XQuery!: An XML query language with side effects

Giorgio Ghelli[1], Christopher Ré[2], and Jérôme Siméon[3]

[1] Università di Pisa
[2] University of Washington
[3] IBM T.J. Watson Research Center

Abstract. As XML applications become more complex, there is a growing interest in extending XQuery with side-effect operations, notably XML updates. However, the presence of side-effects is at odds with XQuery's declarative semantics which leaves evaluation order unspecified. In this paper, we define "XQuery!", an extension of XQuery 1.0 that supports first-class XML updates and user-level control over update application, preserving the benefits of XQuery's declarative semantics when possible. Our extensions can be easily implemented within an existing XQuery processor and we show how to recover basic database optimizations for such a language.

1 Introduction

As XML applications grow in complexity, developers are calling for advanced features in XML query languages. Many of the most requested extensions, such as XML updates, support for references, and variable assignment, involve side-effects. So far, proposed update extensions for XQuery [16, 22, 24, 1] have been based on restricted composability and a "snapshot semantics", where updates are only applied at the end of query execution. This approach preserves as much of XQuery's declarative semantics as possible, but the query cannot use the result of an update for further processing, limiting expressiveness in a way which is not always acceptable for applications.

In this paper, we develop the semantic foundations for extending XQuery 1.0 with side-effect operations in a fully compositional way. We use that framework to define XQuery! (read: "XQuery Bang"), an extension of XQuery 1.0 [3] that supports compositional XML updates and user-level control over update application. We show such a language can be obtained with limited impact on XQuery's declarative semantics and classical optimization techniques. To the best of our knowledge, this is the first complete treatment and implementation of a compositional side-effect extension of XQuery. The semantic framework is characterized by the presence of an operator (`snap`) that allows users to identify declarative fragments within their side-effecting programs, and which enables the recovery of traditional database optimizations.

XQuery! supports the same basic update operations as previous proposals [16, 8, 7]. However, the ability to use updates in any context (e.g., in function calls) and to control update application make it particularly expressive. Compositionality is one of the main design principles in XQuery 1.0, resulting in a language simpler to explain to users and specify. Our experience with a more restricted update language [24] shows that applications often require the additional expressiveness. We illustrate how compositionality between queries and updates in XQuery! can be used to develop a simple Web service that includes logging of service calls.

The contributions of the paper are:

– We provide a formal description of a semantic framework for extending XML query languages with side-effect operations which can appear anywhere in the query.

– We describe a new construct (`snap`) that can be used to control update application. The semantics of that operator enables unlimited nesting. We show how this semantics enables the recovery of standard database optimizations in the presence of side-effects.
– We define XQuery!, an extension to XQuery 1.0 with first-class updates, and illustrate its use on a concrete Web service usecase.
– We describe a simple implementation of XQuery!. We show that such an implementation can easily be obtained from an existing XQuery engine.

The main novelty in our framework lies in the ability to control update application through the `snap` construct. The notion of delaying update application to the end of query evaluation (so called *snapshot* semantics) was first proposed in [22, 16], and has been studied further in [8, 7, 1]. Previous proposals apply that approach to the whole query, while XQuery! provides programmer control of the snapshot scope through the `snap` operator. Languages with explicit control of the snapshot semantics are mentioned explicitly in the XQuery update requirements document [5], and have been explored by the W3C XML update task force [10, 4]. Work on the XL programming language [11] indicates support for fully compositional updates, but not for control of update application. To the best of our knowledge, our work is the first to propose a complete treatment of such an operator, and to explicit its relationship with optimization properties of the language.

Due to space limitations, we restrict the presentation to the main aspects of the language and its semantics. We first introduce XQuery! through a Web service usecase, before giving a formal definition of the language semantics. We then give an overview of our implementation, and discuss optimization issues. More details of the language, its complete formal semantics and more details about the implementation can be found in the complete paper [12].

2 XQuery! Use Case: Adding Logging to an XQuery Web Service

2.1 Snapshot semantics

Before we illustrate the use of XQuery!, we introduce the notion of snapshot semantics. All the update extensions to XQuery we are aware of [22, 16, 8, 7, 1] delay update applications up to the end of query execution, in order to retain the declarative semantics of XQuery. For instance, consider the following query which inserts a new `buyer` element into a list of purchasers for every person selling an item.

```
for $p in $auction//person
for $t in $auction//closed_auction
where $t/buyer/@person = $p/@id
return insert { <buyer person="{$t/buyer/@person}"
                       itemid="{$t/itemref/@item}" /> }
       into { $purchasers }
```

This is a typical join query, and the snapshot semantics ensures that traditional optimization techniques, such as algebraic rewritings and lazy evaluation, can be applied. In XQuery!, where the snapshot semantics if controlled explicitely, the absence of any internal `snap` allows similar optimizations. We come back to this point in more details in Section 4.

In addition, in order to facilitate rewritings, previous proposals limit the use of updates to specific sub-expressions, typically in the return clause of a FLWOR, as in our example. In the rest of the section, we show how a fairly simple Web service application already requires expressiveness that goes beyond that provided by restricted languages.

