Recursion in XQuery: Put Your Distributivity Safety Belt On

Loredana Afanasiev
ISLA, University of Amsterdam
Amsterdam, The Netherlands
lafanasi@science.uva.nl

Torsten Grust
WSI, Universität Tübingen
Tübingen, Germany
torsten.grust@uni-tuebingen.de

Maarten Marx
ISLA, University of Amsterdam
Amsterdam, The Netherlands
marx@science.uva.nl

Jan Rittinger
WSI, Universität Tübingen
Tübingen, Germany
jan.rittinger@uni-tuebingen.de

Jens Teubner
Systems Group, ETH Zurich
Zurich, Switzerland
jens.teubner@inf.ethz.ch

ABSTRACT

We introduce a controlled form of recursion in XQuery, an *inflationary fixed point operator*, familiar from the context of relational databases. This operator imposes restrictions on the expressible types of recursion, but it is sufficiently versatile to capture a wide range of interesting use cases, including Regular XPath and its core transitive closure operator.

While the optimization of general user-defined recursive functions in XQuery appears elusive, we describe how inflationary fixed points can be efficiently evaluated, provided that the recursive XQuery expressions are *distributive*. We test distributivity syntactically and algebraically, and provide experimental evidence that XQuery processors can benefit substantially from this mode of evaluation.

1. INTRODUCTION

The backbone of the XML data model, namely *ordered, unranked trees*, is inherently recursive and it is natural to equip the associated languages with constructs that can recursively query such structures. In XQuery [6], recursion can be achieved only via *recursive user-defined functions*—a construct that admits *arbitrary* types of recursion and largely evades optimization approaches beyond "procedural" improvements like tail-recursion elimination or unfolding.

In this paper, we explore a controlled form of recursion in XQuery, the *inflationary fixed point operator*, familiar in the context of relational databases [1]. While less expressive than user-defined functions, the new operator embraces a family of widespread use cases of recursion, including forms of structural recursion and the pervasive *transitive closure* operator (in particular, it captures *Regular XPath* [29]). Most importantly, this operator admits an algebraic counterpart and systematic optimizations at the algebraic level. We present one such optimization.

The DTD of Figure 1 (taken from [25]) describes recur-

```
<!ELEMENT curriculum (course)*>
<!ELEMENT course prerequisites>
<!ATTLIST course code ID #REQUIRED>
<!ELEMENT prerequisites (pre_code)*>
<!ELEMENT pre_code #PCDATA>
```

Figure 1: Curriculum data (simplified DTD).

```
1   declare function rec_body ($cs) as node()*
2   { $cs/id (./prerequisites/pre_code) };
3  ┌---------------------------------------
4  ¦ declare function fix ($x) as node()*
5  ¦ { let $res := rec_body ($x)
6  ¦   return if (empty ($x except $res))
7  ¦          then $res
8  ¦          else fix ($res union $x)
9  ¦ };
10 └---------------------------------------
11  let $seed := doc ("curriculum.xml")
12                     //course[@code="c1"]
13  return fix (rec_body ($seed))
```

Figure 2: Prerequisites for course "c1" (dashed box marks the fixed point computation).

sive curriculum data, including courses, their lists of prerequisite courses, the prerequisites of the latter, and so on. The XQuery expression of Figure 2 recursively computes all prerequisite courses, direct or indirect, of the course coded with `"c1"` on an instance document `"curriculum.xml"`. The compilation is seeded by the `course` element node with code `"c1"`. For a given sequence `$x` of `course` nodes, function `fix (·)` calls `rec_body (·)` on `$x` to find their direct prerequisites. While new nodes are encountered, `fix (·)` recursively calls itself on the accumulated `course` node sequence. (This query is not expressible in XPath 2.0.) Note that `fix (·)` implements a generic inflationary fixed point computation: only the *seed* (`$seed := ···`) and the *body* (`rec_body (·)`) are specific to the curriculum problem. This motivates us to introduce a syntactic form for this pattern of computation. Unlike in the case of user-defined function, this account of recursion puts the query processor in control in choosing the evaluation strategy. In Section 2, we define the syntax and the semantics of the fixed point operation in XQuery context.

In Section 3, we discuss two algorithms to compute inflationary fixed points, borrowed from the relational world, *Naïve* and *Delta*. *Naïve* is a direct implementation of the fixed point semantics. *Delta* applies a divide-and-conquer evaluation strategy and it is more efficient, but not always

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

correct in the settings of XQuery. Provided that the *body* of the recursion exhibits a *distributivity* property, *Delta* can be safely applied to implement the fixed point computation. In Section 4, we define this property and we show its benefits in terms of correctness of *Delta* and in terms of the relation between the transitive closure and inflationary fixed point operators in the context of XQuery.

Distributivity can be efficiently tested on a syntactical level—a non-invasive approach that can easily be realized on top of existing XQuery processors. In Section 5, we present a syntactic fragment of XQuery that guarantees distributivity. Further, if we adopt a relational view of the XQuery semantics (as in [16, 17]), distributivity can be elegantly and uniformly tested on the familiar algebraic level. In Section 6, we present the algebraic counterpart of the inflationary fixed point operator on top of a relational algebra and we show how to test distributivity. We implement this approach in *MonetDB/XQuery* [7], an open-source XQuery processor.

Compliance with the restriction that distributivity imposes on fixed point expressions is rewarded by significant query runtime savings. In Section 7, we illustrate the effect for the XQuery processors *MonetDB/XQuery* and *Saxon* [22]. Though we mainly speak about fixed point computation in this work, we expect similar performance advantages if an XQuery processor takes distributivity as an indication to allow parallel processing of XQuery (sub)expressions.

In Section 8, we stop by related work on recursion on the XQuery as well as the relational side of the fence, and finally wrap up in Section 9.

2. DEFINING THE FIXED POINT OPERATION IN XQUERY

The subsequent discussion will revolve around the recursion pattern embodied by function `fix(·)` of Figure 2, known as the *inflationary fixed point operator (IFP)* [1]. We will introduce a new syntactic form to accommodate this operator on the XQuery language level and explore its semantics and applications.

In the following, we regard an XQuery expression e_1 containing a free variable `$x` as a function of `$x`, denoted by e_1(`$x`). We write $e_1(e_2)$ to denote $e_1[e_2/\$x]$, *i.e.*, the uniform replacement of all free occurrences of `$x` in e_1 by the value of e_2. We write $e_1(X)$ to denote the result of e_1(`$x`), evaluated on some given document, when `$x` is bound to the sequence of items X. It is always clear from the context which free variable we consider. The function $fv(e)$ returns the set of free variables of expression e.

Further, we introduce *set-equality* ($\stackrel{s}{=}$), a relaxed notion of equality for XQuery item sequences that disregards duplicate items and order, *e.g.*, `(1,"a")` $\stackrel{s}{=}$ `("a",1,1)`. For X_1, X_2 sequences of type `node()*`, we define

$$X_1 \stackrel{s}{=} X_2 \quad \Leftrightarrow \quad \texttt{fs:ddo}(X_1) = \texttt{fs:ddo}(X_2)\ .^1 \qquad \text{(SetEq)}$$

To streamline the discussion, we *only* consider XQuery expressions and sequences of type `node()*` in the following.[2]

[1] `fs:ddo(·)` abbreviates `fs:distinct-doc-order(·)` a function defined in the XQuery Formal Semantics [8].

[2] An extension of our definitions and results to general sequences of type `item()*` is possible but requires the replacement of XQuery's node set operations that we use (`fs:ddo(·)`, `union` and `except`) with the corresponding operations on sequences of items.

Definition 1. **Inflationary Fixed Point.** Given two XQuery expressions e_{seed} and e_{body}(`$x`), we define the *inflationary fixed point of* e_{body}(`$x`) *seeded by* e_{seed} as the sequence res_k obtained in the following manner:

$$\begin{aligned} res_0 &\leftarrow e_{body}(e_{seed}) \\ res_{i+1} &\leftarrow e_{body}(res_i)\ \texttt{union}\ res_i \quad , \quad i \geqslant 0 \end{aligned} \qquad \text{(IFP)}$$

where $k \geqslant 1$ is the minimum number for which $res_k \stackrel{s}{=} res_{k-1}$ and `union` is the XQuery `union` operator. If no such k exists, the inflationary fixed point is *undefined.* ◁

To make this semantics accessible in the XQuery language, we introduce the inflationary fixed point operator

$$\texttt{with \$x seeded by } e_{seed}\ \texttt{recurse } e_{body}\texttt{(\$x)} \qquad (1)$$

as a syntactic extension to XQuery. This `with` construct may be orthogonally composed with the remaining XQuery expression types. Its semantics is the inflationary fixed point of e_{body}(`$x`) seeded by e_{seed}. The expressions `$x`, e_{seed}, and e_{body}(`$x`) are the *variable*, *seed*, and *body* of the inflationary fixed point operator, respectively.

