RC24505 (W0802-110) February 28, 2008
Computer Science

IBM Research Report

Commutativity Analysis in XML Update Languages

Giorgio Ghelli
Università di Pisa
Dipartimento di Informatica
Via Buonarroti 2
I-56127 Pisa
Italy

Kristoffer Rose, Jérôme Siméon
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Commutativity Analysis in XML Update Languages

Giorgio Ghelli[1], Kristoffer Rose[2], and Jérôme Siméon[2]

[1] Università di Pisa, Dipartimento di Informatica
Via Buonarroti 2, I-56127 Pisa, Italy
ghelli@di.unipi.it
[2] IBM T.J. Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598, U.S.A.
krisrose and simeon@us.ibm.com

Abstract. A common approach to XML updates is to extend XQuery with update operations. This approach results in very expressive languages which are convenient for users but are difficult to reason about. Deciding whether two expressions can commute has numerous applications from view maintenance to rewriting-based optimizations. Unfortunately, commutativity is undecidable in most recent XML update languages. In this paper, we propose a conservative analysis for an expressive XML update language that can be used to determine whether two expressions commute. The approach relies on a form of path analysis that computes upper bounds for the nodes that are accessed or modified in a given update expression. Our main result is a commutativity theorem that can be used to identify commuting expressions.

1 Introduction

Most of the proposed XML updates languages [1–5] extend a full-fledged query language such as XQuery [6] with update primitives. To simplify specification and reasoning, some of the first proposals [1, 2, 4] 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, and limits the expressiveness in a way that is not always acceptable for applications. For that reason, more recent proposals [5, 7] give the ability to apply updates in the course of query evaluation. Such languages typically rely on 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 $doc/country return insert {<new/>} into {$x},
count($doc/country/new)
```

Such an example cannot be written in a language based on a snapshot semantics, as the `count` would always return zero. However, it can be written in the XQuery! [5] or the XQueryP [7] proposals, which both rely on an explicit left-to-right evaluation order. Still, such a semantics severely restricts the optimizer's ability for rewritings, unless the optimizer is able to decide that some pairs of expressions commute.

Deciding commutativity, or more generally whether an update and a query *interfere*, has numerous applications, including optimizations based on algebraic 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 interfere, i.e., whether they can be safely commuted or not. Our technique relies on an extension of the path analysis proposed in [8] 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 [9, 10].

Our commutativity analysis serves a similar purpose to independence checking in the relational context [11, 12]. To the best of our knowledge, our work is the first to study such issues in the XML context, where languages are typically much more expressive. A simpler form of static analysis is proposed in [4, 13], suggesting that similar techniques can be used to optimize languages with a snapshot semantics. Finally, commutativity of tree operations is used in transactional models [14, 15], but relies on run-time information while our purpose is static detection.

Problem and examples. In the rest of the paper, we focus on a simple XQuery extension with insertion and deletion operations. The syntax and semantics of that language is essentially that of [5], with updates applied immediately. This language is powerful enough to exhibit the main problems related to commutativity analysis, yet simple enough to allow a complete formal treatment within the space available for this paper. Here are some sample queries and updates in that language.

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
```
delete {$doc/wines/california}
```

U2
```
for $x in $doc/country return
  insert {<new/>} into {$x}
```

U3
```
for $x in
  $doc/country[population < 24]
return
  delete {$x/city}
```

Some of those 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 nodes being inserted. Deciding whether the set of nodes accessed or modified are disjoint quickly becomes 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.

Approach. We rely on a form of abstract interpretation that approximates the set of nodes processed by a given expression. The 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 result of the analysis must make disjointness decidable. In the context of XML updates, *paths* are a natural choice for the approximation of the nodes being accessed or updated, and they satisfy the precision and decidability requirements.

Contributions. The path analysis itself is a relatively intuitive extension of [8] to handle update operations. However, coming up with a sound analysis turns out to be a hard problem for a number of reasons. First of all, we use paths to denote sets of accessed nodes, but the forthcoming updates will change the nodes denoted by the paths that are being accumulated. We need 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 introduce a store-based formalization of the XML data model and a notion of store history that allows us to talk about the effect of each single update and to solve the stability issue. Another challenge is to find a precise definition of which nodes are actually used or updated by a query. For instance, one may argue that **U3** only modifies nodes reached by the path *country/city*. 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 our analysis, this is kept into account by actually inserting into the updated paths of **U3** all the descendants of the deleted expression *country/city*, as detailed in the table below.

U3	accessed paths:	**Q3**	accessed paths:
	`$doc/country`		`$doc//country`
	`$doc/country/population`		`$doc//country//name`
	`$doc/country/city`		
	updated paths:		updated paths:
	`$doc/country/city/descendant-or-self::*`		

In **Q4**, if the returned expression *$x/new/../very_new* were just associated to the path *country/new/../very_new*, the interference with **U2** would not be observed, since the path *country/new/descendant-or-self::**::∗ updated by **U2** refers to a disjoint set of nodes. Hence, the analysis must also consider the nodes traversed by the evaluation of *$x/new/../very_new*, which correspond to the path *country*|*country/new*|*country/new/..*, whose second component intersects with *country/new/descendant-or-self::**::∗. The main contributions of the paper are as follows:

- We propose a form of static analysis that infers paths to the nodes that are *accessed* and *modified* by an expression in that language;
- We present a formal definition of when such an analysis is sound, based on a notion of *store history equivalence*; this formal definition provides a guide for the definition of the inference rules;
- We show the soundness of the proposed path analysis;
- We prove a commutativity theorem, that provides a sufficient condition for the commutativity of two expressions, based on the given path analysis.

Organization. The rest of the paper is organized as follows. Section 2 presents the XML data model and the notion of store history. Section 3 reviews the update language syntax and semantics. Section 4 presents the path analysis and the main soundness theorem. Section 5 presents the commutativity theorem. Section 6 reviews related work, and Section 7 concludes the paper. For space reasons, proofs for the analysis soundness and for the commutativity theorem are provided separately in the extended version of this paper [16].

2 A Store for Updates

We define here the notions of *store* and *store history*, which are used to represent the effect of XML updating expressions. Our store is a simplification of the XQuery Data

Model [17] to the parts that are most relevant to our path analysis. In this formalization we ignore sibling order, since it has little impact on the approach and on the analysis precision.

