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 are 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, schema alteration, and sound typing rules for updates and "transform" queries.

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 [8, 12]. 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, 10, 13, 24]). While no clear winner has emerged so far, the W3C has introduced the XQuery Update Facility [9], 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* [15]. Regular expression types are closely related to tree automata [22] and have been employed in a number of other settings [11]. 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[3] $doc[a[], b[], c[d[]]]$. 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 $doc[d[], b[], c[d[]]]$.

Secondly, 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 take this into account, and are unsound:

Example 2. 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 [9], 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 adjust the types of updated variables after the updates are performed. One trivial, but unsatisfying way to do so is to set updated variables' types to `Any`, a type that matches any XML document. A more appealing alternative is to calculate accurate types for the updated variables, using the same techniques as are needed to predict the results of updates.

Uses of ancestor or sibling XPath axes further complicate typechecking:

Example 3. Consider the update expression:

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

[3] For brevity, we use compact, XDuce-style notation [15] for XML trees and types.

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.

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 [19]. Nevertheless, it is worthwhile to find sound, static overapproximations to the result of an XML query or update. We are more interested in 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, 4] and FLUX [10], and other XML update proposals [13]. However, no prior work applies directly to the W3C's current XQuery Update proposal. While Benedikt et al. [3, 4] considered a language similar to XQuery Update, they did not investigate typechecking. Cheney [10] 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 [13].

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

- *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 handling 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.

For ease of exposition, we first consider type and effect analysis and schema alteration separately in the absence of "transform" queries; in their presence, these problems are mutually dependent. We also leave out the "replace value of" operation [9]. We omit proofs and standard definitions; these are placed in an appendix [5].

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 an effect analysis for update expressions and proves its soundness. Section 5 presents a schema alteration algorithm that applies a static effect to a schema. Section 6 shows how to extend these results to handle "transform" queries. We discuss a prototype implementation in Section 7. Section 8 discusses related and future work and Section 9 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 the appendix.

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 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 label $\mathtt{L}$ to a fresh subtree, collecting the resulting labels in list $\mathtt{L}'$. This judgment is defined formally in the appendix.

Regular expression types Following previous work [15, 11, 10], 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 $\mathtt{T}, \mathtt{T}', \ldots \in TName$ 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:

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

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

Fig. 1. Validation rules

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.*

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 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 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 We say that $\mathtt{T}$ and $\mathtt{T}'$ *may alias*[4] (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 $\sigma \models_{\mathsf{S}} l : \mathtt{T}'$.

A sufficient (but not necessary) condition to establish that $\mathtt{T}$ and $\mathtt{T}'$ do not alias is that the languages corresponding to the two types are disjoint, that is, no tree can match both $\mathtt{T}$ and $\mathtt{T}'$. Disjointness is decidable for regular languages, and for restricted expressions (e.g. 1-unambiguous), tractable procedures are known [23, 20]. An exact algorithm for determining disjointness is also possible, via reduction to tree automata nonemptiness. 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 $\mathtt{T}' \in \mathrm{alias}_{\mathsf{S}}(\mathtt{T})$.

[4] 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.

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

$$\begin{array}{rl} step ::= ax::\phi & \phi ::= * \mid n \mid \texttt{text} \\ ax ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \mid \texttt{parent} \mid \texttt{ancestor} \mid \cdots \end{array}$$

The semantics and static analysis problems for XPath have been well-studied [6, 22]. 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. [11].

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

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 $\texttt{for}\ x \in q\ \texttt{return}\ q'$ 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 (see Section 6). The rules defining these judgments are given in an appendix; here are two illustrative rules:

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

3 Core XQuery Updates

Atomic updates We consider atomic updates of the form:

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

Here, the direction 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.

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

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 $\texttt{insert}\ q\ \texttt{d}\ q_0$, which says to insert a copy of q in position d relative to the value of q_0; deletion $\texttt{delete}\ q_0$, which says to delete the value of q_0; renaming $\texttt{rename}\ q_0\ \texttt{as}\ a$, which says to rename the value of q_0 to a and replacement $\texttt{replace}\ q_0\ \texttt{with}\ q$, 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 $\sigma, \gamma, x \in \texttt{L} \models^\star u \Rightarrow \sigma', \omega$ 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 $\mathrm{sanitycheck}(\omega)$ 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 [9]. 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 [9], which 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 $\texttt{stage}(\iota)$ 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 [8, 11]; 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. [11] and so are relegated to an appendix. The key new rules with respect to previous work are those for node construction and XPath axis steps, respectively:

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

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. We use the following *effect expressions*:

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Semantics of atomic updates

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

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

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

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

Fig. 4. Update application

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

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

Fig. 5. Effect validity rules (regular expression forms omitted)

$$\frac{}{\mathsf{S}; \Gamma \vdash () : \emptyset; \mathsf{S}} \qquad \frac{\mathsf{S}; \Gamma \vdash u_1 : \Omega_1; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma \vdash u_2 : \Omega_2; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash u_1, u_2 : \Omega_1, \Omega_2; \mathsf{S}_2} \qquad \frac{\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma, x : \tau \vdash u : \Omega; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash \mathtt{let}\ x := e\ \mathtt{in}\ u : \Omega; \mathsf{S}_2}$$

$$\frac{\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}_0 \quad \mathsf{S}_0; \Gamma \vdash u_1 : \Omega_1; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma \vdash u_2 : \Omega_2; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash \mathtt{if}\ q\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 : \Omega_1 | \Omega_2; \mathsf{S}_2} \qquad \frac{\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma; x \in \tau \vdash^\star s : \Omega; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash \mathtt{for}\ x \in e\ \mathtt{return}\ s : \Omega; \mathsf{S}_2}$$

$$\frac{\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma \vdash q' : \mathtt{T}; \mathsf{S}_2}{\mathsf{S}; \Gamma \vdash \mathtt{insert}\ q\ \mathtt{d}\ q' : \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}); \mathsf{S}_2} \qquad \frac{\mathsf{S}; \Gamma \vdash q : \mathtt{T}; \mathsf{S}' \quad \mathsf{S}'(\mathtt{T}) = b[\tau]}{\mathsf{S}; \Gamma \vdash \mathtt{rename}\ q\ \mathtt{as}\ a : \mathtt{ren}(\mathtt{T}, a); \mathsf{S}'}$$

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

$$\frac{\mathsf{S}; \Gamma, x : \mathtt{T} \vdash u : \Omega; \mathsf{S}'}{\mathsf{S}; \Gamma; x \in \mathtt{T} \vdash^\star u : \Omega; \mathsf{S}'} \qquad \frac{}{\mathsf{S}; \Gamma; x \in () \vdash^\star u : \epsilon; \mathsf{S}} \qquad \frac{\mathsf{S}; \Gamma; x \in \tau \vdash^\star u : \Omega; \mathsf{S}'}{\mathsf{S}; \Gamma; x \in \tau^* \vdash^\star u : \Omega^*; \mathsf{S}'}$$

$$\frac{\mathsf{S}; \Gamma; x \in \tau_1 \vdash^\star u : \Omega_1; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma; x \in \tau_2 \vdash^\star u : \Omega_2; \mathsf{S}_2}{\mathsf{S}; \Gamma; x \in \tau_1, \tau_2 \vdash^\star u : \Omega_1, \Omega_2; \mathsf{S}_2}$$

$$\frac{\mathsf{S}; \Gamma; x \in \tau_1 \vdash^\star u : \Omega_1; \mathsf{S}_1 \quad \mathsf{S}_1; \Gamma; x \in \tau_2 \vdash^\star u : \Omega_2; \mathsf{S}_2}{\mathsf{S}; \Gamma; x \in \tau_1 | \tau_2 \vdash^\star u : \Omega_1 | \Omega_2; \mathsf{S}_2}$$

Fig. 6. Update effect-inference rules

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, as defined in Figure 6. 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 possible effects an update may have on a schema. Given input type context Γ, schema S and effect Ω we want to infer a suitable output schema S' and type context Δ. The rough idea is as follows:

1. Augment the input schema S to $\tilde{\mathsf{S}}$ 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
3. Simulate the effects of each stage of atomic update application on $\tilde{\mathsf{S}}$.
4. Finally, flatten the updated $\tilde{\mathsf{S}}$ 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 $\tilde{\mathsf{S}} \mapsto doc[\mathtt{T}], \mathtt{T} \mapsto a[\mathtt{U},\mathtt{V}], \mathtt{U} \mapsto b[], \mathtt{V} \mapsto c[]$, and $\Gamma = x : \mathsf{S}$.

Using the schema S we will form a new schema $\tilde{\mathsf{T}}$ 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 $\tilde{\mathsf{S}}$ 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 $\tilde{\mathsf{S}} \mapsto doc[\tilde{\mathtt{T}}_1|\tilde{\mathtt{T}}_2|()], \tilde{\mathtt{T}}_1 \mapsto a[(\mathtt{U},\mathtt{V})^*,\tilde{\mathtt{U}},\tilde{\mathtt{V}}], \tilde{\mathtt{T}}_2 \mapsto b[(\mathtt{U},\mathtt{V})^*,\tilde{\mathtt{U}},\tilde{\mathtt{V}}]$. 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 $x : \tilde{\mathsf{S}}$ to reflect the change.

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

Preprocessing We define the augmented schema $\tilde{\mathsf{S}}$ as follows. For each rule $\mathtt{T} \mapsto a[\tau]$ in S, we introduce rules

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

Initial augmented schema:

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

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

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

Schema changes:

Phase 1:	Phase 2:	Phase 3:	Phase 4:
$\tilde{\mathtt{U}}_r \mapsto b[\tilde{\mathtt{U}}_c] \mid d[\tilde{\mathtt{U}}_c]$	$\tilde{\mathtt{U}}_{\leftarrow} \mapsto \mathtt{V}^*$	$\tilde{\mathtt{V}}_r \mapsto c[\tilde{\mathtt{V}}_c] \mid \mathtt{U}^*$	$\tilde{\mathtt{T}}_r \mapsto a[\tilde{\mathtt{T}}_c] \mid ()$

Result schema (after some equational simplifications):

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

Re-flattened schema:

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

Fig. 7. Detailed example

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 $()$.

Alias analysis Before proceeding, we pre-compute sound aliasing information for S, defining sets $\mathrm{alias}(\mathtt{T}) = \{\mathtt{U} \mid \mathrm{alias}_{\mathsf{S}}(\mathtt{T})\}$.

