An Inflationary Fixed Point Operator in XQuery

Loredana Afanasiev [#1†], Torsten Grust [*2], Maarten Marx [#3], Jan Rittinger [*4‡], Jens Teubner [°5]

[#] *University of Amsterdam*
Amsterdam, The Netherlands
[1] lafanasi@science.uva.nl
[3] marx@science.uva.nl

[*] *Technische Universität München*
Munich, Germany
[2] grust@in.tum.de
[4] rittinge@in.tum.de

[°] *IBM T.J. Watson Research Center*
Hawthorne, NY, USA
[5] teubner@us.ibm.com

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

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

I. Introduction

The backbone of the XML data model, namely *ordered, unranked trees*, is inherently recursive and it is natural to equip the associated languages with constructs that can recursively query such structures. In XQuery [1], recursion can be achieved only via *recursive user-defined functions (RUDFs)*—a construct that admits *arbitrary* types of recursion and largely evades optimization approaches beyond "procedural" improvements like tail-recursion elimination or unfolding.

In this paper, we explore a controlled form of recursion in XQuery, the *inflationary fixed point operator (IFP)*, familiar in the context of relational databases [2]. While less expressive than RUDFs, IFP embraces a family of widespread use cases of recursion, including forms of horizontal and vertical structural recursion and the pervasive *transitive closure* operator (in particular, IFP captures *Regular XPath* [3]). Let us consider an example of recursive data and query need.

Example 1.1: The DTD of Figure 1 (taken from [4]) describes recursive curriculum data, including courses, their lists of prerequisite courses, the prerequisites of the latter, and so on. The XQuery expression of Figure 2 recursively computes all prerequisite courses, direct or indirect, of the course coded with `"c1"`, on an instance document `"curriculum.xml"`. The compilation is seeded by the `course` element node with code `"c1"`. For a given sequence `$x` of `course` nodes, function `fix(·)` calls `rec_body(·)` on `$x` to find their direct prerequisites. While new nodes are encountered, `fix(·)` calls itself on the accumulated `course` node sequence. (This query is not expressible in XPath 2.0.) ◁

†Supported by Netherlands Organization for Scientific Research (NWO), Grant 017.001.190.

‡Supported by German Research Foundation (DFG), Grant GR 2036/2-1.

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

Fig. 1. Curriculum data (simplified DTD).

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

Fig. 2. Prerequisites for course `"c1"` (⌜ ⌝ marks the fixed point computation).

Note that `fix(·)` implements a generic inflationary fixed point computation: only the *seed* (`$seed := ···`) and the *body* (`rec_body(·)`) are specific to the curriculum problem. This motivates the introduction of a syntactic form for this pattern of computation (Section II). Unlike in the case of RUDFs, this account of recursion puts the query processor in control in choosing the evaluation strategy.

Provided that the *body* of the recursion exhibits a *distributivity* property, the computation of IFP is susceptible to systematic optimization. Distributivity may be tested on a syntactical level—a non-invasive approach that can easily be realized on top of existing XQuery processors. Further, if we adopt a relational view of the XQuery semantics (as in [5]), distributivity can be elegantly and uniformly tested on the familiar algebraic level (Section III).

Compliance with the restriction that distributivity imposes on IFP expressions is rewarded by significant query runtime savings. We illustrate the effect for the XQuery processors *MonetDB/XQuery* [6] and *Saxon* [7] (Section IV).

Discussion of related work, further details and proofs are provided in [8].

II. Defining IFP in XQuery

In this section we define an IFP operator in XQuery. We regard an XQuery expression e_1 containing a free variable `$x` as a function of `$x`, denoted by $e_1(\$x)$. We write $e_1(e_2)$

to denote $e_1[e_2/\$x]$, *i.e.*, the uniform replacement of all free occurrences of $x in e_1 by e_2, assuming that the free variables of e_2 are not bound in e_1. Finally, $e_1(X)$ denotes the result of $e_1(\$x)$ when $x is bound to the sequence of items X.

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

$$X_1 \overset{s}{=} X_2 \quad \Leftrightarrow \quad \texttt{fs:ddo}(X_1) = \texttt{fs:ddo}(X_2)\ . \qquad \text{(Seq)}$$

In this paper, we *only* consider XQuery expressions and sequences of type `node()*`.[2]

Definition 2.1: ***(Inflationary Fixed Point)*** Let e_{seed} and $e_{body}(\$x)$ be XQuery expressions. The *inflationary fixed point (IFP) of* $e_{body}(\$x)$ *seeded by* e_{seed} is the XQuery expression

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

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

The semantics of (1) is the sequence Res_k, if it exists, obtained in the following manner:

$$\begin{array}{lcl} Res_0 & \leftarrow & e_{body}(e_{seed}) \\ Res_{i+1} & \leftarrow & e_{body}(Res_i)\ \texttt{union}\ Res_i \quad , \quad i \geqslant 0 \end{array}$$

where $k \geqslant 1$ is the minimum number for which $Res_k \overset{s}{=} Res_{k-1}$. Otherwise, the semantics of (1) is *undefined*. ◁

Note that if expression e_{body} does *not* invoke node constructors (*e.g.*, `element{·}{·}` or `text{·}`), expression (1) operates over a finite domain of nodes and its semantics is always defined.