2.2 The Web Service scenario: updates inside functions

We assume a simple Web service application in which service calls are implemented as XQuery functions organized in a module. Because of space limitations, we focus on the single function `get_item`, which, given an itemid and the userid of the requester, returns the item with the given itemid; the userid is ignored. The server stores the auction document from XMark [23] in a variable `$auction`. The following is a possible implementation for that function using standard XQuery.

```
declare function get_item($itemid,$userid) {
  let $item := $auction//item[@id = $itemid]
  return $item
};
```

Now, let's assume that the Web service wants to log each item access. This can be easily done in XQuery! by adding an insert operation in the body of the function.

```
declare function get_item($itemid,$userid) {
  let $item := $auction//item[@id = $itemid]
  return (
    (::: Logging code :::)
    let $name := $auction//person[@id = $userid]/name return
    insert { <logentry user="{$name}" itemid="{$itemid}"/> }
    into { $log },
    (::: End logging code :::)
    $item
  )
};
```

This simple example illustrates the need for expressions that have a side-effect (the log entry insertion) and also return a value (the item itself). This is a central feature of our approach, which sets it apart from all the previously proposed XML update languages we are aware of [8, 7, 22, 16, 25, 24, 15].

Note that in the above example we use XQuery's sequence construction `(,)` to compose the conditional insert operation with the result `$item`. This is a convenience made possible by the fact that atomic update operations always return the empty sequence.

2.3 Controlling update application

The other central feature of our approach is the ability to control the "snapshot scope". A single global scope is often too restrictive, since many applications, at some stage of their computation, need to analyse their own previous effects. For this reason, XQuery! supports a `snap { `*Expr*` }` operator which evaluates *Expr*, collects its update requests, and applies them at the end of its scope. A `snap` is always implictly present around the top-level query in the main XQuery! module, so that the usual "delay everything" semantics is obtained by default. However, when needed, the code can decide to see its own effects. For example, consider the following simple variant for the logging code, where the log is summarized into an archive once every *$maxlog* insertions. (`snap insert{} into{}` abbreviates `snap{insert{} into{}}`, and similarly for the other update primitives).

```
(::: Logging code :::)
let $name := $auction//person[@id = $userid]/name
```

```
return
 (snap insert { <logentry user="{$name}"
                         itemid="{$item/@id}"/> }
       into { $log },
  if (count($log/logentry) >= $maxlog)
  then (archivelog($log,$archive),
        snap delete $log/logentry )
  else ()
(::: End logging code :::)
```

Here, the `snap` around `insert` makes the insertion happen. The insertion is then visible to the code inside the if-then-else, as required, because XQuery! semantics specifies that the sequence constructor `e1,e2` causes `e1` to be fully evaluated before `e2`. Hence, XQuery! expressive power relies on the combination of the `snap` operator and explicitly defined evaluation order. This is an important departure from XQuery 1.0 semantics, and requires some further discussion.

2.4 Sequence order, evaluation order, and update order

In XQuery 1.0, queries return sequences of items. Although sequences of items are ordered, the evaluation order for most operators is left to the implementation. For instance, in the expression (e_1, e_2), if e_1 and e_2 evaluate respectively to v_1 and v_2, then the value of e_1, e_2 must be v_1, v_2, in this order. However, the engine can evaluate e_2 before e_1, provided the result is presented in the correct sequence order. The only visible effect of this freedom is the fact that, if both expressions e_1 and e_2 were to raise an error, which error is reported may vary from implementation to implementation.

Although that approach is reasonable in an almost-purely functional language as XQuery 1.0, it is widely believed that programs with side-effects are impossible to reason about unless the evaluation order is easy to grasp.[4] For this reason, in XQuery! we adopt the standard semantics used in popular functional languages with side-effects [18, 17], based on the definition of a precise evaluation order. This semantics is easy to understand for a programmer and easy to formalize using the XQuery 1.0 formal semantic style, but is quite constraining for the compiler.[5] However, as we discuss in Section 3, inside an innermost `snap` no side-effect takes place, hence we there recover XQuery 1.0 freedom of evaluation order. XQuery 1.0 allows the processor to evaluate subexpressions in any order, provided that the item sequence is presented in the order which is specified by the formal semantics. Similarly, inside an innermost `snap`, both the pure subexpressions and the update operations can be evaluated in any order, provided that, at the end of the `snap` scope, both the item sequence and the list of update requests are presented in the correct order.

The order of update requests is a bit harder to maintain than sequence order, since a FLWOR expression may generate updates in the *for*, *where*, and *return* clause, while result items are only generated in the *return* clause. For this reason, XQuery! supports alternative semantics for update application, discussed in Section 3.2, which do not depend on order.

In many situations, different scopes for the `snap` would lead to the same result. In such cases, the programmer can adopt a simple criterion: make `snap` scope as broad as possible,

[4] Simon Peyton-Jones: "lazy evaluation and side effects are, from a practical point of view, incompatible" [14].

[5] An interesting alternative is to add a sequencing operator (e.g., `e1;e2`) that forces `e1` to be evaluated before `e2`, while retaining the XQuery 1.0 freedom of evaluation order for the other expressions. This alternative requires a more complex formalization style, and is explored in the full paper [12].

since a broader `snap` favors optimization. A `snap` should only be closed when the rest of the program relies on the effect of the updates.

2.5 Nested snap

Support for nested snap is central to our proposal, and is essential for compositionality. Assume, for example, that a counter is implemented using the following function.

```
declare variable $d := element counter { 0 };

declare function nextid() as xs:integer {
  snap { replace { $d/text() } with { $d + 1 },
         $d }
};
```

The snap around the function body is meant to ensure that any next call effectively returns the next value for the counter. Obviously, the `nextid()` function may be used in the scope of another snap. For instance, the following variant of the logging code computes a new id for every log entry.

```
  (::: Logging code :::)
  let $name := $auction//person[@id = $userid]/name
  return
   (snap insert { <logentry id="{nextid()}"
                            user="{$name}"
                            itemid="{$item/@id}"/> }
         into { $log },
    if (count($log/logentry) >= $maxlog) ...
  (::: End logging code :::)
```

The example shows that the `snap` operator must not freeze the state when its scope is opened, but just delay the updates that are in its immediate scope until the scope closes. Any nested snap opens a nested scope, and makes its updates visible as soon as it is closed. The details of this semantics are explained in Section 3.

