1|1//*|"S"/log//*|V//*⟩
12                let ⟨1⟩$entry := ⟨1; (); 1⟩(element1 entry {}) return
13                   ⟨(); R/id//*|R/act/*//*|"S"/log|1//*|V/*|1/*//*; 1//*|"S"/log//*|V//*⟩ (lines 15+19+21)
14                   (
15                      ⟨(); R/id//*|R/act/*//*; 1//*⟩
16                      do insert
17                         ⟨R/id|R/act/*; R/act/*; ()⟩(⟨R/id; (); ()⟩$id, ⟨R/act/*; R/act/*; ()⟩($act/*))
18                         into ⟨1; (); ()⟩$entry,
19                      ⟨(); "S"/log|1//*; "S"/log//*⟩
20                      do insert ⟨1; (); ()⟩$entry into ⟨"S"/log; "S"/log; ()⟩($state/log),
21                      ⟨(); V/*|R/id|1/*//*; V//*⟩
22                      for ⟨V⟩$valve in ⟨V; V; ()⟩($state/valves/valve) return
23                         ⟨(); V/*|R/id|1/*//* (note 1); V//*⟩
24                         if ⟨(); V/id|R/id; ()⟩(
25                               ⟨(); V/id; ()⟩(data(⟨V/id; V/id; ()⟩($valve/id)))
26                               = ⟨(); R/id; ()⟩(data(⟨R/id; (); ()⟩$id)))
27                         then
28                            ⟨(); V/*|1/*//* (note 2); V//*⟩
29                            if ⟨(); V/close|V/open|1/close|1/open; ()⟩
30                               (($entry/open and $valve/close) or ($entry/close and $valve/open))
31                            then
32                               ⟨(); V/*|1/*//*; V//*⟩(
33                                  ⟨(); V/*; V//* (note 3)⟩
34                                  do delete ⟨V/*; V/*; ()⟩($valve/*),
35                                  ⟨(); 1/*//*; V//*⟩
36                                  do insert ⟨1/*; 1/*; ()⟩($entry/*) into ⟨V; () ()⟩$valve
37                               )
38                            else ()
39                         else ()
40                   ),
41          ⟨(); (); R|R//*⟩
42          do delete $request
43       )
```

where R = "S"/requests/request, V = "S"/valves/valve, and we made the following non-trivial simplifications:
(1) line 23, by Lemma 32: V/id|V/* ⇒ V/*,
(2) line 28, by Lemma 32: V/*|V/open|V/close ⇒ V/*, and 1/*//*|1/open|1/close ⇒ 1/*//*;
(3) line 33, by ⟦V/*|V/*//*⟧ = ⟦V//*⟧: V/*|V/*//* ⇒ V//*

Fig. 5. Path analysis of valve script.

```
let $state := doc("S") return
(  let $entries :=
     for $request in $state//request return
     ( let $id := $request/id return
         for $act in $request/act return
           let $entry := element1 entry {} return
           ( do insert ($id, $act/∗) into $entry,
             $entry),
       do delete $request),
   return
   (  for $valve in $state/valves/valve return
        for $entry in $entries return
          if (data($valve/id) = data($entry/id)) then
           if (($entry/open and $valve/close) or ($entry/close and $valve/open)) then
           ( do delete $valve/∗,
             do insert $entry/∗ into $valve )
           else ()
          else (),
      for $entry in $entries return do insert $entry into $state/log
   )
)
```

Fig. 6. Valve script after optimizations.

the creation of valve entries, because the simplified loop does not access the request nodes itself. The optimized script is shown in Figure 6, and has the nice property that the output is produced in document order of the result document, which could further improve its efficiency.

8. CONCLUSION

In this paper, we have proposed a conservative approach to detect whether two expressions commute in XQueryU, an expressive XML update language with strict evaluation order and immediate update application. The approach relies on a form of *path analysis* which computes an upper bound for the nodes accessed or updated by an expression. As there is a growing need to extend XML languages with imperative features [Carey et al. 2006; Ghelli et al. 2006; Cooper et al. 2006], we believe the kind of analysis we propose here will be crucial for the long-term development of those languages. The development here is essentially formal, and the proposed approach needs to be validated against practical usage scenarios. We are currently working on an implementation of the approach in a popular XQuery engine, and investigating the changes needed to real-life compilers to support the kind of side-effects proposed in recent XQuery-based languages.

