Structural Function Inlining Technique for Structurally Recursive XML Queries*

Chang-Won Park† Jun-Ki Min‡ Chin-Wan Chung‡

†Information Technology Lab., LG Electronics Institute of Technology
‡Division of Computer Science, Korea Advanced Institute of Science and Technology
cwpark@LG-Elite.com, {jkmin,chungcw}@islab.kaist.ac.kr

Abstract

Structurally recursive XML queries are an important query class that follows the structure of XML data. At present, it is difficult for XQuery to type and optimize structurally recursive queries because of polymorphic recursive functions involved in the queries.

In this paper, we propose a new technique called structural function inlining which inlines recursive functions used in a query by making good use of available type information. Based on the technique, we develop a new approach to typing and optimizing structurally recursive queries. The new approach yields a more precise result type for a query. Furthermore, it produces an optimal algebraic expression for the query with respect to the type information. When a structurally recursive query is applied to non-recursive XML data, our approach translates the query into a finitely nested iterations.

We conducted several experiments with commonly used real-life and synthetic datasets. The experimental results show that the number of node lookups by our approach is on the average 3.7 times and up to 279.8 times smaller than that by the XQuery core's current approach in evaluating structurally recursive queries.

*This work was supported by the Brain Korea 21 Project.
†This work was performed while the author was with KAIST.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

XML is the standard language for representing data on the Web. Because of the growing popularity of XML, it is crucial to have a flexible query language capable of querying diverse XML data sources. In this regard, a standard query language called XQuery [2] is currently being developed, which is a strongly typed functional language in which a query is a typed expression. An important feature of XQuery is its support for recursive queries that follow the structure of XML data. We refer to this important query class as structurally recursive queries.

By a structurally recursive query, we mean a query that involves a recursive navigation ("//") [2], a recursive filtering [2], and function calls to user-defined structurally recursive functions [3, 11]. Several use cases [5] and the requirements [4] of XQuery indicate that structurally recursive queries are frequently used, and also meet various requirements such as abilities to query without schema knowledge, to operate on hierarchy, to transform input structures and create new structures, and to preserve the relative hierarchy and sequence of input structures in query results. For example, we often write `//title` to retrieve `title` elements regardless of the structure of data and the locations of `title` elements in the data.

In connection to the whole XQuery language, the XQuery core [10, 11] (a core subset of XQuery) has been established as well. Every XQuery expression is mapped to an expression in the XQuery core, revealing the semantics of the original expression. And then, the mapped expression is typed and optimized by means of the XQuery core's typing rules and equivalence rules. Thus, the XQuery core serves as an algebra for XQuery, and it is crucial for the XQuery core to map, type, and optimize structurally recursive queries properly. However, the XQuery core is still incomplete with regard to structurally recursive queries [10]. (Section 2 reviews how the XQuery core types and optimizes a structurally recursive query in detail.) Several related issues have been raised by the

XML Query Working Group as follows:

- Issues 42 and 163 in the XQuery 1.0 document [2]; and 0008, 0032, and 0098 in the XQuery 1.0 formal semantics document [11] are related to typing structurally recursive queries. For example, a structurally recursive query's type inferred by the XQuery core tends to be imprecise since the query relies on polymorphic recursive functions whose declared return type is usually a generic type such as `xs:AnyType` [10].
- Issues 0008 and 0043 in the XQuery 1.0 formal semantics document [11] are related to optimizing structurally recursive queries. For example, `//title` exhaustively searches whole data only to retrieve relatively small fragments consisting of `title` elements.

In this paper, we provide a solution to the above issues. We have been motivated by the XQuery core's support for projection [10, 11, 13]. The XQuery core translates a projection into nested iterations sensitive to their local types. In this way, the projection is naturally typed and significantly optimized. To achieve similar results, we adapt a well-known technique called function inlining [1] to typing and optimizing structurally recursive queries. The rationale of this adaptation is based on the observation that the main obstacle in typing and optimizing a structurally recursive query is polymorphic recursive functions involved in the query. While function inlining has been used in the programming language community, our function inlining differs significantly in that it inlines a (structurally) recursive function with the guidance of type information. We refer to this kind of function inlining as structural function inlining. The guiding principle is making good use of type information available in both a query and its environment [11] in which it is evaluated.