3 XQuery! Semantics

The original semantics of XQuery is defined in [6] as follows. First, each expression is normalized to a *core* expression. Then, the meaning of core expressions is defined by a semantic judgement $dynEnv \vdash Expr \Rightarrow value$. This judgment states that, in the dynamic context *dynEnv*, the expression *Expr* yields the value *value*, where *value* is an instance of the XQuery data model (XDM).

To support side-effect operations, we extend the data model with a notion of store that maintains the state of the data model being processed. We then extend the semantic judgement so that expressions may change the store, and produce both a value and a list of pending update. In the rest of this section, we introduce the update primitives supported by the XQuery! language, followed by the data model extensions. We then shortly describe normalization, and finally define the new semantic judgment.

3.1 Update primitives

At the language level, XQuery! supports a set of updates primitives, insertion, deletion, replacement, and renaming of XML nodes, which are standard [16, 22, 24, 1, 8, 7]. The language also includes an explicit deep-copy operator, written `copy { ... }`. The full grammar for the XQuery! extension to XQuery 1.0 is given in Appendix A.

The detailed semantics of these primitives is also standard: insertion allows a sequence of nodes to be inserted below a parent in a specified position. Replacement allows a node to be replaced by another, and renaming allows the node name to be updated. Finally, to better deal with aliasing issues in the context of a compositional language, the semantics of the delete operation does not actually delete nodes, but merely *detaches* nodes from their parents. Similarly to what happens with object-oriented languages, if the "deleted" (actually, detached) node is still accessible from a variable, then it can still be queried, or inserted somewhere. The alternative "erase" semantics could be supported as well, but we believe it is slightly more complex to specify, implement, and program with.

GG: This sentence may go

3.2 XDM stores and update requests

Store. To represent the state of XQuery! computation, we need a notion of *store*, which specifies, for each node id, its kind (element, attribute, text...), parent, name, and content. A formal definition can be found in [12, 13, 9]. On this store, we define accessors and constructors corresponding to those of the XDM. Note that this presentation focuses on well-formed documents, and does not consider the impact of types on the data model representation and language semantics.

Update requests. We then define, for each XQuery! update primitive, the corresponding *update request*, which is a tuple that contains the operation name and its parameters, written as "opname(par_1,...,par_n)". For each update request, we define its *application* as a partial function from stores to stores. For example, the application of "insert (*nodeseq*,*nodepar*,*nodepos*)" inserts all nodes of *nodeseq* as children of *nodepar*, after *nodepos*. For each update request we also define some preconditions for its parameters. In the insert case, they include that fact that nodes in *nodeseq* must have no parent, and that *nodepos* must be a child of *nodepar*. When the preconditions are not met, the update application is undefined.

Update lists. An update list, noted as Δ, is a list of update requests. Update lists are collected during the execution of the code inside a given `snap`, and are applied hen the `snap` scope is closed. An update list is an *ordered* list, whose order is fully specified by the language semantics.

Applying an update list to the store. For optimization reasons, XQuery! supports three distinct semantics for update list application: *ordered*, *non-deterministic*, or *conflict-detection*. The programmer chooses the semantics through an optional keyword after each `snap`.

In the *ordered* semantics, the update requests are applied in the order specified by Δ. In the *non-deterministic* approach, the update requests are applied in an arbitrary order. In the *conflict-detection* approach, update application is divided into conflict verification followed by store modification. The first phase tries to prove, by some simple rules, that the update sequence is actually conflict-free, meaning that the ordered application of every permutation of Δ would produce the same result. If verification fails, update application fails. If verification succeeds, the store is modified, and the order of application is immaterial. Hence we get the benefit of determinism with no dependency on the order of updates inside Δ.

The *ordered* approach is simple and deterministic, but imposes more restrictions on the optimizer. The *non-deterministic* approach gives the optimizer more leverage, but non-determinism

makes code development harder, especially in the testing phase. Finally, the *conflict-detection* approach gives the optimizer the same re-ordering freedom as the non-deterministic approach and avoids non-determinism. However, it rules out many reasonable pieces of code, as exemplified in the full paper. Moreover, it can raise run-time failure which may be difficult to understand and to prevent.

Our implementation currently supports all the three semantics. We believe more experience with concrete applications is needed in order to assess the best choice.

3.3 Normalization

Normalization simplifies the semantics specification by first transforming each XQuery! expression into a *core* expression. As a result, the semantics only needs to be defined on the core language. The syntax of XQuery! core for update operations is almost identical to that of the surface language. The only non-trivial normalization effect is the insertion of a deep copy operator around the first argument of `insert`, as specified by the following normalization rule; the same happens to the second argument of `replace`. As with element construction in XQuery 1.0, this copy prevents the inserted tree from having two parents.

$$\frac{\texttt{[insert \{}Expr_1\texttt{\} into \{}Expr_2\texttt{\}]}}{\texttt{insert \{copy \{[}Expr_1\texttt{]\}\} as last into \{[}Expr_2\texttt{]\}}}$$

3.4 Formal semantics

Dynamic evaluation judgment. We extend the XQuery 1.0 semantic judgement "$dynEnv \vdash Expr \Rightarrow value$", in order to deal with delayed updates and side-effects, as follows:

$$store_0;\ dynEnv \vdash Expr \ \Rightarrow\ value;\ \Delta;\ store_1$$

Here, $store_0$ is the initial store, *dynEnv* is the dynamic context, *Expr* is the expression being evaluated, *value* and Δ are the value and the list of update requests returned by the expression, and $store_1$ is the new store after the expression has been evaluated. The updates in Δ have not been applied to $store_1$ yet, but *Expr* may have modified $store_1$ thanks to a nested `snap`, or by allocating new elements.

