Semantics, Types and Effects for XML Updates

Michael Benedikt[1] and James Cheney[2]

[1] Oxford University Computing Laboratory
[2] Laboratory for Foundations of Computer Science, University of Edinburgh

Abstract. The W3C recently released the XQuery Update Facility 1.0, a Candidate Recommendation for an XML update language. It appears likely that this proposal will become standard. XQuery has been equipped with a formal semantics and sound type system, but there has been little work on static analysis or typechecking of XML updates, and the typing rules in the current W3C proposal appear unsound for "transform" queries that perform embedded updates. In this paper, we investigate the problem of *schema alteration*, or synthesizing an output schema describing the result of an update applied to a given input schema. We review regular expression type systems for XQuery, present a core language and semantics for W3C-style XML updates, and develop an effect analysis and schema alteration, which can be used as the basis for sound typechecking for queries involving "transform".

1 Introduction

Query and transformation languages for XML data have been studied extensively, both in the database and programming language communities. The World Wide Web Consortium (W3C) has developed XQuery, a standard XML query language with a detailed formal semantics and type system [1,2]. Most real-world data changes over time, and so it is also important to be able to update XML documents and XML-based data. However, query languages such as XQuery, and transformation languages such as XSLT, provide support only for "functional" computation over immutable data, and are awkward for writing transformations that update part of the data "in-place" while leaving most of the document alone.

There have been a number of proposals and prototype implementations for XML update languages (see for example [3,4,5,6]). While no clear winner has emerged so far, the W3C has introduced the XQuery Update Facility [7] (henceforth called simply "XQuery Update"), combining features from several proposals; this is now supported by many XML database implementations. However, the typechecking and static analysis problems for XQuery Update (and for XML updates more generally) remain ill-understood. In contrast to XQuery, there is no formal semantics; moreover, the proposed typing rules for XQuery Update only ensure that updates are minimally well-formed, and do not show how to compute the type of the document after an update is performed. In fact, as we shall see, the proposed typing rules in the current W3C proposal are unsound.

In this paper we develop a sound type and effect system for XQuery Update based on *regular expression types* [8]. Regular expression types are closely related to tree automata [9] and have been employed in a number of other settings [10]. We show how to infer safe over-approximations for the results of both queries and updates. This is nontrivial because we must consider destructive update at the schema/type level.

A complication is that XQuery Updates have a somewhat involved "snapshot" semantics. An update expression is first *evaluated*, yielding a sequence of atomic update operations; then the atomic update sequence is *sanity-checked* and finally *applied*. Moreover, updates are not applied in the order they were generated (as a programmer might expect) but instead are applied in several phases: insertions and renamings first, then replacements, then deletions.

Example 1. Consider the update:

```
for $y in $x//a delete $y,
for $y in $x//a, $z in $x//d return (insert $z before $y)
```

This deletes all nodes matching $\$x//a$ and inserts copies of all nodes matching $\$x//d$ before the deleted nodes. Suppose the input $\$x$ has type[1] $doc[a[], b[], c[d[]]]$. One might expect that the a node will be deleted first, so that the second update has no effect, yielding result type $doc[b[], c[d[]]]$. However, the informal semantics in the W3C proposal reorders insert operations before deletions, so the actual result type is $doc[d[], b[], c[d[]]]$.

Uses of ancestor or sibling XPath axes further complicate typechecking:

Example 2. Consider the update expression:

```
for $y in $x//a/following::b/parent::c return delete $y
```

Intuitively, this deletes all c nodes that are parents of b nodes that follow some a node in the document. If the input $\$x$ has type $doc[b[c[]^*, a[]^*]]$ then this update has no effect; if $\$x : doc[a[], c[b[]^*]]$ then the output will always have type $doc[a[], c[]?]$; if $\$x : doc[(c[b[]], a[])^*]$ then the output will always have type $\$x : doc[(c[b[]], a[]^+)^?]$.

In the XQuery standard, however, the typing rules for axes such as `following` and `parent` are very conservative: they assume that the result of a query might be any part of the document. This would be disastrous from the point of view of typechecking updates such as the above, however, since we would have to assume that any part of the input could be the target of an update.

Finally, XQuery Update includes a new "transform" query expression that performs updates in the middle of a query. The "transform" expression copies data into new variables and then modifies the copied data. This complicates typechecking because the modified values may be used in subsequent queries. The W3C proposal's typing rules for "transform" do not seem take this into account, and appear unsound:

[1] For brevity, we use compact, XDuce-style notation [8] for XML trees and types.

Example 3. A typical W3C "transform" expression is of the form

```
copy $y := $x  modify delete $y/c  return $y
```

This expression behaves as follows: First we copy the value of $\$x$ and assign it to $\$y$; then we evaluate the modifying expression `delete` $\$y/c$ and apply the resulting updates; finally, we return $\$y$. Suppose $\$x : a[b[], c[]]$. Thus, initially $\$y$ will have the same type. According to the typing rules given in the W3C proposal [7], the return expression will be typechecked with $\$y$ still assigned type $a[b[], c[]]$, so the result of the query will be assigned type $a[b[], c[]]$, but the return value will be of the form $a[b[]]$.

To recover soundness for "transform" expressions it is necessary to ensure that the types of updated variables remain correct after the update is performed. One trivial, but unsatisfying way to do so is to set updated variables' types to `Any`, a type that matches any XML document. Another possibility, perhaps that intended by the W3C proposal, is that that the data should be revalidated after each update snapshot completes (including updates in the modify clause of a transform). Such revalidation could ensure that the type information remains valid after a transform is done, so the above example would result in a run-time type error. However, the current draft is ambiguous on this point: it does specify that some revalidation should take place but does not specify that revalidation should ensure that types are preserved [7, Sec. 2.4.5]. In any case, dynamic revalidation is potentially costly, and would require the schema designer to anticipate all possible changes to the schema in advance, thus precluding typechecking XQuery Update expressions that, for example, add new elements to the document that were not present in the original schema.

As these examples illustrate, it is easy to find pathological updates for which "good" output schemas appear difficult to predict. In fact, in general there may be no schema (based on regular expression types) that exactly captures the output of a query, because the range of a query or update over a regular input language may not be regular [11]. Even typechecking an update given a fixed input and output schema is hard in general, and undecidable for full XQuery Update. Nevertheless, it is worthwhile to find sound, static overapproximations to the result of an XML query or update. We focus on developing a pragmatic approach that demonstrates reasonable behavior on common cases. It is already difficult just to develop a nontrivial sound analysis for the W3C proposal, however, and experimental validation of the practical utility of our approach is beyond the scope of this paper.

Prior work has been done on typechecking and other static analyses for UpdateX [3,12] and FLUX [4], and other XML update proposals [5]. However, no prior work applies directly to the W3C's current XQuery Update proposal. While Benedikt et al. [3,12] considered a language similar to XQuery Update, they did not investigate typechecking. Cheney [4] studied regular expression typechecking for FLUX, an XML update language that is simpler, but also less expressive, than XQuery Update. Ghelli et al. studied *commutativity analysis* for an update language whose semantics differs substantially from the current version [5].

We want to emphasize that we consider a strict sublanguage of XQuery Updates that includes many key features but excludes some complications to the XML data model (such as attributes, processing instructions, etc.). We also leave out the "replace value of" operation [7]. However, extensions to handle these features appear straightforward. We also do not model the optional XML Schema validation, dynamic type-name tags, or revalidation features of XQuery Update. Instead, we adopt a purely structural type system with subtyping based on language inclusion, as in XQuery and much other previous work on typed XML programming languages [1,2,8,10,4]. Thus, our approach applies to any conforming implementation of XQuery Update, independent of whether it implements validation.

In this paper, we consider these related problems for XQuery Updates:

– *pending effect analysis*: given a schema and an update, approximate the possible atomic updates ("effect") generated by the update.
– *schema alteration*: given a schema and an update effect, find an output schema that approximates the results of applying atomic updates described by the effect.

