XQuery!: An XML query language with side effects

Giorgio Ghelli
Università di Pisa

Christopher Ré
University of Washington

Jérôme Siméon
IBM T.J. Watson Research Center

Draft of October 14, 2005, please do not distribute

Abstract

As XML applications become more complex, there is a growing interest in extending XQuery with side-effect operations, notably XML updates. Unfortunately, the presence of side-effects is at odds with XQuery's declarative semantics which favors optimization. In this paper, we propose a semantic framework that enables extending XQuery with side-effect operations, while preserving the benefits of XQuery's declarative semantics when possible. We use that framework to define "XQuery!", an extension of XQuery 1.0 that supports first-class XML updates and user-level control over update application. We show that those extensions can be easily implemented within an existing XQuery processor and 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. The benefits of such extensions are far reaching, giving XQuery the expressive power necessary to support Web service applications [13, 22], scripting applications [3], or more simply to support XML updates over large XML repositories [28].

Because these extensions involve side-effects, they appear to be at odds with the declarative semantics of XQuery 1.0. As a result, previously proposed extensions to XQuery 1.0 restrict the usage of side-effect operations. In this paper, we develop the semantic foundations for extending XQuery 1.0 with side-effects operations in a fully compositional way. We use the resulting framework to define XQuery! (read: "XQuery Bang"), an extension of XQuery 1.0 [4], that supports first-class XML updates and user-level control over update application. We show such a language can be obtained with small impact on XQuery's declarative semantics and optimization techniques.

The semantic framework is a simple extension of the XQuery 1.0 Formal Semantics [8], and builds upon previous experiences in functional languages that support side-effects [21, 19]. The main difference between our framework and a programming language approach is the presence of an operator (`snap`) which allows users to identify declarative fragments within their programs, and inside which we are enable to recovery traditional database optimizations. Although this operator bears some resemblance with nested transactions [14, 20], it is designed to support effectively compositional updates rather than concurrency.

The XQuery! language is a small extension to XQuery 1.0, which supports common update operations over XML documents. Thanks to its ability to use side-effect operations in any context (e.g., in function calls), this makes for a very expressive language, which is strictly more expressive than any previously proposed update language for XML we are aware of [25, 18, 29, 28, 17]. We show how to use XQuery! to build a simple Web service that supports logging of user calls for some of its operations.

The main 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 code.
- We describe a new construct (`snap`) that can be used to control update applications. The semantics of that operator enables unlimited nesting. We describe how, and in which context, this semantics enables the recovery of standard database optimizations in the presence of side-effects.

- We demonstrate the expressive power of the semantics framework by defining 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.

Some of the motivations for our work originate from discussions within the W3C XQuery Update Language Task Force, and from our own experience implementing and using a more restricted XML update language based on XQuery [28]. Composition being one of the foundations for the design of XQuery 1.0, attempting to allow compositional updates seems to be a natural direction, and is mentioned in the XQuery update requirements document [6]. The notion of delaying update applications to facilitate optimization (so called *snapshot* semantics) was first proposed in [25, 18], and has been further used and studied in [10, 9, 1]. Most of the previous proposals applied that semantics only for the whole query, while XQuery! provides control over it at the expression level through the `snap` operator. Languages with explicit control of the snapshot semantics have been explored by the W3C XML update task force [12, 5]. To the best of our knowledge, this work is the first to completely define the semantics of such an operator in a way which allows compositionality, and to explicit its relationship with optimization properties of the language.

The rest of the paper is organized as follows. Section 2 motivates our work using a Web service usecase. Section 3 describes the XQuery! grammar. Section 4 introduces some formal notations and the foundations for the semantic framework. Section 5 gives the language semantics using the proposed framework. Section 6 describes a simple implementation of XQuery! and discusses optimization issues. Section 7 concludes the paper.

2 Adding Logging to an XQuery Web Service

A compositional XML update extension for XQuery is attractive for many reasons. First, XQuery is itself compositional and XQuery developers are accustomed to building complex programs by composing a small set of basic expressions. Second, restricting compositionality often results in ad hoc rules which are difficult for the user to understand. Finally, compositionality brings additional expressiveness that is useful for applications. In the rest of the section we focus on a simple Web service scenario to illustrate the need of an expressive XML language which supports both data querying and side-effects.

2.1 An auction Web service with XQuery 1.0

We assume an application that implements an auction Web service using XQuery. The Web service is deployed as an XQuery module, where each function corresponds to a Web service operation; Web service clients access the Web Service through these functions.

The server stores the auction document from XMark [27]. The user cannot see the whole data, but only access it through the Web service interface. That interface makes two operations available. The first gives users access to all the open auctions. The second gives access to items with a given itemid, and requires the user to also provide his person id (obtained e.g., during login). Those operations can be implemented as follows, assuming the XMark document has been bound to variable `$auction`.

```
declare function auctions() {
  $auction//open_auctions
};

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

The following is a possible query over that Web service, on the client side, accessing all items for which there is an open auction from a happy seller with a current price of less than 100$. Note that the user must obtain an item reference number from the open auctions in order to get access to the item description.

```
let $myid := "person0"
for $o in auctions()//open_auction
where $o/seller/happiness > 8 and $o/current < 100
return get_item($o/itemref/@item,$myid)
```

2.2 Logging using XML updates

Now, let's assume that the Web service wants to log each item access. This cannot be easily done with XQuery 1.0, as there is no way to modify a value (the log), without passing it as a parameter to the function and returning the new value as a result of that function. On the other hand, this can be done transparently using an XML update within the body of the `get_item()` function. Here is a possible implementation of the modified `get_item()` function in XQuery!.

```
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}"
                       date="{current-date()}"/> }
    into { $log },
    (::: End logging code :::)
    $item
  )
};
```

As opposed to the previous program, the `get_item` function not only returns the item, but also adds a log entry for every access. The logging is implemented by an XML update expression that inserts a new `logentry` element into the content of the `$log` variable. Embedding the insert into a function allows the original query code to be easily reused, and makes the logging behavior transparent to the user. This particular example illustrates the need for expressions that have a side-effect (the log entry insertion) and also return a value (the item description).

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 (in addition to the effect they have on the XML data). This kind of function is not supported by any of the previously proposed XML update languages we are aware of [10, 9, 25, 18, 29, 28, 17].

2.3 Controlling update application

The previous example should look natural to programmers familiar with functional languages such as SML [21] or Caml [19], where side effect operations are also supported. Indeed, XQuery shares many design characteristics with those languages and it will not come as a surprise that similar extensions to XQuery are possible. However, XQuery differs from traditional functional programming languages in a few significant ways. First of all, the semantics of XQuery is very liberal in terms of evaluation order and error handling [1] in order to facilitate the work of optimizers. For that reason, several previous proposals [18, 25, 10, 9, 28, 1] have relied on a so-called *snapshot semantics*, which defers update application until the query is completed, hence facilitating the use of traditional database optimizations for the side-effect free part of the query. The same approach is used, by default, in XQuery!: if the `snap` operator (introduced below) is not used, every update that is requested by the query is only executed at the end of the top-level expression in the main XQuery! module, because a `snap` is always implictly present around that top-level expression.

However, a more fine-grained control on when a given update is applied is sometimes necessary. For example, consider the following variant for the logging code, in which the user wants to maintain a log count after each log insertion.

[1] See Section **[2.3.4 Errors and Optimization]** of the XQuery 1.0 specification [4]