2.1 The Store

We assume the existence of disjoint infinite sets of *node ids*, $\mathcal{N}$, the *node kinds*, $\mathcal{K} = \{\texttt{element}, \texttt{text}\}$, *names*, $\mathcal{Q}$, and possible *textual content*, $\mathcal{T}$. A node *location* is used to identify where a document or an XML fragment originates from; it is either a URI or a unique code-location identifier: *loc* ::= *uri* | *code-loc*.

A *uri* typically corresponds to the URI associated to a document and a *code-loc* is used to identify document fragments generated during query evaluation by an element constructor. Now we are ready to define our basic notion of store.

Definition 1 (Store). *A store σ is a quadruple (N,E,R,F) where $N \subset \mathcal{N}$ contains the set of nodes in the document, $E \subset N \times N$ contains the set of edges, $R\colon N \rightarrow loc$ is a partial function mapping some nodes to their location, and the node description $F = (\texttt{kind}_F, \texttt{name}_F, \texttt{content}_F)$ is a triple of partial functions where $\texttt{kind}_F\colon N \rightarrow \mathcal{K}$ maps each node to its kind, $\texttt{name}_F\colon N \rightarrow \mathcal{Q}$ maps nodes to their name (if any), and $\texttt{content}_F\colon N \rightarrow \mathcal{T}$ maps nodes to their text content (if any).*

We use N_σ, E_σ, R_σ, F_σ to denote the N,E,R,F component of σ. When $(m,n) \in E$, we say that m is a parent of n and n is a child of m. A “root” is a node that has no parent.

Finally a store must be “well-formed”: (1) all nodes mapped by R must be root nodes, (2) every non-root node must be the child node of exactly one parent node, (3) the transitive closure E^+ of E must be irreflexive (4) element nodes must have a name and no content; and (5) text nodes must have no name and no children but do have content.

In what follows, every store operation preserves store well-formedness.

2.2 Accessing and updating the store

We assume the standard definitions for the usual accessors (parent, children, descendants, ancestors, name, text-content...), and focus on operations that modify the store (insert,delete, and node creation).[3] We define a notion of *atomic update record*, which captures the dynamic information necessary for each update, notably allowing the update to be re-executed on a store, using the *apply* operation defined below.

Definition 2 (Atomic update records). *Atomic update records are terms with the following syntax:*

$$\texttt{create}(\bar{n},F) \;\mid\; \texttt{R-insert}(n,loc) \;\mid\; \texttt{insert}(E) \;\mid\; \texttt{delete}(\bar{n})$$

Definition 3 (Atomic update application). *The operation apply(σ,u) returns a new store as detailed below, but fails when the listed preconditions do not hold. $\perp$ denotes undefined.*

[3] Note that replace is trivial to add to the framework.

- *$apply(\sigma, \texttt{create}(\bar{n}, F'))$ adds $\bar{n}$ to N and extends F with F'.*
  Preconditions: $\bar{n}$ disjoint from N. $(\bar{n}, (), (), F')$ is a well-formed store.
- *$apply(\sigma, \texttt{R-insert}(n, loc))$ extends R with $n \to loc$.*
  Preconditions: n is a root node and $R(n_c) = \perp$.
- *$apply(\sigma, \texttt{insert}(E'))$ extends E with E'.*
  Preconditions: for each $(n_p, n_c) \in E'$, n_c has no parent in $E \cup E' \setminus \{(n_p, n_c)\}$, and $R(n_c) = \perp$. The transitive closure of $E \cup E'$ is irreflexive.
- *$apply(\sigma, \texttt{delete}(\bar{n}))$ deletes each edge $(n_p, n_c) \in E$ where $n_c \in \bar{n}$.*
  Preconditions: $\bar{n} \subseteq N$.

Definition 4 (Composite updates). *A composite update, Δ, is an ordered sequence of atomic updates: $\Delta \equiv (u_1, \ldots, u_n)$. $apply(\sigma, \Delta)$ denotes the result of applying $u_1 \ldots u_n$ on store σ, in this order.*

We use *created*(Δ) to denote the set of nodes created by Δ. A composite update Δ *respects creation time* iff, for any $\Delta_1, \Delta_2 = \Delta$, no node in *created*$(\Delta_2)$ appears in Δ_1. Hereafter we will always assume that we only work with such Δ's.

Finally, the notion of *updated*(Δ_1) gives a sufficient condition for non-interference ($S\#T$ means that S and T are disjoint).

Definition 5 (Update target). *The* update target *of each update operation is defined as*

$$\begin{array}{ll} updated(\texttt{create}(\bar{n}, F)) & =_{def} \{\} \\ updated(\texttt{R-insert}(n, loc)) & =_{def} \{\} \\ updated(\texttt{insert}(E)) & =_{def} \{n_c \mid (n_p, n_c) \in E\} \\ updated(\texttt{delete}(\bar{n})) & =_{def} \bar{n} \end{array}$$

Property 1. If Δ_1, Δ_2 and Δ_2, Δ_1 both respect creation time, then

$$updated(\Delta_1)\#updated(\Delta_2) \;\Rightarrow\; \text{apply}(\sigma, (\Delta_1, \Delta_2)) = \text{apply}(\sigma, (\Delta_2, \Delta_1))$$

Intuitively, provided that creation time is respected, the only two operations that do not commute are `insert`(n_p, n_c) and `delete`(n_c). Any other two operations either do not interfere at all or they fail in whichever order are applied, as happens for any conflicting `R-insert-R-insert`, `R-insert-insert`, or `insert-insert` pair.

2.3 Store History

Finally, we introduce a notion of store history, as a pair $(\sigma, (u_1, \ldots, u_n))$. 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 $Expr_1, Expr_2$ by analysing the histories $(\sigma, (\Delta_1, \Delta_2))$ and $(\sigma, (\Delta'_2, \Delta'_1))$ produced by their evaluations in different orders, and by proving that, under some conditions, $\Delta_1 = \Delta'_1$ and $\Delta_2 = \Delta'_2$.

Definition 6 (Store history). *A store history $\eta = (\sigma_\eta, \Delta_\eta)$ is a pair formed by a store and a composite update.*

A store history (σ,Δ) can be mapped to a plain store either by $\text{apply}(\sigma,\Delta)$ or by applying *no-delete*(Δ) only, which is the Δ without any deletion. The second mapping $(\text{mrg}((\sigma,\Delta)))$ will be crucial to capture the degree of approximation that store dynamicity imposes over our static analysis.