Prior work on typechecking of queries has not handled upward axes, since they use regular expression types that specify only the hedge or subtree structure of returned nodes, not their position within a larger schema. To handle the interaction of schemas and updates, we develop a type and effect system that can record this information. Hence our approach applies to a language that contains all XPath axis steps.

In many XML processing settings (particularly databases) we can assume a fixed input schema and type declarations for the free variables of the expression, so we do not consider the (likely harder) *schema inference* problem of inferring types for both input variables and results.

Due to space limitations, in the body of the paper we omit full treatment of "transform" queries; however, the omitted material is straightforward given the results in the paper, and is provided in the companion technical report [13]. We omit proofs and standard or straightforward definitions; these are also provided in a companion technical report. The technical report also presents extended examples.

Outline. The rest of this paper is structured as follows: Section 2 reviews core XQuery and schema languages we will use, and Section 3 introduces the atomic update and XQuery Update languages, along with their operational semantics. Section 4 defines a pending effect analysis for update expressions and proves its soundness. Section 5 presents a schema alteration algorithm that applies a pending effect to a schema. We discuss a prototype implementation in Section 6. Section 7 discusses related and future work and Section 8 concludes.

2 Background

W3C XQuery Update 1.0 extends XQuery, which is already a large language. Even restricting attention to a core language, we must present a great deal of background material. In this section we review XML stores, regular expression types, XPath steps, and queries. Whenever possible we omit standard definitions that can be found in previous work or in [13].

XML stores. Let *Loc* be a set of *locations* l. A *location sequence* $\mathtt{L}$ is a list of locations; we write $()$ for the empty location sequence and $\mathtt{L} \cdot \mathtt{L}'$ for sequence composition. A *store* σ is a mapping from locations to *constructors* k, defined as follows:

$$k ::= \mathsf{text}[s] \mid a[\mathtt{L}]$$

where s is a string, a is an element node label and $\mathtt{L}$ is a list of locations. A well-formed store corresponds to an acyclic forest of XML trees. (We follow the XML data model and XQuery semantics in storing data values such as strings using "text nodes" in the store.)

We introduce a *copying* judgment $\sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}'$ that, intuitively, extends σ to a store σ' by copying the subtree under each location in $\mathtt{L}$ to a fresh subtree, collecting the resulting locations in list $\mathtt{L}'$. This judgment is defined formally in [13].

Regular expression types. Following previous work [8,10,4], we employ regular expression types τ for XML queries and updates:

$$\tau ::= () \mid \mathtt{T} \mid a[\tau] \mid \delta \mid \tau, \tau' \mid \tau|\tau' \mid \tau^*$$

Here, δ is the base type of "data" (e.g. strings), and $\mathtt{T}, \mathtt{T}', \ldots \in \mathit{TName}$ are *type names.* We consider *schemas* S mapping type names to types. In order to ensure regularity, we forbid uses of top-level type names in $\mathsf{S}(\mathtt{T})$; for example, both the type definitions $\mathtt{T} \mapsto a[], \mathtt{T}, b[]|()$ and $\mathtt{T}' \mapsto a[\mathtt{T}'], \mathtt{T}'|()$ are forbidden, whereas $\mathtt{T}' \mapsto a[\mathtt{T}']^*$ is allowed (and is equivalent to $\mathtt{T}' \mapsto a[\mathtt{T}'], \mathtt{T}'|())$. Such schemas are called *regular.* A type whose type names are drawn from S is called an S-type.

Regular schemas are very general and flexible, but they are awkward for our purposes. There are two reasons for this. First, we want to be able to typecheck queries and updates involving navigation axes such as `descendant`, `ancestor` and `following` more accurately than the default XQuery approach. Second, it is non-obvious how to apply the effects of updates to general regular schemas.

Both problems can be ameliorated using *flat* schemas. Flat schemas provide an explicit type name for each "part" (e.g. element or data type) in the schema corresponding to a "part" of a document. This makes them more suitable for updating. Flat schemas are defined as follows:

Definition 1. *A* flat type *is a regular expression over type names. A* flat schema *is a schema in which all type definitions are either of the form* $\mathtt{T} \mapsto \delta$*, or* $\mathtt{T} \mapsto a[\tau]$ *where* τ *is a flat type.*

$$\frac{\sigma(l) = a[\mathtt{L}] \quad \sigma \models_{\mathsf{S}} \mathtt{L} : \tau}{\sigma \models_{\mathsf{S}} l : a[\tau]} \quad \frac{\sigma(l) = \mathsf{text}[s]}{\sigma \models_{\mathsf{S}} l : \delta} \quad \frac{}{\sigma \models_{\mathsf{S}} () : ()} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L}_1 : \tau_1 \quad \sigma \models_{\mathsf{S}} \mathtt{L}_2 : \tau_2}{\sigma \models_{\mathsf{S}} \mathtt{L}_1 \cdot \mathtt{L}_2 : \tau_1, \tau_2}$$

$$\frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_1}{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_1 | \tau_2} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_2}{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_1 | \tau_2} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : () \mid \tau, \tau^*}{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau^*} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \mathsf{S}(\mathtt{T})}{\sigma \models_{\mathsf{S}} \mathtt{L} : \mathtt{T}}$$

Fig. 1. Validation rules

In a flat schema, a type name is mapped to either a single element $a[\tau]$ (with flat content type τ) or δ. For example, $\mathtt{X}, (\mathtt{Y}^*, \mathtt{Z})^*$ is a flat type and $\mathtt{X} \mapsto a[\mathtt{X}, (\mathtt{Y}^*, \mathtt{Z})^*]$ is a flat schema rule.

Flat schemas are syntactically more restrictive than general schemas, and hence they are less convenient for users. Fortunately, it is always possible to translate a regular schema S to an equivalent flat schema S', as follows: First introduce new type definitions $\mathtt{T} \mapsto a[\tau]$ for each type of the form $a[\tau]$ occurring in the original schema, rewriting the existing definitions and un-nesting nested element constructors. Then, "inline" all occurrences of the original type names in the schema with their new definitions. Other S-types in a context Γ can also be translated to S'-types in this way. As an example, the flat schema S' corresponding to $\mathtt{Y} \mapsto a[\mathtt{Y}]^*$ is $\mathtt{Z} \mapsto a[\mathtt{Z}^*]$, and the flat S'-type corresponding to the S-type $\mathtt{Y}$ is $\mathtt{Z}^*$.

Validation. We define a *validation* relation $\sigma \models_{\mathsf{S}} \mathtt{L} : \tau$ that states that in store σ and schema S, location sequence $\mathtt{L}$ matches type τ. The rules in Figure 1 define validation.

Aliasing. In determining types for updates, we will have to know whether two types can point to the same thing – this is a critical part of the algorithm in the beginning of Section 5. Types $\mathtt{T}$ and $\mathtt{T}'$ *may alias*[2] (with respect to S) provided that for some σ and $l \in \mathrm{dom}(\sigma)$, we have $\sigma \models_{\mathsf{S}} l : \mathtt{T}$ and $\sigma \models_{\mathsf{S}} l : \mathtt{T}'$.

Equivalently, types $\mathtt{T}$ and $\mathtt{T}'$ do not alias provided that they are disjoint, considered as regular tree languages (with respect to S viewed as a tree automaton). Disjointness is decidable for regular languages, and for restricted expressions (e.g. 1-unambiguous), tractable procedures are known [14,15]. For the purposes of this paper we assume that we are given sound alias sets $\mathrm{alias}_{\mathsf{S}}(\mathtt{T})$ such that if $\mathtt{T}$ and $\mathtt{T}'$ may alias we have $\mathtt{T}' \in \mathrm{alias}_{\mathsf{S}}(\mathtt{T})$.