A brief overview of our approach is as follows: Given a structurally recursive query, it is mapped to structurally recursive functions and function calls to them. The mapped functions embed as much type information as possible into their function bodies from the given query. And then, we apply the structural function inlining to the mapped function calls. The technique can be regarded as a series of cascaded function inlining guided by available type information, and is followed by algebraic simplification. Together with the algebraic simplification, the structural function inlining translates the given query into an optimal algebraic expression with respect to the type information. This approach provides a more precise result type, and the resulting expression does not require useless evaluation with respect to the type information. Furthermore, if a structurally recursive query is applied to non-recursive XML data, the structural function inlining transforms a recursive function call into a finitely nested iterations sensitive to their local types. This effect is similar to that of the XQuery core's relating projection to iteration. As we can see in [2, 5], structurally recursive queries are very commonly applied to non-recursive XML data.

We conducted several experiments with commonly used real-life and synthetic datasets to compare quantitatively resulting expressions produced by our approach and the XQuery core's approach. The experimental results show that the number of node lookups by our approach is on the average 3.7 times and up to 279.8 times smaller than that by the XQuery core's current approach in evaluating structurally recursive queries.

1.1 Contributions and Advantages

Major contributions of our approach are summarized as follows:

- The structural function inlining technique that is the first adaptation of function inlining to typing and optimizing recursive functions;
- An algorithm that implements the structural function inlining;
- An approach based on the structural function inlining to typing and optimizing the structurally recursive query. The approach is illustrated with three representative examples of applying the structural function inlining to recursive navigation, recursive filtering, and regular path expressions;
- Several experiments for quantitative evaluation of our approach with respect to the XQuery core's current approach.

In addition, major advantages of the structural function inlining technique are summarized as follows:

- The technique is always promising because it provides not only a precise result type but also an optimal algebraic expression with respect to available type information.
- When a structurally recursive query is a subquery of a larger query, the technique reveals hidden opportunities for further global optimization by translating a structurally recursive query to nested iterations.

1.2 Organization of the Paper

The rest of the paper is organized as follows. After summarizing related work in the next section, Section 2 reviews the XQuery core's approach to mapping, typing, and optimizing structurally recursive queries. Subsequent sections describe our approach. The basic

idea and the algorithm of the structural function inlining are presented in Section 3, and its application to structurally recursive queries is illustrated with examples in Section 4. The experiments we conducted and the experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

1.3 Related Work

There has been a little work on typing XML queries and optimizing recursive functions. Some excellent work on typing XML queries such as semi-monad [13] and XDuce [15] has affected the type system of the XQuery core significantly. However, the XQuery core cannot properly type recursive XML queries [2, 10, 11]. In this regard, our structural function inlining is a novel technique for typing recursive XML queries.

As to optimizing functions, most of existing optimization techniques [6, 7] treat functions simply as externally defined black boxes accompanying some semantic information. Moreover, they consider nonrecursive functions only, and even the XQuery core cannot optimize recursive functions [2, 10, 11]. In contrast, the structural function inlining optimizes recursive functions to avoid useless evaluation over irrelevant fragments of data. The query pruning [14] similarly optimizes regular path expressions, but it is inapplicable to arbitrary recursive functions containing operations interleaved arbitrarily with navigation since such recursive functions are not transformed to finite automata.

2 Preliminaries

We briefly review how an XQuery expression is mapped to the corresponding algebraic expression and how the algebraic expression is typed and optimized in the XQuery core. For more details including the complete XQuery core syntax, typing rules, and equivalence rules, the reader is referred to the current XQuery 1.0 Formal Semantics document [11].

Mapping. A structurally recursive query involves one or more recursive functions and function calls to them. In the case of a recursive navigation, it is mapped to an expression that consists of a function call to the built-in recursive function `descendant-or-self()` and a projection. For example, `//title` is mapped intermediately to `descendant-or-self($roots)/title`. The variable `$roots` is a distinguished variable bound to a sequence of root nodes [11]. Since the projection is not in the XQuery core, we have the following expression from the subsequent mapping that relates the projection to an iteration [10, 11, 13]:

```
for $v1 in descendant-or-self($roots) return
  typeswitch ($v1) as $v2
    case element title(xs:AnyType) return $v2
    default return ()
```

The `typeswitch` keyword is followed by a parenthesized expression called the operand expression. The variable that follows the `as` keyword is referred to as the operand variable. Whatever the operand expression's type is, the type matches at least one `case` rule. In other words, the `case` rules are exhaustive. The `case` rule that matches first is called the effective case. While the operand expression may have several local types at query-analysis time, the operand variable has a single local type that dynamically determines the effective case at each step of the iteration. This makes the iteration sensitive to the local types of `$v1`. Thus, the mapped expression can be read in English as follows: for each descendant of the root nodes including themselves, either return the descendant if its type is `element title(xs:AnyType)`, or return an empty sequence.