$$\begin{aligned}\text{apply}((\sigma,\Delta)) &=_{\text{def}} \text{apply}(\sigma,\Delta)\\ \text{mrg}((\sigma,\Delta)) &=_{\text{def}} \text{apply}(\sigma,\textit{no-delete}(\Delta))\end{aligned}$$

By abuse of notation we shall (1) implicitly interpret σ as $(\sigma,())$; (2) extend accessors to store histories using the convention that, for any function defined on stores, $f(\eta) =_{\text{def}} f(\text{apply}(\eta))$; (3) when $\eta = (\sigma,\Delta)$ then write $\eta,\Delta' =_{\text{def}} (\sigma,(\Delta,\Delta'))$. We define history difference $\eta \setminus \eta'$ as follows: $(\sigma,(\Delta,\Delta')) \setminus (\sigma,\Delta) =_{\text{def}} \Delta'$.

Definition 7 (Well-formed History). *A history η is well-formed (wf(η)), if mrg(η) and apply(η) are both defined.*

3 Update language

The language we consider is a cut-down version of XQuery! [5] 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. The language has the following syntax; we will use the usual abbreviations for the parent ($\mathbf{p}/..$), child ($\mathbf{p}/\textit{name}$), and descendant ($\mathbf{p}//\textit{name}$) axes. We assume that *code-loc* (See Section 2) is generated beforehand by the compiler.

$$\begin{aligned}\textit{Expr} ::= {} & \$x \mid \textit{Expr}/\textit{axis}::\textit{ntest} \mid \textit{Expr},\textit{Expr} \mid \textit{Expr} = \textit{Expr}\\ & \mid \texttt{let}\ \$x\ \texttt{:=}\ \textit{Expr}\ \texttt{return}\ \textit{Expr} \mid \texttt{for}\ \$x\ \texttt{in}\ \textit{Expr}\ \texttt{return}\ \textit{Expr}\\ & \mid \texttt{if}\ (\textit{Expr})\ \texttt{then}\ \textit{Expr}\ \texttt{else}\ \textit{Expr} \mid \texttt{delete}\ \{\textit{Expr}\}\\ & \mid \texttt{insert}\ \{\textit{Expr}_1\}\ \texttt{into}\ \{\textit{Expr}\} \mid \texttt{element}_{\textit{code-loc}}\{\textit{Expr}\}\{\textit{Expr}\}\\ \textit{axis} ::= {} & \textit{child} \mid \textit{descendant} \mid \textit{parent} \mid \textit{ancestor}\\ \textit{ntest} ::= {} & \textit{text()} \mid \textit{node()} \mid \textit{name} \mid *\end{aligned}$$

The main semantic judgement "$\textit{dEnv} \vdash \eta_0;\textit{Expr} \Rightarrow \eta_1;\bar{n}$" specifies that the evaluation of an expression *Expr*, with respect to a store history η_0 and to a dynamic environment *dEnv* that associates a value to each variable free in *Expr*, produces a value $\bar{n}$ and extends η_0 to $\eta_1 = \eta_0,\Delta$. A value is just a node sequence $\bar{n}$; textual content may be accessed by a function f, but we otherwise ignore atomic values, since they are ignored by path analysis. In an implementation, we would not manipulate the history η_0 but the store $\text{apply}(\eta_0)$, 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.

As an example, we present here the rule for insert expressions; the complete semantics can be found in [16]. Let $\bar{n}_d$ be the descendants-or-self of the nodes in $\bar{n}$. Insert-into uses *prepare-deep-copy* to identify a fresh node $m_i \in \bar{m}_d$ for each node in $\bar{n}_d$, while E_{copy} and F_{copy} reproduce for $E_{\text{apply}(\eta_2)}$ and $F_{\text{apply}(\eta_2)}$ for $\bar{m}_d$, and $\bar{m}$ is the subset of

$\bar{m}_d$ that corresponds to $\bar{n}$. Hence, $\texttt{create}(\bar{m}_d, F_{\text{copy}}), \texttt{insert}(E_{\text{copy}})$ copy $\bar{n}$ and their descendants, while $\texttt{insert}(\{n\} \times \bar{m})$ links the copies of $\bar{n}$ to n. Notice how the rule only depends on apply(η_2), not on the internal structure of η_2.

$$\frac{\begin{array}{c} dEnv \vdash \eta_0; Expr_1 \Rightarrow \eta_1; \bar{n} \\ dEnv \vdash \eta_1; Expr_2 \Rightarrow \eta_2; n \\ (\bar{m}, \bar{m}_d, E_{\text{copy}}, F_{\text{copy}}) = \textit{prepare-deep-copy}(\text{apply}(\eta_2), \bar{n}) \\ \eta_3 = \eta_2,\ \texttt{create}(\bar{m}_d, F_{\text{copy}}), \texttt{insert}(E_{\text{copy}}), \texttt{insert}(\{n\} \times \bar{m}) \end{array}}{dEnv \vdash \eta_0; \texttt{insert \{}\mathit{Expr}_1\texttt{\} into \{}\mathit{Expr}_2\texttt{\}} \Rightarrow \eta_3; ()}$$

It is easy to prove that, whenever $dEnv \vdash \eta_0; Expr \Rightarrow \eta_1; \bar{n}$ holds and η_0 is well-formed, then η_1 is well-formed as well.

4 Path analysis

4.1 Paths and prefixes

We now define the notion of paths that is used in our static analysis. Observe that the paths used by the analysis are not the same as the paths in the target language. For example, they are rooted in a different way, and the steps need not coincide: if we added order to the store, we could add a following-sibling axis to the language, but approximate it with *parent::*/child::* in the analysis.

Definition 8 (Static paths). Static paths*, or simply* paths*, are defined as follows.*

$$\mathbf{p} ::= () \mid loc \mid \mathbf{p}_0|\mathbf{p}_1 \mid \mathbf{p}/axis::ntest$$

where axis denotes any of the axes in the grammar.

Note that paths are always rooted at a given location. In addition, the particular fragment chosen here is such that important operations, notably intersection, can be checked using known algorithms [9, 10].