XPath axes. XPath is an important sublanguage of both XQuery and XQuery Update. XPath steps are expressions of the form:

$$\begin{aligned} \mathit{step} &::= \mathit{ax}::\phi \qquad \phi ::= * \mid n \mid \mathtt{text} \\ \mathit{ax} &::= \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant} \mid \mathtt{parent} \mid \mathtt{ancestor} \mid \cdots \end{aligned}$$

[2] Aliasing means that two names refer to the same thing. In pointer analysis, aliasing usually means that two *variable* names refer to the same memory location. Here, aliasing means two *type* names may match the same store location.

The semantics and static analysis problems for XPath have been well-studied [16,9]. We will abstract away from the details of XPath in this paper, by introducing judgments $\sigma \models l/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \mathtt{L}$ to model XPath step evaluation and $\mathsf{S} \vdash \mathtt{T}/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \tau$ to model static typechecking for XPath steps. For the purposes of this paper, we assume that these relations satisfy the following soundness property:

Lemma 1. *If* $\mathsf{S} \vdash \mathtt{T}/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \tau$ *and* $\sigma \models l/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \mathtt{L}$ *and* $\sigma \models_{\mathsf{S}} l : \mathtt{T}$ *then* $\sigma \models_{\mathsf{S}} \mathtt{L} : \tau$*.*

Environments and type contexts. We employ *(dynamic) environments* γ mapping variables $x, y, \ldots \in \mathit{Var}$ to location sequences $\mathtt{L}$, and *type contexts* (also known as static environments) Γ mapping variables to regular expression types. We write $\bullet$ for an empty environment or type context, and write $\gamma[x := \mathtt{L}]$ for the result of updating a context by binding x to $\mathtt{L}$.

A type context is *flat* if its types are flat. An S-context is a context whose types are S-types. We also write $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ to indicate that $\forall x \in \mathrm{dom}(\Gamma).\ \sigma \models_{\mathsf{S}} \gamma(x) : \Gamma(x)$.

Queries. We introduce a core XQuery fragment, following Colazzo et al. [10].

$$\begin{aligned} q ::= {} & x \mid () \mid q, q' \mid a[q] \mid s \mid x/\mathit{step} \\ \mid {} & \mathtt{if}\ q\ \mathtt{then}\ q_1\ \mathtt{else}\ q_2 \mid \mathtt{let}\ x := q\ \mathtt{in}\ q' \mid \mathtt{for}\ x \in q\ \mathtt{return}\ q' \end{aligned}$$

The empty sequence $()$, element constructor $a[q]$, sequential composition q, q' and string s expressions build XML values. Variables and let-bindings are standard; conditionals branch depending on whether their first argument is nonempty. The expression x/step performs an XPath step starting from x. The iteration expression $\mathtt{for}\ x \in q\ \mathtt{return}\ q'$ evaluates q to $\mathtt{L}$, and evaluates q' with x bound to each location l in $\mathtt{L}$, concatenating the results in order.

We model the operational semantics of queries using a judgment $\sigma, \gamma \models q \Rightarrow \sigma', \mathtt{L}$. Note that the store σ may grow as a result of allocation, for example in evaluating expressions of the form $a[q]$ and s. We employ an auxiliary judgment $\sigma, \gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L}$ that is used for element node construction and later in the semantics of inserts (see Section 3) and transforms [13, Sec. 6]. The rules defining these judgments are given in [13]; here are two illustrative rules:

$$\frac{\sigma, \gamma \models q \Rightarrow \sigma_0, \mathtt{L}_0 \quad \sigma_0, \mathtt{L}_0 \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}}{\sigma, \gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L}} \qquad \frac{\sigma, \gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L} \quad l \notin \mathrm{dom}(\sigma')}{\sigma, \gamma \models a[q] \Rightarrow \sigma'[l := a[\mathtt{L}]], l}$$

Note that the second rule employs the auxiliary "copying" judgment, which simply evaluates a query and makes a fresh copy of the result.

3 Core XQuery Updates

Atomic updates. We consider atomic updates of the form:

$$\begin{aligned}\iota &::= \mathtt{ins}(\mathtt{L}, \mathtt{d}, l) \mid \mathtt{del}(l) \mid \mathtt{repl}(l, \mathtt{L}) \mid \mathtt{ren}(l, a)\\ \mathtt{d} &::= \leftarrow \mid \rightarrow \mid \downarrow \mid \swarrow \mid \searrow\end{aligned}$$

Here, the direction $\mathtt{d}$ indicates whether to insert before ($\leftarrow$), after ($\rightarrow$), or into the child list in first ($\swarrow$), last ($\searrow$) or arbitrary position ($\downarrow$). Moreover, we consider sequences of atomic updates ω with the empty sequence written ϵ and concatenation written $\omega; \omega'$.

Updating expressions. We now define the syntax of *updating expressions*, based roughly on those of the W3C XQuery Update proposal.

$$\begin{aligned}u ::= {} & () \mid u, u' \mid \mathtt{if}\ q\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 \mid \mathtt{for}\ x \in q\ \mathtt{return}\ u \mid \mathtt{let}\ x := q\ \mathtt{in}\ u\\ \mid {} & \mathtt{insert}\ q\ \mathtt{d}\ q_0 \mid \mathtt{replace}\ q_0\ \mathtt{with}\ q \mid \mathtt{rename}\ q_0\ \mathtt{as}\ a \mid \mathtt{delete}\ q_0\end{aligned}$$

The XQuery Update proposal overloads existing query syntax for updates. The () expression is a "no-op" update, expression u, u' is sequential composition, and let-bindings, conditionals, and for-loops are also included. There are four basic update expressions: insertion $\mathtt{insert}\ q\ \mathtt{d}\ q_0$, which says to insert a copy of q in position $\mathtt{d}$ relative to the value of q_0; deletion $\mathtt{delete}\ q_0$, which says to delete the value of q_0; renaming $\mathtt{rename}\ q_0\ \mathtt{as}\ a$, which says to rename the value of q_0 to a and replacement $\mathtt{replace}\ q_0\ \mathtt{with}\ q$, which says to replace the value of q_0 with a copy of q. In each case, the target expression q_0 is expected to evaluate to a single node; if not, evaluation fails.

Semantics. Updates have a multi-phase semantics. First, the updating expression is *evaluated*, resulting in a *pending update list* ω. We model this phase using an update evaluation judgment $\sigma, \gamma \models u \Rightarrow \sigma', \omega$, along with an auxiliary judgment $\sigma, \gamma, x \in \mathtt{L} \models^{\star} u \Rightarrow \sigma', \omega$ that handles for-loops. The rules for these judgments are presented in Figure 2. Note that again the store may grow as a result of allocation, but the values of existing locations in σ do not change in this phase. Next, ω is checked to ensure, for example, that no node is the target of multiple rename or replace instructions. We do not model this sanity-check phase explicitly here; instead we simply introduce an abstract predicate sanitycheck(ω) that checks that ω is a valid update sequence. Finally, the pending updates are *applied* to the store. The semantics of atomic updates is defined using the judgment $\sigma \models \iota \rightsquigarrow \sigma'$ presented in Figure 3.

