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 in which evaluation order is 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 [14, 19, 21, 1] have been based on restricted compositionality 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 [2] 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 [14, 7, 6]. However, the ability to use updates in any context (e.g., in function calls) and to control update application makes it more expressive than previous proposals. 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 [21] 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 this paper are:

- A formal description of a semantic framework for extending XML query languages with side-effect operations which can appear anywhere in the query.
- The description of a new construct (`snap`) that can be used to control update application. The semantics of `snap` enables unlimited nesting and allows the optimizer to recover standard database optimizations, even in the presence of side-effects.
- The definition of XQuery!, an extension to XQuery 1.0 with first-class updates, and an example of its use in a Web service usecase.
- The description of a complete 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 [19, 14], and has been studied further in [7, 6, 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 [4], and have been explored by the W3C XML update task force [9, 3]. Work on the XL programming language [10] 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 [11].

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 [19, 14, 7, 6, 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 for each person who buys an item.

```
for $p in $auction//person
for $t in $auction//closed_auction
where $t/buyer/@person = $p/@id
return insert { <buyer name="{$p/name}"
                       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 is controlled explicitly, 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 the above example. In the rest of this section, we give a more advanced Web service use-case which requires complex composition of side-effects and queries, and control over update application.

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 for now. The server stores the auction document from XMark [20] 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 (
    ( let $name := $auction//person[@id = $userid]/name return
      insert { <logentry user="{$name}" itemid="{$itemid}"/> }
      into { $log }),
    $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).

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 the value returned by atomic update operations is always 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 see the result of previous side-effects. For this reason,

XQuery! supports a `snap { ` *Expr* ` }` operator which evaluates *Expr*, collects its update requests, and makes the effects of these updates visible to the rest of the query. A `snap` is always implicitly present around the top-level query in the main XQuery! module, so that the usual "delay until the end" 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).