```
(::: Logging code :::)
let $name := $auction//person[@id = $bidder]/name
return
 (snap insert { <logentry user="{$name}"
                          itemid="{$item/@id}"
                          date="{current-date()}"/> }
       into { $log },
  snap replace { $log/@count } with { count($log/logentry) })
(::: End logging code :::)
```

In this example, the replace operation is meant to update the number of log entries after the insertion. There are two remarks to make on the example. First, it relies on the first `insert` to be actually applied on `$log`, in order for the `replace` to work with the intended semantics. This is the reason for the presence of the `snap` keyword before the `insert`, which indicates that the effect of the update must be visible right away. Second, it relies on the fact that the operations in the sequence are evaluated in the order specified. Hence, this piece of code works as expected because the `snap` causes the log to be actually updated *and* because XQuery! semantics specifies that the sequence constructor `e1,e2` causes `e1` to be fully evaluated before `e2`.

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, consider 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 does not matter for purely functional programs, but in the presence of side-effects the evaluation order has an impact on the order in which side-effects occur. This is already observable in XQuery 1.0 when considering error handling. For example, if both expressions e_1 and e_2 were to raise an error, which error is reported may vary from implementation to implementation. This approach leaves considerable freedom to the compiler and optimizer which may reorder evaluation without worrying of how that may affect the final result.

Although that approach may be reasonable in an almost-purely functional language as XQuery 1.0, it is widely believed that programs that update data are impossible to reason about unless the evaluation order is easy to grasp.[2] The main reason is that some part of the code may rely on some update to have taken place, as in our previous example.

For this reason, in XQuery! we adopt the standard semantics used in popular functional languages with side-effects [21, 19], 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. Note however, that 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 second alternative requires a more complex formalization style, and is explored in Appendix A.

Obviously, XQuery!'s semantics is more constraining for the compiler. However, as we discuss in Section 5, inside an innermost `snap` no side-effect takes place, hence recovering XQuery 1.0 freedom of evaluation order. Interestingly, the same holds for the update requests that are generated in such scope: the implementation can actually produce them in any order, provided that, at the end of the `snap` scope, the order required by the official semantics is recovered.

This update order is a bit harder to maintain than sequence order, because a FLWOR expression may generate updates in the *for*, *where*, and *return* clause, while the result items are only generated in the *return* clause (see also Section 6). For this reason, XQuery! supports alternative semantics for update application which do not depend on the update order (Section 4).

2.5 Recovering joins with `snap`

XQuery! is geared toward database applications, in which join optimizations are very critical. Here we show how to recover a traditional join optimization in a query which combines a join predicate with side-effects. Consider the

[2]Simon Peyton-Jones: "lazy evaluation and side effects are, from a practical point of view, incompatible" [16].

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 the ones proposed in e.g., [24]. By default, XQuery! always assumes the presence of a `snap` around the outermost expression. This produces the most efficient behavior by default, enabling to use the standard group-by plan in our example. If the above program had been written using a `snap insert` instead, the group-by optimization would be more difficult to detect as one would have to know that the effect of those inserts are not observed in the rest of the query. We come back to that query and the properties that have to be checked in order to recover such optimizations in Section 6.

2.6 Programming with `snap`

In many situations, different scopes for the `snap` may lead to the same result, hence the programmer needs a guideline about `snap` placement. In such cases, a simple criterion can be used: a broader `snap` favors optimization, hence one should always leave `snap` as broad as possible. Smaller scopes should only be used when the rest of the program relies on the effect of the updates, or when writing a library function which should work independently of the context where it is called, as with the counter example above.

Consider, for another example, the following variant of the `get_items` function, which accesses a set of items by name, and logs each item access. The outermost `snap` is needed to specify that update is performed before control is returned to the caller.

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

The broad scope of `snap` ensures that traditional rewritings, such as join detection in our example, can still be used within the function itself. However, the optimization of a piece of code which calls this function would rely on complex proofs of independence of evaluation order with respect to the side effects performed by `get_items`. If the programmer did not use a `snap` at all, the pending updates would be passed to the calling function, until an outer `snap` is closed, and the optimization of the calling code could be easier.

In general, the intuition is that a maximally broad `snap` is good news for the optimizer, which is then allowed to essentially ignore the presence of side-effects. Again, this is why XQuery! always places a `snap` around the outermost piece of code, allowing users to naturally obtain the most efficient behavior. Of course, since every function may in principle execute a `snap`, cross-module optimization becomes quite hard. This may be alleviated by requiring that the possible execution of `snap` is declared in function signatures.

2.7 Nested snap

Support for nested snap 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 = $bidder]/name
  return
   (snap insert { <logentry id="{nextid()}"
                            user="{$name}"
                            itemid="{$item/@id}"/> }
         into { $log },
    snap replace { $log/@count } with { count($log/logentry) })
  (::: End logging code :::)
```

As that example shows, the `snap` operator must not freeze the state of the data model 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 5. Although `snap` is reminiscent of nested transactions, we do not explore this connection, mainly because XQuery 1.0 has no persistency or concurrency model. Extending our approach to a full transactional model is outside the scope of this paper.

2.8 Tree transformations using updates

Finally, let us come back to the case for compositional updates.Another interesting aspect of compositional XML updates is the ability to naturally express some tree transformations that are cumbersome to express in XQuery (similar to the so-called *transform* in [9]). Typical examples are functions that create an external version of a document by hiding some of the input data (e.g., to implement access control discipline over XML documents [2, 26]). In XQuery 1.0, this can only be implemented through functions which copy the data that is not hidden, and explicitly rebuild all the tree structure from the hidden point up to the root. In the presence of updates, the same transformation can be implemented as a composition of a `copy` operation with updates. As an example, we change here the implementation of the `auctions` function to not return the seller and the reserve price for the auctions.

```
  declare function auctions() {
    snap { let $result := copy { $auction }
           return
             (delete { $result//seller },
              delete { $result//reserve },
              $result) }
  };
```

The presence of a `snap` in that function is essential, otherwise an expression calling that function would not see the effect of the deletions until an outer `snap` is closed. However, since the function does not need to query the result of its own updates, there is no reason to choose a snapshot scope that is smaller than the whole function. Note that this update-the-copy style makes it quite easy for the optimizer to realize that the code is not side-effecting any pre-existing object, hence the optimizations of pure functional code still apply. Also, observe that `copy`, not being an update operation, is not affected by snapshot semantics, and is always executed right away.

3 Grammar

Figure 1 shows the grammar of XQuery!, which we use in the rest of this document. This grammar is written as a simple extension of XQuery 1.0 with side-effect expressions, an operation for explicit copying, and an operation for controlling the snapshot semantics. There are two reasons for the presence of the explicit copying. The first reason is its use to support *subtree queries* usecases, as the one presented in Section 2.8. The second reason is that it gives the ability to express the copying involved in the semantics of the insert and replace update operators. Those operators make a copy of the nodes being inserted or replaced. To avoid update conflicts, that copying should be performed during the querying part of the evaluation, and not during the application of the atomic updates. We will come back to this point in Section 5.

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* `}`

Figure 1: XQuery! Grammar

The first expressions are the typical update operations, similar to those proposed in [25, 18, 10, 9, 28]. The optional `snap` keyword before those update operations is for convenience, facilitating the use of the `snap` operator at the finest granularity.

All atomic updates return the empty sequence. Since XQuery flattens sequences, and notably *value*`,() =` *value*, we can use the sequence constructor expression in place of a traditional ML-like sequencing operator. This avoids the need for an additional operator and the addition of a unit type.

The `snap` operator has been described extensively in the previous section. We introduce here an option for the `snap` which allows the user to choose between three different update application semantics (ordered, non-deterministic, and conflict-detection semantics which is the default), explained in the next section. The copy operator returns a deep copy of its input.

4 XQuery! Data Model

This section lays out the formal foundations for the semantics of XQuery!. Instead of going through a complete re-formalization of the language, we indicate required changes and extensions to the existing XQuery 1.0 specifications.

We first reformulate the XQuery data model (XDM) [7] with a notion of *store*, which specifies, for each node id, its kind (element, attribute, text...), parent, children, name, and content. This notion is similar to that proposed in [15, 11]. On this store, we define accessors and constructors corresponding to those of the XDM, plus some update operations. Our definitions of atomic and update lists, and the notion of update conflict, originate from discussions within the W3C XQuery Update Language Task Force.

This extended data model is the basis for the evaluation relation defined, in next section, by extending the evaluation judgment defined in [8] to deal with side-effects.

While the store goes in the direction of formalizing access to documents and collections, this is not our aim here. The access to documents and collections presents some of the same issues which are presented by side-effects, notably sharing and aliasing, but is also related to questions of persistence and concurrency that we prefer to avoid by now.

4.1 The store

The store is an alternative formalization of the XDM, needed to deal with updates. Every program is evaluated with respect to a store. A store is a quadruple composed by an finite set of node ids N, by a set of edges $E \subseteq N \times N$, by a strict total order *ord* on N, and by a tuple of partial functions F, with domain N. We assume that $F = (\text{node-kind}_F, \text{node-name}_F, \text{content}_F)$. For a complete formal treatment of the XDM, F should contain many other partial functions, such as *type-name, base-uri, document-uri, typed-value, nilled, is-id, is-idrefs, prefix, uri, target*, but we ignore them here for simplicity. E and F give the essential information about a node.