Definition 9 (Path Semantics). *For a path* $\mathbf{p}$ *and store* σ*,* $[\![\mathbf{p}]\!]_\sigma$ *denotes the set of nodes selected from the store by the path with the standard semantics [18] except that order is ignored, and* R_σ *is used to interpret the locations loc. The following concepts are derived from the standard semantics:*

Inclusion. *A path* $\mathbf{p}_1$ *is included in* $\mathbf{p}_2$*, denoted* $\mathbf{p}_1 \subseteq \mathbf{p}_2$*, iff* $\forall\sigma\colon [\![\mathbf{p}_1]\!]_\sigma \subseteq [\![\mathbf{p}_2]\!]_\sigma$.

Disjointness. *Two paths* $\mathbf{p}_1, \mathbf{p}_2$ *are disjoint, denoted* $\mathbf{p}_1 \# \mathbf{p}_2$*, iff* $\forall\sigma\colon [\![\mathbf{p}_1]\!]_\sigma \cap [\![\mathbf{p}_2]\!]_\sigma = \emptyset$.

Prefixes. *For each path* $\mathbf{a}$ *we define* $\textit{pref}(\mathbf{a})$ *as follows.*

$\mathbf{a}$	loc	$\mathbf{p}/axis::ntest$	$\mathbf{p}\vert\mathbf{q}$
$\textit{pref}(\mathbf{a})$	$\{loc\}$	$\{\mathbf{p}/axis::ntest\} \cup \textit{pref}(\mathbf{p})$	$\{\mathbf{p}\vert\mathbf{q}\} \cup \textit{pref}(\mathbf{p}) \cup \textit{pref}(\mathbf{q})$

Prefix Closure. *For a path* $\mathbf{a}$ *we write* $\textit{prefclosed}(\mathbf{a})$ *iff* $\forall\mathbf{p}\colon \mathbf{p} \in \textit{pref}(\mathbf{a}) \Rightarrow \mathbf{p} \subseteq \mathbf{a}$.

The *prefixes* of a path are all its initial subpaths, and a path is prefix-closed when it includes all of its prefixes. For example, the paths $/a//b \mid /a \mid /a//b/c$ and $/* \mid /a/b$ are both prefix-closed (the latter because $/a \subseteq /*$).

4.2 The meaning of the analysis

Definition 10 (Path analysis). *Given an expression Expr and a path environment* **pEnv** *which is a mapping from variables to paths, our path-analysis judgment*

$$\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\ \langle \mathbf{a}, \mathbf{u} \rangle$$

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

The **r** path is not actually needed to check commutativity, but is used to infer **u** and **a** for those expression that update, or access, their argument.

There are many reasonable ways to interpret which nodes are “returned” and “accessed” by an expression. For example, a path $\$x//a$ only returns the $\$x$ descendants with an a name but, in a naive implementation, may access every descendant of $\$x$. Deciding what is “updated” is even trickier. This definition should be as natural as possible, should allow for an easy computation of a static approximation and, above all, should satisfy the following property: if what is accessed by Expr_1 is disjoint from what is accessed or updated by Expr_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: an analysis is sound if $\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\ \langle \mathbf{a}, \mathbf{u} \rangle$ and $\mathit{dEnv} \vdash \eta_0; \mathit{Expr} \Rightarrow \eta_1; \bar{n}$ imply that $\bar{n} \subseteq [\![\mathbf{r}]\!]_{\mathrm{apply}(\eta_1)}$. Unfortunately, this is simplistic. Consider the following example:

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

Our rules bind a path $\mathrm{u1}/a$ to $\$x$, and finally deduce a returned path $\mathrm{u1}/a/b$ for the expression above. However, after *delete($x)*, the value of $\$x/b$ is not in $[\![\mathbf{p}]\!]_{\mathrm{apply}(\eta)}$ anymore; the best we can say it is that it is still in $[\![\mathbf{p}]\!]_{\mathrm{mrg}(\eta)}$. 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 $\bar{n} \subseteq [\![\mathbf{r}]\!]_{\mathrm{mrg}(\eta_1)}$, which is weaker than $\bar{n} \subseteq [\![\mathbf{r}]\!]_{\mathrm{apply}(\eta_1)}$ but is stable; we also generalize the notion to environments.

Definition 11 (Approximation). *A path* **p** *approximates a value* $\bar{n}$ *in the store history* η*, denoted* $\mathbf{p} \supseteq_\eta \bar{n}$*, iff* $\bar{n} \subseteq [\![\mathbf{p}]\!]_{mrg(\eta)}$.

A path environment **pEnv** *approximates a dynamic environment dEnv in a store history* η*, denoted* $\mathbf{pEnv} \supseteq_\eta \mathit{dEnv}$*, iff*

$$(\$x \mapsto \bar{n}) \in \mathit{dEnv} \ \Rightarrow\ \exists \mathbf{b}.\ (\$x \mapsto \mathbf{b}) \in \mathbf{pEnv}\ \mathit{and}\ \mathbf{b} \supseteq_\eta \bar{n}$$

Thanks to this “merge” interpretation, a path denotes all nodes that are reached by that path, or were reached by the path in some past version of the current history. This approximation has little impact, because the merge interpretation of a history is still a well-formed store, where every node has just one parent and one name, hence the usual algorithms can be applied to decide path disjointness.

The approach would break if we had, for example, the possibility of moving a node from one parent to another. Formally, $\mathrm{mrg}(\eta)$ may now contain nodes with two parents. In practice, one could not deduce, for example, that $(a/d)\#(b/c/d)$, because $\$x/a/d$ and $\$x/b/c/d$, if evaluated at different times, may actually return the same node, because its parent was moved from $\$x/a$ to $\$x/b/c$ in the meanwhile. Similarly, if nodes could be renamed, then node names would become useless in the process of checking path disjointness.

The commutativity theorem in Section 5 is based on the following idea: assume that $Expr_1$ transforms η_0 into (η_0,Δ) and only modifies nodes reachable through a path $\mathbf{u}$, while $Expr_2$ only depends on nodes reachable through $\mathbf{a}$, such that $\mathbf{u}\#\mathbf{a}$. Because $Expr_1$ only modifies nodes in $\mathbf{u}$, the histories η_0 and (η_0,Δ) are "the same" with respect to $\mathbf{a}$, hence we may evaluate $Expr_2$ either before or after $Expr_1$.