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 $I_{\downarrow}(\mathtt{T}) = \{\mathtt{U} \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \mathtt{ins}(\tau, \downarrow, \mathtt{T}') \in |\Omega|, \mathtt{U} \in |\tau|\}$. 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 $N(\mathtt{T}) = \{b \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \mathtt{ren}(\mathtt{T}', b) \in |\Omega|\}$, 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 $I_{\mathtt{d}}(\mathtt{T}) = \{\tau \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}') \in |\Omega|\}$ 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 $R(\mathtt{T}) = \{\tau \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\, \exists \tau.\, \mathtt{repl}(\mathtt{T}', \tau) \in |\Omega|\}$ 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 $\mathtt{U} \in \mathrm{alias}(\mathtt{T})$, 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 the appendix) 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 Transform queries

In this section, we sketch how to extend the above semantics and type system to handle "transform" queries. We extend the syntax of queries as follows:

$$q ::= \cdots \mid \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q \qquad \theta ::= \theta, x := q \mid \bullet$$

Here, $\bullet$ is an empty binding list. The transform expression $\mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q$ first evaluates the queries in θ and binds them to their corresponding variables in the environment, then runs the update u and applies the results, and finally returns the value of q. Only copied nodes bound to the variables in θ may be updated. The semantics of transforms is given by the rules in Figure 8, where we introduce an auxiliary judgment $\sigma, \gamma \models \theta \Rightarrow \sigma', \delta$ for evaluating the θ-bindings. We use an auxiliary function $\mathrm{target}(\omega)$ to check that ω only updates freshly copied nodes.

Typing rules for transform queries are shown in Figure 9. It is crucial that the operational semantics rules forbid modifying nodes in σ. The soundness of the typing rules rely on this fact since they assume data reachable from Γ is immutable. Using Theorem 3, we can prove Theorem 1, Theorem 2 and Corollary 1 in the presence of transforms. There are some subtleties, due partly to the fact that transforms must only modify copied nodes; these are detailed in the appendix.

$$\frac{\begin{array}{c}\sigma,\gamma \models \theta \Rightarrow \sigma_1,\delta \quad \sigma_1,\gamma \uplus \delta \models u \Rightarrow \sigma_2,\omega \\ \mathrm{sanitycheck}(\omega) \quad \mathrm{target}(\omega) \subseteq \mathrm{dom}(\sigma_1) - \mathrm{dom}(\sigma) \\ \sigma_2 \models \omega \rightsquigarrow \sigma_2' \quad \sigma_2',\gamma \uplus \delta \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma_3,\mathtt{L}\end{array}}{\sigma,\gamma \models \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q \Rightarrow \sigma_4,\mathtt{L}}$$

$$\frac{}{\sigma,\gamma \models \bullet \Rightarrow \sigma,\bullet} \qquad \frac{\sigma,\gamma \models \theta \Rightarrow \sigma',\delta \quad \sigma',\gamma \uplus \delta \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma'',\mathtt{L}}{\sigma,\gamma \models \theta, x := q \Rightarrow \sigma'',\delta[x := \mathtt{L}]}$$

Fig. 8. Semantics of transform queries

$$\frac{\mathsf{S};\Gamma \vdash \theta : \Delta;\mathsf{S}_1 \quad \mathsf{S}_1;\Delta \vdash u : \Omega;\mathsf{S}_2 \quad \mathsf{S}_2,\Delta \vdash \Omega \rightsquigarrow \mathsf{S}_2',\Delta' \quad \mathsf{S}_2';\Gamma,\Delta' \vdash q : \tau;\mathsf{S}_3}{\mathsf{S};\Gamma \vdash \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q : \tau;\mathsf{S}_3}$$

$$\frac{}{\mathsf{S};\Gamma \vdash \bullet : \bullet;\mathsf{S}} \qquad \frac{\mathsf{S};\Gamma \vdash \theta : \Delta;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma,\Delta \vdash q : \tau;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \theta, x := q : \Delta, x : \tau;\mathsf{S}_2}$$

Fig. 9. Typechecking rules for transform queries

7 Implementation

We have developed a prototype implementation in OCaml, to demonstrate feasibility of the approach. We have tested the implementation on a number of examples from the XQuery Update Use Cases [16]. 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 an appendix.

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 $(\tau^*)^* \equiv \tau^*$. Post-processing using full-fledged regular expression simplification might be more useful [25].
- 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 examples in the appendix, this naive analysis is reasonably accurate. However, for more complex updates and schemas, we may need more sophisticated alias analysis to produce useful results. We envision using ideas from region inference [14] or more advanced shape analysis techniques [21] to obtain more accurate alias information.
- 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.

8 Related and future work

There is a great deal of related work on static analysis of fragments of XPath [6], regular expression types and schema languages [15, 17], and XML update language designs [3, 10, 13, 24, 9]. We restrict attention to closely related work.

Cheney developed a typed XML update language called FLUX [10], building on the XQuery type system of Colazzo et al. [11]. FLUX differs significantly from XQuery Update and handles only child and descendant axes.

Static analysis problems besides typechecking have also been studied for XML or object query/update languages. Bierman [7] developed an effect analysis that tracks object-identifier generation side-effects in OQL queries. Benedikt et al. [3, 4] presented static analyses for optimizing updates in UpdateX, a precursor to XQuery Update. Ghelli et al. [13] 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.

There is also prior work on typechecking for XML transformations (see e.g. Møller and Schwartzbach [18] 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 (sublanguages of) XQuery Updates to transducers.

Balmin [1] and Barbosa et al. [2] 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 [20] 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.

In ongoing work, building partly on the type effect analyses in this paper, we are developing a schema-based *independence analysis* for XML queries and updates. 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 would be useful for avoiding expensive recomputation of query results and for managing safe concurrent access to XML databases.

As stated in the introduction, we have prioritized soundness, accessibility and ease of implementation over technical sophistication, but it would be desirable to develop more sophisticated techniques (as outlined in Section 7).

9 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. 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.

References

1. A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.
2. D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE*. IEEE Computer Society, 2004.
3. Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In Daniela Florescu and Hamid Pirahesh, editors, *XIME-P*, 2005.
4. Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Verification of tree updates for optimization. In *CAV*, 2005.
5. Michael Benedikt and James Cheney. 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`.
6. Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1):1–54, 2008.
7. G. M. Bierman. Formal semantics and analysis of object queries. In *SIGMOD*, 2003.
8. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery`.
9. Don Chamberlin and Jonathan Robie. XQuery update facility 1.0. W3C Candidate Recommendation, August 2008. `http://www.w3.org/TR/xquery-update-10/`.
10. James Cheney. FLUX: FunctionaL Updates for XML. In *ICFP*, 2008.
11. Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
12. Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery-semantics/`.
13. Giorgio Ghelli, Kristoffer Rose, and Jérôme Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4):1–47, 2008.
14. Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.
15. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
16. Ioana Manolescu and Jonathan Robie. XQuery update facility use cases. W3C Candidate Recommendation, March 2008. `http://www.w3.org/TR/xqupdateusecases`.
17. Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systens*, 31(3):770–813, 2006.
18. Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, pages 17–36, 2005.
19. Yannis Papakonstantinou and Victor Vianu. Type inference for views of semistructured data. In *PODS*, 2000.
20. Mukund Raghavachari and Oded Shmueli. Efficient revalidation of XML documents. *IEEE Trans. on Knowl. and Data Eng.*, 19(4):554–567, 2007.
21. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

22. Thomas Schwentick. “Automata for XML – A Survey”. *Journal of Computer and Systems Science*, 73:289–315, 2007.
23. R.B. Stearns and H.B. Hunt. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM J. Comput.*, 14:598–611, 1985.
24. Gargi Sur, Joachim Hammer, and Jérôme Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.
25. Alejandro A. R. Trejo Ortiz and Guillermo Fernández Anaya. Regular expression simplification. *Math. Comput. Simul.*, 45(1-2):59–71, 1998.

A Examples

In this Appendix we give a few examples showing that our update typechecker produces reasonable results on real examples, drawn from the W3C's XQuery Update Use Cases.

We focus on thee first collection of queries, involving "relational" XML documents defining users, items and bids, as might be needed in an online auction. There is no DTD specified in the W3C Use Cases recommendation, so we defined our own schema that matches their example data.

```
type Doc          = doc[Items,Users,Bids]
and Items         = items[Item_tuple*]
and Users         = users [User_tuple*]
and Bids          = bids [Bid_tuple*]
and Item_tuple    = item_tuple [Itemno,
                                Description,
                                Offered_by,
                                Start_date,
                                End_date,
                                Reserve_price]
and User_tuple     = user_tuple [Userid,
                                 Name,
                                 Rating?]
and Bid_tuple      = bid_tuple [Userid,
                                Itemno,
                                Bid,
                                Bid_date]
and Itemno         = itemno [string]
and Description    = description [string]
and Offered_by     = offered_by [string]
and Start_date     = start_date [string]
and End_date       = end_date [string]
and Reserve_price = reserve_price [string]
and Userid         = userid [string]
and Name           = name [string]
and Rating         = rating[string]
and Bid            = bid [string]
and Bid_date       = bid_date [string]
```

We skip queries Q3, Q5, Q8 and Q9. Q3 is similar to Q2, from the point of view of typechecking. Queries 5, 8 and 9 illustrate constraint checking facilities that are not part of the W3C XQuery Update proposal or our model. We will illustrate our approach using the remaining queries in the relational data use case.

In several of the examples, we observe that the result schema appears to be "equivalent" to the input schema, or that the input schema is "preserved". By this we mean that the type assigned to `$doc` in the input is equivalent (as a regular tree language) to the type inferred for `$doc` in the result. Formally, we define:

Definition 2 (Schema equivalence). *Let* $(\mathsf{S}, \Gamma), (\mathsf{S}', \Gamma')$ *be pairs each consisting of schema and static environment. We say that* (S, Γ) *is* equivalent *to* (S', Γ') *provided:*

1. $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$*, and*
2. *For each* $x \in \mathrm{dom}(\Gamma)$*, the regular tree language denoted by* $\Gamma(x)$ *in* S *is the same as that denoted by* $\Gamma'(x)$ *in* S'*.*

If Γ *and* Γ' *are obvious from context, we just say that* S *and* S' *are equivalent.*

Example 5 (Q1). The first query is:

```
insert nodes
  <user_tuple>
    <userid>U07</userid>
    <name>Annabel Lee</name>
  </user_tuple>
into $doc/users
```

Effect analysis yields the effect

$$\texttt{ins}(user\_tuple[Userid, Name], \downarrow, Users) \ .$$

Schema alteration produces an output schema that is the same as the input schema except that it redefines:

```
Users -> users[user_tuple[Userid, Name]*,
               user_tuple[Userid, Name, Rating?]*,
               user_tuple[Userid, Name]*]
```

This is equivalent to the original schema.

Example 6 (Q2). Query 2 is:

```
let $uid :=
$doc/users/user_tuple[name="Annabel Lee"]/userid
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1001</itemno>
      <bid>60</bid>
      <bid_date>1999-02-01</bid_date>
    </bid_tuple>
  into $doc/bids
```

The result schema we obtain after typechecking this update is identical to the input schema except for the change:

```
Bids -> bids[bid_tuple[userid[string**],Itemno,Bid,Bid_date]*,
             bid_tuple[Userid,Itemno,Bid,Bid_date],
             bid_tuple[userid[string**],Itemno,Bid,Bid_date]*]
```

It is worth noting that $string^{**} = string$, so the above rule is equivalent to:

```
Bids -> bids[bid_tuple[Userid,Itemno,Bid,Bid_date]*]
```

hence, the result schema is equivalent to the input schema.