Note that if expression e_{body} does *not* invoke node constructors (*e.g.*, `element{·}{·}` or `text{·}`), expression (1) operates over a finite domain of nodes and its semantics is always defined. Otherwise, nodes might be created at each iteration and the semantics of (1) might be undefined. For example, `with $x seeded by () recurse <a>{$x}</a>` generates infinitely many distinct elements, thus it is undefined. When the result is defined, it is always a duplicate-free and document-ordered sequence of nodes, due to the semantics of the set operation `union`. Using the new operator we can express the query from Figure 2 in a concise and elegant fashion:

```
with $x seeded by doc("curriculum.xml")
                        //course[@code="c1"] .    (Q1)
recurse $x/id(./prerequisites/pre_code)
```

Clearly, the user-defined function template `fix(·)` (shown in ⌜ ⌟ in Figure 2) can be used to express fixed point computation by means of existing XQuery functionality. But since `with-seeded by-recurse` is a second-order construct (taking an XQuery variable name and two XQuery expressions as arguments), function `fix(·)` then has to be interpreted as a template in which the recursion body `rec_body(·)` needs to be instantiated to e_{body} (which is shown in Figure 2 for Query Q_1). Higher-order functions are currently not supported in the XQuery language.[3]

2.1 Using IFP to Compute Transitive Closure

Transitive closure is an archetype of recursive computation over relational data as well as over XML instances. For example, Regular XPath [29, 23] extends the navigational fragment of XPath, Core XPath [13], with a transitive closure operator on location paths. We extend this fragment and allow any XQuery expression of type `node()*`.

Definition 2. **Transitive Closure Operator.** Let e be an XQuery expression. The *transitive closure operator*

[3] A recent proposal and proof-of-concept implementation illustrates how first-class function types, lambdas, and their (partial) application could fit into a future version of XQuery's syntax and semantics [28].

$(\cdot)^+$ applied to e is the expression e^+. The semantics of e^+, called the *transitive closure of* e, is the result of

$$e \texttt{ union } e/e \texttt{ union } e/e/e \texttt{ union } \cdots \quad , \qquad \text{(TC)}$$

if it is a finite sequence. Otherwise, the semantics of e^+ is *undefined.* ◁

Analogously to the inflationary fixed point operator, e^+ might be undefined only if e contains node constructors. For example, `<a/>`$^+$ generates infinitely many distinct empty elements tagged with `a`, thus it is undefined.

Operator $(\cdot)^+$ applied to location paths expresses the transitive closure of paths in the XML tree: $(\texttt{child::*})^+ \equiv$ `descendant::*`; $(\texttt{child::*})^+\texttt{/self::a} \equiv$ `descendant::a`. Operator $(\cdot)^+$ applied to arbitrary expressions of type `node()*` expresses the transitive closure of arbitrary relations on nodes. The query from Figure 2 *e.g.*, can be expressed as:

```
doc("curriculum.xml")//course[@code="c1"]/
    (id(./prerequisites/pre_code))+          .  (Q'1)
```

Note that the expression in the scope of the transitive closure operator is a *data-value join* and cannot be expressed in Regular XPath.

Considering the equivalence of (Q_1) and (Q'_1), we can deduce the following translation of $(\cdot)^+$ into the `with-seeded by-recurse` construct. For some e in XQuery, e^+ can be expressed as follows:

```
with $x seeded by . recurse $x/e ,
```

where '`.`' denotes the context node. Unfortunately, this translation is not correct for all e in XQuery (but we will show in Section 4 that it is correct for all e in Regular XPath).

2.2 Fixed Points in SQL:1999

Fixed point computation can be expressed in SQL using the `WITH RECURSIVE` clause introduced with SQL:1999 [24]. The `WITH` clause defines a virtual table, while `RECURSIVE` specifies that the table is recursively defined. To exemplify, consider the table `Curriculum`(`course`, `prerequisite`) as a relational representation of the curriculum data from Figure 1. The prerequisites `P`(`course_code`) of the course with code `'c1'` expressed in Datalog are:

```
P(x)  ←  Curriculum('c1', x)
P(x)  ←  P(y), Curriculum(y, x) .
```

The equivalent SQL code reads:

```
WITH RECURSIVE P(course_code) AS
  (SELECT prerequisite                          ⎫
   FROM   Curriculum                            ⎬ seed
   WHERE  course = 'c1')                        ⎭
 UNION ALL
  (SELECT Curriculum.prerequisite               ⎫
   FROM   P, Curriculum                         ⎬ body
   WHERE  P.course_code = Curriculum.course)    ⎭
SELECT DISTINCT * FROM P;
```

Analogously to the XQuery variant, the query is composed of a *seed* and a *body.* In the seed, table `P` is instantiated with the direct prerequisites of course `'c1'`. In the body, table `P` is joined with table `Curriculum` to obtain the direct prerequisites of the courses in `P`. The results are added to `P`. The computation of the body is iterated until `P` stops growing.

```
res ← e_body(e_seed);
do
    res ← e_body(res) union res;
while res grows ;
return res;
```

(a) Algorithm *Naïve*.

```
res ← e_body(e_seed);
Δ ← res;
do
    Δ ← e_body(Δ) except res;
    res ← Δ union res;
while res grows ;
return res;
```

(b) Algorithm *Delta*.

Figure 3: Algorithms to evaluate the inflationary fixed point of e_{body}(\$x) seeded by e_{seed}. The result is *res*.

```
declare function delta($x,$res) as node()*
{
  let $delta := rec_body($x) except $res
  return if(empty($delta))
         then $res
         else delta($delta,$delta union $res)
};
```

Figure 4: An XQuery formulation of Delta.

The SQL:1999 standard requires engine support for *linear recursion*, *i.e.*, each `RECURSIVE` definition contains at most one reference to a mutually recursively defined table. Note that the recursive table `P` in the example above is defined linearly: it is referenced only once in the `FROM` clause of the body. This syntactic restriction allows for efficient evaluation. In Section 5, we define a similar syntactic restriction for the `with-seeded by-recurse` construct in XQuery.

3. ALGORITHMS FOR IFP

This section describes two algorithms, *Naïve* and *Delta*, commonly used to evaluate fixed point queries in the relational setting. *Delta* is more efficient than *Naïve*, but unfortunately, *Delta* is not always a correct implementation for our `with-seeded by-recurse` extension to XQuery.

Definition 1 of the inflationary fixed point straightforwardly yields the implementation shown in Figure 3(a), commonly referred to as *Naïve* [4]. At each iteration of the **while** loop, $e_{body}(\cdot)$ is executed on the intermediate result sequence *res* until no new nodes are added to it. Note that the recursive function `fix(·)` shown in Figure 2 is the XQuery equivalent of *Naïve*. Another remark is that the old nodes in *res* are fed into $e_{body}(\cdot)$ over and over again. Depending on the nature of $e_{body}(\cdot)$, *Naïve* may involve a substantial amount of redundant computation.

A folklore variation of *Naïve* is the *Delta* algorithm [19] of Figure 3(b). *Delta* implements a divide-and-conquer approach to evaluation. In this variant, $e_{body}(\cdot)$ is invoked only for those nodes that have not been encountered in earlier iterations: the node sequence Δ is the difference between $e_{body}(\cdot)$'s last answer and the current result *res*. In general, $e_{body}(\cdot)$ will process fewer nodes. Thus, *Delta* introduces a significant potential for performance improvement, especially for large intermediate results and computationally expensive recursion bodies.

Figure 4 shows the corresponding XQuery user-defined function `delta(·,·)` which, for Figure 2 and thus Query Q_1, can serve as a drop-in replacement for function `fix(·)` (in line 13, `return delta(rec_body($seed),())` needs to replace its invocation).

Unfortunately, *Delta* is *not always* a valid optimization for the `with-seeded by-recurse` clause as we will see in the following expression:

```
let $seed := <a><b><c/></b></a>
return with $x seeded by $seed
       recurse if (count ($x) = 1)       .     (Q2)
               then $x/* else ()
```

While *Naïve* computes `(a, b)`, *Delta* computes `(a, b, c)`, where `a`, `b`, and `c` denote the elements constructed by the respective subexpressions of the seed. The table below illustrates the progress of the iterations performed by both algorithms.

Iteration	*Naïve* *res*	*Delta* *res*	Δ
0	`(a)`	`(a)`	`(a)`
1	`(a, b)`	`(a, b)`	`(b)`
2	`(a, b)`	`(a, b, c)`	`(c)`
3		`(a, b, c)`	`()`

The culprit in this example is the application `count ($x)` which prohibits an evaluation based on divide-and-conquer with respect to `$x`. *Delta* may *not* be safely applied in this case.

Even though *Delta* does not always compute the inflationary fixed point correctly, we can investigate for which body expressions *Delta* computes the correct result and apply it in those cases. In the next section, we provide a natural semantic property which allows us to trade *Naïve* for *Delta*.