The function node-kind_F is total, specifies the kind of a node, and its value determines whether the value of node-name_F and content_F is defined on n, according to XDM constraints. The edges E model the *parent* property of the XDM, and satisfy XDM constraints (N is a forest, every parent is either a document or an element node…). As in the XDM, every node can be a root, not just a document node. The order *ord* is compatible with the document order implied by the edges E. These details are formalized in the Appendixes. A store defines a notion of valid values, as follows.

> **Definition**: a value is a sequence of items, where an item is either an atomic value or a node id. An atomic value is an element of an atomic type, as defined in the XDM. A value is valid w.r.t. a store (N, E, ord, F) if every node id in the value belongs to N.

The data model includes the XDM accessors and constructors, discussed in the Appendixes.

We add atomic updates, and a deep-copy constructor $\text{deepcopy}(\mathit{store}, \mathit{node})$ which returns a pair $(\mathit{store}', \mathit{node}')$, where store' extends *store* and node' is the root of a newly allocated deep copy of the tree rooted in *node*. This operator is already implicitly present in some XDM operations, such as element construction.

4.2 Atomic updates

A formal definition of atomic updates is needed to precisely discuss the issues of update commutativity and compatibility. We use a specific set of atomic update operations similar to those proposed in [10, 9, 28]. However, the framework we propose is mostly orthogonal to the exact semantics of atomic updates. Hence, we only give here an overview of their semantics; for a full definition see the Appendixes.

An atomic update is a tuple op(p1,…,pn) formed by an operation name op and a list of parameters pi, which are values. The *application* of an atomic update to a store yields a new store or fails. We list here the atomic updates and describe their application on a store. All of them fail when some parameter is not valid w.r.t. the store, or when the other preconditions that we list below are not satisfied. Otherwise, we say that the update is valid on the store.

We describe update applications in the case where every node parameter is of kind *element*. Nodes of the other kinds require some more preconditions and manipulations in order to enforce XDM constraints.

- apply insert (*nodeseq*, *nodepar*, *nodepos*) to (N, E, ord, F)

  Preconditions:

  1. *nodepar* is a single node, *nodepos* is single node.
  2. every node in *nodeseq* is a root in E.
  3. *nodepos* is either a child of *nodepar* or is *nodepar* itself.

  Effects: the nodes in *nodeseq* are inserted as children of *nodepar*, immediately after *nodepos*, and before any other child, by updating both E and *ord*. If *nodepos =nodepar*, then the new nodes become the initial children of *nodepar*. Observe that *nodeseq* is not copied, but is required to be a sequence of root nodes.

- apply delete (*node*) to (N, E, ord, F)

  Precondition: *node* is a single node.

  Effects: the pair $(\mathit{node}', \mathit{node})$ is removed from E, if such pair exists, hence *node* is detached from its parent, and *ord* is updated accordingly. The node, and its descendants, are not actually deleted from the store, since they may still be reachable from a variable.

- apply rename (*node*, *qname*) to (N, E, ord, F)

  Precondition: *node* is a single node, *qname* is a QName.

  Effects: F is updated so that node-name$_F$(*node*) $=$ *qname*.

We do not consider types here; otherwise, every update operation should update the value of type-name$_F$(*node*) to xs:anyType, for any node which is an ancestor or a descendant of the target node in the store before the update or in the store after the update. The validity(*node*) and validation-attempted(*node*) properties should also be updated for the same nodes, to record the fact that node validity cannot be relied upon any more. Static typing would also be heavily affected. Many techniques to face this problem could be studied, but they are out of the scope of this paper.

Hereafter we use $u_1, \ldots, u_n$ to range on atomic updates.

4.3 Update lists

Update lists (Δ) are sequences of atomic updates: $\Delta \equiv (u_1, \ldots, u_n)$. An update list represents a list of updates that are collected during the execution of the code inside a `snap`, and are applied at once when the `snap` scope is closed; the result of such application is denoted by "apply Δ to *store*". In our semantic framework, an update list is an *ordered* list, whose order is fully specified by the language semantics. However, if a processor were allowed to ignore such order, the optimizer would gain some more freedom. For this reason, different approaches can be considered for the definition of the effect of the application of a sequence of updates to a store. We describe here the following three approaches: *ordered*, *non-deterministic*, or *conflict-detection*.

In the *ordered* approach, the atomic updates are applied in the order specified by Δ. In the *non-deterministic* approach, the atomic updates 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 and 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. Actually, we only consider those permutations where every insert precedes any delete. This restriction allows many conflicts to be avoided, as detailed in the next subsection, and still does not depend on the order of Δ.

The *ordered* approach is simple and deterministic, but imposes more restrictions on the optimizer. Consider the following pieces of code, where `cond($x)` is any "simple" condition (i.e., it only depends on `$x` and does not modify the store):

```
ordered-insert     = for $x in $a where cond($x)
                     return snap insert $x into $list

declarative-insert = snap {for $x in $a where cond($x)
                           return insert $x into $bag}
```

In the first case, the programmer is trying to specify that updates should be performed immediately, hence reflecting the order in which they are requested, while in the second case no such indication is given. However, the *ordered* approach makes the two pieces of code semantically equivalent, hence preventing some natural optimizations. Assume, for example, that each element of `$a` is stored in a database, in an arbitrary order, together with its document position; the ordered semantics forces the system to perform a sort operation, or a sorted retrieval, even in the `declarative-insert` case. 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. For example, the declarative-insert code above would raise an exception any time we try and insert more than one element, since the order of two insertions into the same parent affects the result. Hence, in this case, the programmer is forced to write the function in the non-optimizable ordered-insert version, and the optimization advantage over the ordered execution approach case is lost.The fact that the *conflict-detection* approach raises run-time failures, where the same piece of code may fail or not depending on the state of the store, is also a problem.

Conflict-detection can be actually combined with the other approaches, yielding an hybrid approach where we first check for some class of conflicts, raise an error if one is present, and otherwise execute the updates in either an ordered or a non-deterministic approach.

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

4.4 Update conflicts

We say that a set of atomic updates are "in conflict" when the order of their application to some store affects the result. We first discuss binary conflicts, where we distinguish *validity conflicts* and *result conflicts*.

Two updates u_1, u_2 are in a *validity conflict* if a store exists such that they are both valid in the store, but u_1 is not valid after u_2 is applied. The only precondition that may become invalid because of an update is condition 3 on insert (*nodeseq*, *nodepar*, *nodepos*), in case *nodepos* is a child of *nodepar*, since it would be invalidated by a preceding delete (*nodepos*).[3] Since replace is translated as insert after a node followed by a delete of the same node (Section 5.4), every replace operation generates a delete-insert conflict. These conflicts are avoided by stipulating that every insertion is always performed before any deletion.

Two updates u_1, u_2 are in a *result conflict* if a store exists such that applying u_1, u_2 or u_2, u_1 yield different stores. Two operations insert$(\mathit{nodeseq}_1, \mathit{nodepar}, \mathit{nodepos})$, and insert$(\mathit{nodeseq}_2, \mathit{nodepar}, \mathit{nodepos})$, are in result conflict if $\mathit{nodeseq}_1 \neq \mathit{nodeseq}_2$. We also have a result conflict between rename$(\mathit{node}, \mathit{qname}_1)$ and rename$(\mathit{node}, \mathit{qname}_2)$, when $\mathit{qname}_1 \neq \mathit{qname}_2$. The insert-insert conflict is a serious issue, since it arises when one writes the following code, which we believe may commonly occur.

```
snap { for $x in $seq
         ...
         return insert f($x) into $bag
     }
```

A set of operations is conflict-free if no pair is in conflict. We suspect that the opposite implication is true as well. Anyway, we believe that conflicts should be detected by focusing on binary conflicts only.

5 Semantics

We describe the semantics of XQuery!, following the approach of [8]: each expression is normalized to a *core* expression, and the meaning of core expressions is defined.

5.1 XQuery! Core

Figure 2 describes the XQuery! core grammar. It contains the same set of expression as the grammar from the previous section. However, note the absence of optional `snap` keywords in front of atomic updates. We will see in the next section that these are normalized into regular `snap` expressions.

5.2 Normalization

Normalization is trivial for most of the new expressions in XQuery!. Normalization remains unchanged for all other XQuery 1.0 expressions.

In the following normalization rules $\$x_i$ for any i is a fresh variable.