Example 7 (Q4). Query 4 is:

```
let $user := $doc/users/user_tuple[name="Annabel Lee"]
return
  if ($user/rating)
    then replace value of node $user/rating with "B"
    else insert node <rating>B</rating> into $user
```

In our implementation, we translate the "replace value of" subexpression to

```
replace node $user/rating with <rating>B</rating>
```

The result schema we obtain after typechecking this update is identical to the input schema except for the change:

```
User_tuple -> user_tuple[Rating*,Userid,
                         Rating*,Name,
                         Rating*,(Rating|Rating?),
                         Rating*]
```

Because the "insert into" operation is used, we have to assume that a rating element could be inserted anywhere in the child sequence of `user_tuple`. Also, our analysis cannot detect that the insert and replace operations are mutually exclusive.

If we add the clause `as last` to the insert in Q4, we get a more precise type:

```
User_tuple -> user_tuple[Userid,
                         Name,
                         (Rating|Rating?),
                         Rating*]
```

which is equivalent to

```
User_tuple -> user_tuple[Userid,
                         Name,
                         Rating*]
```

The query (modified with "as last") does preserve the schema, but our analysis cannot certify this, since it does not keep track of the fact that the insert and replace are mutually exclusive.

Example 8 (Q6). Query 6 tests deletion behavior, It deletes tuples having to do with a specific user.

```
let $user := $doc/users/user_tuple[name="Dee Linquent"]
let $items := $doc/items/item_tuple[offered_by=$user/userid]
let $bids := $doc/bids/bid_tuple[userid=$user/userid]
return (
```

```
  delete nodes $user,
  delete nodes $items,
  delete nodes $bids
)
```

Our typechecker verifies that this update preserves the schema.

Example 9 (Q7). Query 7 inserts a comment:

```
insert nodes
  <comment>This is a bargain !</comment>
as last into $doc/items/item_tuple[itemno=1002]
```

Our analysis produces a result schema that differs from the input schema only as follows:

```
Item_tuple -> item_tuple[Itemno,
                         Description,
                         Offered_by,
                         Start_date,
                         End_date,
                         Reserve_price,
                         comment[string]*]
```

Note again that we do not know how many comments will be added by the insert. A more precise type would have $comment[string^?]$ instead of $comment[string]^*$. However, the comment must be optional because it may not be added to some item tuples.

B Proofs

For expository purposes, the rules given in the main body of the paper left out a number of details. In this appendix, we present the full system, including rules omitted from the main body of the paper. The modified definitions are more restrictive since we take more care to distinguish between *mutable* and *immutable* parts of the store. The main changes are in Figure 10, Figure 13, Figure 14, and and Figure 15 and are discussed further below.

We consider stores in which each location is annotated with a *mutability annotation*:

$$m ::= \mathsf{ro} \mid \mathsf{rw}$$

Here, ro stands for *read-only* data and rw for *read-write* data. We consider stores to be maps σ from locations to pairs of constructors k and mutability annotations m. We often write such pairs as k^m, e.g. $\sigma(l) = a[\mathtt{L}]^{\mathsf{ro}}$ means that l points to $a[\mathtt{L}]$ and is read-only.

We write $\mathsf{annot}(m, \sigma, \gamma)$ for the result of setting the annotations in locations reachable from γ in σ to m. We will set data to be read-write immediately before applying updates and we re-set it to be read-only after update application finishes. See Figure 15 and Figure 14 for these changes.

The reason we introduce annotations is to ensure that source data referenced from update sequences ω is immutable, and thus neither its structure nor its type can change while applying updates. The update sequence validity rules (Figure 12) are adjusted to require that all input data in insert and replace update instructions is annotated ro. The update application rules Figure 13 are adjusted so that it is only possible to modify labels that are annotated rw.

We introduce read-only versions of the validity judgments $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ and $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \omega : \Omega$ that require all relevant data to be read-only (see Figure 10 for example; the rules for update sequences are similar). We write $\sigma \sqsubseteq \sigma'$ to mean that $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$ and that σ agrees with σ' on all read-only locations.

We write $\mathsf{S} \leq \mathsf{S}'$ to indicate that $\mathrm{dom}(\mathsf{S}) \subseteq \mathrm{dom}(\mathsf{S}')$ and for every $\mathtt{X}$, we have $L_{\mathsf{S}}(\mathtt{X}) \subseteq L_{\mathsf{S}'}(\mathtt{X})$, where $L_{\mathsf{S}}(\mathtt{X})$ is the set of trees matching $\mathtt{X}$ in S.

Since many judgments modify or extend the state or extend the schema, we need the following monotonicity properties:

Lemma 2. *If $\sigma \models \omega \rightsquigarrow \sigma'$ then $\sigma \sqsubseteq \sigma'$.*

Proof. Immediate since atomic update evaluation only modifies nodes labeled rw.

Lemma 3 (Monotonicity). *If $\sigma \sqsubseteq \sigma'$ then:*

1. *$\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ implies $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$*
2. *$\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ implies $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$*
3. *$\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \omega : \Omega$ implies $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}} \omega : \Omega$*

Lemma 4 (Schema monotonicity). *If $\mathsf{S} \leq \mathsf{S}'$ then:*

1. *$\sigma \models_{\mathsf{S}} \mathtt{L} : \tau$ implies $\sigma \models_{\mathsf{S}'} \mathtt{L} : \tau$*
2. *$\sigma \models_{\mathsf{S}} \gamma : \Gamma$ implies $\sigma \models_{\mathsf{S}'} \gamma : \Gamma$*
3. *$\sigma \models_{\mathsf{S}} \omega : \Omega$ implies $\sigma \models_{\mathsf{S}'} \omega : \Omega$*

B.1 Soundness of schema alteration

We will first prove Theorem 3 and then prove the type and effect soundness theorems simultaneously for the query and update languages (including transform queries).

First, let $\tilde{\mathsf{S}}_0$ be the initial augmented schema for S, and let $\tilde{\Gamma}$ be the type context obtained by replacing each type name $\mathtt{T}$ with $\tilde{\mathtt{T}}$. Since $\mathsf{S} \subseteq \tilde{\mathsf{S}}_0$, it is trivial to see that:

Lemma 5. $\mathsf{S} \leq \tilde{\mathsf{S}}_0$.

We prove the soundness of schema alteration in four stages, corresponding to the four stages of update application. Recall that we defined

$$\mathrm{alias}(\mathtt{T}) = \mathrm{alias}_{\mathsf{S}}(\mathtt{T}) \ .$$

We also introduce the following definitions:

$$\frac{\sigma(l) = a[\mathtt{L}]^{\mathsf{ro}} \quad \sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} l : a[\tau]} \quad \frac{\sigma(l) = \mathsf{text}[s]^{\mathsf{ro}}}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} l : \delta} \quad \frac{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \mathsf{S}(\mathtt{T})}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \mathtt{T}}$$

$$\frac{}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} () : ()} \quad \frac{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L}_1 : \tau_1 \quad \sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L}_2 : \tau_2}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L}_1, \mathtt{L}_2 : \tau_1, \tau_2} \quad \frac{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau_i}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau_1 | \tau_2} \quad \frac{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : () \mid \tau, \tau^*}{\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau^*}$$

Fig. 10. Read-only validity rules

$$\frac{\sigma(l) = a[\mathtt{L}]^{m} \quad \sigma \models_{\mathsf{S}} \mathtt{L} : \tau}{\sigma \models_{\mathsf{S}} l : a[\tau]} \quad \frac{\sigma(l) = \mathsf{text}[s]^{m}}{\sigma \models_{\mathsf{S}} l : \delta} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \mathsf{S}(\mathtt{T})}{\sigma \models_{\mathsf{S}} \mathtt{L} : \mathtt{T}}$$

$$\frac{}{\sigma \models_{\mathsf{S}} () : ()} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L}_1 : \tau_1 \quad \sigma \models_{\mathsf{S}} \mathtt{L}_2 : \tau_2}{\sigma \models_{\mathsf{S}} \mathtt{L}_1, \mathtt{L}_2 : \tau_1, \tau_2} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_i}{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau_1 | \tau_2} \quad \frac{\sigma \models_{\mathsf{S}} \mathtt{L} : () \mid \tau, \tau^*}{\sigma \models_{\mathsf{S}} \mathtt{L} : \tau^*}$$

Fig. 11. Validity rules

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

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

$$\frac{}{\sigma \models_{\mathsf{S}} \epsilon : \epsilon} \quad \frac{\sigma \models_{\mathsf{S}} \omega_1 : \Omega_1 \quad \sigma \models_{\mathsf{S}} \omega_2 : \Omega_2}{\sigma \models_{\mathsf{S}} \omega_1, \omega_2 : \Omega_1, \Omega_2} \quad \frac{\sigma \models_{\mathsf{S}} \omega : \Omega_i}{\sigma \models_{\mathsf{S}} \omega : \Omega_1 | \Omega_2} \quad \frac{\sigma \models_{\mathsf{S}} \omega : \epsilon | \Omega; \Omega^*}{\sigma \models_{\mathsf{S}} \omega : \Omega^*}$$

Fig. 12. Effect validity rules

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma \models \mathtt{ins}(\mathtt{L}, \leftarrow, l) \rightsquigarrow \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}]} \quad \frac{\sigma(l) = a[\mathtt{L}']^{\mathsf{rw}}}{\sigma \models \mathtt{ins}(\mathtt{L}, \swarrow, l) \rightsquigarrow \sigma[l := a[\mathtt{L} \cdot \mathtt{L}']^{\mathsf{rw}}]}$$

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma \models \mathtt{ins}(\mathtt{L}, \rightarrow, l) \rightsquigarrow \sigma[l' := a[\mathtt{L}_1 \cdot l \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]} \quad \frac{\sigma(l) = a[\mathtt{L}']^{\mathsf{rw}}}{\sigma \models \mathtt{ins}(\mathtt{L}, \searrow, l) \rightsquigarrow \sigma[l := a[\mathtt{L}' \cdot \mathtt{L}]^{\mathsf{rw}}]}$$

$$\frac{\sigma(l) = a[\mathtt{L}_1 \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma \models \mathtt{ins}(\mathtt{L}, \downarrow, l) \rightsquigarrow \sigma[l := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]} \quad \frac{\sigma(l) = a[\mathtt{L}]^{\mathsf{rw}}}{\sigma \models \mathtt{ren}(l, b) \rightsquigarrow \sigma[l := b[\mathtt{L}]^{\mathsf{rw}}]}$$

$$\frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma \models \mathtt{repl}(l, \mathtt{L}) \rightsquigarrow \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]} \quad \frac{\sigma(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma \models \mathtt{del}(l) \rightsquigarrow \sigma[l' := a[\mathtt{L}_1 \cdot \mathtt{L}_2]^{\mathsf{rw}}]}$$

Fig. 13. Semantics of atomic updates

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

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

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

$$\frac{\sigma,\gamma \models u \Rightarrow \sigma',\omega \quad \mathrm{sanitycheck}(\omega) \quad \mathsf{annot}(\mathsf{rw},\sigma',\gamma) \models \omega \rightsquigarrow \sigma''}{\sigma,\gamma \models u \rightsquigarrow \mathsf{annot}(\mathsf{ro},\sigma'',\gamma)}$$