4. DISTRIBUTIVITY FOR XQUERY

In this section, we define a *distributivity property* for XQuery expressions. We show that distributivity implies the correctness of *Delta* as an implementation for the `with-seeded by-recurse` clause. Moreover, we show that distributivity allows for the elegant formulation of transitive closure (operator $(\cdot)^{+}$) based on our extension to the XQuery syntax.

4.1 Defining Distributivity

A function e defined on sets is *distributive* if, for any non-empty sets X_1 and X_2, $e(X_1 \cup X_2) = e(X_1) \cup e(X_2)$. This property suggests the use of the divide-and-conquer approach taken by algorithm *Delta*, which applies the recursion body to subsets of its input and takes the union of the results. We define a similar property for XQuery expressions using the sequence set-equality $\overset{s}{=}$ defined in Section 2. Recall that in this paper we *only* consider XQuery expressions and sequences of type `node()*`.

Definition 3. **Distributivity Property.** Let e be an XQuery expression. Expression $e(\texttt{\$x})$ is *distributive for* `$x` iff for any non-empty sequences X_1, X_2,

$$e(X_1\,\texttt{union}\,X_2) \quad \overset{s}{=} \quad e(X_1)\,\texttt{union}\,e(X_2)\ . \qquad (2)$$

◁

Note that if `$x` is not a free variable in e, then Equality (2) always holds, thus e is distributive for `$x`.

Proposition 1. *Let e be an XQuery expression. Expression $e(\texttt{\$x})$ is* distributive for `$x` *iff for any node sequence $X \neq$* `()` *and any fresh variable* `$y`,

$$(\texttt{for \$y in \$x return } e(\texttt{\$y}))(X) \quad \overset{s}{=} \quad e(X)\ . \qquad (3)$$

Proof. Consider the following equality: for any sequence $X = (x_1, \ldots, x_n)$, $n \geq 1$,

$$(e(x_1)\ \texttt{union}\ \cdots\ \texttt{union}\ e(x_n)) \quad \overset{s}{=} \quad e(X)\ . \qquad (4)$$

It is easy to see that for any partition X_1 and X_2 of X, *i.e.*, $X_1 \cap X_2 = \emptyset$ and $X_1 \cup X_2 = X$, if Equality (2) holds then Equality (4) holds for X, and vice versa. Thus Equalities (2) and (4) are equivalent.

According to the XQuery Formal Semantics [8], the left-hand side of Equality (3) evaluates to the sequence $(e(x_1)$, $\ldots$, $e(x_n))$, which is set-equal to $(e(x_1)$ `union` $\cdots$ `union` $e(x_n))$, the left-hand side of Equality (4). From the equivalence of Equalities (2) and (4), it follows the equivalence of Equalities (2) and (3). QED

We will use Equality (3) as an *alternative* definition of distributivity.

Proposition 2. *Any XQuery expression that has the form $e(\texttt{\$x}) = \texttt{\$x/}p$ is distributive for* `$x` *if the expression p neither contains (i) free occurrences of* `$x`*, nor (ii) calls to* `fn:position()` *or* `fn:last()` *that refer to the context item sequence bound to* `$x`*, nor (iii) node constructors.*

The proof of this proposition is given in [2].

Expressions of the form `$x/`p where p is a Core XPath (or even Regular XPath) expression are prevalent examples of distributive expressions in XQuery. Note that all Core XPath and Regular XPath expressions satisfy the conditions *(i)* to *(iii)* of Proposition 2 above.

In reverse it is easy to see that `$x[1]` is not distributive for `$x`. For a counterexample, let `$x` be bound to `(<a/>, <b/>)`, then `$x[1]` evaluates to `(<a/>)`, whereas `for $i in $x return $i[1]` evaluates to `(<a/>, <b/>)`.

4.2 Trading *Naïve* for *Delta*

We say that *Delta* and *Naïve* are *equivalent* for a given fixed point expression, if for any XML document (collection) both algorithms produce the same sequence of nodes.

Theorem 1. **Distributivity Guarantees Correctness of *Delta*.** *Consider the expression* `with $x seeded by` e_{seed} `recurse` $e_{body}(\texttt{\$x})$. *If $e_{body}(\texttt{\$x})$ is distributive for* `$x`*, then the algorithm* Delta *correctly computes the inflationary fixed point of $e_{body}(\texttt{\$x})$ seeded by e_{seed}.*

Proof. We show by inductive reasoning that *Delta* and *Naïve* have the same intermediate results, denoted by res_i^{Δ} and res_i, respectively. The equivalence of *Naïve* and *Delta* follows from this. The induction is on i, the iteration number of the **do** $\cdots$ **while** loops.

In its first loop iteration, *Naïve* yields

$$e_{body}\left(e_{body}\left(e_{seed}\right)\right)\ \texttt{union}\ e_{body}\left(e_{seed}\right)$$

which is equivalent to *Delta*'s first intermediate result

$$\left(e_{body}\left(e_{body}\left(e_{seed}\right)\right)\ \texttt{except}\ e_{body}\left(e_{seed}\right)\right)\ \texttt{union}\ e_{body}\left(e_{seed}\right)\ .$$

Suppose that $res_k = res_k^{\Delta}$, for all $k \leq i$. We show in the following that $res_{i+1} = res_{i+1}^{\Delta}$: Starting with the definition from the *Naïve* algorithm (a), we can apply Set-Equality (2) at (b). Note that we are allowed to replace set-equality with strict equality here, since both sequences are in document order and duplicate-free due to the semantics of `union`. At

(c) we apply the inductive step, before taking the definition of *res* in algorithm *Delta* into account (at (d)). Deduction (e) follows from the containment of $e_{body}(res_{i-1})$ in res_i. Taking the definition of Δ in *Delta* into account (at (f)) we get the desired result:

$$\begin{aligned}
res_{i+1} &\underset{(a)}{=} e_{body}\,(res_i)\ \texttt{union}\ res_i \\
&= e_{body}\,((res_i\ \texttt{except}\ \Delta_i)\ \texttt{union}\ \Delta_i)\ \texttt{union}\ res_i \\
&\underset{(b)}{=} e_{body}\,(res_i\ \texttt{except}\ \Delta_i)\ \texttt{union}\ e_{body}\,(\Delta_i)\ \texttt{union}\ res_i \\
&\underset{(c)}{=} e_{body}\,\big(res_i^{\Delta}\ \texttt{except}\ \Delta_i\big)\ \texttt{union}\ e_{body}\,(\Delta_i)\ \texttt{union}\ res_i^{\Delta} \\
&\underset{(d)}{=} e_{body}\,\big(res_{i-1}^{\Delta}\ \texttt{union}\ \Delta_i\ \texttt{except}\ \Delta_i\big)\ \texttt{union} \\
&\quad\ e_{body}\,(\Delta_i)\ \texttt{union}\ res_i^{\Delta} \\
&= e_{body}\,\big(res_{i-1}^{\Delta}\big)\ \texttt{union}\ e_{body}\,(\Delta_i)\ \texttt{union}\ res_i^{\Delta} \\
&\underset{(e)}{=} e_{body}\,(\Delta_i)\ \texttt{union}\ res_i^{\Delta} \\
&= \big(e_{body}\,(\Delta_i)\ \texttt{except}\ res_i^{\Delta}\big)\ \texttt{union}\ res_i^{\Delta} \\
&\underset{(f)}{=} \Delta_{i+1}\ \texttt{union}\ res_i^{\Delta} \\
&\underset{(d)}{=} res_{i+1}^{\Delta}\ .
\end{aligned}$$

QED

In the next section, we discuss one more benefit of distributivity, namely the correctness of an elegant translation of the transitive closure operator into the `with-seeded by-recurse` clause.

4.3 Translating Transitive Closure

Distributivity is also a key to understanding the relation between the transitive closure operator and the `with-seeded by-recurse` clause in our XQuery dialect. Intuitively, if expression e is distributive for the context sequence, then e^{+} is equivalent to the XQuery expression `with $x seeded by . recurse $x/`e, where `$x`, a fresh variable, is a placeholder for the context sequence.

THEOREM 2. *Consider an XQuery expression e and a variable* `$x`*, such that* `$x` $\notin fv(e)$*. If* `$x/`e *is distributive for* `$x`*, then*

$$e^{+} = \texttt{with \$x seeded by . recurse \$x/}e$$

The proof, similar to the proof of Theorem 1, is given in [2].

From Proposition 2 and Theorems 1 and 2 follows that the transitive closure of any Regular XPath expression can be safely computed with *Delta* using the translation in Theorem 2.