Observe that, while the store is modified, the update list Δ is just returned by the expression, exactly as the *value*. This property hints at the fact that an expression which just produces update requests, without applying them, is actually side-effects free, hence can be evaluated with the same approaches used to evaluate pure functional expressions. This is the main reason to use a snapshot semantics: inside the innermost `snap`, where updates are collected but not applied, lazy evaluation techniques can be applied.

Dynamic semantics of XQuery expressions. The presence of stores and Δ means that every judgment in XQuery 1.0 must be extended in order to properly deal with them. Specifically, every semantic judgment which contains at least two subexpressions has to be extended in order to specify which subexpression has to be evaluated first. Consider for example the rule for the sequence constructor.

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr_1 \ \Rightarrow\ value_1;\ \Delta_1;\ store_1 \\ store_1;\ dynEnv \vdash Expr_2 \ \Rightarrow\ value_2;\ \Delta_2;\ store_2 \end{array}}{store_0;\ dynEnv \vdash Expr_1, Expr_2 \ \Rightarrow\ value_1, value_2;\ (\Delta_1, \Delta_2);\ store_2}$$

As written, $Expr_1$ must be evaluated first in order for $store_1$ to be computed and passed for the evaluation of $Expr_2$.

In the case the sub-expressions do not contain any snap, the store remains the same, and evaluation order becomes irrelevant again, bringing back the expected declarative semantics. Of course, Δ_1 must precede Δ_2 in the result, when the *ordered* approach is followed, but this is not harder than preserving the order of ($value_1$, $value_2$); preserving update order is more complex in the case of FLWOR expressions and function calls (see [12]).

Dynamic semantics of XQuery! operations. We have to define the semantics of `copy`, of the update operators, and of `snap`. `copy` just invokes the corresponding operation at the data model level, adding the corresponding nodes to the store. The evaluation of an update operation produces an update request, which is added to the list of the pending update requests produced by the subexpressions, while `replace` produces *two* update requests, insertion and deletion. Here is the semantics of `replace`.

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr_1 \ \Rightarrow\ node;\ \Delta_1;\ store_1 \\ store_1;\ dynEnv \vdash Expr_2 \ \Rightarrow\ nodeseq;\ \Delta_2;\ store_2 \\ store_2;\ dynEnv \vdash \text{parent}(node) \ \Rightarrow\ nodepar;\ ();\ store_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{insert}(nodeseq, nodepar, node), \text{delete}(node)) \end{array}}{store_0;\ dynEnv \vdash \texttt{replace \{}Expr_1\texttt{\} with \{}Expr_2\texttt{\}} \ \Rightarrow\ ();\ \Delta_3;\ store_2}$$

The evaluation produces an empty sequence and an update list. It may also modify the store, but only if either $Expr_1$ or $Expr_2$ modify it. If they only perform allocations or copies, their evaluation can still be commuted or interleaved. If either executes a `snap`, the processor must follow the order specified by the rule, since, for example, $Expr_2$ may depend on the part of the store which has been modified by a `snap` in $Expr_1$. The two update requests produced by the operation are just inserted into the pending update list Δ_3 after every update requested by the two subexpressions. The actual order is only relevant if the *ordered* semantics has been requested for the smallest enclosing `snap`.

The rule for `snap` looks very simple: the `snap` argument is evaluated, it produces its own update list Δ, and Δ is applied to the store.

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr \ \Rightarrow\ value;\ \Delta;\ store_1 \\ store_2 = \text{apply } \Delta \text{ to } store_1 \end{array}}{store_0;\ dynEnv \vdash \texttt{snap \{}Expr\texttt{\}} \ \Rightarrow\ value;\ ();\ store_2}$$

The evaluation of *Expr* may itself modify the store, and this modified store is updated by the `snap`. For example, the following piece of code inserts `<b/><a/><c/>` into `$x`, in this order, since the internal `snap` is closed first, and it only applies the updates in its own scope.

```
snap ordered {  insert {<a/>} into $x,
                snap   { insert {<b/>} into $x },
                insert {<c/>} into $x       }
```

Hence, the formal semantics implicitly specifies a stack-like behavior, reflected by the actual stack-based implementation that we adopted (see [12]). However, the stack needs not be explicitly represented in the formal semantics; it is built into the recursive machinery of the deduction process exploited in the formal semantic definition.

In the appendix we list the semantic rules for the other update operations, and for the most important core XQuery 1.0 expressions.

4 Implementation and Optimization

XQuery! has been fully implemented as an extension to the Galax XQuery engine [21, 20], which includes an optimizer based on a variant of a standard nested-relational algebra. It is not yet fully tested, but has been tried on significantly complex update programs and a version of the compiler is available for download[6]. In this section, we review the modifications that were required to the original Galax compiler to support side-effects, notably changes to the optimizer.

4.1 Data model and run-time

Changes to the data model implementation to support atomic updates were not terribly invasive. The only two significant challenges relate to dealing with document order maintenance, and garbage collection of persistent but unreachable nodes, resulting from the detach semantics. Both of these aspects are beyond the scope of this paper.

The run-time must be modified to support update lists, which are computed in addition to the value for each expression. The way the update lists are represented internally depends on whether the `snap` operator uses the ordered semantics or not (See Section 3.2). Because the nondeterministic and conflict-detection semantics are both independent of the actual order of the atomic updates collected in a `snap` scope, they can be easily implemented using a stack of update lists, where each update list on the stack correspond to a given `snap` scope. The invocation of an update operation adds an update in the update list on the top of the stack. When exiting a `snap`, the top-most delta bag is popped from the stack and applied. In the case of conflict-detection semantics, it is also checked for conflicts, in linear time, using a pair of hash-tables over node ids.