Fig. 14. Update application

Definition 3. *A* type preservation map *(or* TP-map*)* $\boldsymbol{h}$ *is a collection of functions*

$$\begin{aligned}
h &: \mathit{Loc} \to \mathit{Loc}^*\\
h_r &: \mathit{Loc} \to \mathit{Loc}^*\\
h_c &: \mathit{Loc} \to \mathit{Loc}^*\\
h_{\mathtt{d}} &: \mathit{Loc} \to \mathit{Loc}^* \quad (\mathtt{d} \in \{\swarrow,\searrow,\leftarrow,\rightarrow\}\\
h_\downarrow &: \mathit{Loc} \to \mathcal{P}(\mathit{Loc})
\end{aligned}$$

A TP-map $\boldsymbol{h}$ *is said to* preserve types from σ,S to $\sigma',\tilde{\mathsf{S}}$ *provided that:*

1. *For all* $l \in \mathrm{dom}(\sigma)$, $h(l) = h_\leftarrow(l) \cdot h_r(l) \cdot h_\rightarrow(l)$.
2. *For all* $l \in \mathrm{dom}(\sigma)$*, we must have:*
   (a) $l \in \mathrm{dom}(\sigma')$ *and* $\sigma'(l) = b[h_c(l)]^m$ *for some* b, m*, and*
   (b) *Suppose* $\sigma(l) = a[l_1 \cdots l_n]^m$*. Then* $h_c(l)$ *is of the form:*

$$h_c(l) = h_\swarrow(l) \cdot \mathtt{L}'_0 \cdot h(l_1) \cdot \mathtt{L}'_1 \cdots \mathtt{L}'_{n-1} \cdot h(l_n) \cdot \mathtt{L}'_n \cdot h_\searrow(l)$$

   where each $\mathtt{L}'_i$ *consists of locations from* $h_\downarrow(l)$.
3. *If* $\sigma \models_{\mathsf{S}} l : \mathtt{T}$ *then*
   (a) $\sigma' \models_{\tilde{\mathsf{S}}} l' : \tilde{\mathtt{T}}_\downarrow$ *for all* $l' \in h_\downarrow(l)$*,*
   (b) $\sigma' \models_{\tilde{\mathsf{S}}} h_{\mathtt{d}}(l) : \tilde{\mathtt{T}}_{\mathtt{d}}$ *for* $\mathtt{d} \in \{\swarrow,\searrow,\leftarrow,\rightarrow\}$*,*
   (c) $\sigma' \models_{\tilde{\mathsf{S}}} h_c(l) : \tilde{\mathtt{T}}_c$*,*
   (d) $\sigma' \models_{\tilde{\mathsf{S}}} h_r(l) : \tilde{\mathtt{T}}_r$*, and*
   (e) $\sigma' \models_{\tilde{\mathsf{S}}} h(l) : \tilde{\mathtt{T}}$.

We say that initial types are preserved *from initial configuration* σ,S *to updated store* $\sigma',\tilde{\mathsf{S}}$ *provided that there exists a TP-map* $\boldsymbol{h}$ *that preserves types from* σ,S *to* $\sigma',\tilde{\mathsf{S}}$.

The overall goal of the proof is to show that if $\sigma \models \omega \rightsquigarrow \sigma'$ then initial types are preserved from σ, S to $\sigma', \tilde{\mathsf{S}}$. To this end, we will make frequent use of another monotonicity property:

Lemma 6. *If initial types are preserved from σ, S to $\sigma', \tilde{\mathsf{S}}$ and $\tilde{\mathsf{S}} \leq \tilde{\mathsf{S}}'$ then initial types are preserved from σ, S to $\sigma', \tilde{\mathsf{S}}'$.*

Proof. Straightforward. If we have a TP-map $\boldsymbol{h}$ from σ, S to $\sigma', \tilde{\mathsf{S}}$ then conditions 1 and 2 still hold for $\sigma', \tilde{\mathsf{S}}'$, and conditions 3(a)-(e) can be established using monotonicity.

Here is an outline of the proof that schema alteration is sound:

1. We start by assuming $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ and $\sigma \models_{\mathsf{S}} \omega : \Omega$.
2. Show that initial types are preserved from σ, S to $\sigma, \tilde{\mathsf{S}}_0$ (Lemma 7).
3. Define $\tilde{\mathsf{S}}_1$, the augmented schema after the first phase, and observe $\tilde{\mathsf{S}}_0 \leq \tilde{\mathsf{S}}_1$, so initial types are preserved from σ, S to $\sigma, \tilde{\mathsf{S}}_1$. (Lemma 8)
4. Suppose that $\sigma \models_1 \omega \rightsquigarrow \sigma_1$. Show that initial types are preserved from σ, S to $\sigma_1, \tilde{\mathsf{S}}_1$. (Lemma 9)
5. Define $\tilde{\mathsf{S}}_2$, the augmented schema after the second phase, and observe $\tilde{\mathsf{S}}_1 \leq \tilde{\mathsf{S}}_2$, so initial types are preserved from σ, S to $\sigma_1, \tilde{\mathsf{S}}_2$. (Lemma 10)
6. Suppose that $\sigma_1 \models_2 \omega \rightsquigarrow \sigma_2$. Show that initial types are preserved from σ, S to $\sigma_2, \tilde{\mathsf{S}}_2$. (Lemma 11)
7. Define $\tilde{\mathsf{S}}_3$, the augmented schema after the third phase, and observe $\tilde{\mathsf{S}}_2 \leq \tilde{\mathsf{S}}_3$, so initial types are preserved from σ, S to $\sigma_2, \tilde{\mathsf{S}}_3$. (Lemma 12)
8. Suppose that $\sigma_2 \models_3 \omega \rightsquigarrow \sigma_3$. Show that initial types are preserved from σ, S to $\sigma_3, \tilde{\mathsf{S}}_3$. (Lemma 13)
9. Define $\tilde{\mathsf{S}}_4$, the augmented schema after the fourth phase, and observe $\tilde{\mathsf{S}}_3 \leq \tilde{\mathsf{S}}_4$, so initial types are preserved from σ, S to $\sigma_3, \tilde{\mathsf{S}}_4$. (Lemma 14)
10. Suppose that $\sigma_3 \models_4 \omega \rightsquigarrow \sigma_4$. Show that initial types are preserved from σ, S to $\sigma_4, \tilde{\mathsf{S}}_4$. (Lemma 15)
11. Conclude by observing that the preservation of initial types from σ, S to $\sigma_4, \tilde{\mathsf{S}}_4$ implies that $\sigma_4 \models_{\tilde{\mathsf{S}}_4} \gamma : \tilde{\Gamma}$. (Lemma 16)

Lemma 7. *If $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ then initial types are preserved from σ, S to $\sigma, \tilde{\mathsf{S}}_0$.*

Proof. We define a TP-map $\boldsymbol{h}$ as follows. For each $l \in \mathrm{dom}(\sigma)$ with $\sigma(l) = a[\mathtt{L}]^m$, we define:

$$\begin{aligned} h(l) &= l \\ h_r(l) &= l \\ h_c(l) &= \mathtt{L} \\ h_{\downarrow} &= \emptyset \\ h_{\mathtt{d}} &= () \quad (d \in \{\swarrow, \searrow, \leftarrow, \rightarrow\}) \end{aligned}$$

To establish that $\boldsymbol{h}$ preserves types from $\mathsf{S}(\mathtt{X}) \leq \tilde{\mathsf{S}}_0(\tilde{\mathtt{X}})$, parts (1) and (2) are immediate by calculation. For part (3), part (a) follows using Lemma 5, and the other parts are immediate.

Stage 1 We define $\tilde{\mathsf{S}}_1$, the augmented schema after phase 1, as follows:

$$
\begin{aligned}
I_{\downarrow}(\mathtt{T}) &= \{\mathtt{U} \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{ins}(\tau, \downarrow, \mathtt{T}') \in |\Omega|, \mathtt{U} \in |\tau|\}\} \\
N(\mathtt{T}) &= \{b \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \mathtt{ren}(\mathtt{T}', b) \in |\Omega|\} \\
\tilde{\mathsf{S}}_1(\tilde{\mathtt{T}}_{\downarrow}) &= (\bigvee I_{\downarrow}(\mathtt{T}))^* \\
\tilde{\mathsf{S}}_1(\tilde{\mathtt{T}}_r) &= \tilde{\mathsf{S}}_0(\tilde{\mathtt{T}}_r) \mid (\bigvee \{b[\tilde{\mathtt{T}}_c] \mid b \in N(\mathtt{T})\}) \\
\tilde{\mathsf{S}}_1(\mathtt{X}) &= \tilde{\mathsf{S}}_0(\mathtt{X}) \quad \text{otherwise}
\end{aligned}
$$

Lemma 8. $\tilde{\mathsf{S}}_0 \leq \tilde{\mathsf{S}}_1$.

Proof. This is immediate for unmodified type names; for the modified names $\tilde{\mathtt{T}}_{\downarrow}$ and $\tilde{\mathtt{T}}_r$ it is straightforward to see that the new definitions subsume the old ones. Specifically, observe $\tilde{\mathsf{S}}_0(\tilde{\mathtt{T}}_{\downarrow}) = ()$, which is clearly contained in $(\bigvee I_{\downarrow}(\mathtt{T}))^*$, whereas $\tilde{\mathsf{S}}_0(\tilde{\mathtt{T}}_r)$ is self-evidently contained in $\tilde{\mathsf{S}}_0(\tilde{\mathtt{T}}_r) \mid (\bigvee \{b[\tilde{\mathtt{T}}_c] \mid b \in N(\mathtt{T})\})$.

Lemma 9. *Suppose that $\sigma \models_{\mathsf{S}} \omega : \Omega$ and $\sigma \sqsubseteq \sigma_0$, and initial types are preserved from σ, S to $\sigma_0, \tilde{\mathsf{S}}_1$. If $\sigma_0 \models_1 \omega' \rightsquigarrow \sigma_1$ where $|\omega'| \subseteq |\omega|$ then $\sigma \sqsubseteq \sigma_1$ and initial types are preserved from σ, S to $\sigma_1, \tilde{\mathsf{S}}_1$.*

Proof. Proof is by induction on the structure of the derivation of $\sigma_0 \models \omega' \rightsquigarrow \sigma_1$ (see Figure 13). The cases for the empty sequence and (commutative) sequential composition are straightforward, as is the case for the inactive update instructions for stages 2, 3 and 4. The only interesting cases are those for the insertions or renamings that take effect in this stage.