One natural-seeming semantics for update application is simply to apply the updates in ω in (left-to-right) order. However, this naive semantics is not what the W3C proposal actually specifies [7]. Instead, updates are applied in the following order: (1) "insert into" and rename operations, (2) "insert before, after, as first" and "as last" operations, (3) "replace" operations, and finally (4) "delete" operations. (There is an extra stage for "replace value of" operations in [7], which

$$\frac{}{\sigma,\gamma \models () \Rightarrow \sigma,\epsilon} \qquad \frac{\sigma_1,\gamma \models u_1 \Rightarrow \sigma_2,\omega_1 \quad \sigma_2,\gamma \models u_2 \Rightarrow \sigma_3,\omega_2}{\sigma_1,\gamma \models u_1,u_2 \Rightarrow \sigma_3,\omega_1;\omega_2}$$

$$\frac{\sigma_1,\gamma \models q \Rightarrow \sigma_2, l\cdot \mathtt{L} \quad \sigma_2,\gamma \models u_1 \Rightarrow \sigma_3,\omega_1}{\sigma_1,\gamma \models \mathtt{if}\ q\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 \Rightarrow \sigma_3,\omega_1} \qquad \frac{\sigma_1,\gamma \models q \Rightarrow \sigma_2,() \quad \sigma_2,\gamma \models u_2 \Rightarrow \sigma_3,\omega_2}{\sigma_1,\gamma \models \mathtt{if}\ q\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 \Rightarrow \sigma_3,\omega_2}$$

$$\frac{\sigma_1,\gamma \models q \Rightarrow \mathtt{L},\sigma_2 \quad \sigma_2,\gamma[x := \mathtt{L}] \models u \Rightarrow \sigma_3,\omega}{\sigma_1,\gamma \models \mathtt{let}\ x = q\ \mathtt{in}\ u \Rightarrow \sigma_3,\omega} \qquad \frac{\sigma_1,\gamma \models q \Rightarrow \mathtt{L},\sigma_2 \quad \sigma_2,\gamma,x \in \mathtt{L} \models^{\star} u \Rightarrow \sigma_3,\omega}{\sigma_1,\gamma \models \mathtt{for}\ x \in q\ \mathtt{return}\ u \Rightarrow \sigma_3,\omega}$$

$$\frac{\sigma_1,\gamma \models q_1 \overset{\mathsf{copy}}{\Rightarrow} \sigma_2,\mathtt{L}_1 \quad \sigma_2,\gamma \models q_2 \Rightarrow \sigma_3,l_2}{\sigma_1,\gamma \models \mathtt{insert}\ q_1\ \mathtt{d}\ q_2 \Rightarrow \sigma_3,\mathtt{ins}(\mathtt{L}_1,\mathtt{d},l_2)} \qquad \frac{\sigma_1,\gamma \models q \Rightarrow \sigma_2,l}{\sigma_1,\gamma \models \mathtt{delete}\ q \Rightarrow \sigma_2,\mathtt{del}(l)}$$

$$\frac{\sigma_1,\gamma \models q_1 \Rightarrow \sigma_2,l_1 \quad \sigma_2,\gamma \models q_2 \overset{\mathsf{copy}}{\Rightarrow} \sigma_3,\mathtt{L}_2}{\sigma_1,\gamma \models \mathtt{replace}\ q_1\ \mathtt{with}\ q_2 \Rightarrow \sigma_3,\mathtt{repl}(l_1,\mathtt{L}_2)} \qquad \frac{\sigma_1,\gamma \models q \Rightarrow \sigma_2,l}{\sigma_1,\gamma \models \mathtt{rename}\ q\ \mathtt{as}\ a \Rightarrow \sigma_2,\mathtt{ren}(l,a)}$$

$$\frac{}{\sigma,\gamma,x \in () \models^{\star} u \Rightarrow \sigma,\epsilon} \qquad \frac{\sigma_1,\gamma[x := l] \models u \Rightarrow \sigma_2,\omega_1 \quad \sigma_2,\gamma,x \in \mathtt{L} \models^{\star} u \Rightarrow \sigma_3,\omega_2}{\sigma_1,\gamma,x \in l\cdot\mathtt{L} \models^{\star} u \Rightarrow \sigma_3,\omega_1;\omega_2}$$

Fig. 2. Rules for evaluating update expressions to pending update lists

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L},\leftarrow,l) \leadsto \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2]]} \qquad \frac{\sigma(l) = a[\mathtt{L}']}{\sigma \models \mathtt{ins}(\mathtt{L},\swarrow,l) \leadsto \sigma[l := a[\mathtt{L} \cdot \mathtt{L}']]}$$

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L},\rightarrow,l) \leadsto \sigma[l' := a[\mathtt{L}_1 \cdot l \cdot \mathtt{L} \cdot \mathtt{L}_2]]} \qquad \frac{\sigma(l) = a[\mathtt{L}']}{\sigma \models \mathtt{ins}(\mathtt{L},\searrow,l) \leadsto \sigma[l := a[\mathtt{L}' \cdot \mathtt{L}]]}$$

$$\frac{\sigma(l) = a[\mathtt{L}_1 \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L},\downarrow,l) \leadsto \sigma[l := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]]} \qquad \frac{\sigma(l) = a[\mathtt{L}]}{\sigma \models \mathtt{ren}(l,b) \leadsto \sigma[l := b[\mathtt{L}]]}$$

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]}{\sigma \models \mathtt{repl}(l,\mathtt{L}) \leadsto \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]]} \qquad \frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]}{\sigma \models \mathtt{del}(l) \leadsto \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L}_2]]}$$

Fig. 3. Semantics of atomic updates

$$\begin{aligned}
\mathsf{stage}(\mathtt{ins}(\_,\downarrow,\_)) &= 1 \\
\mathsf{stage}(\mathtt{ren}(\_,\_)) &= 1 \\
\mathsf{stage}(\mathtt{ins}(\_,\mathtt{d},\_)) &= 2 \quad (\mathtt{d} \in \{\leftarrow,\rightarrow,\swarrow,\searrow\}) \\
\mathsf{stage}(\mathtt{repl}(\_,\_)) &= 3 \\
\mathsf{stage}(\mathtt{del}(\_)) &= 4
\end{aligned}$$

$$\frac{\sigma_0 \models_1 \omega \leadsto \sigma_1 \quad \sigma_1 \models_2 \omega \leadsto \sigma_2 \quad \sigma_2 \models_3 \omega \leadsto \sigma_3 \quad \sigma_3 \models_4 \omega \leadsto \sigma_4}{\sigma_0 \models \omega \leadsto \sigma_4} \qquad \frac{}{\sigma \models_i \epsilon \leadsto \sigma}$$

$$\frac{\sigma \models_i \omega_j \leadsto \sigma' \quad \sigma' \models_i \omega_k \leadsto \sigma'' \quad \{j,k\} = \{1,2\}}{\sigma \models_i \omega_1,\omega_2 \leadsto \sigma''} \qquad \frac{\sigma \models \iota \leadsto \sigma'}{\sigma \models_{\mathsf{stage}(\iota)} \iota \leadsto \sigma'} \qquad \frac{\mathsf{stage}(\iota) \neq i}{\sigma \models_i \iota \leadsto \sigma}$$

$$\frac{\sigma,\gamma \models u \Rightarrow \sigma',\omega \quad \mathrm{sanitycheck}(\omega) \quad \sigma' \models \omega \leadsto \sigma''}{\sigma,\gamma \models u \leadsto \sigma''}$$

Fig. 4. Update application

we omit.) Subject to these constraints, the order of application within each stage is unspecified. To model this behavior we introduce a judgment $\sigma \models \omega \rightsquigarrow \sigma'$ along with an auxiliary function $\mathsf{stage}(\iota)$ and judgment $\sigma \models_i \omega \rightsquigarrow \sigma'$ for stages $i \in \{1, 2, 3, 4\}$. The rules defining these judgments are shown in Figure 3. Note that the rule for sequential composition permits arbitrary reordering of update sequences (which are also identified up to associativity). Static analyses for the W3C semantics are not in general valid for the naive, "in-order" semantics and vice versa.

The final rule in Figure 4 defines the judgment $\sigma, \gamma \models u \rightsquigarrow \sigma'$, which evaluates an update, checks that the resulting pending update list is valid, and then applies the updates to the store.