5. ASSESSING DISTRIBUTIVITY

Whenever an XQuery processor plans the evaluation of `with $x seeded by` e_{seed} `recurse` e_{body}, knowing the answer to "*Is e_{body} distributive for* `$x`*?*" is particularly valuable: it allows the processor to apply *Delta* for evaluating the inflationary fixed point of e_{body} seeded by e_{seed}. We may legitimately expect *Delta* to be a significantly more efficient fixed point evaluation strategy than *Naïve* (Section 7 will indeed make this evident). While, unfortunately, there is no complete procedure to decide this question[4], still we can safely approximate the answer.

[4]If, for two arbitrary expression e_1, e_2 in which `$x` does not occur free, an XQuery processor could assess whether `if (deep-equal (`e_1`,`e_2`)) then $x else $x[1]` is distributive for `$x`, it could also decide the equivalence of e_1 and e_2 (which is impossible).

5.1 A Syntactic Approximation of Distributivity

In this section, we define a syntactic fragment of XQuery for a variable `$x`, called *the distributivity-safe fragment for* `$x`. The membership of this fragment can be determined in linear time with respect to the size of the expression. We show that distributivity safety implies distributivity. Moreover, this fragment is *expressively complete* for distributivity, *i.e.*, any distributive XQuery expression is expressible in the distributivity-safe fragment of XQuery.

Intuitively, we may apply a divide-and-conquer evaluation strategy for an expression e(`$x`), if any subexpression of e accesses the nodes in `$x` one by one. The most simple example of such subexpression is `for $y in $x return` e(`$y`), where e is an XQuery expression such that `$x` $\notin fv(e)$. On the other hand, we may *not* apply a divide-and-conquer evaluation strategy if any subexpression of e accesses `$x` as a whole. Examples of such problematic subexpressions are `count ($x)` and `$x[1]`, but also the general comparison `$x = 10` which involves existential quantification over the sequence bound to `$x`.

Further, subexpressions whose value is *independent* of `$x` are distributive. The only exception of this rule are XQuery's node constructors, *e.g.*, `element{·}{·}`, which create new node identities upon each invocation. With `$x` bound to `(<a/>,<b/>)`, for example,

$$\texttt{element\{"c"\}\{()\}} \not\overset{s}{=} \begin{array}{l}\texttt{for \$y in \$x return}\\ \texttt{element\{"c"\}\{()\}}\end{array},$$

since the right-hand side will yield a sequence of two distinct element nodes.

We implement these considerations when defining the *distributivity-safe* fragment of XQuery. For practical reasons, in our definition, we use LiXQuery [21], a fragment of XQuery. LiXQuery has a simpler syntax and data model than XQuery, though it preserves the Turing-completeness property. It includes the most important language constructs, three basic types of items: `xs:boolean`, `xs:string`, and `xs:integer` plus four node kinds: `element()`, `attribute()`, `text()`, and `document-node()`. The language has well-defined semantics and it was designed as a convenient tool for studying XQuery language properties. Given our prior remarks on how *node constructors appearing in the recursion body* inhibit distributivity, we deliberately omit their treatment in the upcoming syntactic assessment of distributivity for recursion bodies formulated in LiXQuery. (Note that this does not affect the ability to construct nodes outside the recursion body.)

DEFINITION 4. **Distributivity Safety.** A LiXQuery expression e is called *distributivity-safe for* `$x`, if the rules of Figure 5 can infer $ds_{\texttt{\$x}}(e)$. ◁

The inference rules of Figure 5 assess syntactically the *distributivity safety* $ds_{\texttt{\$x}}(e)$ of an arbitrary LiXQuery input expression e by traversing e's parse tree in a bottom-up fashion. Rules CONST and VAR constitute the base of the fragment, inferring the distributivity safety of LiXQuery expressions that do not contain `$x` free and of variables, including `$x`. Rule CONCAT propagates the distributivity safety of subexpressions. Rules FOR1 and FOR2 ensure that

$$\frac{}{ds_{\texttt{\$x}}(c)}(\textsc{Const}) \quad \frac{}{ds_{\texttt{\$x}}(\$v)}(\textsc{Var}) \quad \frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2) \quad ds_{\texttt{\$x}}(e_3)}{ds_{\texttt{\$x}}(\texttt{if}\,(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3)}(\textsc{If}) \quad \frac{\oplus \in \{\texttt{,},\texttt{|}\} \quad ds_{\texttt{\$x}}(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(e_1 \oplus e_2)}(\textsc{Concat})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(\texttt{for}\ \$v\ \texttt{at}\ \$p\ \texttt{in}\ e_1\ \texttt{return}\ e_2)}(\textsc{For1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2)}{ds_{\texttt{\$x}}(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2)}(\textsc{For2})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(\texttt{let}\ \$v\ \texttt{:=}\ e_1\ \texttt{return}\ e_2)}(\textsc{Let1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2) \quad ds_{\$v}(e_2)}{ds_{\texttt{\$x}}(\texttt{let}\ \$v\ \texttt{:=}\ e_1\ \texttt{return}\ e_2)}(\textsc{Let2})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(c_i)_{i=1\ldots n+1}}{ds_{\texttt{\$x}}\left(\begin{array}{l}\texttt{typeswitch}\,(e_1)\\ \quad \texttt{case}\ \tau_1\ \ \texttt{return}\ c_1\\ \quad\quad\quad \vdots \\ \quad \texttt{case}\ \tau_n\ \ \texttt{return}\ c_n\\ \quad \texttt{default}\ \ \texttt{return}\ c_{n+1}\end{array}\right)}(\textsc{TypeSw}) \quad \frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(e_1/e_2)}(\textsc{Step1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2)}{ds_{\texttt{\$x}}(e_1/e_2)}(\textsc{Step2})$$

$$\frac{\texttt{declare function}\ f(\$v_1,\ldots,\$v_n)\,\{\,e_0\,\} \quad (\texttt{\$x} \in fv(e_i) \Rightarrow ds_{\texttt{\$x}}(e_i) \wedge ds_{\$v_i}(e_0))_{i=1\ldots n}}{ds_{\texttt{\$x}}(f(e_1,\ldots,e_n))}(\textsc{FunCall})$$

Figure 5: Distributivity safety $ds_{\texttt{\$x}}(\cdot)$: A syntactic approximation of the distributivity property for LiXQuery-formulated recursion bodies.

the recursion variable `$x` occurs either in the body e_2 or in the binding expression e_1 of a `for`-iteration but not both. A similar remark applies to Rules Step1, Step2 (in XQuery, the step operator '`/`' essentially describes an iteration over a sequence of type `node()*` [8]) and Rules Let1, Let2. Note that these conditions resemble the linearity constraint of SQL:1999. Rules If and TypeSw ensure that `$x` does not occur free in e_1, the expression in the conditional, while propagating the distributivity safety of subexpressions. Rule FunCall recursively infers the distributivity of the body of a called function if the recursion variable occurs free in the function argument(s).

The rules in Figure 5 can be checked with a single traversal of the parse tree of a LiXQuery expression. Thus the membership to the distributivity-safe fragment is in linear time with respect to the size of an XQuery expression.

Theorem 3. **Soundness.** *Any XQuery expression* e *that is distributivity-safe for a variable* `$x`, i.e., *for which* $ds_{\texttt{\$x}}(e)$ *holds, is also distributive for* `$x`.

The proof of this implication, by induction on the syntactical structure of e, is given in [2].

The distributive-safe fragment does not contain all distributive expressions. For example, `count ($x) >= 1` is not distributivity-safe, but still distributive for `$x`. However, it is interesting to note that the distributivity-safe fragment is expressive complete for distributivity.

Proposition 3. **Expressive completeness.** *Given an XQuery expression* e(`$x`)*, if* e(`$x`) *is distributive for* `$x` *and does not contain node constructors as subexpressions, then it is set-equal to* `for $y in $x return` e(`$y`)*, which is distributivity-safe for* `$x`.

Proof. This is a direct consequence of Rule For2 (Figure 5) and Proposition 1. QED

At the expense of a slight query reformulation, we may provide a "syntactic distributivity hint" to an XQuery processor.

6. DISTRIBUTIVITY AND RELATIONAL XQUERY

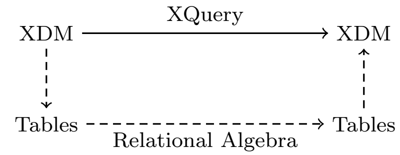


Figure 6: Relational XQuery (dashed path) faithfully implements the XQuery semantics.

In this section we will, literally, follow an alternative route to decide the applicability of *Delta* for the evaluation of the `with $x seeded by` e_{seed} `recurse` e_{body}(`$x`) construct. We leave syntax aside and instead inspect *relational algebraic code* that has been compiled for e_{body}: the equivalent algebraic representation of e_{body} renders the check for the inherently algebraic distributivity property particularly uniform and simple.