This implementation strategy has the virtue that it does not require substantial modifications to the existing XQuery infrastructure. The implementation of the ordered semantics is more involved, as we need to rely on a specialized tree structure to represent the update list in a way which allows the compiler to retain the order in which each update must be applied. We refer to the full paper [12] for more details.

4.2 Compilation architecture

The implementation of XQuery! did not require any major changes to the XQuery processing model or compilation architecture. As for XQuery 1.0, the compilation proceeds by first *parsing* the query into an AST, followed by *normalization*, a phase of syntactic *rewriting*, *compilation* into the XML algebra [21], *optimization* and *evaluation*.

Changes to the parser and normalization are trivial (See Section 3). A number of the syntactic rewritings must be guarded by a judgment which detects whether side effects occur in a given subexpression to avoid changing the semantics for the query. Of course, this is not necessary when the query is guarded by an innermost `snap`, which is a `snap` whose scope contains no other `snap`, nor any call to any function which may cause a `snap` to be evaluated. In this case, all the rewritings immediately apply.

4.3 Changes to the optimizer

Galax uses a rule-based approach in several phases of the logical optimization. Most rewrite rules require some modifications. To illustrate the way the optimizer works, let us consider

[6] `http://xquerybang.cs.washington.edudb.cs.uwashington.edu/`

the following variant of XMark query 8 which, for each person, stores information about the buyers who purchased its items.

```
for $p in $auction//person
let $a :=
  for $t in $auction//closed_auction
  where $t/buyer/@person = $p/@id
  return (insert { <buyer person="{$t/buyer/@person}"
                          itemid="{$t/itemref/@item}" /> }
          into { $purchasers }, $t)
return <item person="{ $p/name }">{ count($a) }</item>
```

Ignoring the insert operation for a moment, the query is identical to XMark 8, and can be evaluated efficiently with an outer join followed by a group by. Such a query plan can be produced using query unnesting techniques such as those proposed in e.g., [21]. Naively evaluated, this query has complexity O($|person| * |closed\_auction|$). Using an outer join/group by with a typed hash join, we can recover the join complexity of O($|person| + |closed\_auction| + |matches|$), resulting in a substantial improvement.

In XQuery!, recall that the query is always wrapped into a top-level snap. Because that top-level snap does not contain any nested snap, the state of the database will not change during the evaluation of the query, and a outer-join/group-by plan can be used. The optimized plan generated by our XQuery! compiler is shown below. The syntax of the query plan is a simplified version of that defined in [21].

```
Snap {
  MapFromItem {
    <person name="{ Input#p/name }">{ count(Input#a) }</person>
   }
   (GroupBy [ Input#p, {
         (insert { <buyer person="{Input#t/buyer/@person}"
                          itemid="{Input#t/itemref/@item}" /> }
          as last into { $purchasers }, Input#t ) }]
    ( LeftOuterJoin( MapFromItem{[p:Input]}
                       ($auction//person ),
                     MapFromItem{[t:Input]}
                       ($auction//closed_auction))
       on { Input#t/buyer/@person = Input#p/@id }
    )
   )
}
```

In general, the optimization rules must be guarded by appropriate preconditions to ensure that not only the resulting value is correct, but also that the order (when applicable) and side-effects are preserved. Those preconditions check for properties related to cardinality and a form of query independence. The former ensures that expressions are evaluated with the correct cardinality, as changing the number of invocation may change the number of effects applied to the data model. The latter is used to check that a part of the query cannot observe the effects resulting from another part of the query, hence allowing certain rewritings to occur.

More specifically, consider the compilation of a join from nested for loops (maps): We must check that the inner branch of a join does not have updates. If the inner branch of the join does have update operations, they would be applied once for each element of the outer loop. Merge join and hash joins are efficient because they only evaluate their inputs once, however

doing so may change the cardinality for the side-effect portion of the query. Additionally, we must ensure that applying these new updates does not change the values returned in the outer branch, thus changing the value returned by the join. The first problem requires some analysis of the query plan, while the latter is difficult to ensure without the use of snap. In our example, if we had used a `snap insert` at line 5 of the source code, the group-by optimization would be more difficult to detect as one would have to know that the effect of the inserts are not observed in the rest of the query. This bears some similarity with the technique proposed in [1], although it is applied here on a much more expressive language.

5 Related work

Nested transactions. The `snap` operator is reminiscent of transactions, since both, in a sense, group update requests and apply them all at once; moreover, both can be nested. However, their purpose and semantics is essentially orthogonal.

Flat transactions are meant to protect a piece of code from concurrently running transactions, and nested transactions also allow the programmer to isolate different concurrent threads in its own code. XQuery has no internal concurrency, and we assume here that external concurrency is dealt with by the system that runs XQuery!, typically by running the top-level expression in its own (flat) transaction. Nested transactions are also used to control the extent of failure propagation. We propose a similar use for the `snap` operator, with no pretence of originality, in [12]. However, `snap` as presented here has nothing to do with either concurrency or failures.

On the other side, without concurrency and failures, transactions have no effect. In particular, after a transaction updates x, its next query to x will return the new value. On the contrary, an update to x requested inside a `snap` scope will not affect the result of queries to x inside the same scope. Hence, transactions isolate against external actions, while `snap` delays internal actions.