This is formalized by defining a notion of history equivalence wrt a path $\eta \sim_{\mathbf{p}} \eta'$, and by proving that the inferred $\mathbf{a}$ and $\mathbf{u}$ and the evaluation relation are related by the following soundness properties.

Parallel evolution from a-equivalent stores, first version:
$\eta'_0 \sim_{\mathbf{a}} \eta_0$ and $dEnv \vdash \eta_0; Expr \Rightarrow (\eta_0,\Delta);\bar{n}$
imply $dEnv \vdash \eta'_0; Expr \Rightarrow (\eta'_0,\Delta);\bar{n}$, i.e. the same $\bar{n}$ and Δ are produced.

Immutability out of u, first version:
$\forall \mathbf{c}\colon \mathbf{c}\#\mathbf{u}$ and $dEnv \vdash \eta_0; Expr \Rightarrow (\eta_0,\Delta);\bar{n}$
imply $\eta_0 \sim_{\mathbf{c}} (\eta_0,\Delta)$.

To define the right notion of path equivalence, consider the Comma rule

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash Expr_1 \Rightarrow \mathbf{r}_1;\ \langle \mathbf{a}_1,\mathbf{u}_1\rangle \\ \mathbf{pEnv} \vdash Expr_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_2,\mathbf{u}_2\rangle\end{array}}{\mathbf{pEnv} \vdash Expr_1, Expr_2 \Rightarrow \mathbf{r}_1|\mathbf{r}_2;\ \langle \mathbf{a}_1|\mathbf{a}_2,\mathbf{u}_1|\mathbf{u}_2\rangle} \quad (\textsc{Comma})$$

The rule says that if $\eta'_0 \sim_{\mathbf{a}_1|\mathbf{a}_2} \eta_0$ then the evaluation of $Expr_1, Expr_2$ gives the same result in both η_0 and η'_0. Our equivalence over $\mathbf{p}$ will be defined as "$\forall \mathbf{p}' \in \mathrm{pref}(\mathbf{p}).\, P(\mathbf{p}')$", so that $\eta'_0 \sim_{\mathbf{a}_1|\mathbf{a}_2} \eta_0$ implies $\eta'_0 \sim_{\mathbf{a}_1} \eta_0$ and $\eta'_0 \sim_{\mathbf{a}_2} \eta_0$. Hence, by induction, if we start the evaluation of $Expr_1, Expr_2$ from $\eta_0 \sim_{\mathbf{a}_1|\mathbf{a}_2} \eta'_0$, then $Expr_2$ will be evaluated against (η_0,Δ) and (η'_0,Δ), but we have still to prove that $\eta_0 \sim_{\mathbf{a}_2} \eta'_0$ implies $(\eta_0,\Delta) \sim_{\mathbf{a}_2} (\eta'_0,\Delta)$. This is another instance of the "stability" problem. In this case, the simplest solution is the adoption of the following notion of path equivalence: two histories η_1 and η_2 are equivalent modulo a path $\mathbf{p}$, denoted $\eta_1 \sim_{\mathbf{p}} \eta_2$, iff:

$$\forall \mathbf{p}' \in \mathrm{pref}(\mathbf{p}).\ \forall \Delta.\ \ [\![\mathbf{p}']\!]_{\mathrm{apply}(\eta_1,\Delta)} = [\![\mathbf{p}']\!]_{\mathrm{apply}(\eta_2,\Delta)}$$

The quantification on Δ makes this notion "stable" with respect to store evolution, which is extremely useful for our proofs, but the equality above actually implies that:

$$\forall \Delta.\ (wf(\eta_1,\Delta) \Rightarrow wf(\eta_2,\Delta)) \wedge (\forall \Delta.\ wf(\eta_2,\Delta) \Rightarrow wf(\eta_1,\Delta))$$

This is too strong, because, whenever two stores differ in one node, the Δ that creates the node can only be added to the store that is missing it. Similarly, it they differ in one edge, the Δ that inserts the edge can only be added to the store that is missing it. Hence, only identical stores can be extended with exactly the same set of Δ's.

So, we have to weaken the requirement. We first restrict the quantification to updates that only create nodes that are fresh in both stores. Moreover, we do not require that $\mathit{wf}(\eta_1,\Delta) \Rightarrow \mathit{wf}(\eta_2,\Delta)$, but only that, for every n of interest, a subset Δ' of Δ exists which can be used to extend η_1 and η_2 so to have n in both. The resulting notion of equivalence is preserved by every update in the language whose path does not intersect pref($\mathbf{p}$); this notion is strong enough for our purposes ($\Delta' \subseteq^i \Delta$ means the Δ' creates and deletes the same edges as Δ, but the inserted edges are a subset).

Definition 12 (Store equivalence modulo a path). *Two stores σ_1 and σ_2 are equivalent modulo a path $\mathbf{p}$, denoted $\sigma_1 \sim_{\mathbf{p}} \sigma_2$, iff:*

$$\begin{aligned}
&\forall \mathbf{p}' \in \mathit{pref}(\mathbf{p}).\ \forall \Delta.\ \mathit{created}(\Delta)\#(N_{\sigma_1} \cup N_{\sigma_2}) \ \wedge\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_1,\Delta)}\\
&\qquad\qquad \Rightarrow\ \exists \Delta' \subseteq^i \Delta.\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_1,\Delta')} \ \wedge\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_2,\Delta')}\\
&\forall \mathbf{p}' \in \mathit{pref}(\mathbf{p}).\ \forall \Delta.\ \mathit{created}(\Delta)\#(N_{\sigma_1} \cup N_{\sigma_2}) \ \wedge\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_2,\Delta)}\\
&\qquad\qquad \Rightarrow\ \exists \Delta' \subseteq^i \Delta.\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_1,\Delta')} \ \wedge\ n \in [\![\mathbf{p}']\!]_{\mathit{apply}(\sigma_2,\Delta')}
\end{aligned}$$

Definition 13 (Store history equivalence modulo a path).