Inferring Types. For functional programs (i.e., queries) on documents, the notion of a valid type for an expression is fairly clear: given a schema S and expression e, a typing is a representation (e.g. by a regular expression type) of a set of trees; it is valid if it represents all of the possible hedges of subtrees returned by the query. Since XML updates modify the input store but do not return a value, the appropriate notion of a valid typing is less familiar. Our goal is to define a typing judgment $\mathsf{S}, \Gamma \vdash \mathtt{u} \rightsquigarrow \mathsf{S}', \Gamma'$ that relates an update $\mathtt{u}$, input schema S and a S-context Γ to a new schema S' and a new S'-context Γ' in which the types of variables in Γ have been adjusted to account for the changes made by the update. The basic correctness criterion we expect for this judgment is that *if the initial store satisfies* Γ *with respect to* S*, then the final store resulting from applying* $\mathtt{u}$ *satisfies the type context* Γ' *with respect to* S'. This property (Corollary 1) is the main result of the paper. Typically, the initial store will consist of a single tree and the environment γ will map a single variable $\$doc$ to the root of the tree. In this case our correctness property guarantees that the portion of the output reachable from this root will satisfy the new schema S'.

4 Type and Effect Analysis

Query result type analysis. First, for queries we would like to define a typechecking judgment $\mathsf{S}; \Gamma \vdash q : \tau$ that calculates return type τ for q when run in context Γ. Previous work on type systems for XML queries has been based on general regular-expression types [1,10]; here, however, we want to infer *flattened* types. To do this in the presence of element-node constructor expressions, we may need to add rules to the schema, so we employ a judgment $\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}'$. The rules are mostly straightforward generalizations of those in Colazzo et al. [10] and so are relegated to [13]. The key new rules with respect to previous work are those for node construction and XPath axis steps, respectively:

$$\frac{\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}' \quad \mathtt{T} \notin \mathrm{dom}(\mathsf{S}')}{\mathsf{S}; \Gamma \vdash a[q] : \mathtt{T}; \mathsf{S}'[\mathtt{T} := a[\tau]]} \qquad \frac{\mathsf{S} \vdash \Gamma(x)/ax :: \phi \overset{\mathsf{step}}{\Rightarrow} \tau}{\mathsf{S}; \Gamma \vdash x/ax :: \phi : \tau; \mathsf{S}}$$

$$\frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau \quad \sigma \models_{\mathsf{S}} l : \mathtt{T}}{\sigma \models_{\mathsf{S}} \mathtt{ins}(\mathtt{L}, \mathtt{d}, l) : \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T})} \quad \frac{\sigma \models_{\mathsf{S}} l : \mathtt{T}}{\sigma \models_{\mathsf{S}} \mathtt{del}(l) : \mathtt{del}(\mathtt{T})}$$

$$\frac{\sigma \models_{\mathsf{S}} l : \mathtt{T}}{\sigma \models_{\mathsf{S}} \mathtt{ren}(l, a) : \mathtt{ren}(\mathtt{T}, a)} \quad \frac{\sigma \models_{\mathsf{S}} l : \mathtt{T} \quad \sigma \models_{\mathsf{S}} \mathtt{L} : \tau}{\sigma \models_{\mathsf{S}} \mathtt{repl}(l, \mathtt{L}) : \mathtt{repl}(\mathtt{T}, \tau)}$$

Fig. 5. Some representative effect validity rules

$$\frac{\begin{array}{c}(\text{if } \mathtt{d} \in \{\downarrow, \swarrow, \searrow\} \text{ then } \mathsf{S}'(\mathtt{T}) \neq \delta) \\ \mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma \vdash q' : \mathtt{T}; \mathsf{S}_2\end{array}}{\mathsf{S}; \Gamma \vdash \mathtt{insert}\ q\ \mathtt{d}\ q' : \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}); \mathsf{S}_2} \quad \frac{\mathsf{S}; \Gamma \vdash q : \mathtt{T}; \mathsf{S}' \quad \mathsf{S}'(\mathtt{T}) \neq \delta}{\mathsf{S}; \Gamma \vdash \mathtt{rename}\ q\ \mathtt{as}\ a : \mathtt{ren}(\mathtt{T}, a); \mathsf{S}'}$$

$$\frac{\mathsf{S}; \Gamma \vdash q : \mathtt{T}; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma \vdash q' : \tau; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash \mathtt{replace}\ q\ \mathtt{with}\ q' : \mathtt{repl}(\mathtt{T}, \tau); \mathsf{S}_2} \quad \frac{\mathsf{S}; \Gamma \vdash q : \mathtt{T}; \mathsf{S}'}{\mathsf{S}; \Gamma \vdash \mathtt{delete}\ q : \mathtt{del}(\mathtt{T}); \mathsf{S}'}$$

Fig. 6. Some representative update effect-inference rules

Theorem 1 (Type Soundness). *If* $\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}'$ *then for all* $\sigma, \gamma, \mathtt{L}, \sigma'$*, if* $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models q \Rightarrow \sigma', \mathtt{L}$ *then* $\sigma' \models_{\mathsf{S}'} \mathtt{L} : \tau$*.*

Update effect analysis. We next turn to the problem of statically approximating the pending update list generated by an update expression. We use the term *(pending) effect* for such static approximations. Effects have the following forms:

$$\Omega ::= \epsilon \mid \Omega; \Omega' \mid \Omega | \Omega' \mid \Omega^* \mid \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}) \mid \mathtt{del}(\mathtt{T}) \mid \mathtt{ren}(\mathtt{T}, a) \mid \mathtt{repl}(\mathtt{T}, \tau)$$

The semantics of effects is defined by the judgment $\sigma \models_{\mathsf{S}} \omega : \Omega$ in Figure 5; we leave out standard rules for regular expression forms. Intuitively, $\sigma \models_{\mathsf{S}} \omega : \Omega$ says that in store σ and schema S, the atomic updates ω match the effect expression Ω.

We use judgments $\mathsf{S}; \Gamma \vdash u : \Omega; \mathsf{S}'$ and $\mathsf{S}; \Gamma; x \in \tau \vdash^{\star} u : \Omega; \mathsf{S}'$ to infer effects for plain and iterative updates respectively. We show some representative rules in Figure 6; the full definition is in the technical report. Note that typechecking an update may also require adding rules to the result schema, because of embedded node-construction (e.g. `insert` $foo[] \downarrow x$).

Theorem 2 (Effect soundness). *If* $\mathsf{S}; \Gamma \vdash u : \Omega; \mathsf{S}'$ *then for all* σ, γ*, if* $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models u \Rightarrow \sigma', \omega$ *then* $\sigma' \models_{\mathsf{S}'} \omega : \Omega$*.*

Type soundness only guarantees that the results of successful executions will match the static type. Dynamic errors may still occur while evaluating a well-formed query. Similarly, update effect soundness only guarantees that the results of a successful update evaluation will match the computed effect, not that evaluation will be free of dynamic errors. We believe our techniques can be modified to issue static warnings about possible dynamic errors in queries, but this is beyond the scope of this paper.

5 Schema Alteration

We now present an algorithm for *schema alteration*, that is, soundly over-approximating the actual effects a pending update may have on a schema. Given input type context Γ, schema S and pending effect Ω we want to infer a suitable output schema S' and type context Γ'. The rough idea is as follows:

1. Augment the input schema S to $\tilde{\mathtt{S}}$ by adding new temporary type names standing for "places" where updates may occur.
2. Determine which type names may match the same store location at run time, using alias analysis (as described in Section 2).
3. Simulate the effects of each stage of atomic update application on $\tilde{\mathtt{S}}$.
4. Finally, flatten the updated $\tilde{\mathtt{S}}$ to S' and update the type context Γ to Γ'.

We first illustrate the above algorithm by an example:

Example 4. Suppose we have effect $\Omega = \mathtt{ins}((\mathtt{U},\mathtt{V}),\swarrow,\mathtt{T}),\mathtt{del}(\mathtt{T}),\mathtt{ren}(\mathtt{T},b)$, with schema S given by rules $\tilde{\mathtt{S}} \mapsto doc[\mathtt{T}], \mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}]$, $\mathtt{U} \mapsto b[]$, $\mathtt{V} \mapsto c[]$, and $\Gamma = x : \mathsf{S}$.