Monads in pure functional languages. Our approach allows the programmer to write essentially imperative code containing code fragments which are purely functional, and hence can be optimized more easily. The motivation is similar to that of monadic approaches in languages such as Haskell [14]. There, the type system distinguishes pieces of pure code, which can be lazily evaluated, from inpure "monadic" pieces of code, whose evaluation order is constrained. The type system will not allow pure code to call monadic code, while monadic code may invoke pure code at will.

The semantics of XQuery! bang requires to go beyond the pure-inpure distinction, notably requiring to distinguish the case where the query has some pending update but no effect. Those pieces of code in XQuery! do not block every optimizations, provided that some "independence" constraints are verified. It seems that these constraints are too complex to be represented through types. Hence, we let the optimizer collect the relevant information, and in particular flag the scope of each innermost `snap` as *pure*. To be fair, we believe that a bit of typing would be useful: the signature of functions coming from other modules should contain an *updating* flag, with the "monadic" rule that a function that calls an updating function is *updating* as well.

We are currently investigating the systematic translation of XQuery! to a core monadic language, which should give us a more complete understanding of the relationship between the two approaches.

Finally, the optimization opportunities enabled by the snapshot semantics are explored in [1]. An important difference is that we consider similar optimization in the context of a fully compositional language.

6 Conclusion

We presented here an extension of XQuery 1.0 which supports programmer-controlled delay of update application, in order to combine the expressive power of side-effects with the optimizability of side-effect free code fragments. The esential feature of this proposal is the free nesting of the `snap` operator, and we described the semantics and implementation of this operator. The proposal leaves many issues open for further investigation, such as static typing, optimization, and transactional mechanisms.

Acknowledgments. We want to thank the members of the W3C XML Query working group update task for numerous discussion on update languages which had a strong influence on our work. Thanks to Daniela Florescu, Don Chamberlin, Ioana Manolescu, Andrew Eisenberg, Kristoffer Rose, Mukund Raghavachari, Rajesh Bordawekar, and Michael Benedikt for their feedback on earlier versions of this draft. Special thanks go to Dan Suciu for proposing `snap` as the keyword used in XQuery!.

References

1. Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P'05*, 2005.
2. Bard Bloom. Lopsided little languages: Experience with XQuery. In *XIME-P'05*, 2005.
3. Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Simeon. XQuery 1.0: An XML query language. W3C Working Draft, April 2005.
4. Don Chamberlin. Communication regarding an update proposal. W3C XML Query Update Task Force, May 2005.
5. Don Chamberlin and Jonathan Robie. XQuery update facility requirements. W3C Working Draft, June 2005.
6. Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jerôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C Working Draft, Aug 2004. `http://www.w3.org/TR/query-semantics`.
7. Daniela Florescu et al. Communication regarding an XQuery update facility. W3C XML Query Working Group, July 2005.
8. Don Chamberlin et al. Communication regarding updates for XQuery. W3C XML Query Working Group, October 2002.
9. Mary Fernández, Jerôme Siméon, and Philip Wadler. *XQuery from the experts*, chapter Introduction to the Formal Semantics. Addison Wesley, 2004.
10. Daniela Florescu. Communication regarding update grammar. W3C XML Query Update Task Force, April 2005.
11. Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: An XML programming language for Web service specification and composition. In *Proceedings of International World Wide Web Conference*, pages 65–76, May 2002.
12. Giorgio Ghelli, Christopher Ré, and Jérôme Siméon. XQuery!: An XML query language with side effects, full paper, September 2005. `www.di.unipi.it/~ghelli/papers/XQueryBangTR.pdf`.
13. Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In *Database and XML Technologies (XSym)*, pages 5–20, May 2004.

14. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In ”Engineering theories of software construction”, ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, 2001.
15. Andreas Laux and Lars Matin. http://www.xmldb.org/xupdate, October 2000.
16. Patrick Lehti. Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit, 2001.
17. Xavier Leroy. *The Objective Caml system, release 3.08, Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, july 2004.
18. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. MIT Press, 1997.
19. Nicola Onose and Jérôme Siméon. XQuery at your Web service. In *Proceedings of International World Wide Web Conference*, New York, NY, May 2004.
20. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. Technical report, AT&T Labs Research, 2005.
21. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta,GA, April 2006.
22. Michael Rys. Proposal for an XML data modification language, version 3, May 2002. Microsoft Corp., Redmond, WA.
23. A. Schmidt, F. Waas, M. Kersten, M. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, August 2002.
24. Gargi M. Sur, Joachim Hammer, and Jérôme Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.
25. I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *SIGMOD*, 2001.

A Grammar

Figure 1 shows the grammar of XQuery!, which extends the grammar of XQuery 1.0 expressions. "`snap insert{} into{}`" abbreviates "`snap {insert{} into{}}`", and similarly for the other update primitives.

Expr ::= ... | *DeleteExpr* | *InsertExpr* | *ReplaceExpr* | *RenameExpr*
| *CopyExpr* | *SnapExpr*

DeleteExpr ::= `snap`? `delete { ` *Expr* ` }`

InsertExpr ::= `snap`? `insert { ` *Expr* ` }` *InsertLocation*

InsertLocation ::= (`as first` | `as last`)? `into { ` *Expr* ` }`
| `before { ` *Expr* ` }`
| `after { ` *Expr* ` }`

ReplaceExpr ::= `snap`? `replace { ` *Expr* ` } with { ` *Expr* ` }`

RenameExpr ::= `snap`? `rename { ` *Expr* ` } to { ` *Expr* ` }`

CopyExpr ::= `copy { ` *Expr* ` }`

SnapExpr ::= `snap` (`nondeterministic` | `ordered`)? `{ ` *Expr* ` }`

Fig. 1. XQuery! Grammar

B Language Semantics

We present here the semantics of all the update operators of XQuery! and of the most important XQuery 1.0 operators, enriched in order to specify their effect of the store and on the delta list. This semantics imposes an evaluation order for each operator.