– If the derivation is of the form

$$
\frac{\sigma_0(l) = a[\mathtt{L}_1 \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\dfrac{\sigma_0 \models \mathtt{ins}(\mathtt{L}, \downarrow, l) \rightsquigarrow \sigma_0[l := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]}{\sigma_0 \models_1 \mathtt{ins}(\mathtt{L}, \downarrow, l) \rightsquigarrow \sigma_0[l := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]}}
$$

then let $\sigma_1 = \sigma_0[l := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]^{\mathsf{rw}}]$. First, to show that $\sigma \sqsubseteq \sigma_1$, observe that $\sigma \sqsubseteq \sigma_0$ by assumption. If l' is a read-only location in σ_1, then it is not l because l is read-write, so $\sigma_0(l') = \sigma_o[l := a[\mathtt{L}_1 \cdot \mathtt{L}_2]^{\mathsf{rw}}](l') = \sigma_1(l')$. Hence, $\sigma \sqsubseteq \sigma_0 \sqsubseteq \sigma_1$.
Next we must show that types are preserved from σ to σ_1. First, note that by assumption, $\mathtt{ins}(\mathtt{L}, \downarrow, l) \in |\omega|$ and so (by induction on the derivation of $\sigma \models_{\mathsf{S}} \omega : \Omega$) there must be some types $\tau, \mathtt{T}$ such that $\sigma \models_{\mathsf{S}} \mathtt{ins}(\mathtt{L}, \downarrow, l) : \mathtt{ins}(\tau, \downarrow, \mathtt{T})$ holds. Moreover, since there is only one rule that can derive this validity judgment (Figure 12), we must also have the antecedents of that rule: $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ and $\sigma \models_{\mathsf{S}} l : \mathtt{T}$. Sn addition, since $\sigma \sqsubseteq \sigma_1$, we also know that $\sigma_1 \models^{\mathsf{ro}}_{\tilde{\mathsf{S}}_1} \mathtt{L} : \tau$.
Now to prove types are preserved from σ to σ_1, by assumption there must exist a TP-map $\boldsymbol{h}$ from σ, S to $\sigma_0, \tilde{\mathsf{S}}_1$. We show how to adjust $\boldsymbol{h}$ to a TP-map $\boldsymbol{h}'$ so that the latter is a TP-map from σ, S to $\sigma_1, \tilde{\mathsf{S}}_1$. We define $\boldsymbol{h}'$ to agree with $\boldsymbol{h}$ away from l,

and modify $\boldsymbol{h}'$ at l as follows:

$$\begin{aligned}
h'(l) &= h(l) = l \\
h'_r(l) &= h_r(l) = l \\
h'_c(l) &= \mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2 \\
h'_\downarrow(l) &= h_\downarrow(l) \cup \{l_1, \ldots, l_n\} \quad \text{where } \mathtt{L} = l_1 \cdots l_n\} \\
h'_{\mathtt{d}}(l) &= h_{\mathtt{d}}(l) = () \quad (d \in \{\swarrow, \searrow, \leftarrow, \rightarrow\})
\end{aligned}$$

We must now show that $\boldsymbol{h}'$ is indeed a TP-map from σ, S to $\sigma_1, \tilde{\mathsf{S}}_1$. Part (1) is immediate since the relevant parts of $\boldsymbol{h}'$ are the same as for $\boldsymbol{h}$. For part (2a), suppose $\sigma(l) = a[l_1, \ldots, l_n]$. It is immediate that $\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2 = h'_\swarrow(l) \cdot h'_c(l) = h'_\searrow(l)$. For (2b), we must have

$$h_c(l) = \mathtt{L}_1 \cdot \mathtt{L}_2 = (\mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i) \cdot (\mathtt{L}''_i \cdots h(l_n) \cdot \mathtt{L}'_n)$$

where each $\mathtt{L}'_i$ mentions only labels in $h_\downarrow(l)$. Also, $h(l) = h'(l) = l$ so we must have

$$h'_c(l) = \mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2 = (\mathtt{L}'_1 \cdot h'(l_1) \cdots h'(l_i) \cdot \mathtt{L}'_i) \cdot \mathtt{L} \cdot (\mathtt{L}''_i \cdot h'(l_{i+1}) \cdot h'(l_n) \cdot \mathtt{L}'_n)$$

that is, $h'_c(l)$ is of the required form (since $h_\swarrow(l) = () = h_\searrow(l)$).
For part (3), the proof proceeds by induction from the leaf nodes in σ to the root. Suppose that $\sigma \models_{\mathsf{S}} l' : \mathtt{T}'$. By assumption, we know that parts (a-e) hold for σ_0. If $l' \neq l$, then it is straightforward to show that parts (a-e) still hold, appealing to part (2) and the induction hypothesis for part (c).
If $l = l'$, then part (a) is straightforward using the well-formedness of $\mathtt{L}$ and definition of $h'_\downarrow(l)$. Part (b) is immediate since the $h_{\mathtt{d}}(l) = ()$ for $\mathtt{d} \neq \downarrow$. For part (c), there is some work to do. By assumption, $h_c(l)$ is of the form

$$h_c(l) = \mathtt{L}_1 \cdot \mathtt{L}_2 = (\mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i) \cdot (\mathtt{L}''_i \cdots h(l_n) \cdot \mathtt{L}'_n)$$

where the sequences $\mathtt{L}'_i$ use labels from $h'_\downarrow(l)$. As shown above,

$$h'_c(l) = \mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2 = (\mathtt{L}'_1 \cdot h'(l_1) \cdots h'(l_i) \cdot \mathtt{L}'_i) \cdot \mathtt{L} \cdot (\mathtt{L}''_i \cdot h'(l_{i+1}) \cdot h'(l_n) \cdot \mathtt{L}'_n)$$

By part (a) we know that $\sigma_1 \models_{\tilde{\mathsf{S}}_1} l_0 : \tilde{\mathtt{T}}_\downarrow$ for each l_0 in $h'_\downarrow(l)$. Moreover, l matches both $\tilde{\mathtt{T}}$ and $\tilde{\mathtt{T}}'$, so $\tilde{\mathtt{T}}$ and $\tilde{\mathtt{T}}'$ may alias. This means that the type names from τ must be included as possibilities in $\tilde{\mathtt{T}}'_\downarrow$. Hence, the subsequence $\mathtt{L}'_i \cdot \mathtt{L} \cdot \mathtt{L}''_i$ also matches $\tilde{\mathtt{T}}'_\downarrow$, so we can establish that $\sigma_1 \models_{\tilde{\mathsf{S}}_1} \mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2 : \tilde{\mathtt{T}}_c$, as desired. Parts (d) and (e) are then immediate.

– If the derivation is of the form

$$\dfrac{\dfrac{\sigma_0(l) = a[\mathtt{L}]^{\mathsf{rw}}}{\sigma_0 \models \mathtt{ren}(l, b) \rightsquigarrow \sigma_0[l := b[\mathtt{L}]^{\mathsf{rw}}]}}{\sigma_0 \models_1 \mathtt{ren}(l, b) \rightsquigarrow \sigma_0[l := b[\mathtt{L}]^{\mathsf{rw}}]}$$

Following a similar line of reasoning to the above, we must have $\sigma \models_{\mathsf{S}} \mathtt{ren}(l, b) : \mathtt{ren}(\mathtt{T}, b)$ for some type $\mathtt{T}$. This case is relatively easy (compared to the case for

"insert into") because we do not have to adjust the TP-map or worry about the types of inserted nodes, only check that the TP-map $\boldsymbol{h}$ that works for σ_0 still works for σ_1. This is the case because for any alternate type $\mathtt{T}'$ for l in the initial store σ, clearly $\mathtt{T}'$ is an alias of $\mathtt{T}$ so the definition of $\tilde{\mathtt{T}}'_r$ in $\tilde{\mathsf{S}}_1$ must include a case of the form $b[\tilde{\mathtt{T}}'_c]$.

Stage 2 We define $\tilde{\mathsf{S}}_2$, the augmented schema after the second stage, as follows:

$$
\begin{aligned}
I_{\mathtt{d}}(\mathtt{T}) &= \{\tau \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{ins}(\tau, \mathtt{d}, \mathtt{T}') \in |\Omega|\} \\
\tilde{\mathsf{S}}_2(\tilde{\mathtt{T}}_{\mathtt{d}}) &= (\bigvee I_{\mathtt{d}}(\mathtt{T}))^* \\
\tilde{\mathsf{S}}_2(\tilde{\mathtt{X}}) &= \tilde{\mathsf{S}}_1(\tilde{\mathtt{X}}) \quad \text{otherwise}
\end{aligned}
$$

Lemma 10. $\tilde{\mathsf{S}}_1 \leq \tilde{\mathsf{S}}_2$

Proof. Straightforward since $\tilde{\mathsf{S}}_1(\tilde{\mathtt{T}}_{\mathtt{d}}) = ()$ which is contained in $(\bigvee I_{\mathtt{d}}(\mathtt{T}))^*$.

Lemma 11. *Suppose that $\sigma \models_{\mathsf{S}} \omega : \Omega$ and $\sigma \sqsubseteq \sigma_1$, and initial types are preserved from σ, S to $\sigma_1, \tilde{\mathsf{S}}_2$. If $\sigma_1 \models_2 \omega' \rightsquigarrow \sigma_2$ where $|\omega'| \subseteq |\omega|$ then $\sigma \sqsubseteq \sigma_2$ and initial types are preserved from σ, S to $\sigma_2, \tilde{\mathsf{S}}_2$.*

Proof. Generally similar to the proof for stage 1. The proof cases for "insert first into" ($\swarrow$) and "insert last into" ($\searrow$) are essentially the same as for "insert into", except that we add the inserted sequences to the appropriate $h'_{\swarrow}$ or $h'_{\searrow}$ as appropriate. Similarly, the cases for "before" and "after" are essentially the same except that we add the inserted nodes to $h'_{\leftarrow}$ or $h'_{\rightarrow}$ as appropriate. We show the case for $\leftarrow$ in detail.

– If the derivation is of the form

$$
\dfrac{\dfrac{\sigma_1(l') = a[\mathtt{L}_1 \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}}{\sigma_1 \models \mathtt{ins}(\mathtt{L}, \leftarrow, l) \rightsquigarrow \sigma_1[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}]}}{\sigma_1 \models_2 \mathtt{ins}(\mathtt{L}, \leftarrow, l) \rightsquigarrow \sigma_1[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2]^{\mathsf{rw}}]}
$$

then let $\sigma_2 = \sigma_1[l' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2]]$. First note that since $\sigma \sqsubseteq \sigma_1$ we also have $\sigma \sqsubseteq \sigma_2$ since σ_1 and σ_2 differ only at l', which is rw.
Next we must show that types are preserved from σ to σ_1. First, note that by assumption, $\mathtt{ins}(\mathtt{L}, \leftarrow, l) \in |\omega|$ and so (by induction on the derivation of $\sigma \models_{\mathsf{S}} \omega : \Omega$) there must be some types $\tau, \mathtt{T}$ such that $\sigma \models_{\mathsf{S}} \mathtt{ins}(\mathtt{L}, \leftarrow, l) : \mathtt{ins}(\tau, \leftarrow, \mathtt{T})$ holds. Moreover, since there is only one rule that can derive this validity judgment (Figure 12), we must also have the antecedents of that rule: $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ and $\sigma \models_{\mathsf{S}} l : \mathtt{T}$. In addition, by monotonicity we also have $\sigma_2 \models^{\mathsf{ro}}_{\tilde{\mathsf{S}}_2} \mathtt{L} : \tau$.
Now to prove types are preserved from σ to σ_2, by assumption there must exist a TP-map $\boldsymbol{h}$ from σ, S to $\sigma_1, \tilde{\mathsf{S}}_2$. We show how to adjust $\boldsymbol{h}$ to $\boldsymbol{h}'$ so that the latter is a TP-map from σ, S to $\sigma_2, \tilde{\mathsf{S}}_2$. We define $\boldsymbol{h}'$ to agree with $\boldsymbol{h}$ away from l and l', and define $\boldsymbol{h}'$ as follows for l and l':

$$
\begin{aligned}
h'(l') &= h(l) \\
h'_r(l') &= h_r(l') \\
h'_c(l') &= \mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2 \\
h'_{\mathtt{d}}(l') &= h_{\mathtt{d}}(l') \quad (d \in \{\swarrow, \searrow, \leftarrow, \rightarrow, \downarrow\})
\end{aligned}
\qquad
\begin{aligned}
h'(l) &= h_{\leftarrow}(l) \cdot \mathtt{L} \cdot h_c(l) \cdot h_{\rightarrow}(l) \\
h'_r(l) &= h_r(l) = l \\
h'_c(l) &= h_c(l) \\
h'_{\leftarrow}(l) &= h_{\leftarrow}(l) \cdot \mathtt{L} \\
h'_{\mathtt{d}}(l) &= h_{\mathtt{d}}(l) \quad (d \in \{\downarrow, \swarrow, \searrow, \rightarrow\})
\end{aligned}
$$

