Chapter 8

An Inflationary Fixed Point Operator for XQuery

In this chapter, we investigate a query processing technique for recursion in XQuery and use an experimental analysis to evaluate our approach. Our work is motivated by the current lack of declarative recursive operators in the language. We propose to introduce an inflationary fixed point (IFP) operator to XQuery and we present an efficient processing technique for it. This approach lifts the burden of optimizing recursive queries from the user’s shoulders and shifts it to the automatic (algebraic) query optimizer.

This chapter is organized as follows: we begin by formally introducing an IFP operator in the setting of XQuery in Section 8.2; in Section 8.3, we present two evaluation algorithms for IFP, one is the direct implementation of the IFP semantics and the other is an optimized variant; we define a distributivity property that allows us to optimize the IFP evaluation in Section 8.4; in Sections 8.5 and 8.6, we provide sufficient syntactic and algebraic conditions for distributivity; we present our implementation of the IFP operator in MonetDB/XQuery and show the gains of the optimization in Section 8.7; in Section 8.8, we address related work on recursion in XQuery as well as on the relational side of the fence; and finally we conclude and discuss future work in Section 8.9.

This chapter is based on work previously published in [Afanasiev et al., 2008, 2009].

8.1 Introduction

The backbone of the XML data model, namely ordered, unranked trees, is inherently recursive and it is natural to equip the associated query languages with constructs that can query such recursive structures. While XPath has a very restricted form of recursion via the recursive axes, e.g., ancestor and descendant, XQuery’s [World Wide Web Consortium, 2007] recursive user-defined functions...
Chapter 8. An Inflationary Fixed Point Operator for XQuery

Chapter 8. An Inflationary Fixed Point Operator for XQuery

(RUDFs) are the key ingredient of its Turing completeness. To obtain expressive power, the designers of the language took a giant leap, however. User-defined functions in XQuery admit arbitrary types of recursion, which makes recursion in the language procedural by nature—a construct that largely evades automatic optimization approaches beyond improvements like tail-recursion elimination or unfolding. This puts the burden of optimization on the user’s shoulders.

To make matters concrete, let us consider a typical example of a recursive data and information need.

8.1.1. Example. The DTD of Figure 8.1 (taken from [Nentwich et al., 2002]) describes recursive curriculum data, including courses, their lists of prerequisite courses, the prerequisites of the latter, and so on. Figure 8.2 shows an XML snippet of data that conforms to the given schema. A student wants to know what the courses are that (s)he needs to pass before being able to follow the course coded with "c1". This query cannot be expressed in XPath 2.0, while in XQuery it can be done only by means of recursive user-defined functions. The XQuery expression of Figure 8.3, for instance, uses the course element node with code "c1" to seed a computation that recursively finds all prerequisite courses, direct or indirect, of course "c1". For a given sequence $x$ of course nodes, function fix(·) calls out to rec_body(·) to find their prerequisites. As long as new nodes are encountered, fix(·) calls itself with the accumulated course node sequence.
8.1. Introduction

 declar function rec_body($cs) as node()*
  { $cs/id(./prerequisites/pre_code) }
};

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

 let $seed := doc("curriculum.xml")
  //course[@code="c1"]
 return fix(rec_body($seed))

Figure 8.3: An XQuery query for computing the prerequisites for the course "c1"
(❑❑❑ marks the fixed point computation).

Note that fix(·) implements a generic fixed point computation: only the initialization (let $seed := · · ·) and the recursion body rec_body(·) are specific to
the curriculum problem. The XQuery expression describes how to get the answer rather than what the answer is, i.e., the expression is procedural rather than declarative. The evaluation strategy encoded by fix(·) is not optimal, since it feeds already discovered course element nodes back into rec_body(·). There are many ways to optimize a fixed point computation, but this task is left to the user—the query engine has little chance to recognize and optimize the particular recursion operator this expression describes.

Another difficulty with the RUDFs is that they do not seem to fit into the algebraic framework commonly adopted by the database community for query optimization (e.g., Natix Physical Algebra (NPA) [Fiebig et al., 2002], or TAX, a tree algebra for XML used by the Timber engine [Jagadish et al., 2001]). Most XQuery engines have an underlying algebra that facilitates optimizations, but since there is no proper algebraic correspondent for user-defined recursive functions, these optimizations cannot be used for recursive queries. On the other hand, working with operators allows an engine to uniformly apply the algebraic reasoning framework.

Thus, the question we are facing is:

8.1. Question. What is a suitable declarative recursive operator in XQuery that is rich enough to cover interesting cases of recursion query needs and that allows for (algebraic) automatic optimizations?

In this chapter, we consider adding a declarative, and thus controlled, form of recursion to XQuery. Our choice falls on the inflationary fixed point (IFP) operator, familiar from the context of relational databases [Abiteboul et al., 1995].
While IFP imposes restrictions on the expressible types of recursion, it encompasses a family of widespread use cases of recursion in XQuery, including structural recursion (navigating recursively in one direction in the XML tree) and the more general and pervasive transitive closure (TC) on path expressions (capturing relations between XML nodes). In particular, XPath extended with IFP captures Regular XPath [ten Cate, 2006b], a version of XPath extended with a transitive closure operator. Most importantly, the IFP operator is susceptible to systematic optimizations.

Our goal is to define and implement an IFP operator in the setting of XQuery. We look at a standard optimization technique for IFP developed in the setting of relational databases and check whether it fits the new setting. The optimization consists of reducing the number of items that are fed into the recursion body, avoiding re-computation of items already obtained in previous steps of the recursion. Provided that the recursion body exhibits a specific distributivity property, this technique can be applied.

Unlike general user-defined XQuery functions, this account of recursion puts the query processor in control to decide whether the optimization may be safely applied. Distributivity may be assessed at a syntactic level—a non-invasive approach that can easily be realized on top of existing XQuery processors. Alternatively, if we adopt a version of relational algebra extended with a special tree aware operator and with the IFP operator for reasoning about XQuery queries (as in [Grust et al., 2004]), the seemingly XQuery-specific distributivity notion turns out to be elegantly modeled at the algebraic level.

To assess the viability of our approach in practice, we integrated the IFP operator into Pathfinder, an open-source XQuery compiler of the relational back-end MonetDB [Boncz et al., 2006a]. The compiler is part of the MonetDB/XQuery system, one of the fastest and most scalable XQuery engines today. Compliance with the restriction that IFP imposes on query formulation is rewarded by significant query runtime savings that the IFP-inherent optimization hook can offer. We document the effect for the XQuery processors MonetDB/XQuery [Boncz et al., 2006a] and Saxon [Kay, 2009].

8.2 Defining an Inflationary Fixed Point Operator for XQuery

In this section, we define an inflationary fixed point operator for XQuery expressions similar to the inflationary fixed point operator defined in the relational setting [Abiteboul et al., 1995]. We consider the XQuery fragment without recursive user defined functions.
8.2. Defining an Inflationary Fixed Point operator for XQuery

Throughout this chapter, 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 a safe substitution $e_1[e_2/\$x]$, i.e., the uniform replacement of all free occurrences of $\$x$ in $e_1$ by $e_2$ making sure that no other free variables get accidentally bound. Finally, $e_1(X)$ denotes 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 ($\sim$), a relaxed notion of equality for XQuery item sequences that disregards duplicate items and order, e.g., $(1,"a") \sim ("a",1,1)$. For $X_1, X_2$ sequences of type $\text{node()}*$, we have

$$X_1 \sim X_2 \iff \text{fs:ddo}(X_1) = \text{fs:ddo}(X_2),$$


To streamline the discussion, in the following we only consider XQuery expressions and sequences of type $\text{node()}*$. An extension of our definitions and results to general sequences of type $\text{item()}*$ is possible but requires the replacement of XQuery’s node set operations that we use ($\text{fs:ddo}()$, $\text{union}$ and $\text{except}$) with the corresponding operations on sequences of items.

8.2.1. Definition (Inflationary Fixed Point Operator). Let $e_{seed}$ and $e_{body}(\$x)$ be XQuery expressions. The inflationary fixed point (IFP) operator applied to $e_{body}(\$x)$ and $e_{seed}$ is the expression

$$\text{with } \$x \text{ seeded by } e_{seed} \text{ recurse } e_{body}(\$x). \quad (8.1)$$

The expressions $e_{body}$, $e_{seed}$, and $\$x$ are called, respectively, the recursion body, seed, and variable of the IFP operator.

The semantics of (8.1), called the IFP of $e_{body}(\$x)$ seeded by $e_{seed}$, is the sequence $Res_k$ obtained in the following manner:

$$Res_0 \leftarrow e_{body}(e_{seed})$$

$$Res_{i+1} \leftarrow e_{body}(Res_i) \text{ union } Res_i, \quad i \geq 0,$$

where $k \geq 1$ is the minimum number for which $Res_k \sim Res_{k-1}$. If no such $k$ exists, the semantics of (8.1) is undefined.

Note that if expression $e_{body}$ does not use node constructors (e.g., $\text{element} \{\cdot\} \{\cdot\}$ or $\text{text} \{\cdot\}$), expression (8.1) 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 (8.1) might be undefined. For example, $\text{with } \$x \text{ seeded by } () \text{ recurse } &a\{\$x\} &a$ 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 $\text{union}$.
8.2.2. Example. Using the new operator we can express the query from Example 8.1.1 in a concise and elegant fashion:

\[
\text{with } \$x \text{ seeded by doc("curriculum.xml")//}\n\text{course[@code="c1"] \quad (Q1)}\n\text{recurse } \$x/\text{id}(.//\text{prerequisites/pre_code})
\]

In XQuery, each specific instance of the IFP operator can be expressed via the recursive user-defined function template \(\text{fix}()\) (shown in \(\ldots\) in Figure 8.3). Since the IFP operator is a second-order construct taking an XQuery variable name and two XQuery expressions as arguments, the function \(\text{fix}()\) has to be interpreted as a template in which the recursion body \(\text{rec_body}()\) needs to be instantiated. Note that XQuery 1.0 does not support higher-order functions. Given this, Expression (8.1) is equivalent to the expression

\[
\text{let } \$x := e_{seed} \text{ return } \text{fix}(\text{rec_body}(\$x)).
\]

8.2.3. Definition. XQuery\(^{-rudf}\) is the XQuery fragment without recursive user defined functions, i.e., where the function dependency graph is acyclic. XQuery\(^{-rudf,+ifp}\) is XQuery\(^{-rudf}\) closed under the IFP operator.

8.2.1 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 [ten Cate, 2006b, Marx, 2004] extends the navigational fragment of XPath, Core XPath [Gottlob and Koch, 2002], with a transitive closure operator defined on location paths. We extend this definition to any XQuery expression of type \(\text{node()}\)*.

8.2.4. Definition (Transitive Closure). Let \(e\) be an XQuery expression. The transitive closure (TC) operator \((\cdot)^*\) applied to \(e\) is the expression \(e^*\). The semantics of \(e^*\) is the result of

\[
e \cup e/e \cup e/e/e \cup e/\ldots, \quad (TC)
\]

if it is a finite sequence. Otherwise, the semantics of \(e^*\) is undefined.

Analogously to the IFP 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.