Example 2.2: Using the new operator we can express the query from Example 1.1 in a concise and elegant fashion:

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

Obviously, (1) is mere syntactic sugar as it can be equivalently expressed via the recursive user-defined function `fix(·)` (shown by ⌐¬ in Figure 2). Since (1) is a second-order operator taking XQuery expressions as arguments, `fix(·)` has to be interpreted as a template in which `rec_body($x)` is replaced with $e_{body}(\$x)$ (XQuery does not support higher-order functions). Then (1) is equivalent to `let $x :=` e_{seed} `return fix(`$e_{body}(\$x)$`)`.

Example 2.3: Like in the relational context, *transitive closure* is an archetype of recursive computation over XML instances. Regular XPath [3], for example, defines the transitive closure of XPath location paths and thus can express forms of horizontal and vertical structural recursion. Let e be a Regular XPath expression and e^+ its transitive closure as defined in [3]. Then e^+ can be expressed using the new IFP operator: `with $x seeded by . recurse $x/`e. Here '.' denotes the context node. ◁

[1] `fs:ddo(·)` abbreviates the function `fs:distinct-doc-order(·)` of the XQuery Formal Semantics [9].

[2] An extension to general sequences of type `item()*` is possible but requires the replacement of XQuery's node set operations (`union`, `except`) with appropriate variants.

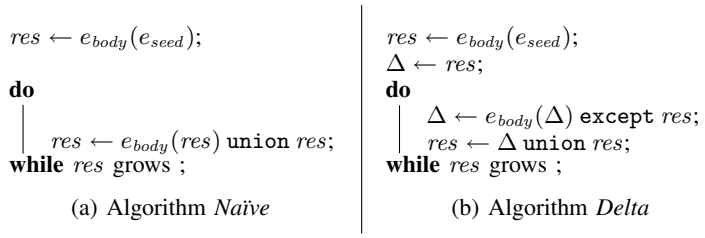

Fig. 3. Algorithms to evaluate the IFP of e_{body} given e_{seed}. Result is res.

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

Fig. 4. An XQuery formulation of Delta.

III. Implementing IFP in XQuery

A. Algorithms for IFP

The semantics of the IFP operator (1) given in Definition 2.1 can be implemented straightforwardly. Figure 3(a) shows the resulting procedure, commonly referred to as *Naïve* [10]. At each iteration of the **while** loop, e_{body} is executed on the intermediate result sequence res until no new nodes are added to it. Note that the old nodes in res are fed into $e_{body}(\cdot)$ over and over again. Depending on the nature of $e_{body}(\cdot)$, *Naïve* may involve a substantial amount of redundant computation.

A now folklore variation of *Naïve* is the *Delta* algorithm [11] of Figure 3(b). In this variant, $e_{body}(\cdot)$ is invoked only for those nodes that have not been encountered in earlier iterations: node sequence Δ is the difference between $e_{body}(\cdot)$'s last answer and the current result res. In general, $e_{body}(\cdot)$ will thus process fewer nodes. *Delta* introduces a significant potential for performance improvement, especially for large intermediate results and computationally expensive recursive bodies (Section IV). Figure 4 shows the corresponding RUDF `delta(·,·)`, which can serve as a drop-in replacement for function `fix(·)` in Figure 2—line 13 then needs to be replaced by `return delta(rec_body($d),())`.

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

Example 3.1: Consider the following expression:

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

When can *Naïve* be safely traded for *Delta*?

B. *Trading* Naïve *for* Delta

Delta correctly computes the IFP operator if it produces the same results as *Naïve* on the same inputs. The algorithms are equivalent, in particular, if both yield the same intermediate res sequences in each iteration of their **while** loops.

In its first loop iteration, *Naïve* yields $e_{body}(e_{body}(e_{seed}))$ `union` $e_{body}(e_{seed})$ which is equivalent to *Delta*'s first intermediate result $(e_{body}(e_{body}(e_{seed}))$ `except` $e_{body}(e_{seed}))$ `union` $e_{body}(e_{seed})$. For the second and further iterations, an inductive proof can show the equivalence of all subsequent intermediate res sequences, if for all sequences X_1, X_2,

$$e_{body}(X_1 \texttt{ union } X_2) \stackrel{s}{=} e_{body}(X_1) \texttt{ union } e_{body}(X_2) \ . \quad (2)$$

Note how (2) resembles the *distributivity property* of functions defined on sets. It suggests a divide-and-conquer evaluation in which e_{body} is applied to (singleton) subsets of the input. We define a similar distributivity property for XQuery.