We must now show that $\boldsymbol{h}'$ is indeed a TP-map from σ, S to $\sigma', \tilde{\mathsf{S}}_1$.
For part (1), we just need to check that the desired property still holds for l and l' in $\boldsymbol{h}'$. For l' there is nothing to prove; for l the desired property is by definition of $\boldsymbol{h}'$.
For part (2a), the proof is straightforward (for both l and l') by definition of $\boldsymbol{h}'$. For (2b), there is nothing to prove for l. For l', suppose $\sigma_1(l) = l_1 \cdots l \cdots l_n$. Since l was present in $\sigma(l')$ originally, we can assume that:

$$h_c(l') = \mathtt{L}_1 \cdot l \cdot \mathtt{L}_2 = (h_{\swarrow}(l) \cdot \mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i) \cdot h(l) \cdot (\mathtt{L}'_{i+i} \cdots h(l_n) \cdot \mathtt{L}'_n \cdot h_{\searrow}(l))$$

where each $\mathtt{L}'_i$ mentions only labels in $h_{\downarrow}(l)$. Also, $h(l) = h_{\leftarrow}(l) \cdot l \cdot h_{\rightarrow}(l)$, since $h_r(l) = l$ (there haven't yet been any updates that can replace or delete l). So, we must have

$$\begin{aligned}\mathtt{L}_1 &= h_{\swarrow}(l) \cdot \mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i \cdot h_{\leftarrow}(l)\\ \mathtt{L}_2 &= h_{\rightarrow}(l) \cdot \mathtt{L}'_{i+i} \cdots h(l_n) \cdot \mathtt{L}'_n \cdot h_{\searrow}(l)\end{aligned}$$

we reason as follows:

$$\begin{aligned}h'_c(l') &= \mathtt{L}_1 \cdot \mathtt{L} \cdot l \cdot \mathtt{L}_2\\ &= h_{\swarrow}(l) \cdot \mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i \cdot h_{\leftarrow}(l) \cdot \mathtt{L} \cdot l \cdot h_{\rightarrow}(l) \cdot \mathtt{L}'_{i+i} \cdots h(l_n) \cdot \mathtt{L}'_n\\ &= \mathtt{L}'_1 \cdot h(l_1) \cdots \mathtt{L}'_i \cdot h'(l) \cdot \mathtt{L}'_{i+i} \cdots h(l_n) \cdot \mathtt{L}'_n \cdot h_{\searrow}(l)\end{aligned}$$

as desired.
For part (3), the proof proceeds by induction from the leaf nodes in σ to the root, and from left to right siblings within a child sequence. Suppose that $\sigma \models_{\mathsf{S}} l'' : \mathtt{T}''$. By assumption, we know that parts (a-e) hold for σ_1. If l'' is neither l nor l', then it is straightforward to show that parts (a-e) still hold, appealing to the induction hypothesis for part (c).
If $l'' = l$, then part (a) is immediate. Part (b) is also immediate except for the case for $h_{\leftarrow}$. For $h_{\leftarrow}$, we need to show that $\sigma_2 \models_{\tilde{\mathsf{S}}_2} h'_{\leftarrow}(l) : \tilde{\mathtt{T}}''_{\leftarrow}$, and we already know that $\sigma_1 \models_{\tilde{\mathsf{S}}_2} h_{\leftarrow}(l) : \tilde{\mathtt{T}}''_{\leftarrow}$. By induction we also have $\sigma_2 \models_{\tilde{\mathsf{S}}_2} h_{\leftarrow}(l) : \tilde{\mathtt{T}}''_{\leftarrow}$. We know that $h'_{\leftarrow}(l) = h_{\leftarrow}(l) \cdot \mathtt{L}$ and $\sigma_2 \models_{\tilde{\mathsf{S}}_2} \mathtt{L} : \tau$. Moreover, since l matches both $\mathtt{T}$ and $\mathtt{T}''$ we know that these two types alias and so τ must have been added as a possibility to $\tilde{\mathtt{T}}''_{\leftarrow}$; so, we can also derive $\sigma_2 \models_{\tilde{\mathsf{S}}_2} h_{\leftarrow}(l) \cdot \mathtt{L} : \tilde{\mathtt{T}}''_{\leftarrow}$, as desired. Part (c) and (d) are immediate since nothing has changed and part (e) is straightforward using parts (c,d,e) and the definition of $\tilde{\mathtt{T}}''$.
If $l'' = l'$ then parts (a) and (b) are immediate since nothing has changed. Part (c) requires work since the content of l' has changed, but using part (2) and the induction hypothesis, we can show that $h_c(l')$ still has type $\tilde{\mathtt{T}}''_c$. Again, parts (d,e) are easy given part (c) since nothing relevant has changed.

Stage 3 We define $\tilde{\mathsf{S}}_3$, the augmented schema after the second stage, as follows:

$$\begin{aligned}R(\mathtt{T}) &= \{\tau \mid \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\ \exists \tau.\ \mathtt{repl}(\mathtt{T}', \tau) \in |\Omega|\}\\ \tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) &= \tilde{\mathsf{S}}_2(\tilde{\mathtt{T}}_r) \mid (\bigvee R(\mathtt{T}))^*\\ \tilde{\mathsf{S}}_3(\tilde{\mathtt{X}}) &= \tilde{\mathsf{S}}_2(\tilde{\mathtt{X}}) \quad \text{otherwise}\end{aligned}$$

Lemma 12. $\tilde{\mathsf{S}}_2 \leq \tilde{\mathsf{S}}_3$.

Proof. Straightforward since $\tilde{\mathsf{S}}_2(\tilde{\mathtt{T}}_r) \leq \tilde{\mathsf{S}}_2(\tilde{\mathtt{T}}_r) \mid (\bigvee R(\mathtt{T}))^*$.

Lemma 13. *Suppose that $\sigma \models_{\mathsf{S}} \omega : \Omega$ and $\sigma \sqsubseteq \sigma_2$, and initial types are preserved from σ, S to $\sigma_2, \tilde{\mathsf{S}}_3$. If $\sigma_2 \models_3 \omega' \rightsquigarrow \sigma_3$ where $|\omega'| \subseteq |\omega|$ then $\sigma \sqsubseteq \sigma_3$ and initial types are preserved from σ, S to $\sigma_3, \tilde{\mathsf{S}}_3$.*

Proof. The proof is generally similar to the proofs for the previous stages; the reasoning dealing with replacement operations is similar to that for insert before and after in the previous phase, except that we adjust h_r instead of $h_\leftarrow$ or $h_\rightarrow$.

Stage 4 We define $\tilde{\mathsf{S}}_4$, the augmented schema after the fourth stage, as follows:

$$\tilde{\mathsf{S}}_4(\tilde{\mathtt{T}}_r) = \begin{cases} \tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) \mid () \text{ if } \exists \mathtt{T}' \in \mathrm{alias}(\mathtt{T}).\mathtt{del}(\mathtt{T}') \in |\Omega| \\ \tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) \qquad \text{otherwise} \end{cases}$$
$$\tilde{\mathsf{S}}_4(\tilde{\mathtt{X}}) = \tilde{\mathsf{S}}_3(\tilde{\mathtt{X}}) \quad \text{otherwise}$$

Lemma 14. $\tilde{\mathsf{S}}_3 \leq \tilde{\mathsf{S}}_4$.

Proof. Straightforward since $\tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) \leq \tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r)$ and $\tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) \leq \tilde{\mathsf{S}}_3(\tilde{\mathtt{T}}_r) \mid ()$.

Lemma 15. *Suppose that $\sigma \models_{\mathsf{S}} \omega : \Omega$ and $\sigma \sqsubseteq \sigma_3$, and initial types are preserved from σ, S to $\sigma_3, \tilde{\mathsf{S}}_4$. If $\sigma_3 \models_4 \omega' \rightsquigarrow \sigma_4$ where $|\omega'| \subseteq |\omega|$ then $\sigma \sqsubseteq \sigma_4$ and initial types are preserved from σ, S to $\sigma_4, \tilde{\mathsf{S}}_4$.*

Proof. Essentially the same as the previous phase; a deletion is basically the same as replacing with $()$.

Postprocessing

Lemma 16. *If $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ and initial types are preserved from σ, S to $\sigma', \tilde{\mathsf{S}}$ then $\sigma' \models_{\tilde{\mathsf{S}}} \gamma : \tilde{\Gamma}$.*

Proof. Let x be a variable in Γ. Then $\sigma \models_{\mathsf{S}} \gamma(x) : \Gamma(x)$. Hence (by induction on location sequences, and using the type preservation assumption), we have $\sigma' \models_{\mathsf{S}} \gamma(x) : \tilde{\Gamma}(x)$ for each x. This completes the proof.