8.2.5. Example. The TC operator applied to location paths expresses the transitive closure of paths in the tree:

\[
\begin{align*}
\text{(child::*)}^* & \equiv \text{descendant::*} , \\
\text{(child::*)}^*/\text{self::a} & \equiv \text{descendant::a} .
\end{align*}
\]
The TC operator applied to any expression of type \( \text{node()}^{*} \) expresses the transitive closure of node relations: the query from Example 8.1.1 can be expressed as

\[
\text{doc("curriculum.xml")//course[@code="c1"]/
{id(.//prerequisites/pre_code)})^*}.
\]

### 8.2.6. Definition.
XQuery\textsuperscript{-rudf,+tc} is the XQuery fragment without recursive user-defined functions extended with the TC operator, i.e., is the XQuery\textsuperscript{-rudf} fragment of XQuery closed under the TC operator.

Note that Regular XPath is a strict fragment of XQuery\textsuperscript{-rudf,+tc}—the expression in the scope of the TC operator in the curriculum example is a data-value join and cannot be expressed in Regular XPath.

### 8.2.7. Remark.
For some expression \( e \), the transitive closure of \( e \) can be expressed using the IFP operator as follows:

\[
e^* \equiv \text{with } x \text{ seeded by . recurse } x/e,
\]

where ‘.’ denotes the context node. In Section 8.4, we define a distributivity property for \( e \) that guarantees the correctness of this translation. We also show that all Regular XPath queries have this property, thus can be expressed in XQuery\textsuperscript{-rudf,+ifp} using this translation.

### 8.2.2 Comparison with IFP in SQL:1999

The IFP operator is present in SQL in terms of the \textsc{WITH RECURSIVE} clause introduced in the ANSI/ISO SQL:1999 standard \cite{Gulutzan and Pelzer, 1999}. The \textsc{WITH} clause defines a virtual table, while \textsc{RECURSIVE} specifies that the table is recursively defined. To exemplify this, consider the table \textsc{Curriculum(course, prerequisite)} as a relational representation of the curriculum data from Figure 8.2. The prerequisites \( P(\text{course, code}) \) of the course with code \( 'c1' \) expressed in Datalog are:

\[
P(x) \leftarrow \text{Curriculum('c1',x)} \\
P(x) \leftarrow P(y),\text{Curriculum(y,x)}.
\]

The equivalent SQL query is:

```sql
WITH RECURSIVE P(course_code) AS 
(SELECT prerequisite 
 FROM Curriculum 
 WHERE course = 'c1') 
UNION ALL 
(SELECT Curriculum.prerequisite 
 FROM P, Curriculum 
 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 result is added to \( P \). The computation of the body is iterated until \( P \) stops growing.

The SQL:1999 standard only 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. The SQL:1999 `WITH RECURSIVE` clause without syntactic restrictions, we call full recursion.

Note that the IFP operator introduced in Definition 8.2.1 does not state any syntactic restriction on the recursive body. In this respect, the IFP in XQuery-rdf, +ifp corresponds to full recursion in SQL. In Section 8.5, we define a syntactic restriction for IFP expressions in XQuery-rdf, +ifp that is similar to linear recursion in SQL.

8.3 Algorithms for IFP

In this section, we describe two algorithms, `Naïve` [Bancilhon and Ramakrishnan, 1986] and `Delta` [Güntzer et al., 1987], commonly used for evaluating IFP queries in the relational setting. `Delta` is more efficient than `Naïve`, but, unfortunately, `Delta` is not always correct for our IFP operator for XQuery.

8.3.1 Naïve

The semantics of the inflationary fixed point operator given in Definition 8.2.1 can be implemented straightforwardly. Figure 8.4(a) shows the resulting procedure, commonly referred to as `Naïve` [Bancilhon and Ramakrishnan, 1986]. At each iteration of the `while` loop, \( e_{body}(\cdot) \) is executed on the intermediate result sequence.
8.3 Algorithms for IFP

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 8.5: An XQuery formulation of Delta.

res until no new nodes are added to it. Note that the recursive function fix(·) shown in Figure 8.3 is the XQuery equivalent of Naïve. Another remark is that the old nodes in $res$ are fed into $e_{body}(·)$ over and over again. Depending on the nature of $e_{body}(·)$, Naïve may involve a substantial amount of redundant computation.

8.3.2 Delta

A folklore variation of Naïve is the Delta algorithm [Güntzer et al., 1987] of Figure 8.4(b). In this variant, $e_{body}(·)$ is invoked only for those nodes that have not been encountered in earlier iterations: the node sequence $\Delta$ is the difference between $e_{body}(·)$’s last answer and the current result $res$. In general, $e_{body}(·)$ will process fewer nodes. Thus, Delta introduces a significant potential for performance improvement, especially for large intermediate results and computationally expensive recursion bodies (see Section 8.7).

Figure 8.5 shows the corresponding XQuery user-defined function delta(·,·) which, for Example 8.1.1 and thus Query Q1, can serve as a drop-in replacement for the function fix(·)—line 14 needs to be replaced by

$$\text{return delta} (\text{rec_body} ($$seed$$),(),).$$

Unfortunately, Delta is not always a valid optimization for the IFP operator in XQuery. Consider the following examples.

8.3.1 Example. Consider expression (Q2) below.

$$\text{let } $seed := (<a/>,<b><c><d/></c></b>)$$
return
with $x$ seeded by $seed$
recurse
if( some $i$ in $x$ satisfies $i/self::a$ )
then $x/* else ()$$ \text{(Q2)}$$

While Naïve computes $(a,b,c,d)$, Delta computes $(a,b,c)$, where $a$, $b$, $c$, and $d$ denote the elements constructed by the respective subexpressions of the seed. The table below illustrates the progress of the iterations performed by both algorithms.
8.3.2. Example. Consider another expression:

```xml
with $x$ seeded by ()
recurse if( count($x) < 10 )
then <a>{$x}</a>
else ()
```

*Naïve* computes a sequence of 10 elements with the tag `a` of different structure: the first element is a tree containing one node, the last one is a tree of depth 9\[^3\]. On the other hand, *Delta* falls into an infinite loop and thus the result is *undefined*.

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

### 8.4 Distributivity for XQuery

In this section, we define a *distributivity property* for XQuery expressions. We show that distributivity implies the correctness of *Delta*. We also show that distributivity allows for the elegant translation of the TC operator into the IFP operator discussed in Section 8.2.1. Unfortunately, determining whether an expression is distributive is undecidable. In the next section though, we present an efficient and expressively complete syntactic approximation of distributivity.

#### 8.4.1 Defining distributivity

A function $e$ defined on sets is *distributive* if for any non-empty sets $X$ and $Y$, $e(X \cup Y) = e(X) \cup e(Y)$. This property suggests a divide-and-conquer evaluation strategy which consists of applying $e$ to subsets of its input and taking union of the results. We define a similar property for XQuery expressions using the sequence set-equality defined by \((\text{SetEq})\) in Section 8.2. Recall that in this chapter we only consider XQuery expressions and sequences of type \(\text{node()}\).\[^*\]

\[^3\]For the interested reader, this expression computes the tree encoding of the first 10 von Neumann numerals: the nodes represent sets of sets and the child relation represents the set membership.

\[^*\]For the interested reader, this expression computes the tree encoding of the first 10 von Neumann numerals: the nodes represent sets of sets and the child relation represents the set membership.
8.4. Distributivity for XQuery

8.4.1. Definition (Distributivity Property). Let \( e(x) \) be an XQuery expression. Expression \( e(x) \) is distributive for \( x \) iff for any non-empty sequences \( X_1, X_2 \),

\[
e(X_1 \cup X_2) \overset{\Delta}{=} e(X_1) \cup e(X_2)
\]  (8.2)

Note that if \( e \) does not contain node constructors and if \( e \) is constant for \( x \), i.e., \( x \) is not a free variable of \( e \), then Eq. (8.2) always holds, thus \( e \) is distributive for \( x \).

8.4.2. Proposition. Let \( e \) be an XQuery expression. Expression \( e(x) \) is distributive for \( x \) iff for any sequence \( X \neq () \) and any fresh variable \( y \),

\[
(\text{for } y \text{ in } x \text{ return } e(y))(X) \overset{\Delta}{=} e(X).
\]  (8.3)

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

\[
(e(x_1) \cup \ldots \cup e(x_n)) \overset{\Delta}{=} e(X).
\]  (8.4)

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 Eq. (8.2) holds then Eq. (8.4) also holds for \( X \), and vice versa. Thus Eq. (8.2) is equivalent to Eq. (8.4).

Conform XQuery formal semantics [World Wide Web Consortium, 2007b], for \( X = (x_1, \ldots, x_n), n \geq 1,\) the left-hand side of Eq. (8.3) equals the sequence concatenation \((e(x_1), \ldots, e(x_n))\). The later sequence is set-equal to \((e(x_1) \cup \ldots \cup e(x_n))\), the left-hand side of Eq. (8.4). From the equivalence of Eq. (8.2) and (8.4), the equivalence of Eq. (8.2) and (8.3) follows.

We will use Eq. (8.3) as an alternative definition of distributivity.

8.4.3. Proposition (Distributivity of Path Expressions). An XQuery expression of the form \( e(x) = x/p \) is distributive for \( x \) if the expression \( p \) neither contains (i) free occurrences of \( x \), nor (ii) calls to \( \text{fn:position()} \) or \( \text{fn:last()} \) that refer to the context item sequence bound to \( x \), nor (iii) node constructors.

Proof. Consider the XQuery Core [World Wide Web Consortium, 2007b] equivalent of \( x/p, \) \text{fs:ddo(for fs:dot in $x$ return $p$)}, which is set-equal to \( \text{for fs:dot in $x$ return $p$} \), where \( \text{fs:dot} \) is a built-in variable that represents the context item. Given conditions (i) to (iii), the left-hand side of Eq. (8.3), \((\text{for $y$ in $x$ return (for $fs:dot$ in $y$ return $p$))}(X)\) is set-equal to the right-hand side, \((\text{for $fs:dot$ in $x$ return $p$})\)(X), for any non-empty \( X \).

8.4.4. Example. Expressions of the form \( x/p \) where \( p \) is a Core XPath or even Regular XPath expression are examples of distributive expressions in XQuery. Note that, by definition, all Core XPath and Regular XPath expressions satisfy conditions (i) to (iii) of Proposition 8.4.3 above.
8.4.5. Example. 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 \((a)\), while
for \$i in \$x return \$i[1] evaluates to \((a, b)\).

In the next section, we establish the main benefit of distributivity, namely that
we can safely trade \textit{Na"ive} for \textit{Delta} for computing distributive IFP expressions.

8.4.2 Trading \textit{Na"ive} for \textit{Delta}
We say that \textit{Delta} and \textit{Na"ive} are
equivalent for a given IFP expression
if for any XML document (collection) both algorithms produce the same sequence of nodes
or both fall into infinite loops.

8.4.6. Theorem (\textit{Delta} computes IFP). Consider the expression with \$x
seeded by \texttt{e seed} recurse \texttt{e body} \$x). If \texttt{e body}(\$x) is distributive for \$x, then the
algorithm \textit{Delta} correctly computes the IFP of \texttt{e body}(\$x) seeded by \texttt{e seed}.

Proof. We show by inductive reasoning that \textit{Delta} and \textit{Na"ive} have the same
intermediate results, denoted by \texttt{res}_i^\Delta and \texttt{res}_i, respectively. The equivalence of
\textit{Na"ive} and \textit{Delta} follows from this. The induction is on \(i\), the iteration number
of the do···while loops.

In its first iteration, \textit{Na"ive} yields \texttt{e rec}(\texttt{e rec}(\texttt{e seed})) union \texttt{e rec}(\texttt{e seed}) which is
equivalent to \textit{Delta}'s first intermediate result (\texttt{e rec}(\texttt{e rec}(\texttt{e seed})) except \texttt{e rec}(\texttt{e seed}))
union \texttt{e rec}(\texttt{e seed}).

Suppose that \texttt{res}_k^\Delta = \texttt{res}_k, for all \(k \leq i\). We show that \texttt{res}_{i+1}^\Delta = \texttt{res}_{i+1}.

By the definition of \textit{Na"ive}, \texttt{res}_{i+1} = \texttt{e body}(\texttt{res}_i) union \texttt{res}_i. Since \texttt{e body}
is distributive for \$x, we can apply Set-Eq. (8.2) to \texttt{e body}(\texttt{res}_i)=\texttt{e body}(\texttt{res}_i except \texttt{Delta}_i) union \texttt{Delta}_i) and obtain

\[
\texttt{res}_{i+1} = (\texttt{e body}(\texttt{res}_i except \texttt{Delta}_i) union \texttt{e body}(\texttt{Delta}_i)) union \texttt{res}_i . \quad (8.5)
\]

Note that we are allowed to replace set-equality with strict equality here, since
both sequences are document ordered and duplicate free due to the semantics of
\texttt{union}.

By induction, \texttt{res}_i^\Delta = \texttt{res}_i and thus the right-hand side of (8.5) can be written
as \((\texttt{e body}(\texttt{res}_i^\Delta except \texttt{Delta}_i) union \texttt{e body}(\texttt{Delta}_i)) union \texttt{res}_i^\Delta). Note that \texttt{res}_i^\Delta is the
disjoin union of \texttt{res}_{i-1}^\Delta and \texttt{Delta}_i. As a result, (8.5) becomes

\[
\texttt{res}_{i+1} = \texttt{e body}(\texttt{res}_{i-1}^\Delta) union \texttt{e body}(\texttt{Delta}_i) union \texttt{res}_i^\Delta . \quad (8.6)
\]

By applying the induction step once more, we obtain

\[
\texttt{res}_{i+1} = \texttt{e body}(\texttt{res}_{i-1}) union \texttt{e body}(\texttt{Delta}_i) union \texttt{res}_i . \quad (8.7)
\]
Since $e_{\text{body}}(\text{res}_{i-1})$ is contained in $\text{res}_i$, it follows that the left-hand side of (8.7) equals $e_{\text{body}}(\Delta_i) \cup \text{res}_i$, which by induction equals $e_{\text{body}}(\Delta_i) \cup \text{res}_i^\Delta$. A final chain of equalities brings us the desired result:

$$
\text{res}_{i+1} = e_{\text{body}}(\Delta_i) \cup \text{res}_i^\Delta \\
= (e_{\text{body}}(\Delta_i) \text{ except } \text{res}_i^\Delta) \cup \text{res}_i^\Delta \\
= \Delta_{i+1} \cup \text{res}_i^\Delta \\
= \text{res}_i^\Delta \\
\text{(8.8)}
$$

QED

We have proven that $\Delta$ can be correctly applied for the evaluation of a distributive IFP expression. In the next section, we discuss one more benefit of distributivity, namely the correctness of the straightforward translation of the TC operator into the IFP operator discussed in Section 8.2.1.

### 8.4.3 Translating TC into IFP

Distributivity is also a key to understanding the relation between the TC operator and the IFP operator in the setting of XQuery. Intuitively, if expression $e$ is distributive for the context sequence, then $e^*$ is equivalent to $\text{with } \$x \text{ seeded by } . \text{ recurse } \$x/e$, where $\$x$, a fresh variable, is a place holder for the context sequence.

#### 8.4.7. Theorem

Consider an XQuery expression $e$ and a variable $\$x$, such that $\$x \notin \text{fv}(e)$. If $\$x/e$ is distributive for $\$x$, then

$$
e^* = \text{with } \$x \text{ seeded by } . \text{ recurse } \$x/e.
\text{(TC2IFP)}
$$

**Proof.** First, we rewrite Definition 8.2.4 of the TC operator. It is not hard to see that the semantics given by (TC') below it is equivalent to the semantics given by (TC)—it is merely a change in representation. Thus, we consider the semantics of $e^*$ to be the sequence of nodes $\text{Res}_{i,k}'$, if it exists, obtained in the following manner:

$$
\Theta_0' \leftarrow e \\
\text{Res}_0' \leftarrow e \\
\Theta_{i+1} \leftarrow \Theta_i/e \\
\text{Res}_{i+1}' \leftarrow \Theta_{i+1} \cup \text{Res}_i', \ i \geq 0,
\text{(TC')}
$$

where $k \geq 1$ is the minimum number for which $\text{Res}_{k}' \equiv \text{Res}_{k-1}'$. Otherwise, $e^*$ is **undefined**. Next, let us compare the $\text{Res}_{i}'$ sequences with the sequences $\text{Res}_i$ obtained conform Definition 8.2.1 while computing the IFP of $\$x/e$ seeded by . (the context node):

$$
\text{Res}_0 \leftarrow ./e \\
\text{Res}_{i+1} \leftarrow \text{Res}_i/e \cup \text{Res}_i, \ i \geq 0.
$$
Chapter 8. An Inflationary Fixed Point Operator for XQuery

If \( x/e \) is distributive for \( x \) then \( Res'_{i+1} = \Theta_i/e \) union \( Res'_i \) is equal to \( Res_i/e \) union \( Res_i = Res_{i+1} \). Below, we prove this equality rigorously. The correctness of the given translation of the TC operator into the IFP operator follows.

Let \( e_{\text{body}} = x/e \) and \( e_{\text{seed}} = . \) (the context node). Further, let \( Res_i, i \geq 0 \) be sequences obtained conform Definition 8.2.1 while computing the IFP of \( e_{\text{body}} \) seeded by \( e_{\text{seed}} \). Similarly, let \( Res'_i, i \geq 0 \) be sequences obtained conform (TC) while computing \( e^* \). We show by induction on \( i \) that \( Res_i \) equals \( Res'_i \). This proof is similar to the proof of Theorem 8.4.6.

The base of the induction holds straightforwardly:

\[
Res_1 = e_{\text{body}}(e_{\text{seed}}) \text{ union } e_{\text{seed}}(e_{\text{seed}})
= ./e/e \text{ union } ./e
= e/e \text{ union } e
= Res'_1.
\]

Suppose that \( Res_j = Res'_{j} \), for all \( j \leq i \). The equality \( Res_{i+1} = Res'_{i+1} \) is easily achieved by applying several times the induction hypothesis and the distributivity of \( e_{\text{body}} \):

\[
Res_{i+1} = e_{\text{body}}(Res_i) \text{ union } Res_i
= e_{\text{body}}(Res'_i) \text{ union } Res_i
= e_{\text{body}}(\Theta_i \text{ union } Res'_{i-1}) \text{ union } Res_i
= e_{\text{body}}(\Theta_i) \text{ union } e_{\text{body}}(Res'_{i-1}) \text{ union } Res_i
= e_{\text{body}}(\Theta_i) \text{ union } e_{\text{body}}(Res_{i-1}) \text{ union } Res_i
= e_{\text{body}}(\Theta_i) \text{ union } Res_i \text{ union } Res_i
= e_{\text{body}}(\Theta_i) \text{ union } Res_i \text{ union } Res_i
= e_{\text{body}}(\Theta_i) \text{ union } Res'_i
= Res'_{i+1}.
\]