$$\eta_1 \sim_{\mathbf{p}} \eta_2 \ \Leftrightarrow_{\mathit{def}}\ \mathit{apply}(\eta_1) \sim_{\mathbf{p}} \mathit{apply}(\eta_2)$$

Since $[\![\mathbf{p}]\!]_{\mathrm{apply}(\eta_1,\Delta)}$ is monotone wrt $\subseteq^i$, the above definition implies that:

$$\eta_1 \sim_{\mathbf{p}} \eta_2 \ \Rightarrow\ (\forall \Delta.\ \mathit{wf}(\eta_1,\Delta) \wedge \mathit{wf}(\eta_2,\Delta)\ \Rightarrow\ [\![\mathbf{p}]\!]_{\mathrm{apply}(\eta_1,\Delta)} = [\![\mathbf{p}]\!]_{\mathrm{apply}(\eta_2,\Delta)})$$

We are now ready for the formal definition of soundness.

Definition 14 (Soundness). *The static analysis $\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\ \langle \mathbf{a},\mathbf{u}\rangle$ is* sound *for the semantic evaluation $\mathit{dEnv} \vdash \eta_0;\mathit{Expr} \Rightarrow \eta_1;\bar{n}$ iff for any well-formed η_0, η_1, dEnv, $\mathbf{pEnv}$, Expr, $\bar{n}$, $\mathbf{r}$, $\mathbf{a}$, $\mathbf{u}$, such that:*

$$\begin{aligned}
&\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\ \langle \mathbf{a},\mathbf{u}\rangle\\
&\mathit{dEnv} \vdash \eta_0;\mathit{Expr} \Rightarrow (\eta_0,\Delta);\bar{n}\\
&\mathbf{pEnv} \supseteq_{\eta_0} \mathit{dEnv}
\end{aligned}$$

the following properties hold.

- ***Approximation by $\mathbf{r}$:*** *$\mathbf{r}$ is an approximation of the result: $\mathbf{r} \supseteq_{\eta_1} \bar{n}$*
- ***Parallel evolution from $\mathbf{a}$-equivalent stores:*** *For any store history η_0', if $\eta_0' \sim_{\mathbf{a}} \eta_0$ and $N_{\eta_0'}\#\mathit{created}(\Delta)$, then $\mathit{dEnv} \vdash \eta_0';\mathit{Expr} \Rightarrow (\eta_0',\Delta);\bar{n}$*
- ***Immutability out of $\mathbf{u}$:*** (1) $\mathbf{u} \supseteq_{\eta_1} \mathit{updated}(\Delta)$
(2) $\forall \mathit{prefclosed}(\mathbf{c}):\ \mathbf{c}\#\mathbf{u}\ \Rightarrow\ \eta_0 \sim_{\mathbf{c}} (\eta_0,\Delta)$.

In the *Parallel evolution* property, the condition $N_{\eta_0'}\#\mathit{created}(\Delta)$ is needed because, if η_0' did already contain some of the nodes that are added by Δ, then it would be impossible to extend η_0' with Δ. This condition is not restrictive, and is needed because we identify nodes in different stores by the fact that they have the same identity. We could relate different store using a node morphism, rather that node identity, but that would make the proofs much heavier.

Immutability has two halves. The first, $\mathbf{u} \supseteq_{\eta_1} \mathit{updated}(\Delta)$, confines the set of edges that are updated to those that are in **u**, and is important to prove that two updates commute if $\mathbf{u}_1\#\mathbf{u}_2$. The second half specifies that, for every **c**#**u**, the store after the update is **c**-equivalent to the store before. Together with *Parallel evolution*, it essentially says that after *Expr* is evaluated, the value returned by any expression Expr_1 that only accesses **c** is the same value returned by Expr_1 before *Expr* was evaluated, and is important to prove that an update and a query commute if $\mathbf{a}_1\#\mathbf{u}_2$. The path **c** must be prefix-closed for this property to hold. For example, according to our rules, $\mathit{delete}(/a/b)$ updates a path $\mathbf{u} = /a/b/\mathit{dos}{::}{*}$. It is disjoint from $\mathbf{c} = /a/b/..$, but still the value of $/a/b/..$ changes after $\mathit{delete}(/a/b)$. This apparent unsoundness arises because **c** is not prefix-closed. If we consider the prefix-closure $\mathbf{a} = /a|/a/b|/a/b/..$ of $/a/b/..$, we notice that **a** is *not* disjoint from **u**.

4.3 Path analysis rules

We present the rules in two groups: selection and update rules.

Selection rules. These rules regard the querying fragment of our language. We extend the rules from [8] for the proper handling of updated paths.

The (Comma) rule has been presented above.

The (Var) rule specifies that variable access does not access the store. One may wonder whether **r** should not be regarded as "accessed" by the evaluation of $x. The doubt is easily solved by referring to the definition of soundness: the value of $x is the same in two stores η_0 and η_0' independently of any equivalence among them, hence the accessed path should be empty. This rule also implicitly specifies that variable access commutes with any other expression. For example, *$x,delete($x)* is equivalent to *delete($x),$x*.

$$\frac{(\$x \mapsto \mathbf{r}) \in \mathbf{pEnv}}{\mathbf{pEnv} \vdash \$x \Rightarrow \mathbf{r};\, \langle (),() \rangle} \quad (\textsc{Var})$$

The (Step) rule specifies that a step accesses the prefix closure of **r**. Technically, the rule would still be sound if we only put $\mathbf{r}|(\mathbf{r}/\mathit{axis}{::}\mathit{ntest})$ in the accessed set. However, the commutativity theorem relies on the fact that, for any expression, its inferred accessed path is prefix-closed, for the reasons discussed at the end of the previous section, and the addition of the prefix closure of **r** does not seem to seriously affect the analysis precision.

$$\frac{\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\, \langle \mathbf{a},\mathbf{u} \rangle}{\mathbf{pEnv} \vdash \mathit{Expr}/\mathit{axis}{::}\mathit{ntest} \Rightarrow \mathbf{r}/\mathit{axis}{::}\mathit{ntest};\, \langle \mathrm{pref}(\mathbf{r}/\mathit{axis}{::}\mathit{ntest})|\mathbf{a},\mathbf{u} \rangle} \quad (\textsc{Step})$$

Iteration binds the variable and analyses the body once. Observe that the analysis ignores the order and multiplicity of nodes.

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\ \langle \mathbf{a}_1, \mathbf{u}_1 \rangle \\ (\mathbf{pEnv} + \$x \mapsto \mathbf{r}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_2, \mathbf{u}_2 \rangle\end{array}}{\begin{array}{c}\mathbf{pEnv} \vdash \texttt{for } \$x \texttt{ in } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2 \\ \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_1 | \mathbf{a}_2, \mathbf{u}_1 | \mathbf{u}_2 \rangle\end{array}} \quad (\text{FOR})$$

Element construction returns the unique constructor location, but there is no need to regard that location as accessed.