Using the schema S we will form a new schema $\tilde{\mathtt{S}}$ extending S with additional type names and instrumented rules based on the rules of S. For example, for the single rule $\mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}]$ we generate three rules:

$$\tilde{\mathtt{T}} \mapsto \tilde{\mathtt{T}}_{\leftarrow}, \tilde{\mathtt{T}}_r, \tilde{\mathtt{T}}_{\rightarrow} \quad \tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] \quad \tilde{\mathtt{T}}_c \mapsto \tilde{\mathtt{T}}_{\swarrow}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{U}}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{V}}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{T}}_{\searrow}$$

Here, the five type names $\tilde{\mathtt{T}}_{\downarrow}$, $\tilde{\mathtt{T}}_{\leftarrow}$, $\tilde{\mathtt{T}}_{\rightarrow}$, $\tilde{\mathtt{T}}_{\swarrow}$, and $\tilde{\mathtt{T}}_{\searrow}$ stand for data inserted "into", "before", "after", "first into", or "last into" $\mathtt{T}$. The type name $\tilde{\mathtt{T}}_r$ stands for the data "replacing" $\mathtt{T}$, and the type name $\tilde{\mathtt{T}}_c$ stands for the "content" of $\mathtt{T}$.

The rest of the auxiliary type names are all initially defined as $()$. Note therefore that each type $\tilde{\mathtt{T}}$ in the augmented schema $\tilde{\mathtt{S}}$ initially is equivalent to $\mathtt{T}$ in S, in the sense that they match the same subtrees.

Next, we simulate the static effects, in order of stage. In stage 1, we perform the rename operation, by altering the definition of $\tilde{\mathtt{T}}_r$ to $a[\tilde{\mathtt{T}}_c]|b[\tilde{\mathtt{T}}_c]$. In stage 2 we simulate effect $\mathtt{ins}((\mathtt{U},\mathtt{V}),\swarrow,\mathtt{T})$ by setting $\tilde{\mathtt{T}}_{\swarrow}$ to $(\mathtt{U},\mathtt{V})^*$. Here we refer to the original types $\mathtt{U}$ and $\mathtt{V}$ in S, which have the same definitions as before. Stage 3 is inactive, and finally in stage 4 we apply the deletion by setting $\tilde{\mathtt{T}}_r$ to $a[\tilde{\mathtt{T}}_c]|b[\tilde{\mathtt{T}}_c]|()$. In this example, there are no other type names that may alias $\mathtt{T}$. Had there been, we would have applied the same changes to the aliases of $\mathtt{T}$.

Finally, we re-flatten the final schema. In this case consider the rule for $\tilde{\mathtt{T}}$. Flattening and simplifying yields $\tilde{\mathtt{S}} \mapsto doc[\tilde{\mathtt{T}}_1|\tilde{\mathtt{T}}_2|()]$, $\tilde{\mathtt{T}}_1 \mapsto a[(\mathtt{U},\mathtt{V})^*,\tilde{\mathtt{U}},\tilde{\mathtt{V}}]$, $\tilde{\mathtt{T}}_2 \mapsto b[(\mathtt{U},\mathtt{V})^*,\tilde{\mathtt{U}},\tilde{\mathtt{V}}]$. Note that this type refers to both the old and new versions of $\mathtt{U}$ and $\mathtt{V}$ (they happen to be the same in this case). We also modify the type context to $x : \tilde{\mathtt{S}}$ to reflect the change.

Another, more elaborate example is shown in Figure 7. We now describe the schema alteration algorithm more carefully.

Initial augmented schema:

$$\begin{array}{llll}
\mathtt{S} \mapsto \mathit{doc}[\mathtt{T}] & \tilde{\mathtt{S}} \mapsto \tilde{\mathtt{S}}_{\leftarrow}, \tilde{\mathtt{S}}_r, \tilde{\mathtt{S}}_{\rightarrow} & \tilde{\mathtt{S}}_r \mapsto a[\tilde{\mathtt{S}}_c] & \tilde{\mathtt{S}}_c \mapsto \tilde{\mathtt{S}}_{\swarrow}, \tilde{\mathtt{S}}_{\downarrow}, \tilde{\mathtt{T}}, \tilde{\mathtt{S}}_{\downarrow}, \tilde{\mathtt{S}}_{\searrow} \\
\mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}] & \tilde{\mathtt{T}} \mapsto \tilde{\mathtt{T}}_{\leftarrow}, \tilde{\mathtt{T}}_r, \tilde{\mathtt{T}}_{\rightarrow} & \tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] & \tilde{\mathtt{T}}_c \mapsto \tilde{\mathtt{T}}_{\swarrow}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{U}}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{V}}, \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{T}}_{\searrow} \\
\mathtt{U} \mapsto b[] & \tilde{\mathtt{U}} \mapsto \tilde{\mathtt{U}}_{\leftarrow}, \tilde{\mathtt{U}}_r, \tilde{\mathtt{U}}_{\rightarrow} & \tilde{\mathtt{U}}_r \mapsto b[\tilde{\mathtt{U}}_c] & \tilde{\mathtt{U}}_c \mapsto \tilde{\mathtt{U}}_{\swarrow}, \tilde{\mathtt{U}}_{\downarrow}, \tilde{\mathtt{U}}_{\searrow} \\
\mathtt{V} \mapsto c[] & \tilde{\mathtt{V}} \mapsto \tilde{\mathtt{V}}_{\leftarrow}, \tilde{\mathtt{V}}_r, \tilde{\mathtt{V}}_{\rightarrow} & \tilde{\mathtt{V}}_r \mapsto c[\tilde{\mathtt{V}}_c] & \tilde{\mathtt{V}}_c \mapsto \tilde{\mathtt{V}}_{\swarrow}, \tilde{\mathtt{V}}_{\downarrow}, \tilde{\mathtt{V}}_{\searrow}
\end{array}$$

All other new type names are initialized to ().
Effect:

$$|\Omega| = \{\mathtt{ins}(\mathtt{V}, \leftarrow, \mathtt{U}), \mathtt{ren}(\mathtt{U}, d), \mathtt{repl}(\mathtt{V}, \mathtt{U}^*), \mathtt{del}(\mathtt{T})\}$$

$$\text{Schema changes: } \left\{ \begin{array}{llll} \text{Phase 1:} & \text{Phase 2:} & \text{Phase 3:} & \text{Phase 4:} \\ \tilde{\mathtt{U}}_r \mapsto b[\tilde{\mathtt{U}}_c] | d[\tilde{\mathtt{U}}_c] & \tilde{\mathtt{U}}_{\leftarrow} \mapsto \mathtt{V}^* & \tilde{\mathtt{V}}_r \mapsto c[\tilde{\mathtt{V}}_c] | \mathtt{U}^* & \tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] | () \end{array} \right.$$

Result schema (after some equational simplifications):

$$\begin{array}{llll}
\mathtt{S} \mapsto \mathit{doc}[\mathtt{T}] & \tilde{\mathtt{S}} \mapsto \tilde{\mathtt{S}}_r, & \tilde{\mathtt{S}}_r \mapsto a[\tilde{\mathtt{S}}_c] & \tilde{\mathtt{S}}_c \mapsto \tilde{\mathtt{T}}, \\
\mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}] & \tilde{\mathtt{T}} \mapsto \tilde{\mathtt{T}}_r & \tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] | () & \tilde{\mathtt{T}}_c \mapsto \tilde{\mathtt{U}}, \tilde{\mathtt{V}} \\
\mathtt{U} \mapsto b[] & \tilde{\mathtt{U}} \mapsto \tilde{\mathtt{V}}^*, \tilde{\mathtt{U}}_r & \tilde{\mathtt{U}}_r \mapsto b[\tilde{\mathtt{U}}_c] | d[\tilde{\mathtt{U}}_c] & \tilde{\mathtt{U}}_c \mapsto () \\
\mathtt{V} \mapsto c[] & \tilde{\mathtt{V}} \mapsto \tilde{\mathtt{V}}_r & \tilde{\mathtt{V}}_r \mapsto c[\tilde{\mathtt{V}}_c] | \mathtt{U}^* & \tilde{\mathtt{V}}_c \mapsto ()
\end{array}$$