Note that all the expressions above are unions of sequences of nodes, thus we can safely replace all set-equalities with strict equalities.

**8.4.8. REMARK.** Any Regular XPath expression \( e^* \) is equivalent to the IFP expression with \( x \) seeded by . recurse \( x/e \). Moreover, Delta correctly evaluates this expression.

**Proof.** Any Regular XPath expression \( x/e \) is of the form described by (i) to (iii) in Proposition 8.4.3, thus it is distributive for \( x \). Then, by Theorem 8.4.7, \( e^* \) is equivalent to with \( x \) seeded by . recurse \( x/e \). By Theorem 8.4.6, Delta correctly computes \( e^* \).

Distributivity gives us a clean translation of the TC operator into the IFP operator. In the general case, we do not know whether there is a translation from XQuery \( ^{\text{rudf},+tc} \) into XQuery \( ^{\text{rudf},+ifp} \) and vice versa. In Section 8.8, we will discuss...
related work on the expressive power of the TC and IFP operators in the context of XPath.

We have seen two benefits of distributivity: distributive TC expressions can be translated into distributive IFP expressions, and then Delta can be applied for their evaluation. Further, we need to be able to determine which expressions are distributive. Unfortunately, in the next section, we show that the distributivity property is undecidable.

### 8.4.4 Undecidability of the distributivity property

When we plan the evaluation of with $x$ seeded by $e_{seed}$ recur $e_{body}$, knowing the answer to “Is $e_{body}$ distributive for $x$?” allows us to apply Delta for computing the answer. Can we always answer this question, i.e., is there an implementable characterization—sufficient and necessary conditions—for the distributivity property? The answer is no.

**Theorem.** The problem of determining whether a given XQuery expression $e(x)$ is distributive for $x$ is undecidable.

**Proof.** Consider two arbitrary expressions $e_1$, $e_2$, in which $x$ does not occur free. If an XQuery processor could assess whether $\text{if (deep-equal}(e_1, e_2))$ then $x$ else $x[1]$ is distributive for $x$, it could also decide the equivalence of $e_1$ and $e_2$. Since the equivalence problem for XQuery is undecidable (as follows from the Turing-completeness of the language [Kepser, 2004]), determining whether an expression is distributive with respect to some variable is also undecidable. QED

In spite of this negative result, in practice, safe approximations of distributivity are still worth while to consider. In the next two sections, we present two such approximations, one at the syntactic level of XQuery, and the other at the level of an algebra underlying XQuery.

### 8.5 A syntactic approximation of distributivity

In this section, we define a syntactic fragment of XQuery parametrized by a variable $x$, called the **distributivity-safe fragment**. This syntactic fragment bares analogy with the syntactic fragment defined by the linearity condition in SQL:1999 (see Section 8.2.2). We show that all expressions that are distributivity-safe for a variable $x$ are distributive for that variable. Moreover, membership in this fragment can be determined in linear time with respect to the size of the expression. Since distributivity is undecidable (see Section 8.4.4), the distributivity-safe fragment does not contain all distributive expressions. We give an example of a distributive expression that is not distributivity-safe. Nevertheless, we show that this fragment is **expressively complete** for distributivity, i.e., any distributive
The XQuery expression is expressible in the distributivity-safe fragment of XQuery. Moreover, membership in this fragment can be determined in linear time with respect to the size of the expression.

In Section 8.5.1, we define the distributivity-safe fragment. We state the soundness of the fragment with respect to distributivity. We also state and prove the expressive completeness. In Section 8.5.2, we provide the proof of soundness.

8.5.1 The distributivity-safe fragment of XQuery

In the following, we define the distributivity-safe fragment of XQuery that implies distributivity. We consider LiXQuery \cite{Hidders et al., 2004}, a simplified version of XQuery. Then we define an auxiliary fragment of LiXQuery, the position and last guarded fragment. Finally, we define the distributivity-safe fragment as a fragment of LiXQuery.

LiXQuery is a simplified version of XQuery that preserves Turing-completeness. LiXQuery has a simpler syntax and data model than XQuery. It includes the most important language constructs, 3 basic types of items: \texttt{xs:boolean}, \texttt{xs:string}, and \texttt{xs:integer}; and 4 types of nodes: \texttt{element()}, \texttt{attribute()}, \texttt{text()}, and \texttt{document-node()}. The syntax of LiXQuery is given in Figure A.1 in Appendix A.

This language has a well-defined semantics and it was designed as a convenient tool for studying properties of the XQuery language. We consider static-type(\cdot) to be the mapping of LiXQuery expressions to their static type, conform XQuery’s formal semantics \cite{World Wide Web Consortium, 2007b}.

Let LiXQuery$^{−nc}$ be the fragment of LiXQuery without node constructors. We define an auxiliary fragment of LiXQuery$^{−nc}$ that contains the built-in functions \texttt{position()} and \texttt{last()} only as subexpressions of the second argument of the path and filter operators. In Section 8.5.2, we will relate this fragment to a semantic notion of context position and size independence.

8.5.1. Definition (Position and Last Guarded). An XQuery expression \( e \) is called position and last guarded, plg(\( e \)), if it can be generated using the syntactic rules in Figure 8.6.

The inference rules \texttt{Atomic} and \texttt{Closure} in Figure 8.6 define the LiXQuery$^{−nc}$ fragment that does not contain \texttt{position()} and \texttt{last()} at all, while the rules \texttt{Path} and \texttt{Filter} allow these two functions as subexpressions of the second argument of the path and filter operators.

Using the position and last guarded fragment, we define the distributivity-safe fragment of LiXQuery$^{−nc}$. But first, we give an intuition of this fragment.

Intuitively, we may apply a divide-and-conquer evaluation strategy for an expression \( e(\$x) \), if any subexpression of \( e \) processes the nodes in \( \$x \) one by one. The most simple example of such a subexpression is \texttt{for \$y in \$x return \( e(\$y) \)}, where \( e \) is in LiXQuery$^{−nc}$ and \( \$x \) does not occur free in \( e \). On the other hand, we may not apply a divide-and-conquer evaluation strategy if any subexpression
8.5. A syntactic approximation of distributivity

\[ e \text{-atomic} \quad e \neq \text{position}() \quad e \neq \text{last}() \]

\[ \ominus \in \{/, //\} \quad \text{plg}(e_1) \quad \text{plg}(e_2) \]

\[ \text{plg}(e_1 \ominus e_2) \]

\[ \ominus \text{-any operator or function name} \quad \text{plg}(e_i), 1 \leq i \leq n \]

\[ \text{plg}(\ominus (e_1, \ldots, e_n)) \]

**Figure 8.6**: Position and last guarded \( \text{plg}() \): A fragment of LiXQuery\(^{-nc} \) that contains \( \text{position}() \) and \( \text{last}() \) only in the second argument of the path and filter operators.

\[ ds_{\mathbf{x}}(\$\mathbf{x}) \]

\[ ds_{\mathbf{x}}(e) \]

\[ ds_{\mathbf{x}}(e_1) \]

\[ ds_{\mathbf{x}}(e_2) \]

\[ ds_{\mathbf{x}}(e_0) \]

\[ ds_{\mathbf{x}}(e_1[i \leq n+1]) \]

\[ ds_{\mathbf{x}}(\text{if } (e_1) \text{ then } e_2 \text{ else } e_3) \]

\[ ds_{\mathbf{x}}(\text{for } v \text{ at } p \text{ in } e_1 \text{ return } e_2) \]

\[ ds_{\mathbf{x}}(\text{for } v \text{ in } e_1 \text{ return } e_2) \]

\[ ds_{\mathbf{x}}(\text{let } v := e_1 \text{ return } e_2) \]

\[ ds_{\mathbf{x}}(\text{let } v := e_1 \text{ return } e_2) \]

\[ ds_{\mathbf{x}}(\text{let } v := e_1 \text{ return } e_2) \]

**Figure 8.7**: Distributivity-safety \( ds_{\mathbf{x}}(\cdot) \): A syntactic approximation of the distributivity property for LiXQuery\(^{-nc} \) expressions.
Chapter 8. An Inflationary Fixed Point Operator for XQuery

of $e$ inspects $\$x$ as a whole. Examples of such problematic subexpressions are $\text{count}(\$x)$ and $\$x[1]$, but also the general comparison $\$x = 10$ which involves existential quantification over the sequence bound to $\$x$.

Subexpressions whose value is \textit{independent} of $\$x$, on the other hand, are distributive. The only exception of this rule are XQuery’s node constructors, \textit{e.g.}, $\text{element} \{ \cdot \} \{ \cdot \}$, which create new node identities upon each invocation. With $\$x$ bound to $(<a/>, <b/>)$, for example,

$$\text{element}\{"c"}\{()\} \neq \text{for } y \text{ in } \$x \text{ return element}\{"c"\}\{()\},$$

since the right-hand side will yield a sequence of two distinct element nodes. This is the reason why we consider the distributivity-safe fragment inside LiXQuery$^{-nc}$.

\textbf{8.5.2. Definition (Distributivity-safety).} An XQuery expression $e$ is said to be \textit{distributivity-safe for} $\$x$, indicated by $\text{ds}_{\$x}(e)$, if it can be generated using the syntactic rules in Figure 8.7.

The rules $\texttt{Var}$ and $\texttt{Const}$ in Figure 8.7 define the base of the fragment, while the other inference rules define the closure with respect to a number of language operators. Note that the rules $\texttt{For1}$ and $\texttt{For2}$ ensure that the recursion variable $\$x$ occurs either in the body $e_2$ or in the range expression $e_1$ of a \texttt{for}-iteration but not both. This condition is closely related with the linearity constraint of SQL:1999 (for an in-depth discussion on this, see Section 8.8). A similar remark applies to Rules $\texttt{Let1}$, $\texttt{Let2}$, $\texttt{Path1}$ and $\texttt{Path2}$. One condition of $\texttt{Filter}$ involves $\texttt{plg}(\cdot)$ defined in Definition 8.5.1, another asks $e_2$ to be of static type $\texttt{xs:boolean}$. In order to check the latter, an engine should implement \textit{static type checking}. This is an optional functionality (note that the expressive completeness does not depend on this rule). Also note how the rule $\texttt{FunCall}$ requires the distributivity for every function argument of the function body if the recursion variable occurs free in that argument.

\textbf{8.5.3. Remark.} All the rules of Definitions 8.5.1 and 8.5.2 can be checked in parallel with a single traversal of the parse tree of a LiXQuery$^{-nc}$ expression. Checking membership of LiXQuery can thus be done in linear time with respect to the size of an XQuery expression (the size of an expression is the number of subexpressions, which equals the size of the parse tree).

Finally, we can state the property that we most desire of the distributivity-safe fragment, the soundness with respect to distributivity.

\textbf{8.5.4. Theorem (Soundness).} Any XQuery expression $e$ that is distributivity-safe for a variable $\$x$, \textit{i.e.}, for which $\text{ds}_{\$x}(e)$ holds, is also distributive for $\$x$. 
A formal proof in the settings of LiXQuery is given in Section 8.5.2. The proof is by induction on the structure of the expression.

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 expressively complete for distributivity.

**8.5.5. Proposition (Expressive completeness).** If an XQuery expression `e($x)` is distributive for `$x` and it 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 the rule `FOR2` (Figure 8.7) and Proposition 8.4.2. QED

Thus, at the expense of a slight reformulation of the query, we may provide a “syntactic distributivity hint” to an XQuery processor.

In the next section, we provide the proof of Theorem 8.5.4 in the setting of LiXQuery.

### 8.5.2 Distributivity-safety implies distributivity

In this section we prove the soundness of the distributivity-safety rules with respect to the distributivity property in the context of LiXQuery. The result transfers directly to XQuery, since LiXQuery is its fragment. Before proceeding with the proof of Theorem 8.5.4 we first cover the basics of the LiXQuery semantics. The complete definition can be found in [Hidders et al., 2004].