[3] If one considers an erase operation which deletes nodes from *N*, then any operation which has *node* in its parameters would be in validity conflict with erase (*node*).

```
CExpr            ::= ...                              (:  XQuery 1.0 expressions :)
                   | CDeleteExpr                      (:  Delete atomic update :)
                   | CInsertExpr                      (:  Insert atomic update :)
                   | CReplaceExpr                     (:  Replace atomic update :)
                   | CRenameExpr                      (:  Rename atomic update :)
                   | CCopyExpr                        (:  Deep copy expression :)
                   | CSnapExpr                        (:  Snap expression :)
CDeleteExpr      ::= delete { CExpr }
CInsertExpr      ::= insert { CExpr } CInsertLocation
CInsertLocation  ::= (as first | as last) into { CExpr }
                   | before { CExpr }
                   | after { CExpr }
CReplaceExpr     ::= replace { CExpr } with { CExpr }
CRenameExpr      ::= rename { CExpr } to { CExpr }
CCopyExpr        ::= copy { CExpr }
CSnapExpr        ::= snap (nondeterministic | ordered )? { CExpr }
```

Figure 2: Core XQuery! Grammar

Normalization of Delete

A delete expression with a snap modifier is normalized as a composition of snap and delete.

$$\frac{[\![\texttt{snap \{delete \{}Expr_1\texttt{\}\}}]\!]}{\texttt{snap \{}[\![\texttt{delete \{}Expr_1\texttt{\}}]\!]\texttt{\}}}$$

A delete expression is normalized into iteration over a core delete expression. Remember that the delete expression can take a sequence of nodes as input, while an atomic delete takes a single node as input.

$$\frac{[\![\texttt{delete \{}Expr_1\texttt{\}}]\!]}{\texttt{for } \$x_0 \texttt{ in \{}[\![Expr_1]\!]\texttt{\} return delete \{}\$x_0\texttt{\}}}$$

Normalization of Insert

An insert expression with a snap modifier is normalized as a composition of snap and insert.

$$\frac{[\![\texttt{ snap insert \{}Expr_1\texttt{\} }((\texttt{as first}|\texttt{as last})?\texttt{ into}|\texttt{before}|\texttt{after})\texttt{ \{}Expr_2\texttt{\}}]\!]}{\texttt{snap \{}[\![\texttt{insert \{}Expr_1\texttt{\} }((\texttt{as first}|\texttt{as last})?\texttt{ into}|\texttt{before}|\texttt{after})\texttt{ \{}Expr_2\texttt{\}}]\!]\texttt{\}}}$$

An insert expression is normalized into a core insert expression. In the case of an insert into without a modifier, it is treated as inserting as the last node within the given element.

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

$$\frac{[\![\texttt{insert \{}Expr_1\texttt{\} }((\texttt{as first}|\texttt{as last})\texttt{ into}|\texttt{before}|\texttt{after})\texttt{ \{}Expr_2\texttt{\}}]\!]}{\texttt{insert \{copy \{}[\![Expr_1]\!]\texttt{\}\} }((\texttt{as first}|\texttt{as last})?\texttt{ into}|\texttt{before}|\texttt{after})\texttt{ \{}[\![Expr_2]\!]\texttt{\}}}$$

Note the presence of an explicit copy operation, introduced during normalization to make sure that copying the input node is performed during the querying part of the processing, before update application. This avoids some classes of interactions between different updates, hence some update conflicts, that would otherwise happen if the copying were done at the atomic update level. For instance, consider the following query.

```
snap {
  let $x := <a/>
  return (insert <b/> into $x,
          insert <c/> into $y,
          $x)
}
```

Since the two insertions are enclosed in the same `snap` scope, they should not interact. However, assume that `copy`, instead of preceding update application, happened as part of insertion. If the two insertions are executed in the textual order, then they have no interaction, and *$x* has no *c* descendant. But if we first insert *c* into *$y*, then copy *$y*, and finally insert the copy, then *$x* ends with a *c* descendant. By copying everything in advance, this conflict is avoided. We believe this is the best choice, but one must be aware that this implies that the following piece of code is equivalent to the code above, hence does *not* insert a *c* descendant into *$x*.

```
snap {
  let $x := <a></a>
  return (insert <c/> into $y,
          insert <b/> into $x,
          $x)
}
```

Normalization of Replace

A `replace` expression with a `snap` modifier is normalized as a composition of `snap` and `replace`.

$$\frac{\mathbf{[}\texttt{snap replace } \{\mathit{Expr}_1\} \texttt{ with } \{\mathit{Expr}_2\}\mathbf{]}}{\texttt{snap } \{\mathbf{[}\texttt{replace } \{\mathit{Expr}_1\} \texttt{ with } \{\mathit{Expr}_2\}\mathbf{]}\}}$$

A `replace` expression is compiled into a core replace expression, where the value to be inserted is copied first.

$$\frac{\mathbf{[}\texttt{replace } \{\mathit{Expr}_1\} \texttt{ with } \{\mathit{Expr}_2\}\mathbf{]}}{\texttt{replace } \{\mathbf{[}\mathit{Expr}_1\mathbf{]}\} \texttt{ with } \{\texttt{copy } \{\mathit{Expr}_2\}\}}$$

Normalization of Rename

A `rename` expression with a `snap` modifier is normalized as a composition of `snap` and `rename`.

$$\frac{\mathbf{[}\texttt{snap rename } \{\mathit{Expr}_1\} \texttt{ to } \{\mathit{Expr}_2\}\mathbf{]}}{\texttt{snap } \{\mathbf{[}\texttt{rename } \{\mathit{Expr}_1\} \texttt{ to } \{\mathit{Expr}_2\}\mathbf{]}\}}$$

A `rename` expression is compiled into a core `rename` expression.

$$\frac{\mathbf{[}\texttt{rename } \{\mathit{Expr}_1\} \texttt{ to } \{\mathit{Expr}_2\}\mathbf{]}}{\texttt{rename } \{\mathbf{[}\mathit{Expr}_1\mathbf{]}\} \texttt{ to } \{\mathit{Expr}_2\}}$$

Normalization of Copy

A `copy` expression is normalized to a core `copy` expression.

$$\frac{\mathbf{[}\texttt{copy } \{\mathit{Expr}_1\}\mathbf{]}}{\texttt{copy } \{\mathbf{[}\mathit{Expr}_1\mathbf{]}\}}$$

Normalization of Snap

A `snap` expression is normalized into a core `snap` expression.

$$\frac{[\texttt{snap } \{\mathit{Expr}_1\}]}{\texttt{snap } \{[\mathit{Expr}_1]\}}$$

5.3 Dynamic semantics: the judgment

The semantics of XQuery 1.0 if formally described using inference rules notations [8].

The main judgment is the following:

$$\mathit{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{value}$$

which means: given the dynamic context *dynEnv*, the expression *Expr* yields the value *value*. To deal with delayed updates and side-effects, we extend it 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 atomic updates 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 atomic updates, 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.

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.

Evaluation order is irrelevant if the subexpressions contain no updates, or if they contain updates but no `snap`.[4] Consider for example the rule for the sequence constructor. The two versions below specify two different evaluation orders, left-to-right and right-to-left, respectively (in XQuery! we adopt the first one).

$$\frac{\begin{array}{c}\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}_1\ \Rightarrow\ \mathit{value}_1;\ \Delta_1;\ \mathit{store}_1\\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_2\ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_2\end{array}}{\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}_1, \mathit{Expr}_2\ \Rightarrow\ \mathit{value}_1, \mathit{value}_2;\ (\Delta_1, \Delta_2);\ \mathit{store}_2}$$

$$\frac{\begin{array}{c}\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}_2\ \Rightarrow\ \mathit{value}_2;\ \Delta_2;\ \mathit{store}_1\\ \mathit{store}_1;\ \mathit{dynEnv} \vdash \mathit{Expr}_1\ \Rightarrow\ \mathit{value}_1;\ \Delta_1;\ \mathit{store}_2\end{array}}{\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}_1, \mathit{Expr}_2\ \Rightarrow\ \mathit{value}_1, \mathit{value}_2;\ (\Delta_1, \Delta_2);\ \mathit{store}_2}$$

If the subexpressions do not apply updates, the two rules produce the same list of values and the same list of pending updates, *in the same order*, although they specify a different evaluation order. 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 (Section 6).

In the Appendixes we present the semantics of the most important core XQuery 1.0 expressions, each with an evaluation order specified. In the Appendixes we also present a different approach, where the evaluation order is left mostly unconstrained, as happens with XQuery 1.0.

[4] Evaluation order is actually relevant to decide which failures can be raised.

5.4 Dynamic semantics of the new operations

We have to define the semantics of `copy`, of the update operators, and of `snap`. `copy` just invokes the corresponding operation on the current store. The evaluation of an update operation produces an atomic update, which is added to the list of the pending atomic updates produced by the subexpressions, while `replace` produces *two* atomic updates, insertion and deletion.