Definition 3.2: ***Distributivity property for XQuery.*** Let $e($`$x`$)$ be an XQuery expression. Then e is *distributive for* `$x` if, for any sequence $X \neq$ `()`,

$$e'(X) \quad \stackrel{s}{=} \quad e(X) \ , \quad (3)$$

where $e'($`$x`$) =$ `for $y in $x return` $e($`$y`$)$ and `$y` is a fresh variable. $\triangleleft$

Equalities (3) and (2) are equivalent, thus we arrive at the following sufficient condition for the applicability of *Delta*:

Theorem 3.3: Let $e_{body}($`$x`$)$ and e_{seed} be XQuery expressions. If e_{body} is distributive for `$x`, then the algorithm *Delta* computes the IFP of $e_{body}($`$x`$)$ seeded by e_{seed}.

Path expressions are a prevalent example of distributive expressions in XQuery. Any expression of the form `$x`$/e$ is distributive for `$x` if e contains neither *(i)* calls to `fn:position()` and `fn:last()` that refer to the context sequence bound to `$x`, nor *(ii)* free occurrences of `$x`, nor *(iii)* node constructors. Note that the body of (Q1) from Example 2.2 satisfies these conditions, thus *Delta* can be applied for its evaluation.

In contrast, expression `$x[1]` is *not* distributive for `$x`. Let `$x` bound to the result of `(<a/>,<b><c/></b>)` and let a, b, and c denote the respective elements. Then `$x[1]` evaluates to a, while `for $y in $x return $y[1]` yields (a,b). Effectively, this invalidates Equation (3). Another example is the body of (Q2) from Example 3.1. For the same binding of `$x`, we obtain c and the empty sequence, respectively.

C. Testing distributivity

If an XQuery engine can determine whether the body of an IFP expression is distributive it can automatically apply *Delta*. Unfortunately, distributivity is an undecidable property. Still, we can safely approximate the answer. We provide a syntactic fragment of XQuery who's membership can be efficiently checked. The syntactic rules describing the fragment are quite verbose, though.

Distributivity can be checked more efficiently if the XQuery processor is equipped with an algebraic rewrite engine. *Pathfinder*, the query compiler behind the *MonetDB/XQuery* system [6], *e.g.*, uses a purely relational query formulation to analyze and optimize XQuery [5]. In [8], we extended the algebra of Pathfinder with the IFP operator μ. A set of rewrite rules repeatedly applies the algebraic equivalent of Equation 2 to propagate the relational union operator $\cup$ upwards in the recursion body of μ. Once $\cup$ reaches μ, distributivity is detected and we can use Algorithm *Delta* to implement the fixed point operation. With only a small number of modifications to the original query optimizer, this algebraic approach turns out to cover a larger distributive fragment than the syntactic fragment.

TABLE I
EVALUATION TIMES FOR QUERY FROM EXAMPLE 1.1.

Data size	***MonetDB/XQuery***		***Saxon-SA 8.9***	
	Naïve	*Delta*	*Naïve*	*Delta*
medium	183 ms	135 ms	1,308 ms	1,040 ms
large	1,466 ms	646 ms	3,485 ms	2,176 ms

IV. PRACTICAL IMPACT OF DISTRIBUTIVITY AND *Delta*

We implemented the IFP operator in *MonetDB/XQuery 0.18* and enhanced its algebraic compiler front-end *Pathfinder* (*i)* to process the IFP operator (Definition 2.1), and (*ii)* to implement the algebraic distributivity test. Whenever the distributivity test succeeds, *Delta* is applied, otherwise *Naïve*.

To quantify the performance advantage that can be realized by using *Delta*, we ran a number of real-world queries over documents of various sizes. Table I lists the execution times we observed for Expression (Q1) from Example 2.2 over XML instances that contained about 800 ("medium") and 4,000 ("large") `course` elements. Thanks to the *Delta* algorithm, MonetDB/XQuery scales linearly with document sizes for this query.

To demonstrate that *any* XQuery processor can benefit from the techniques we explored here, we also expressed both evaluation alternatives using XQuery (cf. Figures 2 and 4) and ran them on Saxon-SA 8.9 [7]. As shown in Table I, Saxon too can benefit from optimized IFP evaluation.

REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML query language," W3C Recommendation, 2007.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison Wesley, 1995.

[3] B. ten Cate, "Expressivity of XPath with Transitive Closure," in *Proc. PODS*, 2006, pp. 328–337.

[4] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: A Consistency Checking and Smart Link Generation Service," *ACM Transactions on Internet Technology*, vol. 2, no. 2, 2002.

[5] T. Grust, S. Sakr, and J. Teubner, "XQuery on SQL Hosts," in *Proc. VLDB*, 2004.

[6] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine," in *Proc. SIGMOD*, 2006.

[7] M. Kay, "The Saxon XSLT and XQuery Processor." [Online]. Available: http://saxon.sf.net/

[8] L. Afanasiev, T. Grust, M. Marx, J. Rittinger, and J. Teubner, "An Inflationary Fixed Point Operator in XQuery," Nov. 2007. [Online]. Available: http://arxiv.org/abs/0711.3375v1

[9] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics," W3C Recommendation, 2007.

[10] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," in *Proc. SIGMOD*, 1986.

[11] U. Güntzer, W. Kieling, and R. Bayer, "On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration," in *Proc. ICDE*, 1987.