**LiXQuery in nutshell**

An *XML store*, denoted by `St`, contains the XML documents and collections that are queried, and also the XML fragments that are created during the evaluation of an expression. The *query evaluation environment* of an XML store, denoted by `En = (a, b, v, x, k, m)`, consists of:

- a partial function `a` that maps a function name to its formal arguments;
- a partial function `b` that maps a function name to its body;
- a partial function `v` that maps variable names to their values;
- `x`, which is undefined or an item of the XML store, and indicates the context item;
- `k`, which is undefined or an integer and gives the position of the context item in the context sequence;
• \( m \), which is undefined or an integer and gives the size of the context sequence.

We use \( En[a(n) \mapsto y] \) (\( En[b(n) \mapsto y] \), \( En[v(n) \mapsto y] \)) to denote the environment that is equal to \( En \) except that the function \( a \) (\( b \), \( v \)) maps the name \( n \) to the item \( y \). Similarly, we let \( En[x \mapsto y] \) (\( En[k \mapsto z] \), \( En[m \mapsto z] \)) denote changing the environment \( En \) only by attributing \( x \) a new item \( y \) (attributing \( k \), \( m \) a new integer value \( z \)).

We denote a sequence of items by \( S = (y_1, y_2, \ldots, y_n) \), the empty sequence by \( (\cdot) \), and the concatenation of two sequences \( S \) and \( S \) by \( S \). The set of items in a sequence \( S \) is \( \text{Set}(S) \). Given a sequence of nodes \( S \) in an XML store \( St \), we denote \( \text{Ord}_{St}(S) \) to be the unique sequence \( S' = (y_1', y_2', \ldots, y_n') \), such that \( \text{Set}(S) = \text{Set}(S') \) and \( y'_1 \ll_{St} \cdots \ll_{St} y'_n \), where \( \ll_{St} \) is a total order on the nodes of the store \( St \) denoting the document order. Using this notation, we can rewrite the equivalence \( \text{SetEq} \) into: \( S = S' \) iff \( \text{Ord}_{St}(\text{Set}(S)) = \text{Ord}_{St}(\text{Set}(S')) \).

Further, the semantics of LiXQuery is defined by a set of semantic rules. We write \( St, En \vdash e \Rightarrow V \) to denote that the evaluation of expression \( e \) against the XML store \( St \) and environment \( En \) of \( St \) results in a sequence \( V \) of values of \( St \). For an example of a semantic rule, let us take the Concatenation Rule [Hidders et al., 2004]:

\[
\frac{St, En \vdash e_1 \Rightarrow V', St', En \vdash e_2 \Rightarrow V''}{St, En \vdash e_1, e_2 \Rightarrow V' \circ V''}
\]

Given this, we can write the definition of distributivity for \( \$x \) in terms of LiXQuery semantics. Let \( St \) be a store, \( En[v(x) \mapsto (x_1, \ldots, x_n)] \) an environment that binds \( \$x \) to a non-empty sequence of items \( (x_1, \ldots, x_n) \). Applying the LiXQuery semantic rules on both sides of Eq. (8.2) of Definition 8.4.1 we obtain the following: \( V_1 \circ \cdots \circ V_n = V \), where \( St, En[v(x) \mapsto (x_1, \ldots, x_n)] \vdash e \Rightarrow V \) and \( St, En[v(x) \mapsto x_i] \vdash e \Rightarrow V_i \), for \( 1 \leq i \leq n \). Thus, \( e \) is distributive for \( \$x \) if

\( \text{Ord}_{St}(\text{Set}(V_1 \circ \cdots \circ V_n)) = \text{Ord}_{St}(\text{Set}(V)) \).

One last remark is that the path operators / and // are defined to be left associative, i.e., \( e_1/e_2/e_3 \) means \( (e_1/e_2)/e_3 \).

Proving the soundness

Before giving the proof of Theorem 8.5.4 we define a notion of context position and size independence and prove two lemmas.

8.5.6. Definition (Context Position and Size Independence). A LiXQuery expression \( e \) is context position and size independent (c.p. and s. ind.) if

\[\text{In fact, the semantic rules of LiXQuery are of the form } St, En \vdash e \Rightarrow (St', V), \text{ where } St' \text{ might be a new XML store and } V \text{ is a sequence of values of } St'. \text{ But since we consider LiXQuery without node constructors, the evaluation of expression } e \text{ against the XML store } St \text{ and the environment } En \text{ results always in the same XML store } St \text{ and a sequence } V \text{ of values of } St.\]
for any XML store $S_t$, environment $E_n$, and any sequence of items $V$, we have $S_t, E_n \vdash e \Rightarrow V$ if and only if $S_t, E_n[m \mapsto 1][k \mapsto 1] \vdash e \Rightarrow V$.

In other words, $e$ is c.p. and s. ind. if no matter what the values for the environment parameters $m$ and $k$ are, $e$ evaluates to the same result as if $m$ and $k$ are set to 1. An expression that is c.p. and s. ind. does not use other information about the context sequence than the context item. We can interpret this property as distributivity for the context sequence.

The expression $a/b/c$ is obviously c.p. and s. ind., while $\text{position}()>1$ is not, if the context sequence contains more than one item. In fact, the position and last guarded fragment defined by $\text{plg}()$ (Definition 8.5.1) is an approximation of the context position and size independent fragment of LiXQuery:

**8.5.7. Lemma.** Any LiXQuery expression $e$ that is position and last guarded, i.e., $\text{plg}(e)$, is also c.p. and s. independent.

**Proof.** The proof goes by induction on the structure of $e$. The base case consists of checking the implication for atomic expressions satisfying the rule $\text{Atomic}$ in Figure 8.6. Below, we list all atomic expressions in LiXQuery, grouped by clause in the BNF definition of the language:

- $(\text{Var})$: "\$" $(\text{Name})$
- $(\text{BuiltIn})$: "true()", "false()", "position()", "last()"
- $(\text{Step})$: ".", ".", $(\text{Name})", @$(\text{Name})", ",@*", "text()"
- $(\text{Literal})$: $(\text{String}), (\text{Integer})$
- $(\text{EmpSeq})$: ""

By the conditions of Rule $\text{Atomic}$, $e$ cannot be $\text{position}()$ or $\text{last}()$. None of the semantic rules for the remaining atomic expressions refer to $k$ or $m$, thus $e$ evaluates to the same sequence of items irrespective of the value of these parameters: $S_t, E_n \vdash e \Rightarrow V$ and $S_t, E_n[k \mapsto 1][m \mapsto 1] \vdash e \Rightarrow V$, for any $S_t$ and $E_n$.

Next, we prove the induction step for the expressions defined by Rules $\text{Path}$ and $\text{Filter}$ and $\text{Closure}$

**Path and Filter expressions.** Let $e = e_1 \odot e_2$, where $\odot \in \{/, //, []\}$, and let $\text{plg}(e)$. By the rules $\text{Path}$ and $\text{Filter}$, $\text{plg}(e_1)$. Suppose also that $e_1$ is c.p. and s. independent. We prove that $S_t, E_n \vdash e_1 \odot e_2$ is equivalent with $S_t, E_n[k \mapsto 1][m \mapsto 1] \vdash e_1 \odot e_2$. By the semantic rules for both path and filter expressions (see Rules (18) and (17), in [Hidders et al., 2004]), $e_1$ is evaluated first: $S_t, E_n \vdash e_1 \Rightarrow (x_1, \ldots, x_m)$. Since $e_1$ is c.p. and s. independent, $S_t, E_n[k \mapsto 1][m \mapsto 1] \vdash e_1 \Rightarrow (x_1, \ldots, x_m)$. Further, $e_2$ is evaluated for each item in the result sequence of $e_1$: $S_t, E_n[x \mapsto x][k \mapsto i][m \mapsto m] \vdash e_2, 1 \leq i \leq m$. Note that the values of $k$ and $m$ are changed by the semantics of these operators and that the result of the evaluation of $e_2$ does not depend on the initial context position.
and size. Thus, no matter which of the three operators we consider, the end result of \(e\) evaluated against \(St\) and \(En\) is the same as evaluated against \(St\) and \(En[\mathbf{k} \mapsto 1][\mathbf{m} \mapsto 1]\).

**Other expressions.** Let \(e = \odot(e_1, \ldots, e_n), n \geq 1\) be a complex expression, where \(\odot(\cdot, \ldots)\) is any operator or function in the language. Suppose \(plg(e)\) then by the rule **Closure** \(plg(e_i), 1 \leq i \leq n\). Suppose also that \(e_i\) is c.p. and s. independent. For \(\odot\) equal to one of the path operators or the filter operator, we have already proved that \(e\) is c.p. and s. independent. The semantic rules of the remaining operators and functions in the languages do not refer to the parameters \(\mathbf{k}\) and \(\mathbf{m}\), thus \(e\) is trivially c.p. and s. independent. QED

**8.5.8. Lemma.** For any LiXQuery expression \(e\) and variable \(\mathbf{x} \notin \text{fv}(e)\), \(e\) is distributive for \(\mathbf{x}\).

**Proof.** Let \(St\) be an XML store and \(En\) an environment that binds \(\mathbf{x}\) to a non-empty sequence of size \(n\). Suppose that \(St, En \vdash e \Rightarrow (x_1, \ldots, x_m)\). In this case, the result of the corresponding for-expression is a sequence constructed by concatenating \((x_1, \ldots, x_m)\) \(n\) times: \(St, En \vdash \text{for } \mathbf{y} \text{ in } \mathbf{x} \text{ return } e \Rightarrow (x_1, \ldots, x_m) \odot (x_1, \ldots, x_m) \odot \cdots \odot (x_1, \ldots, x_m)\), which is set-equal to \((x_1, \ldots, x_m)\).

Note that \(\mathbf{y} \notin \text{fv}(e)\) and the result of \(e\) does not depend on the binding of \(\mathbf{y}\).

QED

**Proof of Theorem 8.5.4.** As before, the proof goes by induction on the structure of \(e\).

The base case consists of checking the implication for atomic expressions and for expressions that do not contain \(\mathbf{x}\) as a free variable (constant w.r.t. \(\mathbf{x}\)). First, suppose \(e\) is an expression for which \(\mathbf{x} \notin \text{fv}(e)\). By the rule **Const** in Figure 8.7, \(e\) is distributivity-safe for \(\mathbf{x}\) and by Lemma 8.5.8, \(e\) is distributive for \(\mathbf{x}\). Second, suppose \(e = \mathbf{x}\), distributivity-safe for \(\mathbf{x}\) by the rule **Var**. Let \(St\) be a store and \(En\) an environment that binds \(\mathbf{x}\) to the non-empty sequence of items \((x_1, \ldots, x_n)\), then \(St, En \vdash \mathbf{x} \Rightarrow (x_1, \ldots, x_n)\) and \(St, En \vdash \text{for } \mathbf{y} \text{ in } \mathbf{x} \text{ return } \mathbf{y} \Rightarrow (x_1) \odot \cdots \odot (x_n)\). Thus expression \(e\) is distributive for \(\mathbf{x}\).

The induction step consists of checking the implication for the complex expressions defined by the rest of the distributivity-safety rules in Figure 8.7. The induction hypothesis (IH) is: any distributive-safe for \(\mathbf{x}\) subexpression of \(e\) is distributive for \(\mathbf{x}\). We suppose that \(\mathbf{x} \in \text{fv}(e)\), otherwise \(e\) was already considered in the base case. Further, let \(St\) be a store, \(En\) an environment that binds \(\mathbf{x}\) to the non-empty sequence of items \((x_1, \ldots, x_n)\).

**If expressions.** Suppose \(e = \text{if } (e_1) \text{ then } e_2 \text{ else } e_3\) and \(ds_{\mathbf{x}}(e)\), then by the rule **If** \(ds_{\mathbf{x}}(e_2), ds_{\mathbf{x}}(e_3)\), and \(\mathbf{x} \notin \text{fv}(e_1)\). Let \(St, En \vdash e_1 \Rightarrow b, St, En \vdash e_2 \Rightarrow V\) and \(St, En \vdash e_3 \Rightarrow V'\), where \(b\) is a boolean value, \(V\) and \(V'\) are sequences of items.

Suppose \(b\) is true, then \(St, En \vdash e \Rightarrow V\) (otherwise, \(St, En \vdash e \Rightarrow V'\)). By Lemma 8.5.8, \(e_1\) is distributive for \(\mathbf{x}\), thus it yields the same boolean value for any
8.5. A syntactic approximation of distributivity

binding of $x$ to a singleton $x_i$: $St, En[v(x) \mapsto x_i] \vdash e_1 \Rightarrow true$, $1 \leq i \leq n$. From this we obtain: if $St, En[v(x) \mapsto x_i] \vdash e_2 \Rightarrow V_i$ then $St, En[v(x) \mapsto x_i] \vdash e \Rightarrow V_i$, for any $1 \leq i \leq n$. By the IH, $e_2$ is distributive for $x$, so $V \mathrel{*} V_1 \circ \cdots \circ V_n$. The reasoning is identical when $b$ is false, thus $e$ is distributive for $x$.