Acknowledgments. The anonymous reviewers have been so helpful that we may almost list them as coauthors. Thanks for that.

REFERENCES

Benedikt, M., Bonifati, A., Flesca, S., and Vyas, A. 2005a. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P'05*.

Benedikt, M., Bonifati, A., Flesca, S., and Vyas, A. 2005b. Verification of tree updates for optimization. In *CAV*. 379–393.

Benedikt, M., Fan, W., and Kuper, G. M. 2005. Structural properties of XPath fragments. *Theor. Comput. Sci. 336,* 1, 3–31.

Biron, P. V. and Malhotra, A. 2001. XML schema part 2: Datatypes. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502`.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. 2007. XQuery 1.0: An XML query language.

Bray, T., Hollander, D., and Layman, A. 1999. Namespaces in XML. W3C recommendation, World Wide Web Consortium. Jan. `http://www.w3.org/TR/REC-xml-names`.

Carey, M., Chamberlin, D., Florescu, D., and Robie, J. 2006. Programming with XQuery. Draft submitted for publication.

Chamberlin, D., Florescu, D., Melton, J., Robie, J., and Siméon, J. 2007. XQuery update facility. W3C Working Draft.

Cooper, E., Lindley, S., Wadler, P., and Yallop, J. 2006. Links: Web programming without tiers. Unpublished Manuscript.

Dekeyser, S., Hidders, J., and Paredaens, J. 2003. Instance indepedent concurrency control for semistructured databases. In *SEBD2003*. 323–334.

Dekeyser, S., Hidders, J., and Paredaens, J. 2004. A transaction model for xml databases. *World Wide Web 7,* 1, 29–57.

Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., and Wadler, P. 2006. XQuery 1.0 and XPath 2.0 formal semantics, W3C Recommendation.

Elkan, C. 1990. Independence of logic database queries and updates. In *PODS*. 154–160.

Fernández, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N. 2007. XQuery 1.0 and XPath 2.0 data model.

Ghelli, G., Ré, C., and Siméon, J. 2006. XQuery!: An XML query language with side effects. In *DataX Workshop*. Lecture Notes in Computer Science. Munich, Germany.

Ghelli, G., Rose, K., and Siméon, J. 2007. Commutativity analysis in xml update languages. In *Proceedings of International Conference on Database Theory (ICDT)*. Barcelona, Spain.

Hammerschmidt, B. C., Kempa, M., and Linnemann, V. 2005. On the Intersection of XPath Expressions. In *Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS 2005)*. Montreal, Canada, 49–57.

Hidders, J., Paredaens, J., and Vercammen, R. 2006. On the expressive power of xquery-based update languages. In *XSym*. 92–106.

Lanin, V. and Shasha, D. 1986. A symmetric concurrent b-tree algorithm. In *FJCC*. 380–389.

Lehti, P. 2001. Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit.

Levy, A. Y. and Sagiv, Y. 1993. Queries independent of updates. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*. Morgan Kaufmann, 171–181.

Marian, A. and Siméon, J. 2003. Projecting XML documents. In *Proceedings of International Conference on Very Large Databases (VLDB)*. Berlin, Germany, 213–224.

Miklau, G. and Suciu, D. 2004. Containment and equivalence for a fragment of XPath. *J. ACM 51,* 1, 2–45.

Tatarinov, I., Ives, Z., Halevy, A., and Weld, D. 2001. Updating XML. In *SIGMOD*.

Tofte, M. and Talpin, J.-P. 1997. Region-based memory management. *Inf. Comput. 132,* 2, 109–176.

Wadler, P. 1999. Two semantics of XPath. Discussion note for W3C XSLT Working Group.

Received July 2007; revised February 2008; accepted April 2008