The metavariables used in the result position are normative. This means that, if a judgment in the premise uses *node* in the result position, as in:

$$\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr} \ \Rightarrow\ \mathit{node};\ \Delta_1;\ \mathit{store}_1,$$

the judgment can only be applied if *Expr* evaluates to a value which is a node; the same is true for the metavariables node_i (for any i), *nodepar*, *nodepos*. The metavariable *nodeseq* ranges over node sequences, and *name* ranges over qnames.

We now present the semantics of the new operators.

Semantics of replace:

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

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 atomic updates 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`.

Semantics of delete:

A delete operator evaluates its argument and inserts an atomic update into Δ.

$$\frac{\begin{array}{c}\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr} \ \Rightarrow\ \mathit{node};\ \Delta_1;\ \mathit{store}_1\\ \Delta_2 = (\Delta_1, \mathrm{delete}\ \mathit{node})\end{array}}{\mathit{store}_0;\ \mathit{dynEnv} \vdash \texttt{delete \{}\mathit{Expr}\texttt{\}} \ \Rightarrow\ ();\ \Delta_2;\ \mathit{store}_1}$$

Semantics of insert:

Insert Location Judgments

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

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

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

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

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash \text{is\_first\_child}(node)\ \Rightarrow\ \text{true};\ store_0;\ () \\ store_0;\ dynEnv \vdash \text{parent}(node)\ \Rightarrow\ nodepar;\ store_0;\ () \end{array}}{store_0;\ dynEnv \vdash \texttt{before \{}node\texttt{\}}\ \Rightarrow\ (nodepar, nodepar);\ store_0;\ ()}$$

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash \text{is\_first\_child}(node)\ \Rightarrow\ \text{false};\ store_0;\ () \\ store_0;\ dynEnv \vdash \text{parent}(node)\ \Rightarrow\ nodepar;\ store_0;\ () \\ store_0;\ dynEnv \vdash \text{preceding\_sibling}(node)\ \Rightarrow\ nodepos;\ store_0;\ () \end{array}}{store_0;\ dynEnv \vdash \texttt{before \{}node\texttt{\}}\ \Rightarrow\ (nodepar, nodepos);\ store_0;\ ()}$$

Main Insert Judgment

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr_1\ \Rightarrow\ nodeseq;\ \Delta_1;\ store_1 \\ store_1;\ dynEnv \vdash Expr_2\ \Rightarrow\ node_2;\ \Delta_2;\ store_2 \\ store_2;\ dynEnv \vdash InsertLocation\ node_2\ \Rightarrow\ (nodepar, nodepos);\ ();\ store_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{insert}(nodeseq, nodepar, nodepos)) \end{array}}{store_0;\ dynEnv \vdash \texttt{insert \{}Expr_1\texttt{\}}\ InsertLocation\ \texttt{\{}Expr_2\texttt{\}}\ \Rightarrow\ ();\ \Delta_3;\ store_2}$$

Semantics of rename:

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr_1\ \Rightarrow\ node;\ \Delta_1;\ store_1 \\ store_1;\ dynEnv \vdash Expr_2\ \Rightarrow\ name;\ \Delta_2;\ store_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \text{rename}(node, name)) \end{array}}{store_0;\ dynEnv \vdash \texttt{rename \{}Expr_1\texttt{\} to \{}Expr_2\texttt{\}}\ \Rightarrow\ ();\ \Delta_3;\ store_2}$$

Semantics of copy:

$$\frac{\begin{array}{c} store_0;\ dynEnv \vdash Expr\ \Rightarrow\ node_1;\ \Delta_1;\ store_1 \\ (store_2, node_2) = \text{deepcopy}(store_1, node_1) \end{array}}{store_0;\ dynEnv \vdash \texttt{copy \{}Expr\texttt{\}}\ \Rightarrow\ node_2;\ store_2;\ ()}$$

Semantics of `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.

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

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 evaluated 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 describe in Section 6. 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.

6 Implementation

In this section, we describe the implementation of XQuery!, focusing on the `snap` operator and on the three semantics for update application described in Section 4. The nondeterministic and conflict-detection semantics only differ in the fact that the first is missing the conflict-detection phase; we call them the *unordered semantics*. They are easier to implement, as they do not require the processor to keep track of update order.

Our implementation is done on top of the Galax XQuery engine [24, 23], which includes an optimizer based on a variant of a standard nested-relational algebra. The implementation of the update extension works over both XML in main-memory, and the Jungle persistent store [30].

6.1 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 an XML algebra, optimization and evaluation.

The XQuery core is extended as described in Section 5. A number of 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, a large number of rewritings immediately apply.

In the rest of this section, we focus on the changes required to the run-time and to the optimizer.

6.2 Changes to the data model

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.

6.3 Nondeterministic and conflict-detection semantics

The nondeterministic and conflict-detection semantics are both independent on the actual order of the atomic updates collected in a `snap` scope. Their implementation is based on a stack of *delta bags*, which are unordered sequences of atomic updates, implemented as lists whose internal order is ignored. When entering a snap expression, an empty delta bag is pushed on the top of the stack. The invocation of an update operation generates a delta, which represents an atomic update as a tuple containing the operation name and the node ids of the parameters. The delta is then inserted into the delta bag currently at the top of the stack. When exiting a `snap`, the top-most delta bag is popped from the stack, reordered so that insertions precede deletions, 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. Specifically, since the order inside a delta bag is irrelevant, there is no need to actually enforce the left-to-right evaluation order inside an innermost `snap`.

6.4 Ordered semantic

In the ordered semantic, the main challenge is to preserve the left-to-right order of the atomic updates dictated by the formal semantic. Since the actual evaluation order may differ from this, we introduce a data structure to keep track of the update order dynamically. This data structure is a recursive tree like structure, which we call a *nested delta list*. This is most easily explained by the following snippet of ML-style code.

```
type nested_delta_list =
  |  DeltaList of nested_delta_list list
  |  Concrete_Update of atomic_update
```

To illustrate why it is necessary to have a nested list instead of a simple list, consider the following simple query:

```
for $x in (1,2, insert { text { 3 } } as last into { $log },
             4, insert { text { 5 } } as last into { $log }, 6 )
return foo($x)
```

The implementation needs to ensure that the insert of `3` and `5` come before any insert that comes from `foo`. However, since the outer branch contains no `snap`, the compiler may decide to evaluate the FLWOR expression in a pipelined fashion, hence interleaving the evaluation of the internal sequence and of `foo($x)`. To preserve the update order, a nested list is created inside the current update list, and the `insert` of `3` is put there. Insertions generated by `foo` will be added after the nested list, while the insertions generated by the inner sequence will be added inside the nested list, after the `insert` of 3. Hence, by flattening this structure, updates are read in the correct order. Since XQuery expressions can be arbitrarily nested, the corresponding data structure must be nested as well.

6.5 Non-interleaved evaluation

In the previous section we have seen that the interleaved evaluation which is implied by a cursor-iterator lazy implementation is not a problem in a FLWOR expression that invokes no `snap`. However, an internal `snap` may force the materialization of the sequence which *for* iterates upon, either to ensure that the *return* expression sees all the effects of the *for* sequence expression, or to ensure that the *for* expression sees none of the effects of the *return* expression. Consider for example the following expression:

```
1. for $x in
2.   snap nondeterministic {
3.     for $y in $list
4.     where condition($y)
5.     return $y//a, insert {<y>{$y}<y/>} as last into {$log}
6.  }
7. return $x//b, insert {<x ycount="{count($log)"}>{$x}<x/>}
8.                as last into {$log}
```

The language semantics imposes that every *y* element precedes every *x* element in the log (lines 5, 7), and that *ycount* is always the size of the log up to the latest previous *y* element (line 7). While the *nested delta list* technique can solve the order problem, the dependency introduced by the `snap` (line 2) and the `count` (line 7) requires the materialization of the result of the snap expression.

The problem is orthogonal to the presence of the `nondeterministic` keyword, which we put here in order to allow multiple insertions under the same parent.

6.6 Changes to the optimizer

Again the non-deterministic and conflict-detection semantics differ only in the conflict-detection phase, and as such impact the logical optimizer in the same way. The judgments for the ordered case are often similar, apart from cases when join-reordering optimizations are used.

Galax uses a rule-based approach in several phases of logical optimization. Most rewrite rules required some modification. Details of these modifications and correctness proofs are beyond the scope of this work. The most common modifications were those needed to ensure that expressions are evaluated with the correct cardinality, or that ordering constraints are respected.