**Type-switch expressions.** The proof for the type-switch expressions that are defined by Rule \textbf{[TypeSw]} is similar to the proof for if-expressions.

**Path expressions.** Suppose $e = e_1/e_2$ (the case for ‘/’ is identical) and $ds_{x}(e)$, then $e$ must satisfy Rule \textbf{[Path1]} or Rule \textbf{[Path2]}

Suppose $e$ satisfies Rule \textbf{[Path1]}, then $x \notin fv(e_1)$ and $ds_{x}(e_2)$. First, let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and, since $e_1$ is constant w.r.t. $x$, $St, En[v(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$, $1 \leq i \leq n$. Second, let $St, En[x \mapsto y_j] \vdash e_2 \Rightarrow V_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$. By the IH, $e_2$ is distributive for $x$ and thus, $V_j = V_{i,j} \circ \cdots \circ V_{n,j}$, for $1 \leq j \leq m$. Following the semantic rule for path expressions in [Hidders et al., 2004], $St, En \vdash e \Rightarrow \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_j))$ and $St, En[v(x) \mapsto x_i] \vdash e \Rightarrow \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_{i,j})\bigcup_{1 \leq j \leq m} \text{Set}(V_{i,j}))$, finally, from the distributivity for $x$ of $e_2$, it follows that

$$\text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_j)) = \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_{i,j})) \circ \cdots \circ \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_{n,j}))$$

which means that $e$ is distributive for $x$.

In the other case, if $e$ satisfies Rule \textbf{[Path2]}, then $ds_{x}(e_1)$, $x \notin fv(e_2)$ and $plg(e_2)$. Let $St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$ and $St, En[v(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1, \ldots, y_m)$, $1 \leq i \leq n$. By the IH, $e_1$ is distributive for $x$, thus $(y_1, \ldots, y_m) = (y_1, \ldots, y_1) \circ \cdots \circ (y_1, \ldots, y_m)$, which means that for any $1 \leq k \leq m$, exists $1 \leq i \leq n$ and $1 \leq j \leq m$, such that $y_k = y_j$, and vice versa, for any $1 \leq i \leq n$ and $1 \leq j \leq m$, exists $1 \leq k \leq m$ with the same property. Next, let $St, En[x \mapsto y_k] \vdash e_2 \Rightarrow V_k$ and $St, En[x \mapsto y_j] \vdash e_2 \Rightarrow V_{i,j}$, for respective $k$, $i$, and $j$. Note that during the evaluation of $e_2$ we disregard the values of the parameters $k$ and $m$, and the binding of the variable $x$: the former is allowed by the fact that $plg(e_2)$ and, by Lemma 8.5.7, $e_2$ is c.p. and s. independent; the latter is allowed by the fact that $e_2$ is constant w.r.t. $x$. Finally, observe that for any $1 \leq k \leq m$, there is $1 \leq i \leq n$ and $1 \leq j \leq m$, such that $V_k = V_{i,j}$, and vice versa. This implies that $\text{Ord}_{St}(\bigcup_{1 \leq k \leq m} \text{Set}(V_k)) = \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_{i,j})) \circ \cdots \circ \text{Ord}_{St}(\bigcup_{1 \leq j \leq m} \text{Set}(V_{n,j}))$, which means that $e$ is distributive for $x$.

For expressions. Suppose $e$ is a for-expression and $ds_{x}(e)$, then $e$ must satisfy Rule \textbf{[For1] or Rule [For2]} in Figure 8.7. If $e$ satisfies Rule \textbf{[For1]} then the proof is similar to the proof for the path expressions satisfying Rule \textbf{[Path1]}. And if $e$ satisfies Rule \textbf{[For2]} then the proof is similar to the one for path expressions satisfying Rule \textbf{[Path2]}. Note that such expressions do not contain the positional variable “at $p$”. This condition forms the counterpart of the position and last guarded condition in the case of path expressions.
Let expressions. Suppose \( e = \text{let } \$v := e_1 \text{ return } e_2 \) and \( ds_{\$x}(e) \). By the IH, \( e_2 \) is distributive for \( \$x \) and the distributivity for \( \$x \) of \( e \) follows straightforwardly.

Suppose \( e \) satisfies Rule \[\text{LET1}\] then \( \$x \not\in fv(e_1) \) and \( ds_{\$x}(e_2) \). By the IH, \( e_2 \) is distributive for \( \$x \) and the distributivity for \( \$x \) of \( e \) follows straightforwardly.

Suppose \( e \) satisfies Rule \[\text{LET2}\], then \( ds_{\$x}(e_1) \), \( \$x \not\in fv(e_2) \) and \( ds_{\$x}(e_2) \). By the IH, \( e_1 \) is distributive for \( \$x \) and \( e_2 \) is distributive for \( \$v \). Let \( St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m) \) and \( St, En[v(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1^1, \ldots, y_i^i, \ldots, y_m^m), 1 \leq i \leq n \). Since \( e_2 \) is distributive for \( \$x \), \( (y_1, \ldots, y_m) \mapsto (y_1^1, \ldots, y_m^m \circ \cdots \circ (y_1^n, \ldots, y_m^n) \), which means that for any \( 1 \leq k \leq m \), exists \( 1 \leq i \leq n \) and \( 1 \leq j \leq m_i \), such that \( y_k = y_{i}^j \), and vice versa, for any \( 1 \leq i \leq n \) and \( 1 \leq j \leq m_i \). Last, let \( St, En[v(x) \mapsto x_i], En[v(x) \mapsto y_k] \vdash e_2 \Rightarrow V_j \), for \( 1 \leq i \leq n \), \( 1 \leq j \leq m_i \). Note that since \( e_2 \) is constant w.r.t. \( \$x \), the binding of this variable does not influence the result of \( e_2 \). And again, since \( e_2 \) is distributive for \( \$v \), it follows that \( V^i = V_1^i \circ \cdots \circ V_{m_i}^i, 1 \leq i \leq n \). We saw before that for any \( k \), there exist \( i \) and \( j \), such that \( y_k = y_{i}^j \), which implies \( V_k = V_{i}^j \), and vice versa, for any \( i \) and \( j \), there exists a \( k \), with the same property. This, finally, implies that \( V = V_1 \circ \cdots \circ V_m = V_1^1 \circ \cdots \circ V_{m_1}^1 \circ \cdots \circ V_{m_n}^n = V^1 \circ \cdots \circ V^n \), which means that \( e \) is distributive for \( \$x \).

Filter expressions. Suppose \( e = e_1 \{ e_2 \} \) and \( ds_{\$x}(e) \), then by the rule \[\text{FILTER}\] \( ds_{\$x}(e_1) \), \( \$x \not\in fv(e_2) \), \( plg(e_2) \) and \( e_2 \) is static type \( \text{xs:boolean} \). Let \( St, En \vdash e_1 \Rightarrow (y_1, \ldots, y_m) \) and \( St, En[v(x) \mapsto x_i] \vdash e_1 \Rightarrow (y_1^1, \ldots, y_i^i, \ldots, y_m^m), 1 \leq i \leq n \). By the IH, \( e_1 \) is distributive for \( \$x \), thus \( (y_1, \ldots, y_m) \mapsto (y_1^1, \ldots, y_m^m \circ \cdots \circ (y_1^n, \ldots, y_m^n) \), which means that for any \( 1 \leq k \leq m \), there exists \( 1 \leq i \leq n \) and \( 1 \leq j \leq m_i \), such that \( y_k = y_{i}^j \), and vice versa, for any \( 1 \leq i \leq n \) and \( 1 \leq j \leq m_i \), there exists \( 1 \leq k \leq m \) with the same property. Next, let \( St, En[x \mapsto y_k] \vdash e_2 \Rightarrow b_k \) and \( St, En[x \mapsto y_j^i] \vdash e_2 \Rightarrow b_j^i \), where \( b_k \) and \( b_j^i \) are booleans, for all \( k, i, j \). Note that during the evaluation of \( e_2 \) we disregard the values of the parameters \( b \) and \( m \), and the binding of the variable \( \$x \): the former is allowed by the fact that \( plg(e_2) \) and, by Lemma \[\text{S.5.1} \] \( e_2 \) is c.p. and s. independent; the latter is allowed by the fact that \( e_2 \) is constant w.r.t. \( \$x \). It is clear that if \( y_k = y_{i}^j \) then \( b_k = b_j^i \). This means that \( y_k \) is contained in the result of \( e \) evaluated against \( St \) and \( En \), iff \( y_j^i \) is contained in the concatenation of the results of \( e \) evaluated against \( St \) and \( En[v(x) \mapsto x_i] \). Thus, the respective result sequences are set-equal and \( e \) is distributive for \( \$x \).

Other expressions. Suppose \( e = e_1 \oplus e_2 \), where \( \oplus \in \{\,, \} \) or \( e = f(e_1, \ldots, e_l) \), a function call, and \( ds_{\$x}(e) \). By the rules \[\text{CONCAT}\] and \[\text{FUNCTION}\] \( ds_{\$x}(e_i) \), for all \( 1 \leq i \leq l \). Then the distributivity for \( \$x \) of \( e \) follows directly from the distributivity for \( \$x \) of \( e_i \), which follows from the IH. QED
8.6 An algebraic approximation of distributivity

XQuery is a syntactically rich language with many equivalent ways of expressing the same information need. Reasoning about the queries at the syntactic level is cumbersome. Most XQuery engines adopt algebras as a convenient formalism for query normalization and optimization. One would also expect that reasoning about distributivity is more elegant at the algebraic level.

In this section, we follow an algebraic route for checking the applicability of $\Delta$ for the evaluation of the IFP of an XQuery expression $e_{body}$. We adopt a relational approach to XML data modeling and XQuery evaluation, and instead of performing the distributivity test at the syntactic level, we inspect the relational algebraic query plan compiled for $e_{body}$. As we would expect, the algebraic representation of $e_{body}$ renders the check for the distributivity property particularly robust and simple.

In the following, we first sketch the relational approach to XQuery that we follow. Then we define an algebraic distributivity property that is equivalent to the XQuery distributivity property and present an incomplete but effective test for it. We also discuss this approach in comparison with the syntactic approach presented in the previous section.

**Relational XQuery.** The alternative route we take in this section builds on the Pathfinder project, which fully implements a purely relational approach to XQuery. Pathfinder compiles instances of the XQuery Data Model (XDM) and XQuery expressions into relational tables and algebraic plans over these tables, respectively, and thus follows the dashed path in Figure 8.8. The translation strategy (i) preserves the XQuery semantics (including compositionality, node identity, iteration and sequence order), and (ii) yields relational plans which rely on regular relational query engine technology [Grust et al., 2004].

The compiler emits a dialect of relational algebra that mimics the capabilities of modern SQL query engines. The algebra operators are presented in Table 8.1. The row numbering operator $\rho_{a:/b_1,\ldots,b_n}/p$ compares with SQL:1999’s `ROW_NUMBER()` OVER (PARTITION BY p ORDER BY b_1,\ldots,b_n) and correctly implements the order semantics of XQuery on the (unordered) algebra. Other non-
Chapter 8. An Inflationary Fixed Point Operator for XQuery

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_{a_1:b_1,...,a_n:b_n}$</td>
<td>project onto cols $a_i$, rename $b_i$ into $a_i$</td>
</tr>
<tr>
<td>$\sigma_{b}$</td>
<td>select rows with column $b = \text{true}$</td>
</tr>
<tr>
<td>$\bowtie_p$</td>
<td>join with predicate $p$</td>
</tr>
<tr>
<td>$\bowtie_q$</td>
<td>iterated evaluation of rhs argument (APPLY)</td>
</tr>
<tr>
<td>$\times$</td>
<td>Cartesian product</td>
</tr>
<tr>
<td>$\cup$</td>
<td>union</td>
</tr>
<tr>
<td>$\setminus$</td>
<td>difference</td>
</tr>
<tr>
<td>$\text{count}_{a:/b}$</td>
<td>aggregates (group by $b$, result in $a$)</td>
</tr>
<tr>
<td>$\bowtie_{a:(b_1,...,b_n)}$</td>
<td>$n$-ary arithmetic/comparison operator $\circ$</td>
</tr>
<tr>
<td>$\bowtie_{a:(b_1,...,b_n)}/{p}$</td>
<td>ordered row numbering (by $b_1,...,b_n$)</td>
</tr>
<tr>
<td>$\bowtie_{\alpha:n}$</td>
<td>XPath step join (axis $\alpha$, node test $n$)</td>
</tr>
<tr>
<td>$\varepsilon,\tau,...$</td>
<td>node constructors</td>
</tr>
<tr>
<td>$\mu,\mu^\Delta$</td>
<td>fixpoint operators</td>
</tr>
</tbody>
</table>

Table 8.1: Relational algebra dialect emitted by the Pathfinder compiler.

Textbook operators, like $\varepsilon$ or $\bowtie$, are merely macros representing “micro plans” composed of standard relational operators: expanding $\bowtie_{\alpha:n}(q)$, for example, would reveal $\text{doc} \bowtie_p q$, where $p$ is a conjunctive range predicate that realizes the semantics of an XPath location step along axis $\alpha$ with node test $n$ between the context nodes in $q$ and the encoded XML document $\text{doc}$. Dependent joins $\bowtie$—also named CROSS APPLY in Microsoft SQL Server’s SQL dialect Transact-SQL—like $\bowtie$ are a logical concept and can be replaced by standard relational operators