Theorem 4. *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 $\sigma' \models_{\mathsf{S}'} \gamma : \Gamma'$.*

Proof. Straightforward, chaining the above lemmas.

$$\frac{}{\sigma, \gamma \models x \Rightarrow \sigma, \gamma(x)} \qquad \frac{}{\sigma, \gamma \models () \Rightarrow \sigma, ()}$$

$$\frac{\sigma, \gamma \models q_1 \Rightarrow \sigma_2, \mathtt{L}_1 \quad \sigma_2, \gamma \models q_2 \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma \models q_1, q_2 \Rightarrow \sigma_3, \mathtt{L}_1 \cdot \mathtt{L}_2} \qquad \frac{\sigma, \gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma_2, \mathtt{L} \quad l \notin \mathrm{dom}(\sigma_2)}{\sigma, \gamma \models a[q] \Rightarrow \sigma_2[l := a[\mathtt{L}]], l}$$

$$\frac{\sigma, \gamma \models q \Rightarrow \sigma_2, l \cdot \mathtt{L} \quad \sigma_2, \gamma \models q_1 \Rightarrow \sigma_3, \mathtt{L}_1}{\sigma, \gamma \models \mathtt{if}\ q\ \mathtt{then}\ q_1\ \mathtt{else}\ q_2 \Rightarrow \sigma_3, \mathtt{L}_1} \qquad \frac{\sigma, \gamma \models q \Rightarrow \sigma_2, () \quad \sigma_2, \gamma \models q_2 \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma \models \mathtt{if}\ q\ \mathtt{then}\ q_1\ \mathtt{else}\ q_2 \Rightarrow \sigma_3, \mathtt{L}_2}$$

$$\frac{\sigma, \gamma \models q_1 \Rightarrow \sigma_2, \mathtt{L} \quad \sigma_2, \gamma[x := \mathtt{L}] \models q_2 \Rightarrow \sigma_3, \mathtt{L}'}{\sigma, \gamma \models \mathtt{let}\ x = q_1\ \mathtt{in}\ q_2 \Rightarrow \sigma_3, \mathtt{L}'}$$

$$\frac{\sigma, \gamma \models q_1 \Rightarrow \sigma_2, \mathtt{L} \quad \sigma_2, \gamma, x \in \mathtt{L} \models^\star q_2 \Rightarrow \sigma_3, \mathtt{L}'}{\sigma, \gamma \models \mathtt{for}\ x \in q_1\ \mathtt{return}\ q_2 \Rightarrow \sigma_3, \mathtt{L}'}$$

$$\frac{\sigma \models \gamma(x)/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \mathtt{L}}{\sigma, \gamma \models x/ax :: \phi \Rightarrow \sigma, \mathtt{L}} \qquad \frac{\sigma, \gamma \models q \Rightarrow \sigma_0, \mathtt{L}_0 \quad \sigma_0, \mathtt{L}_0 \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}}{\sigma, \gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L}}$$

$$\frac{}{\sigma, \gamma, x \in () \models^\star q \Rightarrow \sigma, ()} \qquad \frac{\sigma, \gamma[x := l] \models q \Rightarrow \sigma_2, \mathtt{L}_1 \quad \sigma, \gamma, x \in \mathtt{L} \models^\star q \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma, x \in l \cdot \mathtt{L} \models^\star q \Rightarrow \sigma_3, \mathtt{L}_1 \cdot \mathtt{L}_2}$$

$$\frac{\begin{array}{c}\sigma, \gamma \models \theta \Rightarrow \sigma_1, \delta \quad \mathsf{annot}(\mathsf{rw}, \sigma_1, \delta), \gamma \uplus \delta \models u \Rightarrow \sigma_2, \omega \\ \mathrm{sanitycheck}(\omega) \\ \sigma_2 \models \omega \rightsquigarrow \sigma_2' \quad \mathsf{annot}(\mathsf{ro}, \sigma_2', \delta), \gamma \uplus \delta \models q \Rightarrow \sigma_3, \mathtt{L}\end{array}}{\sigma, \gamma \models \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q \Rightarrow \sigma_4, \mathtt{L}}$$

$$\frac{}{\sigma, \gamma \models \bullet \Rightarrow \sigma, \emptyset} \qquad \frac{\sigma, \gamma \models \theta \Rightarrow \sigma', \delta \quad \sigma', \gamma \uplus \delta \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma'', \mathtt{L}}{\sigma, \gamma \models \theta, x := q \Rightarrow \sigma'', \delta[x := \mathtt{L}]}$$

Fig. 15. Query evaluation rules

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

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

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

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

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

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

Fig. 16. Update expression evaluation

$$\frac{\sigma(l) = \mathsf{text}[s] \quad l' \notin \mathrm{dom}(\sigma)}{\sigma, l \overset{\mathsf{copy}}{\mapsto} \sigma[l' := \mathsf{text}[s]^{\mathsf{ro}}], l'} \qquad \frac{\sigma(l) = a[\mathtt{L}] \quad \sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}' \quad l' \notin \mathrm{dom}(\sigma')}{\sigma, l \overset{\mathsf{copy}}{\mapsto} \sigma'[l' := a[\mathtt{L}']^{\mathsf{ro}}], l'}$$

$$\frac{}{\sigma, () \overset{\mathsf{copy}}{\mapsto} \sigma, ()} \qquad \frac{\sigma, \mathtt{L}_1 \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}_1' \quad \sigma', \mathtt{L}_2 \overset{\mathsf{copy}}{\mapsto} \sigma'', \mathtt{L}_2'}{\sigma, \mathtt{L}_1 \cdot \mathtt{L}_2 \overset{\mathsf{copy}}{\mapsto} \sigma'', \mathtt{L}_1' \cdot \mathtt{L}_2'}$$

Fig. 17. Copying rules

B.2 Type and effect soundness

The full inference rule systems for query result type and update effect analysis are shown in Figure 18 and Figure 19, respectively.

Before proving soundness of these type and effect analyses, we prove a soundness property for the copying judgment:

Lemma 17 (Copy Soundness). *If* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ *and* $\sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}'$ *then* $\sigma \sqsubseteq \sigma'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L}' : \tau$.

Proof. Straightforward by induction on the structure of the derivation of $\sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}'$, using inversion as appropriate on validity assumptions.

We leave out the details of the semantics and static analysis judgments for XPath steps, and just assume that they satisfy:

Proposition 1 (XPath soundness). *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* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} l : \mathtt{T}$ *then* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$.

We now prove soundness for the type and effect analysis. In this appendix we consider the full query and update language, in which queries and updates are mutually recursive. Similarly, the semantics and static analysis judgments for queries and update are also mutually recursive, so we must prove soundness simultaneously:

Theorem 5 (Type and effect soundness).

1. *If* $\mathsf{S}; \Gamma \vdash q : \tau; \mathsf{S}'$ *then for all* $\sigma, \gamma, \mathtt{L}, \sigma'$, *if* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models q \Rightarrow \sigma', \mathtt{L}$ *then* $\sigma \sqsubseteq \sigma'$ *and* $\mathsf{S} \leq \mathsf{S}'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \mathtt{L} : \tau$.
2. *If* $\mathsf{S}; \Gamma; x \in \tau \vdash^{\star} q : \tau'; \mathsf{S}'$ *then for all* $\sigma, \gamma, \mathtt{L}, \mathtt{L}', \sigma'$, *if* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ *and* $\sigma, \gamma, x \in \mathtt{L} \models^{\star} q \Rightarrow \sigma', \mathtt{L}'$ *then* $\sigma \sqsubseteq \sigma'$ *and* $\mathsf{S} \leq \mathsf{S}'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \mathtt{L}' : \tau'$.
3. *If* $\mathsf{S}; \Gamma \vdash \theta : \Delta; \mathsf{S}'$ *then for all* $\sigma, \gamma, \delta, \sigma'$, *if* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma, \theta \models^{\star} \sigma' \Rightarrow \delta$, *then* $\sigma \sqsubseteq \sigma'$ *and* $\mathsf{S} \leq \mathsf{S}'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \delta : \Delta$.
4. *If* $\mathsf{S}; \Gamma \vdash u : \Omega; \mathsf{S}'$ *then for all* $\sigma, \gamma, \omega, \sigma'$, *if* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models u \Rightarrow \sigma', \omega$ *then* $\sigma \sqsubseteq \sigma'$ *and* $\mathsf{S} \leq \mathsf{S}'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \omega : \Omega$.
5. *If* $\mathsf{S}; \Gamma; x \in \tau \vdash^{\star} u : \Omega; \mathsf{S}'$ *then for all* $\sigma, \gamma, \mathtt{L}, \omega, \sigma'$, *if* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \mathtt{L} : \tau$ *and* $\sigma, \gamma, u \models^{\star} \sigma' \Rightarrow \omega$, *then* $\sigma \sqsubseteq \sigma'$ *and* $\mathsf{S} \leq \mathsf{S}'$ *and* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \omega : \Omega$.

Proof. By simultaneous inductions on the structure of derivations. The proof steps involving queries are generally similar to those for Core μXQ [11], with the main difference being the extra schema result parameter and the handling of $a[q]$ expressions.

For part (1), we show the cases for $a[q]$ and transform queries in detail.

$$\frac{}{\mathsf{S};\Gamma \vdash x : \Gamma(x);\mathsf{S}} \quad \frac{\mathsf{S};\Gamma \vdash q_1 : \tau_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma, x:\tau_1 \vdash q_2 : \tau_2;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{let}\ x := q_1\ \texttt{in}\ q_2 : \tau_2;\mathsf{S}_2} \quad \frac{}{\mathsf{S};\Gamma \vdash () : ();\mathsf{S}}$$

$$\frac{\mathsf{S};\Gamma \vdash \mathsf{q}_1 : \tau_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma \vdash \mathsf{q}_2 : \tau_2;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \mathsf{q}_1, \mathsf{q}_2 : \tau_1, \tau_2;\mathsf{S}_2} \quad \frac{\mathsf{S};\Gamma \vdash q : \mathsf{S}';\tau \quad \mathtt{T} \notin \mathrm{dom}(\mathsf{S}')}{\mathsf{S};\Gamma \vdash a[q] : \mathtt{T};\mathsf{S}'[\mathtt{T} := a[\tau]]}$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/ax :: \phi \overset{\mathsf{step}}{\Rightarrow} \tau}{\mathsf{S};\Gamma \vdash x/ax :: \phi : \tau;\mathsf{S}} \quad \frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_0 \quad \mathsf{S}_0;\Gamma \vdash q_1 : \tau_1;\mathsf{S}_1 \quad \mathsf{S};\Gamma \vdash q_2 : \tau_2;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{if}\ q\ \texttt{then}\ q_1\ \texttt{else}\ q_2 : \tau_1|\tau_2;\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash q_1 : \tau_1;\mathsf{S}_0 \quad \mathsf{S}_0;\Gamma;x \in \tau_1 \vdash^\star q : \tau_2;\mathsf{S}'}{\mathsf{S};\Gamma \vdash \texttt{for}\ x \in e_1\ \texttt{return}\ q_2 : \tau_2;\mathsf{S}'}$$

$$\frac{\mathsf{S};\Gamma, x{:}T \vdash q : \tau;\mathsf{S}'}{\mathsf{S};\Gamma;x \in T \vdash^\star q : \tau;\mathsf{S}'} \quad \frac{}{\mathsf{S};\Gamma;x \in () \vdash^\star q : ();\mathsf{S}} \quad \frac{\mathsf{S};\Gamma;x \in \tau \vdash^\star q : \tau';\mathsf{S}'}{\mathsf{S};\Gamma;x \in \tau^* \vdash^\star q : (\tau')^*;\mathsf{S}'}$$

$$\frac{\mathsf{S};\Gamma;x \in \tau_1 \vdash^\star q : \tau_1';\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma;x \in \tau_2 \vdash^\star q : \tau_2';\mathsf{S}_2}{\mathsf{S};\Gamma;x \in \tau_1, \tau_2 \vdash^\star q : \tau_1', \tau_2';\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma;x \in \tau_1 \vdash^\star q : \tau_1';\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma;x \in \tau_2 \vdash^\star q : \tau_2';\mathsf{S}_2}{\mathsf{S};\Gamma;x \in \tau_1|\tau_2 \vdash^\star q : \tau_1'|\tau_2';\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash \theta : \Delta;\mathsf{S}_1 \quad \mathsf{S}_1;\Delta \vdash u : \Omega;\mathsf{S}_2 \quad \mathsf{S}_2, \Delta \vdash \Omega \rightsquigarrow \mathsf{S}_2', \Delta' \quad \mathsf{S}_2';\Gamma, \Delta' \vdash q : \tau;\mathsf{S}_3}{\mathsf{S};\Gamma \vdash \texttt{copy}\ \theta\ \texttt{modify}\ u\ \texttt{return}\ q : \tau;\mathsf{S}_3}$$

$$\frac{}{\mathsf{S};\Gamma \vdash \bullet : \bullet;\mathsf{S}} \quad \frac{\mathsf{S};\Gamma \vdash \theta : \Delta;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma, \Delta \vdash q : \tau;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \theta, x := q : \Delta, x : \tau;\mathsf{S}_2}$$