```
( 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 ())),
```

Here, the `snap` around `insert` makes the insertion happen. The insertion is visible to the code inside the subsequent if-then-else because XQuery! semantics imposes that in the sequence constructor (`e1,e2`), `e1` be evaluated before `e2`. Hence, XQuery!'s ability to support the above example relies on the combination of the `snap` operator and of an explicit evaluation order. This is an important departure from XQuery 1.0 semantics, and is discussed in the next subsection.

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, 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.

2.4 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. This freedom is visible for instance, if both expressions e_1 and e_2 were to raise an error, as which of those errors 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 [16, 15], 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

---

[4] Simon Peyton-Jones: "lazy evaluation and side effects are, from a practical point of view, incompatible" [13].

style, but is quite constraining for the compiler. However, as we discuss in Section 3, inside an innermost `snap` no side-effect takes place, hence we recover XQuery 1.0 freedom of evaluation order in those cases. In other words, 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.

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 ensures that the counter function performs as expected, returning an increasing value after each call. 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 it is closed. 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 [5] as follows. First, each expression is normalized to a *core* expression. Then, the meaning of core expressions is defined by a semantic judgment $dynEnv \vdash Expr \Rightarrow value$. 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 instances that are being processed. It should be clear from the discussion in Section 2 that only snap expressions actually modify the store. We extend the semantic judgment so that expressions may modify the store, and produce both a value and a list of pending updates. 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 standard updates primitives: insertion, deletion, replacement, and renaming of XML nodes [14, 19, 21, 1, 7, 6]. 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 [11].

The detailed semantics of these primitives is also standard: insertion allows a sequence of nodes to be inserted below a parent at 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. If a "deleted" (actually, detached) node is still accessible from a variable, it can still be queried, or inserted somewhere. Although this approach requires garbage collection, we believe it is slightly simpler to specify, implement, and program with than the alternative "erase" semantics.

3.2 XDM stores and update requests

Store. To represent the state of XQuery! computation, we need a notion of *store*, which specifies the valid node ids and, for each node id, its kind (element, attribute, text...), parent, name, and content. A formal definition can be found in [11, 12, 8]. 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, its *application* is a partial function from stores to stores. 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 the 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, denoted Δ, is a list of update requests. Update lists are collected during the execution of the code inside a snap, and are applied when 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 semantics for update list application: *ordered*, *non-deterministic*, and *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* semantics, the update requests are applied in an arbitrary order. In the *conflict-detection* semantics, 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.

The *ordered* approach is simple and deterministic, but imposes more restrictions on the optimizer. The *non-deterministic* approach is simpler to implement and optimize, but 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 while avoiding non-determinism. However, it rules out many reasonable pieces of code, as exemplified in the full paper. Moreover, it can raise run-time failures 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, so that 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.

$$\frac{[\![\text{insert } \{\mathit{Expr}_1\} \text{ into } \{\mathit{Expr}_2\}]\!]}{\text{insert } \{\text{copy } \{[\![\mathit{Expr}_1]\!]\}\} \text{ as last into } \{[\![\mathit{Expr}_2]\!]\}}$$

3.4 Formal semantics

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

$$\mathit{store}_0; \mathit{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{value}; \Delta; \mathit{store}_1$$

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 XQuery! 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 $Expr_2$.

If a sub-expression is guaranteed not to invoke a `snap`, the compiler can again choose evaluation order as in the original XQuery 1.0 semantics for that sub-expression. 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 [11]).

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`. The metavariables express constraints on rule applicability: *node* and *nodepar* can only be matched to node ids, and *nodeseq* only to a sequence of node ids.

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

The evaluation produces an empty sequence and an update list Δ_3. 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` {*Expr*} looks very simple: *Expr* is evaluated, it produces its own update list Δ, Δ is applied to the store, and the value of *Expr* is returned.

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

The evaluation of *Expr* may itself modify the store, and `snap` updates this modified store. 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 [11]).

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 implemented as an extension to the Galax XQuery engine [18, 17], and a preliminary version is available for download[5]. 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 corresponds to a given `snap` scope, and where the order inside a list is irrelevant. The invocation of an update operation adds an update to the update list on the top of the stack. When exiting a `snap`, the top-most update list is popped from the stack and

---

[5] `http://xquerybang.cs.washington.edu/`

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 an existing compiler. The implementation of the ordered semantics is more involved, as we must rely on a specialized tree structure to represent the update lists in a way that allows the compiler to lazily evaluate FLWOR expressions and still retain the order in which each update must be applied. We refer to the full paper [11] for more details.

4.2 Compilation architecture

Implementing XQuery! does not require modifications to the XQuery processing model. The XQuery! compiler in Galax proceeds by first *parsing* the query into an AST and *normalization*, followed by a phase of syntactic *rewriting*, *compilation* into the XML algebra of [18] with some simple update extensions, *optimization* and finally *evaluation*.

Changes to parsing and normalization are trivial (See Section 3). To preserve the XQuery! semantics, some of the syntactic rewritings must be guarded by a judgment checking whether side effects may occur in a given sub-expression. Of course, this is not necessary when the query is guarded by an innermost `snap`, i.e., a `snap` whose scope contains no other `snap`, nor any call to any function which may cause a `snap` to be evaluated. Inside such innermost `snap`, all the rewritings immediately apply.

4.3 XQuery! 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 the following variant of XMark query 8 which, for each person, stores information about the purchased 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 essentially the same as 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., [18]. A naive nested-loop evaluation 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 used for that query plan is that of [18], where algebraic operators are written as follows:

$$\mathsf{Op}[\mathsf{p}_1,\ldots,\mathsf{p}_i]\{\mathsf{DOp}_1,\ldots,\mathsf{DOp}_h\}(\mathsf{Op}_1,\ldots,\mathsf{Op}_k)$$

with Op being the operator name; p_i's being the static parameters of the operator; DOp_i's being dependent sub-operators; and Op_i's are input (or independent) operators. [...] stands for tuple construction, # for tuple field access, and IN is the input of dependent sub-operators (as passed by the operator from the result of its independents sub-operators). Path expressions, constructors, and update operators are written in XQuery! syntax for conciseness.

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

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 the values of side-effects are preserved. Those preconditions check for properties related to cardinality and a notion of independence between expressions. 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 store. 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 groups update requests to apply them all at once, which is reminiscent of transactions. However, their purpose and semantics

are essentially orthogonal. Flat transactions are meant to protect a piece of code from concurrently running transactions, while nested transactions allow the programmer to isolate different concurrent threads within its own code.

On the other side, without concurrency and failures, transactions have no effect. In particular, within a given transaction, access to a variable x that has been modified will return the new value for that variable. 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. In a nutshell, 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 [13]. In those approaches, the type system distinguishes between purely functional code, which can be lazily evaluated, from impure "monadic" code, for which 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.

An XQuery! processor must also distinguish the case where the query has some pending updates but no effect, going beyond the pure-impure distinction. 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.

Snapshot semantics and optimization. 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 a semantic framework, and an extension of XQuery 1.0 that supports fully compositional updates. The main contribution of our work is the definition of a snap operator which enables control over update application and supports arbitrary nesting. We described a prototype implementation which is available for download. Many important issues are still open for research, such as static typing, optimization, and transactional mechanisms. We are currently working on those issues.

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. 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.
3. Don Chamberlin. Communication regarding an update proposal. W3C XML Query Update Task Force, May 2005.
4. Don Chamberlin and Jonathan Robie. XQuery update facility requirements. W3C Working Draft, June 2005.
5. 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`.
6. Daniela Florescu et al. Communication regarding an XQuery update facility. W3C XML Query Working Group, July 2005.
7. Don Chamberlin et al. Communication regarding updates for XQuery. W3C XML Query Working Group, October 2002.
8. Mary Fernández, Jerôme Siméon, and Philip Wadler. *XQuery from the experts*, chapter Introduction to the Formal Semantics. Addison Wesley, 2004.
9. Daniela Florescu. Communication regarding update grammar. W3C XML Query Update Task Force, April 2005.
10. 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.
11. Giorgio Ghelli, Christopher Ré, and Jérôme Siméon. XQuery!: An XML query language with side effects, full paper, 2005.
    `http://xquerybang.cs.washington.edu/papers/XQueryBangTR.pdf`.
12. 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.
13. 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.
14. Patrick Lehti. Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit, 2001.
15. 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.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised).* MIT Press, 1997.
17. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. Technical report, AT&T Labs Research, 2005.
18. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta,GA, April 2006.
19. Michael Rys. Proposal for an XML data modification language, version 3, May 2002. Microsoft Corp., Redmond, WA.
20. 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.
21. Gargi M. Sur, Joachim Hammer, and Jérôme Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.