Relational XQuery. This alternative route is inspired by various approaches that compile instances of the XQuery Data Model (XDM) and XQuery expressions into relational tables and algebraic plans over these tables, respectively, and thus follow the dashed path in Figure 6. The *Pathfinder* project[5] fully implements such a purely relational approach to XQuery. Here we use the translation strategy of Pathfinder that has been carefully designed to *(i)* faithfully preserve the XQuery semantics (including compositionality, node identity, iteration and sequence order), and *(ii)* yield relational plans which exclusively rely on regular relational query engine technology (no specific operators or index structures are required, in particular) [16, 17] . We use the generated plans as a tool to reason over the distributivity of the associated XQuery expression.

The compiler emits a dialect of relational algebra that mimics the capabilities of modern SQL query engines (Table 1). The row numbering operator $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{p}}$ directly compares with SQL:1999's `ROW_NUMBER() OVER (PARTITION BY p ORDER BY` $\mathsf{b}_1,\ldots,\mathsf{b}_n$`)` and correctly implements the or-

[5] `http://www.pathfinder-xquery.org/`

Operator	Semantics
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$	project onto col.s a_i, rename b_i into a_i
σ_{b}	select rows with column $\mathsf{b} = \mathit{true}$
$\bowtie_p$	join with predicate p
$\ddot{\bowtie}_q$	iterated evaluation of rhs argument (`APPLY`)
$\times$	Cartesian product
$\cup$	union
$\setminus$	difference
$\mathtt{count}_{\mathsf{a}:/\mathsf{b}}$	aggregates (group by b, result in a)
$\circledcirc_{\mathsf{a}:\langle\mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}$	n-ary arithmetic/comparison operator $\circ$
$\varrho_{\mathsf{a}:\langle\mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{p}}$	ordered row numbering (by $\mathsf{b}_1,\ldots,\mathsf{b}_n$)
$⏍_{\alpha::n}$	XPath step join (axis α, node test n)
$\varepsilon, \tau, \ldots$	node constructors
μ, μ^{Δ}	fixpoint operators

Table 1: Relational algebra dialect emitted by the Pathfinder compiler.

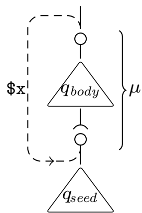
der semantics of XQuery on the (unordered) algebra. Other non-textbook operators, like ε or ⏍, merely are macros representing "micro plans" composed of standard relational operators: expanding $⏍_{\alpha::n}(q)$, for example, reveals $\mathsf{doc} \bowtie_p q$, where p is a conjunctive range predicate that realizes the semantics of an XPath location step along axis α with node test n between the context nodes in q and the encoded XML document doc. Dependent joins $\ddot{\bowtie}$—also named `CROSS APPLY` in Microsoft SQL Server's SQL dialect *Transact-SQL*—like ⏍ are only a logical concept and can be replaced by standard relational operators [12].

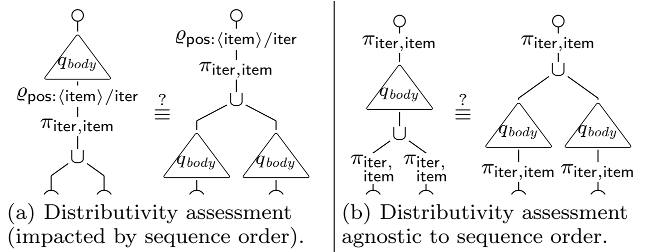
The plans operate over relational encodings of XQuery item sequences held in flat (1NF) tables with an iter|pos|item schema. In these tables, columns iter and pos are used to properly reflect `for`-iteration and sequence order, respectively. Column item carries encodings of XQuery items, *i.e.*, atomic values or nodes. The inference rules driving the compilation procedure are described in [17]. The result is a DAG-shaped query plan where the sharing of sub-plans primarily coincides with repeated references to the same variable in the input XQuery expression.

Since our current work is concerned with distributivity assessment (as opposed to query *evaluation*—but see Section 7), we rephrase the compilation of XQuery `for` expressions of [17] (see Rule FOR′ in Appendix A) to make use of the dependent join operator $\ddot{\bowtie}$. With correlations now made explicit, we can assess distributivity purely based on algebraic equivalences.

6.1 Is Expression e_{body} Distributive? (An Algebraic Account)

An occurrence of our extension `with $x seeded by` e_{seed} `recurse` $e_{body(\$x)}$ in a source XQuery expression will be compiled into a plan fragment as shown here on the left. In the following, let q denote the algebraic query plan that has been compiled for XQuery expression e. Operator μ, the algebraic representation of algorithm *Naïve* (Figure 3(a)), iterates the evaluation of the algebraic plan for e_{body} and feeds its output ⚲ back to its input ⅄ until the IFP is reached. If we can guarantee that the plan for e_{body} is distributive, we may safely trade μ for its *Delta*-based variant μ^{Δ} which, in general, will feed significantly less items back in each iteration (see Figure 3(b) and Section 7).





(a) Distributivity assessment (impacted by sequence order). (b) Distributivity assessment agnostic to sequence order.

Figure 7: Algebraic distributivity assessment.

We defined the necessary distributivity property (Definition 3) based on the XQuery operator `union`. In the algebraic setting, the XQuery `union` operation is compiled to an expression that faithfully implements the XQuery order requirements—for each iteration the result is ordered by the node rank in column item (Appendix A illustrates the compilation of `union`, which had been omitted in [17]):

$$e_1 \ \mathtt{union}\ e_2 \Mapsto \varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{item}}\left(q_1 \cup q_2\right)\right) .$$

Straightforward application of this translation to Definition 3 allows us to express the distributivity criterion based on the equivalence of relational plans. If we can prove the equivalence of the two plans in Figure 7(a), we know that the XQuery expression q_{body} must be distributive.

The condition expressed in Figure 7(a), however, is slightly more restrictive than necessary. It is a prerequisite for distributivity that the recursion body q_{body} does not inspect sequence positions in its input. For a distributive q_{body} it must be legal to omit the row-numbering operator $\varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle/\mathsf{iter}}$ in the left-hand side of Figure 7(a) and discard all position information in the inputs of sub-plan q_{body} (using $\pi_{\mathsf{iter},\mathsf{item}}$).[6] Further, since Definition 3 is indifferent to sequence order, we are also free to disregard the row-numbering operator on top of the right-hand-side plan and place a projection $\pi_{\mathsf{iter},\mathsf{item}}$ on top of both plans to make the order indifference explicit. Proving the equivalence illustrated in Figure 7(b), therefore, is sufficient to decide distributivity. This equality is the algebraic expression of the divide-and-conquer evaluation strategy: evaluating e_{body} over a composite input (lhs, ⅄∪⅄) yields the same result as the union of the evaluation of e_{body} over a partitioned input (rhs).

The equivalence criterion in Figure 7(b) suggests an assessment of distributivity based on algebraic rewrites. If we can successfully "push" a union operator $\cup$ through the sub-plan q_{body}, its corresponding XQuery expression e_{body} must be distributive and we can safely trade μ for μ^{Δ} to compute the fixed point.

To this end, we use a set of algebraic *rewrite rules* (Figure 8) that try to move a union operator upwards the plan DAG. To avoid ambiguity or infinite loops during the rewrite process, we *mark* the union operator (indicated as $\uplus$) in the left-hand-side plan q_{left} of Figure 7(b), before we start rewriting. We then exhaustively apply the rule set in Figure 8 to each sub-plan in q_{left} in a bottom-up fashion. Since

[6]Since order indifference proves valuable also for other reasons, Pathfinder's query compiler readily omits position information in this sense [15].

$$\frac{\otimes \in \{\pi, \sigma, \circledcirc, \lrcorner\}}{\otimes\,(q_1 \uplus q_2) \rightarrow (\otimes\,(q_1)) \uplus (\otimes\,(q_2))} \quad (\textsc{Unary})$$

$$\frac{\otimes \in \{\cup, \times, \bowtie, \ddot{\bowtie}\}}{(q_1 \uplus q_2) \otimes q_3 \rightarrow (q_1 \otimes q_3) \uplus (q_2 \otimes q_3)} \quad (\textsc{Binary1})$$

$$\frac{\otimes \in \{\cup, \times, \bowtie, \ddot{\bowtie}\}}{q_1 \otimes (q_2 \uplus q_3) \rightarrow (q_1 \otimes q_2) \uplus (q_1 \otimes q_3)} \quad (\textsc{Binary2})$$

$$\frac{}{(q_1 \uplus q_2) \cup (q_3 \uplus q_4) \rightarrow (q_1 \cup q_3) \uplus (q_2 \cup q_4)} \quad (\textsc{Union})$$

Figure 8: An algebraic approximation of the distributivity property for arbitrary XQuery expressions.

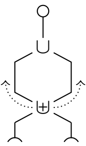
each rule in the set strictly moves the marked union operator upwards the plan, termination of the process is guaranteed. Further, the number of operators n in q_{body} is an upper bound for the number of rewrites needed to push $\uplus$ through q_{body}; n itself is bound by the size of e_{body} (we have seen the same complexity bound for the syntactic analysis of Section 5.1).