Fig. 18. Query result-type analysis rules

$$\frac{}{\mathsf{S};\Gamma \vdash () : \emptyset;\mathsf{S}} \quad \frac{\mathsf{S};\Gamma \vdash u_1 : \Omega_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma \vdash u_2 : \Omega_2;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash u_1, u_2 : \Omega_1;\Omega_2;\mathsf{S}_2} \quad \frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma, x : \tau \vdash u : \Omega;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{let}\ x := e\ \texttt{in}\ u : \Omega;\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_0 \quad \mathsf{S}_0;\Gamma \vdash u_1 : \Omega_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma \vdash u_2 : \Omega_2;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{if}\ q\ \texttt{then}\ u_1\ \texttt{else}\ u_2 : \Omega_1|\Omega_2;\mathsf{S}_2} \quad \frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma;x \in \tau \vdash^\star s : \Omega;\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{for}\ x \in e\ \texttt{return}\ s : \Omega;\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma \vdash q' : \mathtt{T};\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \texttt{insert}\ q\ \mathtt{d}\ q' : \texttt{ins}(\tau, \mathtt{d}, \mathtt{T});\mathsf{S}_2} \quad \frac{\mathsf{S};\Gamma \vdash q : \mathtt{T};\mathsf{S}' \quad \mathsf{S}'(\mathtt{T}) = b[\tau]}{\mathsf{S};\Gamma \vdash \texttt{rename}\ q\ \texttt{as}\ a : \texttt{ren}(\mathtt{T}, a);\mathsf{S}'}$$

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

$$\frac{\mathsf{S};\Gamma, x : \mathtt{T} \vdash u : \Omega;\mathsf{S}'}{\mathsf{S};\Gamma;x \in \mathtt{T} \vdash^\star u : \Omega;\mathsf{S}'} \quad \frac{}{\mathsf{S};\Gamma;x \in () \vdash^\star u : \epsilon;\mathsf{S}} \quad \frac{\mathsf{S};\Gamma;x \in \tau \vdash^\star u : \Omega;\mathsf{S}'}{\mathsf{S};\Gamma;x \in \tau^* \vdash^\star u : \Omega^*;\mathsf{S}'}$$

$$\frac{\mathsf{S};\Gamma;x \in \tau_1 \vdash^\star u : \Omega_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma;x \in \tau_2 \vdash^\star u : \Omega_2;\mathsf{S}_2}{\mathsf{S};\Gamma;x \in \tau_1, \tau_2 \vdash^\star u : \Omega_1, \Omega_2;\mathsf{S}_2}$$

$$\frac{\mathsf{S};\Gamma;x \in \tau_1 \vdash^\star u : \Omega_1;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma;x \in \tau_2 \vdash^\star u : \Omega_2;\mathsf{S}_2}{\mathsf{S};\Gamma;x \in \tau_1|\tau_2 \vdash^\star u : \Omega_1|\Omega_2;\mathsf{S}_2}$$

Fig. 19. Update effect-inference rules

– In this case the typing and operational derivations are of the forms:

$$\frac{\mathsf{S};\Gamma \vdash q : \mathsf{S}';\tau \quad \mathtt{T} \notin \mathrm{dom}(\mathsf{S}')}{\mathsf{S};\Gamma \vdash a[q] : \mathtt{T};\mathsf{S}'[\mathtt{T} := a[\tau]]} \qquad \frac{\sigma,\gamma \models q \Rightarrow \sigma',\mathtt{L} \quad l \notin \mathrm{dom}(\sigma')}{\sigma,\gamma \models a[q] \Rightarrow \sigma'[l := a[\mathtt{L}]], l}$$

Moreover, $\sigma,\gamma \models q \Rightarrow \sigma',\mathtt{L}$ implies that $\sigma,\gamma \models q \Rightarrow \sigma_0,\mathtt{L}_0$ and $\sigma_0,\mathtt{L}_0 \overset{\mathsf{copy}}{\mapsto} \sigma',\mathtt{L}$. So by induction we have that $\sigma \sqsubseteq \sigma_0$ and $\mathsf{S} \leq \mathsf{S}'$. By monotonicity, $\sigma_0 \models^{\mathsf{ro}}_{\mathsf{S}'} \mathtt{L}_0 : \tau$. In addition, using Lemma 17 we have that $\sigma \sqsubseteq \sigma'$ and $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \mathtt{L} : \tau$. To conclude we need to show that $\sigma \sqsubseteq \sigma'[l := a[\mathtt{L}]]$ (immediate since l is fresh) and $\sigma'[l := a[\mathtt{L}]] \models^{\mathsf{ro}}_{\mathsf{S}'[\mathtt{T}:=a[\tau]]} l : \mathtt{T}$, but this is immediate using the rules in Figure 1.

– For a transform query, suppose the typing derivation is of the form:

$$\frac{\begin{array}{c}\sigma,\gamma \models \theta \Rightarrow \sigma_1,\delta \quad \mathsf{annot}(\mathsf{rw},\sigma_1,\delta),\gamma \uplus \delta \models u \Rightarrow \sigma_2,\omega \\ \mathrm{sanitycheck}(\omega) \\ \sigma_2 \models \omega \rightsquigarrow \sigma_2' \quad \mathsf{annot}(\mathsf{ro},\sigma_2',\delta),\gamma \uplus \delta \models q \Rightarrow \sigma_3,\mathtt{L}\end{array}}{\sigma,\gamma \models \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q \Rightarrow \sigma_4,\mathtt{L}}$$

and the operational derivation is of the form:

$$\frac{\mathsf{S};\Gamma \vdash \theta : \Delta;\mathsf{S}_1 \quad \mathsf{S}_1;\Delta \vdash u : \Omega;\mathsf{S}_2 \quad \mathsf{S}_2,\Delta \vdash \Omega \rightsquigarrow \mathsf{S}_2',\Delta' \quad \mathsf{S}_2';\Gamma,\Delta' \vdash q : \tau;\mathsf{S}_3}{\mathsf{S};\Gamma \vdash \mathtt{copy}\ \theta\ \mathtt{modify}\ u\ \mathtt{return}\ q : \tau;\mathsf{S}_3}$$

By induction (part (3)), we have that $\sigma_1 \models^{\mathsf{ro}}_{\mathsf{S}_1} \delta : \Delta$ and $\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}_2} \omega : \Omega$ where $\sigma \sqsubseteq \sigma_1 \sqsubseteq \sigma_2$ and $\mathsf{S} \leq \mathsf{S}_1 \leq \mathsf{S}_2$. Moreover, by Theorem 3, we have that $\sigma_2' \models^{\mathsf{ro}}_{\mathsf{S}_2'} \delta : \Delta'$ where $\sigma_2 \sqsubseteq \sigma_2'$ and $\mathsf{S}_2 \leq \mathsf{S}_2'$. Finally, by induction we have that $\sigma_3 \models^{\mathsf{ro}}_{\mathsf{S}_3} \mathtt{L} : \tau$ where $\sigma_2' \sqsubseteq \sigma_3$ and $\mathsf{S}_2' \leq \mathsf{S}_3$, which completes the proof.

Parts (2) and (3) are straightforward.
For part (4), we show illustrative cases for basic updating expressions:

– If the derivations are of the form

$$\frac{\mathsf{S};\Gamma \vdash q : \tau;\mathsf{S}_1 \quad \mathsf{S}_1;\Gamma \vdash q_0 : \mathtt{T};\mathsf{S}_2}{\mathsf{S};\Gamma \vdash \mathtt{insert}\ q\ \mathtt{d}\ q_0 : \mathtt{ins}(\tau,\mathtt{d},\mathtt{T});\mathsf{S}_2} \qquad \frac{\sigma_1,\gamma \models q \overset{\mathsf{copy}}{\Rightarrow} \sigma_2,\mathtt{L}_1 \quad \sigma_2,\gamma \models q_0 \Rightarrow \sigma_3,l_2}{\sigma_1,\gamma \models \mathtt{insert}\ q\ \mathtt{d}\ q_0 \Rightarrow \sigma_3,\mathtt{ins}(\mathtt{L}_1,\mathtt{d},l_2)}$$

Then by part (1), we know that $\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}_1} \mathtt{L}_1 : \tau$ and $\sigma_3 \models^{\mathsf{ro}}_{\mathsf{S}_2} l_2 : \mathtt{T}$, where $\sigma_1 \sqsubseteq \sigma_2 \sqsubseteq \sigma_3$ and $\mathsf{S} \leq \mathsf{S}_1 \leq \mathsf{S}_2$. So we can derive

$$\frac{\dfrac{\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}_1} \mathtt{L}_1 : \tau}{\sigma_3 \models^{\mathsf{ro}}_{\mathsf{S}_2} \mathtt{L}_1 : \tau} \quad \sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}_2} l_2 : \mathtt{T}}{\sigma_3 \models^{\mathsf{ro}}_{\mathsf{S}_3} \mathtt{ins}(\mathtt{L}_1,\mathtt{d},l_2) : \mathtt{ins}(\tau,\mathtt{d},\mathtt{T})}$$

– If the derivations are of the form:

$$\frac{\sigma_1,\gamma \models q \Rightarrow \sigma_2,l}{\sigma_1,\gamma \models \mathtt{delete}\ q \Rightarrow \sigma_2,\mathtt{del}(l)} \qquad \frac{\mathsf{S};\Gamma \vdash q : \mathtt{T};\mathsf{S}'}{\mathsf{S};\Gamma \vdash \mathtt{delete}\ q : \mathtt{del}(\mathtt{T});\mathsf{S}'}$$

then by part (1), we have $\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}'} l : \mathtt{T}$ where $\sigma_1 \sqsubseteq \sigma_2$ and $\mathsf{S} \leq \mathsf{S}'$. To conclude we can derive:

$$\frac{\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}'} l : \mathtt{T}}{\sigma_2 \models^{\mathsf{ro}}_{\mathsf{S}'} \mathtt{del}(l) : \mathtt{del}(\mathtt{T})}$$

Finally, part (5) is straightforward.

Corollary 2. *Suppose* $\mathsf{S}, \Gamma \vdash u \rightsquigarrow \mathsf{S}', \Gamma'$ *and* $\sigma \models^{\mathsf{ro}}_{\mathsf{S}} \gamma : \Gamma$ *and* $\sigma, \gamma \models u \rightsquigarrow \sigma'$. *Then* $\sigma' \models^{\mathsf{ro}}_{\mathsf{S}'} \gamma : \Gamma'$.

Proof. Straightforward, combining Theorem 5 and Theorem 4.