We come back to the example given in Section 2.5. Here, for each person, we insert all his n purchases into *$purchasers*, and return an item which indicates its name and the value of n.

```
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>
```

Our goal is here to recover a join and group-by plan. The syntax of the query plan below is a simplified version of that detailed in [24]. Naively evaluated, this query has complexity O($|person| * |closed\_auction|$). Using the plan below with a typed hash join, we can recover the join complexity of O($|person| + |closed\_auction| + |matches|$), which is quite a substantial real world saving.

```
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 }
    )
   )
}
```

If the insert had a `snap` immediately around it, join recovering would be difficult. One key property of the hash join is the ability to materialize at least one branch; we call this property "independence". With `snap insert`, the materialized branch may be changed by the `snap`, which would violate independence. This independence property is easily verified if no `snap` is found inside the query fragment; the outer `snap` creates no problem.

7 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. Some versions of the language we describe here have been discussed in the Update Task Force of the W3C XQuery Working group, although the proposal of `snap` free nesting, the formalization of the semantics, and the implementation, are authors contributions. We want to thank Dana Florescu, Don Chamberlin, Ioana Manolescu, Kristoffer Rose, Mukund Raghavachari, Rajesh Bordawekar, and Michael Benedikt for their feedback on earlier versions of this paper. We thank 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] Michael Benedikt and Irini Fundulaki. XML subtree queries: Specification and composition. In *Tenth International Symposium on Database Programming Languages (DBPL)*, 2005.

[3] Bard Bloom. Lopsided little languages: Experience with XQuery. In *XIME-P'05*, 2005.

[4] 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.

[5] Don Chamberlin. Communication regarding an update proposal. W3C XML Query Update Task Force, May 2005.

[6] Don Chamberlin and Jonathan Robie. XQuery update facility requirements. W3C Working Draft, June 2005.

[7] XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, April 2005.

[8] 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`.

[9] Daniela Florescu et al. Communication regarding an XQuery update facility. W3C XML Query Working Group, July 2005.

[10] Don Chamberlin et al. Communication regarding updates for XQuery. W3C XML Query Working Group, October 2002.

[11] Mary Fernández, Jerôme Siméon, and Philip Wadler. *XQuery from the experts*, chapter Introduction to the Formal Semantics. Addison Wesley, 2004.

[12] Daniela Florescu. Communication regarding update grammar. W3C XML Query Update Task Force, April 2005.

[13] 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.

[14] Jim Gray. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, 1992.

[15] 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.

[16] 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.

[17] Andreas Laux and Lars Matin. http://www.xmldb.org/xupdate, October 2000.

[18] Patrick Lehti. Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit, 2001.

[19] 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.

[20] Nancy A. Lynch and Michael Merritt. Introduction to the theory of nested transactions. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 278–305, 1986.

[21] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. MIT Press, 1997.

[22] Nicola Onose and Jérôme Siméon. XQuery at your Web service. In *Proceedings of International World Wide Web Conference*, New York, NY, May 2004.

[23] Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. Technical report, AT&T Labs Research, 2005.

[24] Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta,GA, April 2006.

[25] Michael Rys. Proposal for an XML data modification language, version 3, May 2002. Microsoft Corp., Redmond, WA.

[26] Arnaud Sahuguet and Bogdan Alexe. Sub-document queries over XML with XSQuirrel. In *Proceedings of International World Wide Web Conference*, pages 268–277, 2005.

[27] 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.

[28] Gargi M. Sur, Joachim Hammer, and Jérôme Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[29] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *SIGMOD*, 2001.

[30] Avinash Vyas, Mary F. Fernandez, and Jérôme Siméon. The simplest XML storage manager ever. In *XIME-P 2004*, pages 37–42, Paris, France, June 2004.

A Open Issues

In this section, we list open issues and in some cases discuss some possible directions to address those issues. Most of those issues typically relate to extensions to the framework proposed above.

A.1 Evaluation order

XQuery (notice, no bang here) leaves a great freedom in evaluation order, which is quite important to allow for some database optimizations.

First of all, XQuery allows any operator to evaluate the operands in any order, with the only exception of conditional and typeswitch which cannot start evaluating a branch until they know which branch should be selected. We discuss here a variant of XQuery!, lazy-XQuery!, where we keep this approach and only add a new operator *e1;e2*, which cannot start evaluating *e2* until *e1* has been fully evaluated.

In XQuery, evaluation can be partial: evaluation can stop when the final outcome is determined modulo failure, that is, when the final outcome is either a failure or a value V, and the processor can return V in this case.[5] We adopt the same rule for lazy-XQuery!. In this case, however, the "final outcome" also includes any effect on the store and any pending update. In other words, the processor may skip a subexpression that, apart from possibly failing, would be otherwise irrelevant for the returned value *and* for the state of store.

Lazy-XQuery! is essentially a conservative extension of XQuery, but its formalization requires a drastic change to the formal semantic style (the same change should be applied to XQuery formal specification if one wanted to formally specify failure handling, hence evaluation order).

To formalize lazy-XQuery! evaluation order, we believe one must adopt a small-step semantic approach, where the typical judgement is the following, which specifies that, when the store, dynamic environment, and pending update list are *store*, *dynEnv*, Δ, the processor may reduce *Expr* to *Expr*$'$ and update the other components to *store*$'$, Δ'.

$$\mathit{store}; \mathit{dynEnv}; \mathit{Expr}; \Delta \ \Rightarrow\ \mathit{store}'; \mathit{Expr}'; \Delta'$$

This style is verbose but very easy to read. For example, the rule for `snap` is the following. (Metavariables are normative; this means that *value* below is only matched by a value.)

$$\frac{\begin{array}{c}\mathit{store}; \mathit{dynEnv}; \mathit{Expr}; () \ \Rightarrow\ \mathit{store}'; \mathit{value}; \Delta' \\ \mathit{store}'' = \text{apply } \Delta' \text{ to } \mathit{store}'\end{array}}{\mathit{store}; \mathit{dynEnv}; \texttt{snap}\{\mathit{Expr}\}; \Delta \ \Rightarrow\ \mathit{store}''; \mathit{value}; \Delta}$$

The rule specifies that inside a `snap` scope the outer pending updates are not seen. The operator fully evaluates its body and, at the end of the evaluation, it applies the collected pending updates to the store and returns the computed value.

The rules below explain the difference between "," and ";": evaluation of the second subexpression is allowed in ";" when the first is fully evaluated, while the second subexpression can be freely evaluated in the "," case. The difference is between the second and the fourth rule.

[5]Quoting [4]: At an intermediate stage during evaluation of the sequence, some of its items will be known and others will be unknown. If, at such an intermediate stage of evaluation, a processor is able to establish that there are only two possible outcomes of evaluating Q, namely the value V or an error, then the processor may deliver the result V without evaluating further items in the operand E.

$$\frac{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}_1; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'_1; \Delta'}{\mathit{store}; \mathit{dynEnv}; (\mathit{Expr}_1, \mathit{Expr}_2); \Delta \Rightarrow \mathit{store}'; (\mathit{Expr}'_1, \mathit{Expr}_2); \Delta'}$$

$$\frac{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}_2; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'_2; \Delta'}{\mathit{store}; \mathit{dynEnv}; (\mathit{Expr}_1, \mathit{Expr}_2); \Delta \Rightarrow \mathit{store}'; (\mathit{Expr}_1, \mathit{Expr}'_2); \Delta'}$$

$$\frac{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}_1; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'_1; \Delta'}{\mathit{store}; \mathit{dynEnv}; (\mathit{Expr}_1; \mathit{Expr}_2); \Delta \Rightarrow \mathit{store}'; (\mathit{Expr}'_1; \mathit{Expr}_2); \Delta'}$$

$$\frac{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}_2; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'_2; \Delta'}{\mathit{store}; \mathit{dynEnv}; (\mathit{value}; \mathit{Expr}_2); \Delta \Rightarrow \mathit{store}'; (\mathit{value}; \mathit{Expr}'_2); \Delta'}$$

These rules allow evaluation of $(\mathit{Expr}_1, \mathit{Expr}_2)$ to proceed by interleaving evaluation of Expr_1 and evaluation of Expr_2; of course, they must be completed by the following general rule.

$$\frac{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'; \Delta' \qquad \mathit{store}'; \mathit{dynEnv}; \mathit{Expr}'; \Delta' \Rightarrow \mathit{store}''; \mathit{Expr}''; \Delta''}{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}; \Delta \Rightarrow \mathit{store}''; \mathit{Expr}''; \Delta''}$$