Once the rule set does not permit any further rewrites, we compare the rewritten plan q'_{left} with the right-hand side plan q_{right} of Figure 7(b) for structural equality. Such equality guarantees the equivalence of both plans and, hence, the distributivity of e_{body}.

Figure 9 shows the rewrites involved to determine the distributivity of e_{body} for Query Q_1 (Section 2). We place a marked union operator $\uplus$ as the input to the algebraic plan q_{body} obtained for the recursion body of Query Q_1. The resulting plan corresponds to the left-hand side of Figure 7(b). Applying the equivalence rules Unary, Binary1, and again Rule Unary pushes $\uplus$ up to the plan root, as illustrated in Figures 9(b), 9(c), and 9(d), respectively. The final plan (Figure 9(d)) is structurally identical to the right-hand side of Figure 7(b), with q_{body} instantiated with the recursion body in Query Q_1. We can conclude distributivity for q_{body} and, in consequence, for the recursion body in Query Q_1.

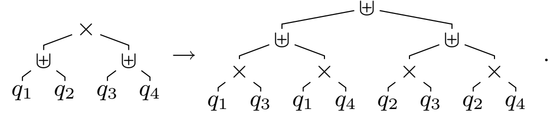
Rewriting in Detail

Zooming in from the plan to the operator level, we now provide justification for the equivalence rules in Figure 8. Operators π, σ, and $\circledcirc$ of Rule Unary are defined in row-by-row fashion and are distributive as such. As mentioned earlier, operator $\lrcorner$ of Rule Unary can be rewritten into a join. The expanded operator then matches Rule Binary2, where parameter e_1 is the XML document relation doc. For operators $\cup$, $\times$, and $\bowtie$, the equivalences in Rules Binary1, Binary2, and Union follow textbook-style plan rewriting. The distributivity of $\ddot{\bowtie}$ (Rules Binary1 and Binary2) follows from its definition in Appendix A.



Note that the bottom-up traversal in combination with the DAG-shaped plans may lead to a situation where two marked union operators $\uplus$ appear on either side of a binary operator. A trivial example is the plan shown on the right, where $\uplus$ is pushed up along both branches of the relational plan for `$x union $x`.

If this happens with operator $\cup$ in the middle, Rule Union uses the associativity of $\cup$ to re-order the input arguments and change markings such that only one instance of $\uplus$ remains left. Pushing two instances of $\uplus$ through $\times$, $\bowtie$, or $\ddot{\bowtie}$, by contrast, takes two rewrites along Rules Binary1 and Binary2, yielding, *e.g.*,



Most likely, the resulting plan will not satisfy the eventual test for structural equality. The plan analyzer can abort the rewrite process early in this case and report non-distributivity. It is the *lack* of a rule like Rule Union for $\times$, $\bowtie$, and $\ddot{\bowtie}$ that implements the restriction on occurrences of the recursion variable `$x` that we saw in Section 5.1 (`$x` $\notin fv(e)$ in the premises of Figure 5) or the single-occurrence requirement in SQL:1999-style recursion (Section 2.2).

For other non-distributive input, the application of the rules in Figure 8 typically leads to a situation where one or more instances of $\uplus$ "get stuck" in places other than the plan root. This happens, *e.g.*, when non-distributive operators such as difference ($\setminus$), row numbering (ϱ), or aggregation operators are encountered during the $\uplus$ push-up. Since the structural equality check is bound to fail in such an event, the algebraic distributivity analyzer may choose to abort rewriting early and report the query as non-distributive. (The query analyzer of Pathfinder follows this strategy, for instance.)

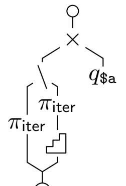
6.2 Coping with Syntactic Variation

The virtue of an algebraic test for distributivity is its concise specification in terms of the rewrite set in Figure 8. Let us now see how both approaches to distributivity analysis react to examples that are non-straightforward.

In both setups, the assessment of distributivity can become rather intricate. Consider, *e.g.*, the XQuery expression

```
let $a := doc("a.xml")
return with $x seeded by e_seed
       recurse if ($x/self::b)          } e_body     .     (Q3)
               then () else $a          }
```

The syntactic as well as the algebraic approximation report this query as non-distributive. And, indeed, if the seed e_{seed} contains an element labeled b, the output of algorithm *Naïve* does not contain the document a.xml, whereas the application of *Delta* would return a.xml as part of its result. After the first evaluation of e_{body}, the implementation in Figure 3(b) no longer knows about the existence of the b node in the seed, once it reaches the invocation of $e_{body}(\Delta)$ (line 4 in Figure 3(b)). The algorithm would thus (wrongly) emit the content of `$a` as the result of the `with-seeded by-recurse` clause.



The compiled plan for the body expression in Query Q_3 explains how the algebraic distributivity analyzer made the right decision. Since the difference operator $\setminus$ (used to implement the check for non-existence) is non-distributive, the rewrite does not succeed in pushing up a union operator through the plan. Rule If takes the account in the syntactic analysis and prevents the

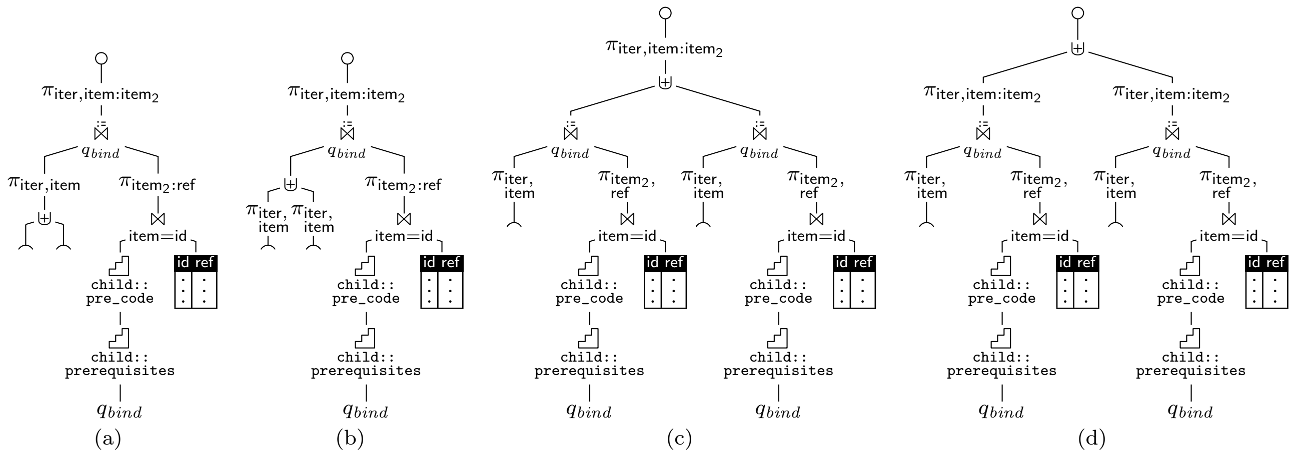


Figure 9: Transformation of the recursion body e_{body} of Query Q_1. (Rewrites proceed from left to right.)

appearance of the seed variable in an `if-then-else` condition.

Now consider a variant of Query Q_3 that instantiates the recursion body e_{body} with

$$e'_{body} = \texttt{if (\$x/self::b) then \$a else ()} \ .$$

(Note the swapped roles of the `then` and `else` branches.) This subtle change made e'_{body} a distributive expression and, hence, algorithm *Delta* a valid means to evaluate the query. Both algorithms now return the document node bound to `$a` only if a `b` element can be found among the nodes in e_{seed}.[7]

The syntactic approximation of Figure 5 concludes that the new recursion body will not be distributive-safe either, since variable `$a` still occurs in the clause's condition. This situation may easily be remedied by extending the rule set of Figure 5 (though we omit details here). It is interesting to see, however, how an algebraic analyzer handles the asymmetry of the `then`/`else` branches in XQuery's conditional expressions. The compiled plan for the rewritten recursion body now looks like the one shown here. It is easy to see that the "push-up" of the relational union will succeed straightforwardly. The structural equality test will then return a positive answer to the distributivity safety of e'_{body}.

Algebraic distributivity assessment plays its full trump, however, if it is paired with algebraic instruments that help further abstraction from syntactical equivalences. The query compiler of Pathfinder, *e.g.*, will discover the equivalence of

```
if (empty ($x/self::b)) then () else $a
```

and expression e'_{body} above and produce identical plan DAGs for both expressions. The distributivity safety then becomes easy to detect by the algebraic checker. Detecting such interplay between the `if`, `then`, and `else` clauses of an XQuery conditional might remain a challenge, however, for an analysis based on syntax only.

[7] e'_{body} is set-equivalent to the query `for $y in $x/self::b return $a`.