Re-flattened schema:

$$\mathtt{S} \mapsto \mathit{doc}[\mathtt{T}] \quad \mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}] \quad \mathtt{U} \mapsto b[] \quad \mathtt{V} \mapsto c[]$$

$$\tilde{\mathtt{S}} \mapsto a[\tilde{\mathtt{T}}|()] \quad \tilde{\mathtt{T}} \mapsto a[\mathtt{V}^*, (\tilde{\mathtt{U}}_1|\tilde{\mathtt{U}}_2), (\tilde{\mathtt{V}}_0|\mathtt{U}^*)] \quad \tilde{\mathtt{U}}_1 \mapsto b[] \quad \tilde{\mathtt{U}}_2 \mapsto d[] \quad \tilde{\mathtt{V}}_0 \mapsto c[]$$

Fig. 7. Detailed example of schema alteration

Preprocessing. First we pre-compute sound approximate aliasing information for S, computing the set $\text{alias}(\mathtt{T}) = \text{alias}_{\mathsf{S}}(\mathtt{T})$ for each $\mathtt{T}$. We next define the augmented schema $\tilde{\mathsf{S}}$ as follows. For each rule $\mathtt{T} \mapsto a[\tau]$ in S, we introduce rules

$$\tilde{\mathtt{T}} \mapsto \tilde{\mathtt{T}}_{\leftarrow}, \tilde{\mathtt{T}}_r, \tilde{\mathtt{T}}_{\rightarrow} \quad \tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] \quad \tilde{\mathtt{T}}_c \mapsto \tilde{\mathtt{T}}_{\swarrow}, h(\tau), \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{T}}_{\searrow}$$

where h is the (unique) regular expression homomorphism satisfying $h(\mathtt{U}) = \tilde{\mathtt{T}}_{\downarrow}, \tilde{\mathtt{U}}$ for all $\mathtt{U}$ in S. We map all other new type names in $\tilde{\mathsf{S}}$ to ().

Effect application. We now apply the effects to the augmented schema. The behavior of an effect is applied to the effect's target type name and all of its aliases. We will ignore the regular expression structure of effects and just consider the set of atomic effects, written $|\Omega|$. Similarly, we write $|\tau|$ for the set of all type names mentioned in τ. We also write $\bigvee\{\tau_1, \ldots, \tau_n\}$ for the regular expression $\tau_1 | \cdots | \tau_n$.

Phase 1: To simulate insert–into operations, for each type name $\mathtt{T}$, we define the set $I_{\downarrow}(\mathtt{T}) = \{\mathtt{U} \mid \exists \mathtt{T}' \in \text{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{ins}(\tau, \downarrow, \mathtt{T}') \in |\Omega|, \mathtt{U} \in |\tau|\}$. We then replace rule $\tilde{\mathtt{T}}_{\downarrow} \mapsto ()$ with $\tilde{\mathtt{T}}_{\downarrow} \mapsto (\bigvee I_{\downarrow}(\mathtt{T}))^*$ in $\tilde{\mathsf{S}}$. To simulate renamings, for each type name $\mathtt{T}$, we define the set $N(\mathtt{T}) = \{b \mid \exists \mathtt{T}' \in \text{alias}(\mathtt{T}).\ \mathtt{ren}(\mathtt{T}', b) \in |\Omega|\}$, and replace rule $\tilde{\mathtt{T}}_r \mapsto \tau_0$ with $\tilde{\mathtt{T}}_r \mapsto \tau_0 \mid \bigvee\{b[\tilde{\mathtt{T}}_c] \mid b \in N(\mathtt{T})\}$.

Phase 2: To simulate the remaining insert operations, we define the set $I_{\mathtt{d}}(\mathtt{T}) = \{\tau \mid \exists \mathtt{T}' \in \text{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}') \in |\Omega|\}$ and then replace rule $\tilde{\mathtt{T}}_{\mathtt{d}} \mapsto ()$ with $\tilde{\mathtt{T}}_{\mathtt{d}} \mapsto (\bigvee I_{\mathtt{d}}(\mathtt{T}))^*$ for each type name $\mathtt{T}$ and direction $\mathtt{d} \in \{\leftarrow, \rightarrow, \swarrow, \searrow\}$.

Phase 3: To simulate replacement operations, we construct the set $R(\mathtt{T}) = \{\tau \mid \exists \mathtt{T}' \in \text{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{repl}(\mathtt{T}', \tau) \in |\Omega|\}$ of possible replacements for each $\mathtt{T}$, and replace the rule $\tilde{\mathtt{T}}_r \mapsto \tau_0$ with $\tilde{\mathtt{T}}_r \mapsto \tau_0 \mid \bigvee R(\mathtt{T})$.

Phase 4: To simulate deletions, for each $\mathtt{T}$, if $\mathtt{del}(\mathtt{U}) \in |\Omega|$ for some $\mathtt{U} \in \text{alias}(\mathtt{T})$, replace the rule $\tilde{\mathtt{T}}_r \mapsto \tau_0$ with $\tilde{\mathtt{T}}_r \mapsto \tau_0 \mid ()$.

Postprocessing. Once we have finished symbolically updating $\tilde{\mathsf{S}}$, we also update Γ to $\tilde{\Gamma}$ by replacing each binding $x : \tau$ in Γ with $x : \tilde{\tau}$, where $\tilde{\tau}$ is the regular expression obtained by replacing each $\mathtt{T}$ with $\tilde{\mathtt{T}}$. We also flatten $\tilde{\mathsf{S}}$ and $\tilde{\Gamma}$ to obtain S' and Γ'.

We will write $\mathsf{S}, \Gamma \vdash \Omega \rightsquigarrow \mathsf{S}', \Gamma'$ to indicate that given input schema S and typing context Γ, symbolically evaluating Ω yields flattened output schema S' and typing context Γ'. We also define $\mathsf{S}, \Gamma \vdash u \rightsquigarrow \mathsf{S}', \Gamma'$ as meaning that $\mathsf{S}; \Gamma \vdash u : \Omega; \mathsf{S}''$ and $\mathsf{S}'', \Gamma \vdash \Omega \rightsquigarrow \mathsf{S}', \Gamma'$ hold for some S'' and Ω.

Correctness. The main correctness properties (proved in [13]) are:

Theorem 3. *Suppose* $\mathsf{S}, \Gamma \vdash \Omega \rightsquigarrow \mathsf{S}', \Gamma'$. *If* $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma \models_{\mathsf{S}} \omega : \Omega$ *and* $\sigma \models \omega \rightsquigarrow \sigma'$ *then* $\sigma' \models_{\mathsf{S}'} \gamma : \Gamma'$.

Corollary 1. *Suppose* $\mathsf{S}, \Gamma \vdash u \rightsquigarrow \mathsf{S}', \Gamma'$ *and* $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models u \rightsquigarrow \sigma'$. *Then* $\sigma' \models_{\mathsf{S}'} \gamma : \Gamma'$.

6 Implementation

We have developed a prototype implementation of type and effect analysis and schema alteration in OCaml, to demonstrate feasibility of the approach. We have tested the implementation on a number of examples from the XQuery Update Use Cases [17]. For these small updates and schemas, schema alteration takes under 0.1s. Space limitations preclude a full discussion of examples; we discuss the accuracy of the resulting schemas in [13]. However, there are several possible avenues for improvement:

– Currently flattening produces large numbers of temporary type names, increasing the size of output and limiting readability. An obvious approach would be to do flattening only "on demand", when further navigation effect application requires exploration of the schema below a certain type name.
– Both effect application and flattening can produce redundancy in type expressions. Currently we simplify the regular expression types in the output schema using basic rules such as $(), \tau \equiv \tau \equiv \tau, ()$ and $(\tau^*)^* \equiv \tau^*$. Postprocessing using full-fledged regular expression simplification might be more useful [18].