We present now the rule to evaluate *for-return*. For typographic reasons, we use $\mathit{Expr} \rightarrow \mathit{Expr}'$ to abbreviate an evaluation rule which ignores all the other metavariables. i.e. to abbreviate the rule:

$$\frac{}{\mathit{store}; \mathit{dynEnv}; \mathit{Expr}; \Delta \Rightarrow \mathit{store}; \mathit{Expr}'; \Delta}$$

Here are the rules for *for*. The combination of the first two rules allows the processor to evaluate the return clause for each item in the binding clause as soon as this item is ready, without waiting for the full binding sequence to be evaluated. The sequence order of the result is preserved, but the evaluation order is unconstrained.

$$\begin{array}{l}\texttt{for } \$x \texttt{ in } (\mathit{Expr}_1, \mathit{Expr}_2) \texttt{ return } \mathit{Expr} \\ \qquad \rightarrow (\texttt{for } \$x \texttt{ in } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}), (\texttt{for } \$x \texttt{ in } \mathit{Expr}_2 \texttt{ return } \mathit{Expr})\end{array}$$

$$\frac{\mathit{store}; (\mathit{dynEnv} + \$x \Rightarrow \mathit{item}); \mathit{Expr}; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'; \Delta'}{\mathit{store}; \mathit{dynEnv}; \texttt{for } \$x \texttt{ in } \mathit{item} \texttt{ return } \mathit{Expr}; \Delta \Rightarrow \mathit{store}'; \mathit{Expr}'; \Delta'}$$

$$\texttt{for } \$x \texttt{ in } () \texttt{ return } \mathit{Expr} \rightarrow ()$$

A.2 Dealing with validated documents

One of the most difficult, and unexplored, question, relating to XML updates, is how to deal with validated document. There are two important classes of constraints that must be verified on documents during query processing:

Data model constraints Some constraints must hold on the XQuery data model. Those are strong constraints which must be always verified during evaluation. Most notably, the type annotation on each element, and attribute node, must be consistent with the content of the node. As a result, XML update operations must make sure they always maintain those constraints. Other issues have to do with the constraint that text nodes contained in other nodes must have non empty content and cannot be followed by a text node sibling. This is much simpler to deal with.

Validation constraints Other constraints have to do with validity of the document. In the presence of updates, it is easy to change the document in a way that changes the validity property of some nodes. Users may want to make sure that their updates always preserve the validity according to the original schema. This is a constraint which may be temporarily violated for the needs of certain applications.

In the presence of XML update operations, several questions relating to XML validation can be raised: (1) what language operations are necessary to support applications dealing with validated documents? (2) how do those operations make sure to address the two classes of constraints mentioned above? (3) Assuming the presence of an in-place validation operation, is it possible to support static typing?

For, the first two questions, one possible approach is outlined below, based on the used of a 'revalidate' expression which performs in-place revalidation of a given tree. The most important remark here is to note that revalidation is treated as any other atomic updates. The first step is to extend the set of atomic updates presented in Section 4 with a revalidate operation, as follows:

- revalidate node

  This operation revalidates a node and its subtree. Node identity is unaffected by this operator but its type annotation, typed value, etc. are changed. When validate is applied, all ancestors of the node have their types changed to xs:anyType.

The second step is to extend the grammar and core grammar with the corresponding revalidate expression, as follows.

Expr::=	...	
|	*RevalidateExpr*	(: Atomic validate inplace :)
RevalidateExpr :=	"`snap`"? "`revalidate`" "{" *Expr* "}"	
CExpr ::=	...	(: All existing XQuery core expressions :)
|	*CRevalidateExpr*	(: Validate inplace :)
CRevalidateExpr :=	"`snap`"? "`revalidate`" "{" *CExpr* "}"	

The third step is to add the corresponding normalization rule from the language grammar to the language core grammar.

Normalization of Revalidate

$$\frac{[[\texttt{revalidate }\{\mathit{Expr}_1\}]]}{\texttt{revalidate }\{[[\mathit{Expr}_1]]\}}$$

$$\frac{[[\,\texttt{snap revalidate }\{\mathit{Expr}_1\}]]}{\texttt{snap }\{\texttt{revalidate }\{\texttt{copy }\{[[\mathit{Expr}_1]]\}\}\}}$$

Finally, the last step is to provide the semantics of revalidate over an existing data model tree.

Semantics of revalidate:

$$\frac{\begin{array}{c}\mathit{store}_0;\ \mathit{dynEnv} \vdash \mathit{Expr}\ \Rightarrow\ \mathit{node};\ \mathit{store}_1;\ \Delta_1\\ \Delta_2 = (\Delta_1, \text{revalidate } \mathit{node})\end{array}}{\mathit{store}_0;\ \mathit{dynEnv} \vdash \texttt{revalidate }\{\mathit{Expr}\}\ \Rightarrow\ ();\ \mathit{store}_1;\ \Delta_2}$$

This approach is attractive since it treats revalidation as any other atomic update, and gives the programmer much controll about validation time. However, a programmer may desire writing code that tentatively applies some updates, try to validate the result, and leaves data unchanged in case validation fails. One could add a modifier to the snap keyword, so that `snap ifvalid {Expr}` collects all updates generated by `Expr`, tentatively applies them, but rolls back the effects in case any of the modified items cannot be revalidated.

This effect cannot be achieved with the operators we propose, since XQuery has no exception handling mechanism. We propose such a mechanism in the next subsection.

A.3 Exception handling

We propose a standard `try` {*Expr*$_1$} `otherwise` {*Expr*$_2$} construct which executes *Expr*$_1$ but passes the control to *Expr*$_2$ if *Expr*$_1$ fails. Every pending update that had been collected but not yet applied at failure time will be discarded. If failure happens at `snap` time, i.e. if the application of a composite update fails, every atomic update that was part of the failed composite update is discarded.

Finally, we have to discuss the effect of `snap` expressions which had been completed inside the `try` scope but before the failure. Consider the following piece of code.

```
try { snap { update1 },
      update2,
      error()
    }
catch { E2 }
```

While *update2* is discarded, as discussed, it is not clear whether *update1* should be undone as well.

We first observe that both choices are equally easy to formalize. The following rule dictates that `update1` is discarded, by evaluating $Expr_2$ in *store*. We call it the “rollback rule”.

$$\frac{\begin{array}{c} store;\ dynEnv \vdash Expr_1 \ \Rightarrow\ fail;\ \Delta_1;\ store_1 \\ store;\ dynEnv \vdash Expr_2 \ \Rightarrow\ value;\ \Delta_2;\ store_2 \end{array}}{store;\ dynEnv \vdash \texttt{try}\ \{Expr_1\}\ \texttt{otherwise}\ \{Expr_2\} \Rightarrow value;\ \Delta_2;\ store_2}$$

The next rule, instead, only discards update2, since it uses $store_1$ to evaluate $Expr_2$. We call it the “discard rule”.

$$\frac{\begin{array}{c} store;\ dynEnv \vdash Expr_1 \ \Rightarrow\ fail;\ \Delta_1;\ store_1 \\ store_1;\ dynEnv \vdash Expr_2 \ \Rightarrow\ value;\ \Delta_2;\ store_2 \end{array}}{store;\ dynEnv \vdash \texttt{try}\ \{Expr_1\}\ \texttt{otherwise}\ \{Expr_2\} \Rightarrow value;\ \Delta_2;\ store_2}$$

The discard rule is easier to implement, but the rollback version seems more useful and natural.

Assume, for example, that a programmer wants to perform a set of updates, validate the result, and rollback the updates in case validation fails. With the rollback semantics, this can be obtained by the following piece of code, which either returns `$x` as it was or it returns it updated and validated. Revalidation must be performed after the first snap is closed, otherwise it would not see the effect of the updates, but its failure must imply the rollback of the previously snapped updates which took place in the scope of `try`, in order to reach the desired effect.

```
try { snap { ...updates to $x...},
      snap { revalidate { $x } }
    }
otherwise { $x }
```

A.4 Delete: detach vs. erase semantics

The next open issue relates to the definition of the semantics for the delete operation. As written, the semantics do not actually delete nodes, but merely *detach* the given node from its parent. This brings a very clean semantics, notably in the presence of variables where some deleted nodes may still be reachable from an actual variable. For instance, consider the following query.

```
let $featured := auction()//item[@featured] return
(snap delete auction//*[@featured][last()],
 count($featured))
```

One key question is how the variable is affected by the deletion of one of the nodes it contains. Following the *detach* semantics, those nodes are still reachable and can be counted. This may not be the prefered semantics in the case the nodes are actually stored in a persistant repository and if the user expects the nodes to actually disappear from that store. An alternative is an *erase* semantics which effectively removes the nodes from the store. This semantics requires some book-keeping in the formal definition of the store to ’mark’ the nodes which are effectively deleted. A possible drawback of that semantics though is that it is unclear what to do with nodes pointed from a given variable: do they actually disappear, is an error raised if the variable is accessed, etc. Note that the two semantics coincide in the case where deleted nodes are not reachable from any variable.