7. QUERYING WITH DISTRIBUTIVITY SAFETY BELT ON

Recasting a recursive XQuery query as an inflationary fixed point computation imposes restrictions. Such recasting, however, also puts the query processor into control since the applicability of a promising optimization, trading *Naïve* for *Delta*, becomes effectively decidable. This section provides the evidence that significant gains can indeed be realized, much like in the relational domain.

To quantify the impact, we implemented the two fixed point operator variants μ and μ^{Δ} (Section 6.1) in *MonetDB/XQuery 0.22* [7], an efficient and scalable XQuery processor that consequently implements the Relational XQuery approach (Section 6). Its algebraic compiler front-end *Pathfinder* has been enhanced *(i)* to process the syntactic form `with-seeded by-recurse`, and *(ii)* to implement the algebraic distributivity check. All queries in this section were recognized as being distributive by *Pathfinder*. To demonstrate that any XQuery processor can benefit from optimized fixed point evaluation in the presence of distributivity, we also performed the transition from *Naïve* to *Delta* on the XQuery source level and let *Saxon-SA 8.9* [22] process the resulting user-defined recursive queries (cf. Figures 2 and 4). All experiments were conducted on a Linux-based host (64 bit), with two 3.2 GHz Intel Xeon® CPUs, 8 GB of primary and 280 GB SCSI disk-based secondary memory.

Table 2 summarizes our observations for four query types, chosen to inspect the systems' behavior for growing input XML instance sizes and varying result sizes at each recursion level (the maximum recursion depth ranged from 5 to 33).

7.1 XMark Bidder Network

To assess scalability, we computed a bidder network—recursively connecting the sellers and bidders of auctions (Figure 10)—over XMark [27] XML data of increasing size (from scale factor 0.01, small, to 0.33, huge). If *Delta* is used to compute the inflationary fixed point of this network, *MonetDB/XQuery* (2.1 to 3.4 times faster) as well as *Saxon* (1.2 to 2.7 times faster) benefit significantly. Most importantly, note that the number of nodes in the network grows quadratically with the input document size. Algorithm *Delta* feeds significantly less nodes back in each re-

Query	*MonetDB/XQuery*		*Saxon-SA 8.9*		Total # of Nodes Fed Back		Recursion Depth
	Naïve	*Delta*	*Naïve*	*Delta*	*Naïve*	*Delta*	
Bidder network (small)	404 ms	190 ms	2,307 ms	1,872 ms	40,254	9,319	10
Bidder network (medium)	5,144 ms	2,135 ms	15,027 ms	7,284 ms	683,225	122,532	16
Bidder network (large)	40,498 ms	14,351 ms	123,316 ms	52,436 ms	5,694,390	961,356	15
Bidder network (huge)	1,344,806 ms	389,946 ms	1,959,749 ms	723,600 ms	87,528,919	9,799,342	24
Romeo and Juliet	1,332 ms	458 ms	1,150 ms	818 ms	37,841	5,638	33
Curriculum (medium)	200 ms	145 ms	1,308 ms	1,040 ms	12,301	3,044	18
Curriculum (large)	1,509 ms	687 ms	3,485 ms	2,176 ms	127,992	19,780	35
Hospital (medium)	695 ms	469 ms	1,301 ms	1,290 ms	99,381	50,000	5

Table 2: Naïve vs. Delta: Comparison of query evaluation times and total number of nodes fed back.

```
let $lengths := for $speech in doc("r_and_j.xml")//SPEECH
                let $rec := with $x seeded by ($speech/preceding-sibling::SPEECH[1], $speech) (: pair of speakers :)
                            recurse $x/following-sibling::SPEECH[1][SPEAKER = preceding-sibling::SPEECH[2]/SPEAKER]
                return count($rec)
return max($lengths)
```

Figure 11: Romeo and Juliet dialogs query.

```
declare variable $doc := doc("auction.xml");

declare function bidder($in as node()*) as node()*
{ let $b := $doc//open_auction
                  [seller/@person = $in/@id]
                /bidder/personref
  return $doc//people/person[@id = $b/@person]
};

for $p in $doc//people/person
return <person>
        { $p/@id }
        { data((with $x seeded by $p
                  recurse bidder($x))/@id) }
       </person>
```

Figure 10: XMark bidder network query.

cursion level which positively impacts the complexity of the value-based join inside recursion payload `bidder(·)`: for the huge network, *Delta* exactly feeds those 10 million nodes into `bidder(·)` that make up the result—*Naïve* repeatedly revisits intermediate results and processes 9 times as many nodes.

7.2 Romeo and Juliet Dialogs

Far less nodes are processed by a recursive expression that queries XML markup of Shakespeare's Romeo and Juliet[8] to determine the maximum length of any uninterrupted dialog (see Figure 11). Seeded with `SPEECH` element nodes, each level of the recursion expands the currently considered dialog sequences by a single `SPEECH` node given that the associated `SPEAKER`s are found to alternate (horizontal structural recursion along the `following-sibling` axis). Although the recursion is shallow (depth 6 on average), Table 2 shows how both, *MonetDB/XQuery* and *Saxon*, completed evaluation up to 3 times faster because the query had been specified in a distributive fashion.

7.3 Transitive Closures

Two more queries, taken directly from related work [25, 10], compute transitive closure problems (we generated the data instances with the help of ToXgene [5]). The first query implements a consistency check over the curriculum data (cf. Figure 1) and finds courses that are among their own prerequisites (Rule 5 in the Curriculum Case Study in Appendix B of [25]). Much like for the bidder network query, the larger the query input (medium instance: 800 courses, large: 4,000 courses), the better *MonetDB/XQuery* as well as *Saxon* exploited *Delta*.

The last query in the experiment explores 50,000 hospital patient records to investigate a hereditary disease [10]. In this case, the recursion follows the hierarchical structure of the XML input (from patient to parents), recursing into subtrees of a maximum depth of 5. Again, *Delta* makes a notable difference even for this computationally rather "light" query.

We believe that this renders this particular controlled form of XQuery recursion and its associated distributivity notion attractive, even for processors that do not implement a dedicated fixed point operator (like *Saxon*).

[8] `http://www.ibiblio.org/xml/examples/shakespeare/`

8. MORE RELATED WORK

Bringing adequate support for recursion to XQuery is a core research matter on various levels of the language. While the efficient evaluation of the recursive XPath axes (*e.g.*, `descendant` or `ancestor`) is well understood by now [3, 18], the optimization of recursive user-defined functions has been found to be tractable only in the presence of restrictions: [26, 14] propose exhaustive inlining of functions but require that functions are *structurally* recursive (use axes `child` and `descendant` to navigate into subtrees only) over *acyclic* schemata to guarantee that inlining terminates. Note that, beyond inlining, this type of recursion does not come packaged with an effective optimization hook comparable to what the inflationary fixed point offers.

The distinguished use case for inflationary fixed point computation is transitive closure. This is also reflected by the advent of XPath dialects like Regular XPath [29] and the inclusion of a dedicated `dyn:closure(·)` construct in the EXSLT function library [9]. We have seen applications

in Section 7 [25, 10] and recent work on data integration and XML views adds to this [11].

In the domain of relational query languages, *Naïve* is the most widely described algorithmic account of the inflationary fixed point operator [4]. Its optimized *Delta* variant, in focus since the 1980's, has been coined *delta iteration* [19], *semi-naïve* [4], or *wavefront* [20] strategy in earlier work. Since our work rests on the adaption of these original ideas to the XQuery Data Model and language, the large "relational body" of work in this area should be directly transferable, even more so in the Relational XQuery context.

The adoption of inflationary fixed point semantics by Datalog and SQL:1999 with its `WITH RECURSIVE` clause (Section 2) led to investigations of the applicability of *Delta* for these recursive relational query languages. For stratified Datalog programs [1], *Delta* is applicable in *all* cases: positive Datalog maps onto the distributive operators of relational algebra (π, σ, $\bowtie$, $\cup$, $\cap$) while stratification yields partial applications of the difference operator $x \setminus R$ in which R is fixed ($f(x) = x \setminus R$ is distributive).

SQL:1999, on the other hand, imposes rigid *syntactical* restrictions [24] on the iterative fullselect (recursion body) inside `WITH RECURSIVE` that make *Delta* applicable: grouping, ordering, usage of column functions (aggregates), and nested subqueries are ruled out, as are repeated references to the virtual table computed by the recursion. Replacing this coarse syntactic check by an algebraic distributivity assessment (Section 6) would render a larger class of queries admissible for efficient fixed point computation.

9. WRAP-UP

This paper may be read in two ways:

(i) As a proposal to add an inflationary fixed point construct, along the lines of `with-seeded by-recurse`, to XQuery (this topic has actually been discussed by the W3C XQuery working group in the very early XQuery days of 2001[9] but then dismissed because the group aimed for a first-order language design at that time).

(ii) As a guideline for query authors as well XQuery processor designers to check for and then exploit distributivity during the evaluation of recursive queries.