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\ \langle \mathbf{a}_1, \mathbf{u}_1 \rangle \\ \mathbf{pEnv} \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_2, \mathbf{u}_2 \rangle\end{array}}{\begin{array}{c}\mathbf{pEnv} \vdash \texttt{element}_{\mathit{code\text{-}loc}}\{\mathit{Expr}_1\}\{\mathit{Expr}_2\} \\ \Rightarrow \mathit{code\text{-}loc};\ \langle \mathbf{a}_1 | \mathbf{a}_2, \mathbf{u}_1 | \mathbf{u}_2 \rangle\end{array}} \quad (\text{ELT})$$

Local bindings just returns the result of evaluating the body, but the accesses and side effects of both subexpressions are both considered.

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\ \langle \mathbf{a}_1, \mathbf{u}_1 \rangle \\ (\mathbf{pEnv} + \$x \mapsto \mathbf{r}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_2, \mathbf{u}_2 \rangle\end{array}}{\mathbf{pEnv} \vdash \texttt{let } \$x \texttt{ := } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_1 | \mathbf{a}_2, \mathbf{u}_1 | \mathbf{u}_2 \rangle} \quad (\text{LET})$$

The conditional approximates the paths by merging the results of both branches.

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r}_0;\ \langle \mathbf{a}_1, \mathbf{u}_1 \rangle \\ \mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\ \langle \mathbf{a}_2, \mathbf{u}_2 \rangle \\ \mathbf{pEnv} \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\ \langle \mathbf{a}_3, \mathbf{u}_3 \rangle\end{array}}{\begin{array}{c}\mathbf{pEnv} \vdash \texttt{if } (\mathit{Expr}) \texttt{ then } \mathit{Expr}_1 \texttt{ else } \mathit{Expr}_2 \\ \Rightarrow \mathbf{r}_1 | \mathbf{r}_2;\ \langle \mathbf{a}_1 | \mathbf{a}_2 | \mathbf{a}_3, \mathbf{u}_1 | \mathbf{u}_2 | \mathbf{u}_3 \rangle\end{array}} \quad (\text{IF})$$

Update rules. The second set of rules deals with update expressions.

The first rule is the one for delete. The "updated path" **u** is extended 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 in $\mathbf{r}/\mathit{descendant\text{-}or\text{-}self}::*$ is affected by the deletion. Assume, for example, that $(\$x \mapsto \mathit{loc}) \in \mathbf{pEnv}$, $(\$x \mapsto n) \in \mathit{dEnv}$, and n is the root of a tree of the form $\langle a\rangle\langle b\rangle\langle c/\rangle\langle b/\rangle\langle a/\rangle$.

The evaluation of `delete` $\{\$x/b\}$ would change the semantics of $\$x//c$, although this path does not explicitly traverse loc/b. This is correctly dealt with, since the presence of $\mathit{loc}/b/\mathit{descendant\text{-}or\text{-}self}::*$ in **u** means: every path that is not disjoint from $\mathit{loc}/b/\mathit{descendant\text{-}or\text{-}self}::*$ may be affected by this operation, and, by Definition 9, $\mathit{loc}//c$ is *not* disjoint from $\mathit{loc}/b/\mathit{descendant\text{-}or\text{-}self}::*$.

Observe that `delete` $\{\$x/b\}$ also affects expressions that do not end below $\$x/b$, such as "$\$x/b/..$". This is not a problem either, since the accessed path **a** computed for the expression $\$x/b/..$ is actually $\mathit{loc}|(\mathit{loc}/b)|(\mathit{loc}/b/..)$, and the second component is not disjoint from $\mathit{loc}/b/\mathit{descendant\text{-}or\text{-}self}::*$.

$$\frac{\mathbf{pEnv} \vdash \mathit{Expr} \Rightarrow \mathbf{r};\ \langle \mathbf{a}, \mathbf{u} \rangle}{\mathbf{pEnv} \vdash \texttt{delete } \{\mathit{Expr}\} \Rightarrow ();\ \langle \mathbf{a}, \mathbf{u} | (\mathbf{r}/\mathit{descendant\text{-}or\text{-}self}::*) \rangle} \quad (\text{DELETE})$$

Similarly, `insert {`Expr_1`} into {`Expr_2`}` may modify every path that ends with descendants of Expr_2. Moreover, it depends on all the descendants of Expr_1, since it copies all of them.

$$\frac{\begin{array}{c}\mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\, \langle \mathbf{a}_1, \mathbf{u}_1 \rangle \\ \mathbf{pEnv} \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\, \langle \mathbf{a}_2, \mathbf{u}_2 \rangle\end{array}}{\begin{array}{c}\mathbf{pEnv} \vdash \texttt{insert } \{\mathit{Expr}_1\} \texttt{ into } \{\mathit{Expr}_2\} \\ \Rightarrow ();\, \langle \mathbf{a}_1 | \mathbf{a}_2 | (\mathbf{r}_1/\mathit{descendant\text{-}or\text{-}self}::*), \mathbf{u}_1 | \mathbf{u}_2 | (\mathbf{r}_2//*) \rangle\end{array}} \quad (\text{INSERTCHILD})$$

4.4 Soundness Theorem

Theorem 1 (Soundness of the analysis). *The static analysis rules presented in Section 4.3 are sound.*

Soundness is proved by induction, showing that the soundness properties are preserved by each rule. A detailed presentation of the soundness proof for the most important rules can be found in [16].

5 Commutativity Theorem

Our analysis is meant as a tool to prove for specific expressions whether they can be evaluated on a given store in any order or, put differently, whether they *commute*.