The following sketches the semantics for a delete operation which supports the erase semantics.

- delete node (erase semantics)

  delete a node. Accessing a node in a deleted tree is an error, however an insert into a deleted node area within a snap is ignored. This will be discussed further in the section on conflicts.

With that proposed semantics, the following changes to the notion of conflicts must also be made. First we need to define a notion of delete conflict.

Two updates u_1, u_2 are in *delete conflict* if we u_1 is a delete and u_2 access a portion of the subtree deleted by u_1.

Ordered Deterministic In the ordered semantics, a delete conflict can occur.

Unordered Nondeterministic There are also no delete conflicts, since we can order any deletes at the end of the update sequence.

Unordered Determinsitic Remains unchanged.

Semantics of delete (erase semantics):

Finally, here is the inference rule for the delete with erase semantics.

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

B Formal Definition of the Data Model

B.1 The Store

A store S is a quadruple $(N, E, \textit{ord}, F = (\text{node-kind}_F, \text{node-name}_F, \text{content}_F))$, with the constraints specified below. The *respect XQDM* predicate is defined later. We only consider three kinds here, to exemplify the techniques. In the rest of the section we will just consider the element kind. $A \rightharpoonup B$ is the set of partial functions from A to B. $\mathcal{N}$ is the infinite set of all possible node ids.

Constraints on a store

$$\begin{array}{l} N \subseteq \mathcal{N} \quad N \text{ is finite} \\ E \subseteq N \times N \\ \textit{ord} \subseteq N \times N \text{ is a strict total order on } N \\ \text{node-kind}_F : N \to \{\text{element}, \text{attribute}, \text{text}\} \\ \text{node-name}_F : N \rightharpoonup \text{QNames} \\ \text{content}_F : N \rightharpoonup \text{Strings} \\ E, \textit{ord}, F \text{ respect XQDM} \end{array}$$

To formalize *respect XQDM* we need a relation $E\text{-rel}(m, n)$ that specifies that m and n are part of a same document.[6] $\text{imm-sib}_{E,ord}(m, n)$ means that m and n are siblings, and m immediately precedes n. We write $E(m, n)$ to signify $(m, n) \in E$, and similarly for $\textit{ord}(m, n)$.

$$\begin{array}{lll} E^*(m,n) & \Leftrightarrow_{def} & m = n \ \vee\ (\exists m'.\ E(m,m') \ \wedge\ E^*(m',n)) \\ E\text{-rel}(m,m') & \Leftrightarrow_{def} & \exists n.\ E^*(n,m) \ \wedge\ E^*(n,m') \\ \text{imm-sib}_{E,ord}(m,m') & \Leftrightarrow_{def} & \exists n.\ E(n,m) \ \wedge\ E(n,m') \ \wedge\ \neg\exists m''.\ (\textit{ord}(m,m'') \ \wedge\ \textit{ord}(m'',m')) \end{array}$$

We can now define the *respect XQDM* predicate, as the conjunction of the following three sentences. N_{el}, N_{at}, and N_{tx} denote the subsets of N where node-kind_F evaluates to, respectively, element, attribute, text.

[6] We should actually speak of document *fragments*, since we have no document node here.

1. Order respects document order, different documents are not mixed together, attributes precede element siblings:

$$\begin{array}{l} E(m,m') \;\Rightarrow\; \mathit{ord}(m,n) \\ E\text{-rel}(m,m') \;\wedge\; E\text{-rel}(n,n') \;\wedge\; \neg E\text{-rel}(m,n) \;\wedge\; \mathit{ord}(m,n) \;\Rightarrow\; \mathit{ord}(m',n') \\ E(m,n) \;\wedge\; E(m,n') \;\wedge\; n \in N_{at} \;\wedge\; n' \in N_{el} \;\Rightarrow\; \mathit{ord}(n,n') \end{array}$$

2. Constraints on the roles of different node kinds:

$$\begin{array}{l} E \subseteq N_{el} \times N \\ \text{node-name}_F : (N_{el} \cup N_{at}) \rightarrow \text{QNames} \\ \text{content}_F : N_{at} \rightarrow \text{Strings} \end{array}$$

3. Constraints the text content of documents:

$$\begin{array}{l} \text{imm-sib}_{E,ord}(m,m') \;\wedge\; m \in N_{tx} \;\Rightarrow\; m' \notin N_{tx} \\ E(m,n) \;\wedge\; n \in N_{tx} \;\Rightarrow\; \text{content}_F(n) \neq \text{""} \end{array}$$

B.2 Store Equivalence

Two stores (N, E, ord, F) and $(N', E', \mathit{ord}', F')$ are equivalent if there exists a permutation σ of $\mathcal{N}$ such that:

$$\begin{array}{l} N = \sigma(N') \\ E'(m,n) \;\Leftrightarrow\; E(\sigma(m),\sigma(n)) \\ \mathit{ord}'(m,n) \;\Leftrightarrow\; \mathit{ord}(\sigma(m),\sigma(n)) \\ \text{content}_{F'} = \text{content}_F \circ \sigma \\ \text{node-name}_{F'} = \text{node-name}_F \circ \sigma \\ \text{node-kind}_{F'} = \text{node-kind}_F \circ \sigma \end{array}$$

Every store operation that increases N will only be defined modulo store equivalence.

B.3 Accessors

We formalize each accessor as a function that is applied to a store $S = (N, E, \mathit{ord}, F)$ and to other parameters, and returns a value which is valid in S.

$$\begin{array}{llll} \text{dm:nodekind}((N,E,\mathit{ord},F),n) & =_{def} & \text{node-kind}_F(n) & \\ \text{dm:children}((N,E,\mathit{ord},F),n) & =_{def} & [n' \mid E(n,n') \;\wedge\; n' \in (N_{el} \cup N_{tx})] & \text{ordered by } \mathit{ord} \\ \text{dm:attributes}((N,E,\mathit{ord},F),n) & =_{def} & [n' \mid E(n,n') \;\wedge\; n' \in N_{at}] & \text{ordered by } \mathit{ord} \\ \text{dm:parent}((N,E,\mathit{ord},F),n) & =_{def} & [n' \mid E(n',n)] & \\ \text{allchildren}((N,E,\mathit{ord},F),n) & =_{def} & [n' \mid E(n,n') \;\wedge\; n' \in N] & \text{ordered by } \mathit{ord} \end{array}$$

B.4 Building new trees

We define two mutually recursive constructors, newnode(store, $\mathit{children}$, $\mathit{F\text{-}value}$) and deepcopy(store, node), both returning a pair $(\mathit{store}', \mathit{node}')$.

- newnode((N, E, ord, F), $\mathit{children}$, $\mathit{F\text{-}value}$) allocates a new node *node*, whose node-kind, node-name, and content are specified by the tuple *F-value*, performs a deep copy of the nodes in the *children* list, extends E so that *node* is the parent of these new nodes, and extends *ord* according to the document order rules. It returns the extended store and the new node.

- deepcopy((N, E, ord, F), node) is defined as

$$\begin{array}{lll} \text{deepcopy}((N,E,\mathit{ord},F),\mathit{node}) = \text{newnode} & (& (N,E,\mathit{ord},F), \\ & & \text{allchildren}((N,E,\mathit{ord},F),\mathit{node}), \\ & & (\text{node-kind}_F(\mathit{node}), \text{node-name}_F(\mathit{node}), \text{content}_F(\mathit{node})) \\ &) & \end{array}$$

We also define the newelement(store, $\mathit{children}$, name) constructor as

$$\text{newelement}(\mathit{store},\mathit{children},\mathit{name}) = \text{newnode}(\mathit{store},\mathit{children},(\text{element},\mathit{name},\uparrow))$$

C Language Semantics

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

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

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

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

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

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

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

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

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

Figure 3: XQuery! Semantics of Non-Update Operations