We have seen how such distributivity checks can be used to safely unlock the optimization potential, namely algorithm *Delta*, that comes tightly coupled with the inflationary fixed point semantics. *MonetDB/XQuery* implements this distributivity check on the algebraic level and significantly benefits whenever the *Delta*-based operator μ^{Δ} may be used for fixpoint computation. Even if the approach is realized on the coarser syntactic level *on top of* an existing XQuery processor, feeding back less nodes in each recursion level yields substantial performance improvements.

Remember that the distributivity notion suggests a divide-and-conquer evaluation strategy in which parts of a computation may be performed independently (before a merge step forms the final result). Beyond recursion, this may lead to improved XQuery compilation strategies for back-ends that can exploit such independence, *e.g*, set-oriented relational query processors (cf. loop-lifting [16]) as well as parallel or distributed execution platforms.

[9] `http://www.w3.org/TR/2001/WD-query-semantics-20010607/` (Issue 0008).

Acknowledgments. We thank Massimo Franceschet for input in early stages of this work. Loredana Afanasiev is supported by the Netherlands Organization for Scientific Research (NWO), Grant 017.001.190. Jan Rittinger is supported by the German Research Foundation (DFG), Grant GR 2036/2-1.

10. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[2] Loredana Afanasiev. Distributivity for XQuery Expressions. Technical report, University of Amsterdam, 2007.

[3] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, 2002.

[4] Francois Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. SIGMOD*, 1986.

[5] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. ToXgene: A template-based Data Generator for XML. In *Proc. SIGMOD*, 2002.

[6] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, 2007.

[7] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, 2006.

[8] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, 2007.

[9] EXSLT: A Community Initiative to Provide Extensions to XSLT.

[10] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. SMOQE: A System for Providing Secure Access to XML. In *Proc. VLDB*, 2006.

[11] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In *Proc. ICDE*, 2007.

[12] César A. Galindo-Legaria and Milind M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. SIGMOD*, 2001.

[13] Georg Gottlob and Christoph Koch. Monadic Queries over Tree-Structured Data. In *Logic in Computer Science*, pages 189–202, Los Alamitos, CA, USA, July22–25 2002. IEEE Computer Society.

[14] Maxim Grinev and Dmitry Lizorkin. XQuery Function Inlining for Optimizing XQuery Queries. In *Proc. ADBIS*, 2004.

[15] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE*, 2007.

[16] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.

[17] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Twente Data Management Workshop (TDM)*, 2004.

$$\frac{\Gamma;\mathsf{loop};\Delta \vdash e_1 \mapsto (q_1,\Delta_1) \qquad q_v \equiv \pi_{\mathsf{iter}:1,\mathsf{pos}:1,\mathsf{item}}(q_{bind}) \quad \Gamma + \{\$v \mapsto q_v\};\pi_{\mathsf{iter}}(q_v);\Delta_1 \vdash e_2 \mapsto (q_2,\Delta_2)}{\Gamma;\mathsf{loop};\Delta \vdash \texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2 \mapsto \left(\pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_1,\mathsf{item}:\mathsf{item}_2}\left(\varrho_{\mathsf{pos}_1:\langle\mathsf{pos},\mathsf{pos}_2\rangle/\mathsf{iter}}\left(q_1 \ltimes_{q_{bind}} \left(\pi_{\mathsf{pos}_2:\mathsf{pos},\mathsf{item}_2:\mathsf{item}}(q2)\right)\right)\right),\Delta_2\right)} \quad (\text{For}')$$

$$\frac{\Gamma;\mathsf{loop};\Delta \vdash e_1 \mapsto (q_1,\Delta_1) \qquad q_v \equiv \pi_{\mathsf{iter}:1,\mathsf{pos}:1,\mathsf{item}}(q_{bind}) \qquad q_p \equiv \pi_{\mathsf{iter}:1,\mathsf{pos}:1,\mathsf{item}:\mathsf{pos}_1}(q_{bind}) \quad \Gamma + \{\$v \mapsto q_v\} + \{\$p \mapsto q_p\};\pi_{\mathsf{iter}}(q_v);\Delta_1 \vdash e_2 \mapsto (q_2,\Delta_2)}{\Gamma;\mathsf{loop};\Delta \vdash \texttt{for } \$v \texttt{ at } \$p \texttt{ in } e_1 \texttt{ return } e_2 \mapsto \left(\pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_3,\mathsf{item}:\mathsf{item}_2}\left(\varrho_{\mathsf{pos}_3:\langle\mathsf{pos}_1,\mathsf{pos}_2\rangle/\mathsf{iter}}\left(\left(\varrho_{\mathsf{pos}_1:\langle\mathsf{pos}\rangle/\mathsf{iter}}(q_1)\right) \ltimes_{q_{bind}} \left(\pi_{\mathsf{pos}_2:\mathsf{pos},\mathsf{item}_2:\mathsf{item}}(q2)\right)\right)\right),\Delta_2\right)} \quad (\text{For-At})$$

$$\frac{\Gamma;\mathsf{loop};\Delta \vdash e_1 \mapsto (q_1,\Delta_1) \qquad \Gamma;\mathsf{loop};\Delta_1 \vdash e_2 \mapsto (q_2,\Delta_2)}{\Gamma;\mathsf{loop};\Delta \vdash e_1 \texttt{ union } e_2 \mapsto \left(\varrho_{\mathsf{pos}:\langle\mathsf{item}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{item}}(q_1 \cup q_2)\right),\Delta_2\right)} \quad (\text{Union})$$

Figure 12: Compilation rules For′, For-At, and Union.

[18] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB*, 2003.

[19] Ulrich Güntzer, Werner Kießling, and Rudolf Bayer. On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In *Proc. ICDE*, 1987.

[20] Jiawei Han, Ghassan Z. Qadah, and Chinying Chaou. The Processing and Evaluation of Transitive Closure Queries. In *Proc. EDBT*, 1988.

[21] Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. LiXQuery: A Foundation for XQuery Research. *SIGMOD Record*, 3(4), 2005.

[22] Michael Kay. The Saxon XSLT and XQuery Processor.

[23] Maarten Marx. Conditional XPath. *ACM Transactions on Database Systems (TODS)*, 30(4), 2005.

[24] Jim Melton and Alan R. Simon. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[25] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2), 2002.

[26] Chang-Won Park, Jun-Ki Min, and Chin-Wan Chung. Structural Function Inlining Technique for Structurally Recursive XML Queries. In *Proc. VLDB*, 2002.

[27] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[28] John Snelson. Higher Order Functions for XQuery, 2008. Technical Note submitted to W3C, `http://snelson.org.uk/~jpcs/higher_order_functions/index.cgi`.

[29] Balder ten Cate. Expressivity of XPath with Transitive Closure. In *Proc. PODS*, pages 328–337, 2006.

APPENDIX

A. ALGEBRAIC XQUERY COMPILATION

In [17] we describe a compilation procedure to transform arbitrary XQuery expressions into relational algebra. The core of this compiler is described in terms of inference rules that define function $\mapsto$ ("compiles to"). In these rules, a judgment of the form

$$\Gamma;\mathsf{loop};\Delta \vdash e \mapsto (q,\Delta')$$

indicates that, given

1. an environment Γ that maps the free XQuery variables $\$v$ in e to their algebraic representations q_v,
2. a relation loop that describes the iteration context of e,
3. a set of live (or reachable) XML nodes Δ,

the XQuery expression e compiles to the algebraic plan q with an associated set of possibly modified live nodes Δ'. The compilation procedure ensures that any such plan q evaluates to a ternary table with schema iter|pos|item which encodes the item sequence result of e. A row $\langle i,p,v\rangle$ in this table may invariably be read as "in iteration i, e assumes value v at the sequence position corresponding to p's rank in column pos."

XQuery `for` loops and variable binding. Here, to aid our distributivity test, we propose an alternative formulation of inference Rule For in [17] which is used to compile XQuery `for` loops of the form `for` $\$v$ `in` e_1 `return` e_2. The new Rule For′ makes the dependence of e_2 on the `for`-bound variable $\$v$ explicit in terms of the *dependent join* operator $\ltimes$ ($\mathcal{A}^{\times}$ in [12]). We adopt the operator's definition from [12]:

$$q_1 \ltimes_{q_{bind}} q_2 = \bigcup_{r \in q_1} \left(\{r\} \times q_2\left[\{r\}/q_{bind}\right]\right),$$

where $q[x/y]$ denotes the consistent replacement of free occurrences of y in q by x. This definition mirrors the semantics of the XQuery `for` loop construct: q_2 (the loop body) is treated like a function with parameter q_{bind} which is iteratively evaluated for each row r of table q_1. (Rule For-At shows how an XQuery `for` loop with positional variable `for` $\$v$ `at` $\$p\ldots$ may be compiled in terms of $\ltimes$.)

Figure 12 further includes compilation Rule Union which can compile XQuery's `union` operation over node sequences—this rule did not originally occur in [17].