Definition 15 (Commutativity). *We shall use* $[\![\mathit{Expr}]\!]_\eta^{\mathit{dEnv}}$ *as a shorthand for the pair* $(\mathit{apply}(\eta'), \mathit{bag\text{-}of}(\bar{n}))$ *such that* $\mathit{dEnv} \vdash \eta; \mathit{Expr} \Rightarrow \eta'; \bar{n}$*, and where* $\mathit{bag\text{-}of}(\bar{n})$ *forgets the order of the nodes in* $\bar{n}$*.*

Two expressions Expr_1 *and* Expr_2 commute *in* $\mathbf{pEnv}$*, written* $\mathit{Expr}_1 \overset{\mathbf{pEnv}}{\longleftrightarrow} \mathit{Expr}_2$*, iff, for all* η *and* dEnv *such that* $\mathbf{pEnv} \supseteq_\eta \mathit{dEnv}$*, the following equality holds:*

$$[\![\mathit{Expr}_1, \mathit{Expr}_2]\!]_\eta^{\mathit{dEnv}} = [\![\mathit{Expr}_2, \mathit{Expr}_1]\!]_\eta^{\mathit{dEnv}}$$

Hence, $\mathit{Expr}_1 \overset{\mathbf{pEnv}}{\longleftrightarrow} \mathit{Expr}_2$ means that the order of evaluation of Expr_1 and Expr_2 only affects the order of the result. We explicitly do not require that the order of the individual nodes updated by the expressions is preserved.

Theorem 2 (Commutativity). *Consider two expressions and their analyses in* $\mathbf{pEnv}$*:*

$$\mathbf{pEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathbf{r}_1;\, \langle \mathbf{a}_1, \mathbf{u}_1 \rangle$$
$$\mathbf{pEnv} \vdash \mathit{Expr}_2 \Rightarrow \mathbf{r}_2;\, \langle \mathbf{a}_2, \mathbf{u}_2 \rangle$$

If the updates and accesses obtained by the analysis are independent then the expressions commute, in any environment that respects $\mathbf{pEnv}$*:*

$$\mathbf{u}_1 \# \mathbf{a}_2, \mathbf{a}_1 \# \mathbf{u}_2, \mathbf{u}_1 \# \mathbf{u}_2 \Rightarrow \mathit{Expr}_1 \overset{\mathbf{pEnv}}{\longleftrightarrow} \mathit{Expr}_2$$

Commutativity is our main result. The proof can be found in [16]. It follows the pattern sketched in Section 4, after Definition 11. The proof 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 following five properties, where only *Stability* is non-trivial.

$$\mathbf{p} \supseteq_{\eta_0} \bar{n} \;\Rightarrow\; \mathbf{p} \supseteq_{\eta_0,\Delta} \bar{n} \qquad \text{(Stability)}$$

$$\text{for each } \mathbf{p},\ \sim_{\mathbf{p}} \text{ is an equivalence relation} \qquad \text{(Equivalence)}$$

$$\mathbf{p}\#(\mathbf{q}_0|\mathbf{q}_1) \;\Leftrightarrow\; \mathbf{p}\#\mathbf{q}_0 \wedge \mathbf{p}\#\mathbf{q}_1 \qquad (|\#)$$

$$\eta_0 \sim_{(\mathbf{q}_0|\mathbf{q}_1)} \eta_1 \;\Rightarrow\; \eta_0 \sim_{\mathbf{q}_0} \eta_1 \qquad (|\sim)$$

$$\mathbf{q}_0 \subseteq \mathbf{q}_0|\mathbf{q}_1 \qquad (|\subseteq)$$

6 Related work

Numerous update languages have been proposed in the last few years [1–5]. Some of the most recent proposals [5, 7] are very expressive, as they provide the ability to observe the effect of updates during query evaluation. Although [7] 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 [5]. Very little work has been done so far on optimization or static analysis for such XML update languages, a significant exception being the work by Benedikt et al [4, 13]. 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 in [8], which we extend here by considering side effects.

Independence between updates and queries has been studied in the relational context [11, 12]. The problem becomes more difficult in the XML context because of the expressivity of existing XML query 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 [14, 15], but these papers rely on dynamic knowledge while we are interested in static commutativity properties, hence the technical tools involved are quite different.

7 Conclusion

In this paper, we have proposed a conservative approach to detect whether two expressions commute in 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 in an expression. As there is a growing need to extend XML languages with imperative features [7, 5, 19], we believe the kind of analysis we propose here will be essential for the long-term development of those languages. We are currently exploring the use of our commutativity analysis for the purpose of algebraic optimization of XML update languages.

References

1. Chamberlin, D., Florescu, D., Robie, J.: XQuery update facility. W3C Working Draft (2006)
2. Lehti, P.: Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit (2001)
3. Tatarinov, I., Ives, Z., Halevy, A., Weld, D.: Updating XML. In: SIGMOD. (2001)
4. Benedikt, M., Bonifati, A., Flesca, S., Vyas, A.: Adding updates to XQuery: Semantics, optimization, and static analysis. In: XIME-P'05. (2005)
5. Ghelli, G., Ré, C., Siméon, J.: XQuery!: An XML query language with side effects. In: DataX Workshop. Lecture Notes in Computer Science, Munich, Germany (2006)
6. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML query language (2006)
7. Carey, M., Chamberlin, D., Florescu, D., Robie, J.: Programming with XQuery. Draft submitted for publication (2006)
8. Marian, A., Simeon, J.: Projecting XML documents. In: Proceedings of International Conference on Very Large Databases (VLDB), Berlin, Germany (2003) 213–224
9. Benedikt, M., Fan, W., Kuper, G.M.: Structural properties of xpath fragments. Theor. Comput. Sci. **336**(1) (2005) 3–31
10. Miklau, G., Suciu, D.: Containment and equivalence for a fragment of xpath. J. ACM **51**(1) (2004) 2–45
11. Elkan, C.: Independence of logic database queries and updates. In: PODS. (1990) 154–160
12. Levy, A.Y., Sagiv, Y.: Queries independent of updates. In: 19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings, Morgan Kaufmann (1993) 171–181
13. Benedikt, M., Bonifati, A., Flesca, S., Vyas, A.: Verification of tree updates for optimization. In: CAV. (2005) 379–393
14. Dekeyser, S., Hidders, J., Paredaens, J.: A transaction model for xml databases. World Wide Web **7**(1) (2004) 29–57
15. Lanin, V., Shasha, D.: A symmetric concurrent b-tree algorithm. In: FJCC. (1986) 380–389
16. Ghelli, G., Rose, K., Siméon, J.: Commutativity analysis in XML update languages (2006) `http://www.di.unipi.it/~ghelli/papers/UpdateAnalysis.pdf`.
17. Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 data model (2006)
18. Wadler, P.: Two semantics of xpath. Discussion note for W3C XSLT Working Group (1999) http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.
19. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers (2006) Unpublished Manuscript.