The metavariables *node*, $node_i$ (for any i), *nodepar*, *nodepos* range over node ids, *nodeseq* ranges over node id sequences, and *name* ranges over qnames. The metavariables are *normative*, i.e. they express constraints on rule applicability. This means that, if a judgment in the premise uses *node* in the result position, as in:

$$store_0;\ dynEnv \vdash Expr \ \Rightarrow\ node;\ \Delta_1;\ store_1,$$

the judgment can only be applied if *Expr* evaluates to a value which is a node.

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr} \Rightarrow \textit{node}_1;\ \Delta_1;\ \textit{store}_1 \quad (\textit{store}_2, \textit{node}_2) = \text{deepcopy}(\textit{store}_1, \textit{node}_1)}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{copy \{}\textit{Expr}\texttt{\}} \Rightarrow \textit{node}_2;\ \textit{store}_2;\ ()}$$

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr} \Rightarrow \textit{value};\ \Delta;\ \textit{store}_1 \quad \textit{store}_2 = \text{apply } \Delta \text{ to } \textit{store}_1}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{snap \{}\textit{Expr}\texttt{\}} \Rightarrow \textit{value};\ ();\ \textit{store}_2}$$

$$\frac{\begin{array}{c}\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr}_1 \Rightarrow \textit{node};\ \Delta_1;\ \textit{store}_1 \\ \textit{store}_1;\ \textit{dynEnv} \vdash \textit{Expr}_2 \Rightarrow \textit{name};\ \Delta_2;\ \textit{store}_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{rename}(\textit{node}, \textit{name}))\end{array}}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{rename \{}\textit{Expr}_1\texttt{\}} \texttt{ to \{}\textit{Expr}_2\texttt{\}} \Rightarrow ();\ \Delta_3;\ \textit{store}_2}$$

$$\frac{\begin{array}{c}\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr}_1 \Rightarrow \textit{node};\ \Delta_1;\ \textit{store}_1 \\ \textit{store}_1;\ \textit{dynEnv} \vdash \textit{Expr}_2 \Rightarrow \textit{nodeseq};\ \Delta_2;\ \textit{store}_2 \\ \textit{store}_2;\ \textit{dynEnv} \vdash \text{parent}(\textit{node}) \Rightarrow \textit{nodepar};\ ();\ \textit{store}_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{insert}(\textit{nodeseq}, \textit{nodepar}, \textit{node}), \text{delete}(\textit{node}))\end{array}}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{replace \{}\textit{Expr}_1\texttt{\}} \texttt{ with \{}\textit{Expr}_2\texttt{\}} \Rightarrow ();\ \Delta_3;\ \textit{store}_2}$$

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr} \Rightarrow \textit{node};\ \Delta_1;\ \textit{store}_1 \quad \Delta_2 = (\Delta_1, \text{delete } \textit{node})}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{delete \{}\textit{Expr}\texttt{\}} \Rightarrow ();\ \Delta_2;\ \textit{store}_1}$$

$$\frac{\begin{array}{c}\textit{store}_0;\ \textit{dynEnv} \vdash \textit{Expr}_1 \Rightarrow \textit{nodeseq};\ \Delta_1;\ \textit{store}_1 \\ \textit{store}_1;\ \textit{dynEnv} \vdash \textit{Expr}_2 \Rightarrow \textit{node}_2;\ \Delta_2;\ \textit{store}_2 \\ \textit{store}_2;\ \textit{dynEnv} \vdash \textit{InsertLocation node}_2 \Rightarrow (\textit{nodepar}, \textit{nodepos});\ ();\ \textit{store}_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{insert}(\textit{nodeseq}, \textit{nodepar}, \textit{nodepos}))\end{array}}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{insert \{}\textit{Expr}_1\texttt{\}}\ \textit{InsertLocation}\ \texttt{\{}\textit{Expr}_2\texttt{\}} \Rightarrow ();\ \Delta_3;\ \textit{store}_2}$$

Insert Location Judgments:

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{as last into \{}\textit{node}\texttt{\}} \Rightarrow (\textit{nodepar}, \textit{nodepos});\ \textit{store}_0;\ ()}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{into \{}\textit{node}\texttt{\}} \Rightarrow (\textit{nodepar}, \textit{nodepos});\ \textit{store}_0;\ ()}$$

$$\frac{}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{as first into \{}\textit{node}\texttt{\}} \Rightarrow (\textit{node}, \textit{node});\ \textit{store}_0;\ ()}$$

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \text{last\_child\_otherwise\_self}(\textit{node}) \Rightarrow (\textit{nodepos});\ \textit{store}_0;\ ()}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{as last into \{}\textit{node}\texttt{\}} \Rightarrow (\textit{node}, \textit{nodepos});\ \textit{store}_0;\ ()}$$

$$\frac{\textit{store}_0;\ \textit{dynEnv} \vdash \text{parent}(\textit{node}) \Rightarrow (\textit{nodepar});\ \textit{store}_0;\ ()}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{after \{}\textit{node}\texttt{\}} \Rightarrow (\textit{nodepar}, \textit{node});\ \textit{store}_0;\ ()}$$

$$\frac{\begin{array}{c}\textit{store}_0;\ \textit{dynEnv} \vdash \text{is\_first\_child}(\textit{node}) \Rightarrow \text{true};\ \textit{store}_0;\ () \\ \textit{store}_0;\ \textit{dynEnv} \vdash \text{parent}(\textit{node}) \Rightarrow \textit{nodepar};\ \textit{store}_0;\ ()\end{array}}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{before \{}\textit{node}\texttt{\}} \Rightarrow (\textit{nodepar}, \textit{nodepar});\ \textit{store}_0;\ ()}$$