The plans operate over relational encodings of XQuery item sequences held in flat (1NF) tables with an $\text{iter}|\text{pos}|\text{item}$ schema. In these tables, columns $\text{iter}$ and $\text{pos}$ are used to properly reflect for-iteration and sequence order, respectively. Column $\text{item}$ carries encodings of XQuery items, i.e., atomic values or nodes. The inference rules driving the translation procedure from XQuery expressions into algebraic query plans are described in [Grust et al., 2004] and [Afanasiev et al., 2009]. 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. Further details of Relational XQuery do not affect our present discussion of distributivity or IFP evaluation and may be found in [Grust et al., 2004, Afanasiev et al., 2009].

In the following, we extend the algebra with two algebraic fixed point operators corresponding to the fixed point computation algorithms discussed in Section 8.3 and assess distributivity based purely on algebraic equivalences.
8.6.1 An algebraic account of distributivity

An occurrence of the new with $x$ seeded by $e_{seed}$ recur $e_{body}$ form in a source XQuery expression will be compiled into a plan fragment as shown here on the right. In the following, let $q$ denote the algebraic query plan that has been compiled for XQuery expression $e$. Operator $\mu$, the algebraic representation of the algorithm Na"{i}ve (Figure 8.4(a)), iterates the evaluation of the algebraic plan for $e_{body}$ and feeds its output $\varphi$ back to its input $\iota$ until the IFP is reached. If we can guarantee that the plan for $e_{body}$ is distributive, we may safely trade $\mu$ for its Delta-based variant $\mu^\Delta$ which, in general, will feed significantly less items back in each iteration (see Figure 8.4(b) and Section 8.7).

In Section 8.4, we defined the distributivity property of XQuery expressions based on the XQuery operator union (see Definition 8.4.1). In the algebraic setting, the XQuery union operation is compiled to the following expression that implements the XQuery order requirements—for each iteration the result is ordered by the node rank in column item (see Afanasiev et al. 2009 for the compilation rule):

$$ q_{\text{pos:(item)/iter}} \\
\; \\
\; \\
\; q_1 \cup q_2 $$

A straightforward application of this translation to Definition 8.4.1 allows us to express the distributivity criterion based on the equivalence of relational plans. If we can prove the set-equality of the two plans in Figure 8.9(a), we know that the XQuery expression $q_{body}$ must be distributive. This equality is the algebraic expression of the divide-and-conquer evaluation strategy: evaluating $e_{body}$ over a composite input (left-hand side, $_1\cup_2$) yields the same result as the union of the evaluation of $e_{body}$ over a partitioned input (right-hand side).

Given that the distributivity property is undecidable (see Theorem 8.4.9), we propose an effective approximation to distributivity. First, we loosen up the condition expressed in Figure 8.9(a). One prerequisite for distributivity is that the recursion body $q_{body}$ does not inspect sequence positions in its input. Thus, for a distributive $q_{body}$ it must be legal to omit the row-numbering operator $q_{\text{pos:(item)/iter}}$ in the left-hand side of Figure 8.9(a) and discard all position information in the inputs of sub-plan $q_{body}$ (using $\pi_{\text{iter,item}}$). Further, since the set-equality (used to define distributivity) 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 $\pi_{\text{iter,item}}$ on top of both plans to make the order indifference explicit. Proving the equivalence illustrated in Figure 8.9(b), therefore, is sufficient to decide distributivity.

Further, we propose an assessment of distributivity based on algebraic rewrites.
Chapter 8. An Inflationary Fixed Point Operator for XQuery

\[ \rho_{\text{pos}:(\text{item})/\text{iter}, \text{iter}, \text{item}} = \rho_{\text{iter}, \text{iter}, \text{item}} \]

(a) Algebraic equivalent of the distributivity property: \( e_{\text{body}} \) is distributive if and only if its algebraic counterpart \( q_{\text{body}} \) satisfies this set-equality.

(b) Distributivity assessment agnostic to sequence order.

Figure 8.9: Algebraic distributivity assessment.

\[ \otimes \in \{ \pi, \sigma, \ominus, \oslash \} \]

\[ \otimes (q_1 \uplus q_2) \rightarrow (\otimes (q_1)) \uplus (\otimes (q_2)) \quad \text{(Unary)} \]

\[ \otimes \in \{ \times, \otimes, \ominus, \oslash \} \]

\[ (q_1 \uplus q_2) \otimes q_3 \rightarrow (q_1 \otimes q_3) \uplus (q_2 \otimes q_3) \quad \text{(Binary1)} \]

\[ \otimes \in \{ \times, \otimes, \ominus, \oslash \} \]

\[ q_1 \otimes (q_2 \uplus q_3) \rightarrow (q_1 \otimes q_2) \uplus (q_1 \otimes q_3) \quad \text{(Binary2)} \]

\[ (q_1 \uplus q_2) \uplus (q_3 \uplus q_4) \rightarrow (q_1 \cup q_3) \uplus (q_2 \cup q_4) \quad \text{(Union)} \]

Figure 8.10: An algebraic approximation of the distributivity property for arbitrary XQuery expressions.
8.6. An algebraic approximation of distributivity

If we can successfully “push” a union operator $\cup$ through the sub-plan $q_{\text{body}}$ in the left-hand side of Figure 8.9(b) to obtain the right-hand side, its corresponding XQuery expression $e_{\text{body}}$ must be distributive and we can safely trade $\mu$ for $\mu^\Delta$ to compute the fixed point.

To this end, we use a set of algebraic rewrite rules (Figure 8.10) that try to move a union operator upwards through the DAG. To avoid ambiguity or infinite loops during the rewrite process, we mark the union operator (indicated as $\sqcup$) in the left-hand-side plan $q_{\text{left}}$ of Figure 8.9(b) before we start rewriting. We then exhaustively apply the rule set in Figure 8.10 to each sub-plan in $q_{\text{left}}$ in a bottom-up fashion. Since each rule in the set strictly moves the marked union operator upwards inside the plan, termination of the process is guaranteed. Further, the number of operators $n$ in $q_{\text{body}}$ is an upper bound for the number of rewrites needed to push $\sqcup$ through $q_{\text{body}}$; $n$ itself is bounded by the size of $e_{\text{body}}$ (we have seen the same complexity bound for the syntactic analysis of Section 8.5).

Once the rule set does not permit any further rewrites, we compare the rewritten plan $q'_{\text{left}}$ with the right-hand side plan $q_{\text{right}}$ of Figure 8.9(b) for structural equality. This type of equality guarantees the equivalence of both plans and, hence, the distributivity of $e_{\text{body}}$.

Figure 8.11 shows the rewrites involved to determine the distributivity of $e_{\text{body}}$ for Query Q1 (Section 8.2). We place a marked union operator $\sqcup$ as the input to the algebraic plan $q_{\text{body}}$ obtained for the recursion body of Query Q1. The resulting plan corresponds to the left-hand side of Figure 8.9(b). Applying the equivalence rules [UNARY, BINARY1] and again Rule [UNARY] pushes $\sqcup$ up to the plan root, as illustrated in Figures 8.11(b), 8.11(c), and 8.11(d), respectively. The final plan (Figure 8.11(d)) is structurally identical to the right-hand side of Figure 8.9(b) with $q_{\text{body}}$ instantiated with the recursion body in Query Q1. We can conclude distributivity for $q_{\text{body}}$ and, consequently, for the recursion body in Query Q1.

To prove the soundness of this approach it is enough to acknowledge the correctness of the rewrite rules in Figure 8.10. Once union has been pushed through the algebraic plan of $e_{\text{body}}$ and the equality in Figure 8.9(b) holds, we can conclude that the expression is distributive and apply $\Delta$ for its evaluation. For more details, we refer to Afanasiev et al., 2009.

8.6.2 Algebraic vs. syntactic approximation

Compared to the syntactic approximation $ds(\cdot)$, the above algebraic account of distributivity draws its conciseness from the fact that the rather involved XQuery semantics and substantial number of built-in functions nevertheless map to a small number of algebraic primitives (given suitable relational encodings of the XDM). Further, for these primitives, the algebraic distributivity property is readily decided.

To make this point, consider this equivalent slight variation of Query Q1 in
Chapter 8. An Inflationary Fixed Point Operator for XQuery

Figure 8.11: The query plan transformation involved in determining the distributivity of $e_{body}$ for Query $Q_1$. The union operator $\biguplus$ marks the input to the algebraic plan $q_{body}$ obtained for the recursion body of Query $Q_1$. Applying the equivalence rules [UNARY, BINARY], and again Rule [UNARY] pushes $\biguplus$ up to the plan root, as illustrated in Figures 8.11(b), 8.11(c), and 8.11(d), respectively.
which variable $x$ now occurs free in the argument of function id(·):

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

If we unfold the implementation of the XQuery built-in function id(·) (effectively, this expansion is performed when Rule [FUNCALL] recursively invokes $ds_x(\cdot)$ to assess the distributivity of the function body of id(·)), we obtain

$$\text{with } x \text{ seeded by }$$
$$\text{doc("curriculum.xml")/course[@code="c1"]}$$
$$\text{recurse}$$
$$\text{for } c \text{ in doc("curriculum.xml")/course}$$
$$\text{where } c/@code = x/prerequisite/pre_code$$
$$\text{return } c.$$ 

The syntactic approximation will flag the recursion body as non-distributive because of the presence of the where clause (Section 8.5). Even if we rewrite the filter condition using an if-construct, the expression still remains not distributivity-safe due to the occurrence of the variable $x$ in the condition. While the algebraic approach is not affected by the two variations, the rule set of Figure 8.7 needs to be extended with a specific rule for id(·) to be able to infer its actual distributivity.

For each syntactic rule in Figure 8.7 we can prove that the corresponding algebraic plan passes the test for distributivity. Thus, the algebraic approach determines a larger fragment of distributive expressions and it is more succinct and easier to work with than the syntactic approach.

In spite of the fact that the approximation is bound to a particular (relational) algebra, we believe that this approach can easily be adapted for other algebras for XQuery.

### 8.7 Practical impact of distributivity and Delta

Exchanging RUDFs for the IFP operator limits the expressive power of the language. However, it also puts the query optimizer in control while the user is spared the trouble of deciding which algorithm should be used for the fix point computation. Trading Naive for Delta is a promising optimization and in the previous sections we showed that it can be effectively decided. In this section, we provide experimental 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 $\mu$ and $\mu^\Delta$ in MonetDB/XQuery 0.18 [Boncz et al., 2006a], an efficient and scalable XQuery processor that implements the Relational XQuery approach (Section 8.6). Its algebraic compiler front-end Pathfinder has been enhanced (i) to
Table 8.2: Naïve vs. Delta: Comparison of query evaluation times and total number of nodes fed to the recursion body.

<table>
<thead>
<tr>
<th>Query</th>
<th>Naïve Depth</th>
<th>Delta Depth</th>
<th>Naïve Total # of nodes fed to body</th>
<th>Delta Total # of nodes fed to body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Romeo and Juliet</td>
<td>6,795 ms</td>
<td>1,260 ms</td>
<td>1,150 ms</td>
<td>818 ms</td>
</tr>
<tr>
<td>Curriculum (medium)</td>
<td>183 ms</td>
<td>135 ms</td>
<td>1,308 ms</td>
<td>1,040 ms</td>
</tr>
<tr>
<td>Hospital (medium)</td>
<td>734 ms</td>
<td>497 ms</td>
<td>1,301 ms</td>
<td>1,290 ms</td>
</tr>
<tr>
<td>Bidder network (small)</td>
<td>9.3 ms</td>
<td>4.5 ms</td>
<td>721 ms</td>
<td>367 ms</td>
</tr>
<tr>
<td>Bidder network (large)</td>
<td>98.3 ms</td>
<td>57.2 ms</td>
<td>1,303 ms</td>
<td>1,297 ms</td>
</tr>
</tbody>
</table>

Input.
8.7. Practical impact of distributivity and Delta

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

declare function bidder ($in as node()* as node()*
{ for $id in $in/@id