– We have implemented a simple, but inaccurate alias analysis: we assume that two types alias if they have the same root element label. For the DTD-based examples in [13], this is exact. However, for more complex updates and schemas, we may need more sophisticated alias analysis to produce useful results.
– Type and effect inference appears to be worst-case exponential in the presence of nested for-loops. In practice, typical queries and updates are small and of low nesting depth, so we expect the size of the schema to be the dominant factor. The type, effect and schema alteration algorithms appear to be polynomial in the size of the schema for fixed expressions. Further study of the complexity of our analysis in the worst case or for typical cases may be of interest.

7 Related and Future Work

There is a great deal of related work on static analysis of fragments of XPath [16], regular expression types and schema languages [8,19], and XML update language designs [3,4,5,6,7]. We restrict attention to closely related work.

Cheney developed a typed XML update language called Flux [4], building on the XQuery type system of Colazzo et al. [10]. Flux differs significantly from XQuery Update and handles only child and descendant axes, but its semantics is much simpler.

Static analysis problems besides typechecking have also been studied for XML or object query/update languages. Bierman [20] developed an effect analysis that tracks object-identifier generation side-effects in OQL queries. Benedikt et al. [3,12] presented static analyses for optimizing updates in UpdateX, a precursor to XQuery Update. Ghelli et al. [5] present a *commutativity analysis* for an XML update language. Roughly speaking, two updates u_1, u_2 commute if they have the same side-effects and results no matter which order they are run. Their update language also differs from XQuery Update in important ways: in particular, the intended semantics of the W3C proposal (as formalized in this paper, for example) seems to imply that u_1, u_2 will always have the same (potential) side-effects as u_2, u_1.

There is also prior work on typechecking for XML transformations (see e.g. Møller and Schwartzbach [21] for an overview). Much of this work focuses on decidable subproblems where both input and output schemas are given in advance, whereas we focus on developing sound, practical schema alteration techniques for general queries and updates. Also, there is no obvious mapping from XQuery Updates to transducers.

Balmin [22] and Barbosa et al. [23] present efficient dynamic techniques for checking that atomic updates preserve a fixed schema. These techniques are exact, but impose run-time overhead on all updates, and do not deal with changes to schemas. Raghavachari and Shmueli [15] give efficient algorithms for revalidating data after updates to either the data or schema, but their approach places stronger restrictions on schemas. It would be interesting to combine static and dynamic revalidation techniques.

Building partly on the type effect analyses in this paper, we develop a schema-based *independence analysis* for XML queries and updates [24]. A query q and update u are statically *independent* if, roughly speaking, for any initial store, running q yields the same results as applying u and then running q. Static independence checking is useful for avoiding expensive recomputation of query results and perhaps also for managing safe concurrent access to XML databases.

Aliasing is a fundamental problem in static analysis, and has been studied in a wide variety of previous contexts. We envision applying ideas from region inference [25] or more advanced shape analysis techniques [26] to obtain more accurate alias information. Aliasing also arises in object-oriented programming languages in settings such as type-safe object reclassification [27] and typestates [28], and ideas from this work may also be useful for dealing with aliasing in XML updates.

8 Conclusions

XML update languages are an active area of study, but so far little is known about typechecking and static analysis for such languages. In this paper we have given an operational semantics for the W3C's XQuery Update Facility 1.0 and developed the first (to our knowledge) sound type system for this language (although many details are relegated to the technical report [13]). As a Candidate Recommendation, XQuery Update is still a work in progress and we hope that our work will help improve the standard as well as provide a foundation for future study of XML updates.

Acknowledgments. Cheney was supported by a Royal Society Research Fellowship and by EPSRC grant EP/F028288/1, "Heterogeneous and Permanent Data". Benedikt is supported in part by EPSRC EP/G004021/1 (the Engineering and Physical Sciences Research Council, UK).

References

1. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML query language. W3C Recommendation (January 2007), http://www.w3.org/TR/xquery
2. Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation (January 2007), http://www.w3.org/TR/xquery-semantics/
3. Benedikt, M., Bonifati, A., Flesca, S., Vyas, A.: Adding updates to XQuery: Semantics, optimization, and static analysis. In: Florescu, D., Pirahesh, H. (eds.) XIME-P (2005)
4. Cheney, J.: FLUX: FunctionaL Updates for XML. In: ICFP (2008)
5. Ghelli, G., Rose, K., Siméon, J.: Commutativity analysis for XML updates. ACM Trans. Database Syst. 33(4), 1–47 (2008)
6. Sur, G., Hammer, J., Siméon, J.: UpdateX - an XQuery-based language for processing updates in XML. In: PLAN-X (2004)

7. Chamberlin, D., Robie, J.: XQuery update facility 1.0. W3C Candidate Recommendation (August 2008), `http://www.w3.org/TR/xquery-update-10/`
8. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)
9. Schwentick, T.: Automata for XML – A Survey. Journal of Computer and Systems Science 73, 289–315 (2007)
10. Colazzo, D., Ghelli, G., Manghi, P., Sartiani, C.: Static analysis for path correctness of XML queries. J. Funct. Program. 16(4-5), 621–661 (2006)
11. Papakonstantinou, Y., Vianu, V.: Type inference for views of semistructured data. In: PODS (2000)
12. Benedikt, M., Bonifati, A., Flesca, S., Vyas, A.: Verification of tree updates for optimization. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 379–393. Springer, Heidelberg (2005)
13. Benedikt, M., Cheney, J.: Types, effects, and schema evolution for XQuery update facility 1.0, `http://homepages.inf.ed.ac.uk/jcheney/publications/-drafts/w3c-update-types-tr.pdf`
14. Stearns, R., Hunt, H.: On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. SIAM J. Comput. 14, 598–611 (1985)
15. Raghavachari, M., Shmueli, O.: Efficient revalidation of XML documents. IEEE Trans. on Knowl. and Data Eng. 19(4), 554–567 (2007)
16. Benedikt, M., Koch, C.: XPath leashed. ACM Comput. Surv. 41(1), 1–54 (2008)
17. Manolescu, I., Robie, J.: XQuery update facility use cases. W3C Candidate Recommendation (March 2008), `http://www.w3.org/TR/xqupdateusecases`
18. Trejo Ortiz, A.A.R., Anaya, G.F.: Regular expression simplification. Math. Comput. Simul. 45(1-2), 59–71 (1998)
19. Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and complexity of XML Schema. ACM Transactions on Database Systems 31(3), 770–813 (2006)
20. Bierman, G.M.: Formal semantics and analysis of object queries. In: SIGMOD (2003)
21. Møller, A., Schwartzbach, M.I.: The design space of type checkers for XML transformation languages. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 17–36. Springer, Heidelberg (2004)
22. Balmin, A., Papakonstantinou, Y., Vianu, V.: Incremental validation of XML documents. ACM Transactions on Database Systems 29(4), 710–751 (2004)
23. Barbosa, D., Mendelzon, A.O., Libkin, L., Mignet, L., Arenas, M.: Efficient incremental validation of XML documents. In: ICDE. IEEE Computer Society, Los Alamitos (2004)
24. Benedikt, M., Cheney, J.: Schema-based independence analysis for XML updates. In: VLDB (2009)
25. Henglein, F., Makholm, H., Niss, H.: Effect types and region-based memory management. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages. MIT Press, Cambridge (2005)
26. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst. 20(1), 1–50 (1998)
27. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: Fickle: Dynamic object re-classification. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 130–149. Springer, Heidelberg (2001)
28. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17(2), 1–34 (2008)