$$\frac{\begin{array}{c}\textit{store}_0;\ \textit{dynEnv} \vdash \text{is\_first\_child}(\textit{node}) \Rightarrow \text{false};\ \textit{store}_0;\ () \\ \textit{store}_0;\ \textit{dynEnv} \vdash \text{parent}(\textit{node}) \Rightarrow \textit{nodepar};\ \textit{store}_0;\ () \\ \textit{store}_0;\ \textit{dynEnv} \vdash \text{preceding\_sibling}(\textit{node}) \Rightarrow \textit{nodepos};\ \textit{store}_0;\ ()\end{array}}{\textit{store}_0;\ \textit{dynEnv} \vdash \texttt{before \{}\textit{node}\texttt{\}} \Rightarrow (\textit{nodepar}, \textit{nodepos});\ \textit{store}_0;\ ()}$$

Fig. 2. XQuery! Semantics of Update Operations

$$\frac{\begin{array}{c}\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{item}_1, \ldots, \mathit{item}_m;\ \Delta;\ \mathit{store}_1 \\ \mathit{for}\ i\ \mathit{in}\ 1 \ldots m:\ \mathit{store}_i;\ (\mathit{dynEnv} + x \Rightarrow \mathit{item}_i) \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_i;\ \Delta_i;\ \mathit{store}_{i+1} \\ \Delta' = (\Delta, \Delta_1, \ldots, \Delta_m)\end{array}}{\mathit{store}_0;\ \mathit{dynEnv} \vdash \texttt{for}\ \ x\ \ \texttt{in}\ \{\mathit{Expr}_1\}\ \texttt{return}\ \ \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_1, \ldots, \mathit{value}_n;\ \Delta';\ \mathit{store}_{m+1}}$$

$$\frac{\begin{array}{c}(f \Rightarrow \text{fun}(x_1, \ldots, x_m, \mathit{Expr}) : T_1, \ldots, T_n \rightarrow T) \in \mathit{funEnv} \\ \mathit{for}\ j\ \mathit{in}\ 1 \ldots m:\ \mathit{store}_j;\ \mathit{dynEnv} \vdash \mathit{Expr}_j \ \Rightarrow\ \mathit{value}_j;\ \Delta_j;\ \mathit{store}_{j+1} \\ \mathit{store}_{m+1};\ (\mathit{dynEnv} + x_1 \Rightarrow \mathit{value}_1 + \ldots + x_m \Rightarrow \mathit{value}_m) \vdash \mathit{Expr} \ \Rightarrow\ \mathit{value}';\ \Delta;\ \mathit{store}_{m+2} \\ \Delta' = (\Delta_1, \ldots, \Delta_m, \Delta)\end{array}}{\mathit{store}_1;\ \mathit{dynEnv} \vdash f(\mathit{Expr}_1, \ldots, \mathit{Expr}_m) \ \Rightarrow\ \mathit{value}';\ \Delta';\ \mathit{store}_{m+2}}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{name};\ \Delta;\ \mathit{store}_1 \\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value};\ \Delta;\ \mathit{store}_2 \\ (\mathit{store}_3, \mathit{node}) = \texttt{NewElement}(\mathit{store}_2, \mathit{name}, \mathit{value})\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \texttt{element\{}\mathit{Expr}_1\texttt{\}\{}\mathit{Expr}_2\texttt{\}} \ \Rightarrow\ \mathit{node};\ \Delta;\ \mathit{store}_3}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{value}_1;\ \Delta_1;\ \mathit{store}_1 \\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_2 \\ \Delta = (\Delta_1, \Delta_2)\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1, \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_1, \mathit{value}_2;\ \Delta;\ \mathit{store}_2}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{value}_1;\ \Delta_1;\ \mathit{store}_1 \\ \mathit{store}_1;\ (\mathit{dynEnv} + x \Rightarrow \mathit{value}_1) \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_2 \\ \Delta = (\Delta_1, \Delta_2)\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \texttt{let}\ \ x\ \ \texttt{:=}\ \ \mathit{Expr}_1\ \ \texttt{return}\ \ \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_2}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr} \ \Rightarrow\ \texttt{true};\ \Delta_1;\ \mathit{store}_1 \\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{value};\ \Delta_2;\ \mathit{store}_2 \\ \Delta = (\Delta_1, \Delta_2)\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Expr}_1\ \ \texttt{else}\ \ \mathit{Expr}_2 \ \Rightarrow\ \mathit{value};\ \Delta;\ \mathit{store}_2}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr} \ \Rightarrow\ \texttt{false};\ \Delta_1;\ \mathit{store}_1 \\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value};\ \Delta_2;\ \mathit{store}_2 \\ \Delta = (\Delta_1, \Delta_2)\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Expr}_1\ \ \texttt{else}\ \ \mathit{Expr}_2 \ \Rightarrow\ \mathit{value};\ \Delta;\ \mathit{store}_2}$$

$$\frac{\begin{array}{c}\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1 \ \Rightarrow\ \mathit{value}_1;\ \Delta_1;\ \mathit{store}_1 \\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_2 \ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_2 \\ b = \texttt{equal}(\mathit{value}_1, \mathit{value}_2) \\ \Delta = (\Delta_1, \Delta_2)\end{array}}{\mathit{store};\ \mathit{dynEnv} \vdash \mathit{Expr}_1 = \mathit{Expr}_2 \ \Rightarrow\ b;\ \Delta;\ \mathit{store}_2}$$

Fig. 3. XQuery! Semantics of Non-Update Operations