  let $b := $doc/open_auction[seller/@person = $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 8.12: XMark bidder network query.

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 IFP 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 [Kay, 2009] process the resulting user-defined recursive queries
(cf. Figures 8.3 and 8.5). 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.

In accordance with the micro-benchmarking methodology developed in Part I,
we identify which query parameters might influence the performance of the IFP
computation. There are three such parameters: (i) the complexity of the recur-
sion body, (ii) the size of the input fed to the recursion body during the queries
computation, and (iii) the depth of the recursion. Our goal is to measure the
practical gains of the proposed optimization on real-life examples, rather than
a thorough investigation of the precise impact of these parameters. Thus, for
our experiment, we chose four queries on different XML data sets that are both
natural (borrowed from related literature) and that cover different values of these
parameters. We leave a more thorough investigation in the style of the MemBeR
micro-benchmarking for future work.

Table 8.2 summarizes our measurements of query evaluation time, total size
of the input fed to the recursion body during the recursive computation, and
recursion depth, for the four queries. We varied the data instance sizes to test
for scalability. Note that varying the data instance size we influence both the
recursion body input size (ranging from 12K to 87M nodes for Naïve and from
3K to 9M nodes for Delta) and the recursion depth (ranging from 5 to 33). Below
we describe each query and its performance.
Chapter 8. An Inflationary Fixed Point Operator for XQuery

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

Figure 8.13: Romeo and Juliet dialog query.

XMark Bidder Network. The first query computes a bidder network—recursively connecting the sellers and bidders of auctions—over XMark Schmidt et al. [2002] XML data (see Figure 8.12). We vary the data size from small (1MB, scale factor 0.01) to huge (37MB, scale factor 0.33). If Delta is used to compute the IFP of this network, MonetDB/XQuery as well as Saxon benefit significantly: 2.2 to 3.3 times faster and 1.2 to 2.7 times faster, respectively. Note that the number of nodes in the network (the same as the total number of nodes fed to body) grows quadratically with the input document size. Algorithm Delta feeds significantly fewer nodes to the recursion body, bidder(·), at each recursion level which positively impacts the complexity of the value-based join inside the function: for the huge network, Delta feeds exactly those 10 million nodes into bidder(·) that make up the result, while Naïve repeatedly revisits intermediate results and processes 9 times as many nodes.

Romeo and Juliet Dialogs. Far less nodes are processed by a recursive expression that queries XML markup of Shakespeare’s Romeo and Juliet [5] to determine the maximum length of any uninterrupted dialog (see Figure 8.13). 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 SPEAKERs are found to alternate. This query expresses horizontal structural recursion along the following-sibling axis. Although the recursion is shallow (depth 6 on average), Table 8.2 shows how both, MonetDB/XQuery and Saxon, completed evaluation up to 5 times faster because the query had been specified in a distributive fashion.

Curriculum. The following query, (Q1), was first presented in Example 8.1.1 and served as the leading example throughout the chapter. This query is borrowed directly from related work Nentwich et al. [2002] (Rule 5 in the Curriculum Case Study in Appendix B). It implements a consistency check over the curriculum data (cf. Figure 8.1) and finds courses that are among their own prerequisites.

8.8. Related work

let $hospital := doc("hospital.xml")/hospital
for $patient in $hospital/patient
where
  (with $x seeded by $patient
   recurse $x/parent/patient)/visit/treatment/test
  and
  $patient/visit/treatment[contains(medication, "headache")])
return $patient/pname

Figure 8.14: Hospital records query.

We generated the data instances for this query with the help of ToXgene [Barbosa et al., 2002].

Much like for the bidder network query, the larger the query input (medium instance: 800 courses, large: 4,000 courses), the bigger the benefit of Delta, for both query engines.

Hospital records. The last query explores 50,000 hospital patient records to investigate a hereditary disease. The query, shown in Figure 8.14, is taken from [Fan et al., 2006]. We generated the corresponding data instances with the help of ToXgene [Barbosa et al., 2002]. 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 five. Again, Delta makes a notable difference even for this computationally rather “light” query.

In conclusion, this experiment renders the particular controlled form of XQuery recursion that we propose and its associated distributivity notion attractive, even for processors that do not implement a dedicated fixed point operator (like Saxon).

8.8 Related work

Achieving adequate support for recursion in XQuery is an important research topic. Recursion exists at different levels of the language, starting with the essential recursive XPath axes (e.g., descendant or ancestor) and ending with the recursive user-defined functions. While efficient evaluation of the recursive axes is well understood by now [Al-Khalifa et al., 2002, Grust et al., 2003], the optimization of recursive user-defined functions has been found to be tractable only in the presence of restrictions: [Park et al., 2002, Grinev and Lizorkin, 2004] 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. Beyond inlining, the recursive user-defined functions do not come packaged with an effective optimization hook comparable to what the inflationary fixed point operator offers.
A prototypical use case for inflationary fixed point computation is transitive closure of arbitrary path expressions. This is also reflected by the advent of XPath dialects like Regular XPath [ten Cate, 2006b] and the inclusion of a dedicated \texttt{dyn:closure}(-) construct in the EXSLT function library [EXSLT, 2006]. In Section 8.7, we have seen two applications relying on transitive closure [Nentwich \textit{et al.}, 2002, Fan \textit{et al.}, 2006] and recent work on data integration and XML views adds to this [Fan \textit{et al.}, 2007].

The adoption of inflationary fixed point semantics by Datalog and SQL:1999 with its \texttt{WITH RECURSIVE} clause (Section 8.2) led to an intense search for efficient evaluation techniques for inflationary fixed point operators in the domain of relational query languages. The \textit{Naïve} algorithm implements the inflationary fixed semantics directly and it is the most widely described algorithm [Bancilhon and Ramakrishnan, 1986]. Its optimized \textit{Delta} variant, in focus since the 1980’s, has been coined \textit{delta iteration} [Güntzer \textit{et al.}, 1987], \textit{semi-naïve} [Bancilhon and Ramakrishnan, 1986], or \textit{wavefront} [Han \textit{et al.}, 1988] strategy in earlier work. Our work rests on the adaptation of these algorithms to the XQuery Data Model and language.

While \textit{Naïve} is applicable to all accounts of inflationary fixed points, \textit{Delta} is mainly applicable under syntactic restrictions, such as \textit{linear} recursion. For stratified Datalog programs [Abiteboul \textit{et al.}, 1995], \textit{Delta} is applicable in all cases, since positive Datalog maps onto the distributive operators of relational algebra ($\pi$, $\sigma$, $\times$, $\cup$, $\cap$) 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 \textit{syntactic} restrictions [Melton and Simon, 2002] on the iterative fullselect (recursion body) inside \texttt{WITH RECURSIVE} that make \textit{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. The distributivity-safe syntactic fragment introduced in Section 8.5.1 is essentially the XQuery counterpart of the linearity condition. We saw in Section 8.6 that replacing this coarse syntactic check by an elegant algebraic distributivity assessment renders a larger class of queries admissible for efficient fixed point computation.

Another well-known algorithm in the relational world, called \textit{Smart}, is presented again only for linear recursion [Ioannidis, 1986]. \textit{Smart} targets the inflationary fixed point computation of \textit{relational operators} specifically and it performs better than \textit{Delta} on shallow recursion. In the settings of XQuery, were the recursion body is any expression, \textit{Smart} is less applicable.

### 8.9 Conclusions and discussions

The problem we faced in this chapter is the lack of \textit{declarative recursive operators} in XQuery that allow for (algebraic) automatic optimizations. As a solution, we
introduced a declarative IFP operator for XQuery, borrowed from the context of relational databases. This operator covers a family of widespread use cases of recursion in XQuery, including the transitive closure of path expressions, while also being susceptible to systematic optimizations. We adopt an optimization technique widely used in relational databases and adapt it to the XQuery settings. This optimization relies on a distributivity property of XQuery expressions that can be effectively detected at the syntactic level. Furthermore, if we adopt a relational approach to XQuery evaluation, then distributivity can be detected more conveniently and effectively at the underlying algebraic level. Nevertheless, the IFP operator and the optimization technique that we propose can be easily implemented on top of any XQuery engine.

We integrated the IFP operator into the MonetDB/XQuery system and assessed the practical gain of our approach on real-life use cases. MonetDB/XQuery implements a relational approach to XQuery query evaluation and it is one of the fastest and most scalable XQuery engines today. We also experimented with Saxon, a popular open-source XQuery engine implementing a native approach to query evaluation. Our experiments showed significant performance gain (up to five times faster query evaluation times) on both engines. The main advantage of our approach—relying on a declarative recursive operator—is that this gain is obtained automatically, thus lifting the burden put on the user by the RUDFs.

While the empirical evidence is there, a foundational question remains: how feasible it is to do static analysis for recursive queries specified by means of the IFP operator. Specifically, are there substantial fragments of XQuery with the IFP operator for which static analysis tasks such as satisfiability are decidable? We address this question in the next chapter, Chapter 9.

Our choice of declarative recursive operator fell naturally on the IFP operator due to its success in relational databases. As we have shown, its good properties transfer to the XQuery setting. Nevertheless, there are other recursive operators, including other types of fixed points, such as the least fixed point operator, worth investigating. For example, a good understanding of the theoretical properties of the IFP operator for XQuery, such as its expressive power, is still missing. In Chapter 9 we study the theoretical properties of the IFP operator in the setting of the navigational core of XPath.

In spite of the fact that IFP covers a large class of recursive query needs in XQuery, some natural recursive operations cannot be expressed with it or it is very cumbersome, e.g., recursive XML construction (XML transformations) and recursive aggregates. It remains an open question what set of declarative recursive operators would be most natural to implement in the XQuery settings. This set should: (i) cover the most useful, commonly used, recursive query needs, and (ii) be easily implementable and susceptible to automatic optimizations.