Typing. Typing a function call checks whether each argument type is subsumed by the corresponding formal parameter type, and then takes the declared return type for its type. However, this may not provide useful type information when the return type is, for instance, `xs:AnyType`. The return type of a polymorphic recursive function that accepts any XML data is usually declared as `xs:AnyType` [10]. Because of such functions, the type of a structurally recursive query tends to be typed imprecisely.

On the other hand, a recursive navigation is typed differently by an ad hoc approach [11] that uses an internal typing function `recfactor()`. Consider the expression `descendant-or-self($roots)/title` mapped from `//title`. After translating the projection into an iteration, we have to determine the type of `descendant-or-self()` to type the entire expression. Supposing that the type of `$roots` is `Bib` in Figure 1(a), the current ad hoc approach invokes `recfactor()` with `Bib`. Then `recfactor()` collects all the types encountered in a recursive traversal of `Bib`, and gives the following type:

```
( element bib(Book*) | element book(attribute year(xs:CDATA),
··· ) | attribute year(xs:CDATA) | xs:CDATA | element
title(xs:string) | xs:string | ··· | element first(xs:string) )
min 1 max *.
```

This type is any sequence of choices each of which consists of all the types encountered in the recursive traversal of `Bib`. Here, we omit several types to be met in the recursive traversal, but it is easy to find them in the type definitions. The lower bound `min 1` happens when a single element `bib` of type `Bib` has no `book` elements. It should be clear that the upper bound is unbounded, and hence `max *`. Based on the type above, we can infer that the type of `descendant-or-self($roots)/title` is `element title(xs:string)*` as expected.

Optimization. The XQuery core has a rich set of equivalence rules. With such rules, we can eliminate unnecessary expressions, reorder expressions, or dis-

```
type Bib =
  element bib(Book*)
type Book =
  element book
  (attribute year(xs:CDATA),
   element title(xs:string),
   (element author(element last(xs:string),
                   element first(xs:string)))+)
```

(a) Type definitions

```
let $bib0 =
  <bib>
    <book year="1994">
      <title>TCP/IP Illustrated</title>
      <author>
        <last>Stevens</last><first>W.</first>
      </author>
    </book>
    <book year="1992">
      <title>Advanced Programming ...</title>
      <author>
        <last>Stevens</last><first>W.</first>
      </author>
    </book>
    <book year="2000">
      <title>Data on the Web</title>
      <author>
        <last>Abiteboul</last><first>Serge</first>
      </author>
      <author>
        <last>Buneman</last><first>Peter</first>
      </author>
      <author>
        <last>Suciu</last><first>Dan</first>
      </author>
    </book>
    <book year="1999">
      <title>The Economics ...</title>
      <author>
        <last>Gerbarg</last><first>Darcy</first>
      </author>
    </book>
  </bib> : Bib
```

(b) Data of the type `Bib`

Figure 1: An example of type definitions and XML data

tribute computations. However, it is hard to optimize structurally recursive queries because of functions. For example, `descendant-or-self($roots)/title` is inefficiently evaluated, since we must consider all the descendants of `$roots`. Furthermore, it is also difficult to optimize an expression by reordering expressions as in the conventional cost based optimization because of *(1)* the inherent order in XML data, and *(2)* the absence of various physical operators and their cost formulae for each core operation. However, we can think of static optimization such as determining whether a query (or a subquery) is type-invalid early by inspecting the type information to avoid useless evaluation over potentially large amounts of irrelevant data.

3 Structural Function Inlining

3.1 Basic Idea

The main obstacle in typing and optimizing a structurally recursive query is the functions involved in the query. To get rid of them, we inline the corresponding function body in place of each function call.

Before further discussion, we introduce some notations. For a function call `f()`, function inlining is

```
{{getBib($bib0)}}
==> for $n in $bib0 return
      typeswitch ($n) as $x
        case element bib(xs:AnyType) return $x
        default return ()
```

(a) Naive function inlining

```
{{getBib($bib0)}}
==> for $n in $bib0 return
      typeswitch ($n) as $x
        case element bib(Book*) return $x
        default return ()
```

(b) Structural function inlining

Figure 2: Two inlining examples

denoted by $\{\{$`f()`$\}\}$. `T(E)` denotes the type of an expression `E`. In addition, we define a symbol `<:` denoting the subtype relation. If t_1 is a subtype of t_2, we write t_1`<:`t_2. Finally, the intersection $t_1 \wedge t_2$ of types t_1 and t_2 is the largest type t such that t`<:`t_1 and t`<:`t_2.

Let us consider the following non-recursive function:

```
define function getBib(xs:AnyType $a) returns xs:AnyType
{
  for $n in $a return
    typeswitch ($n) as $x
      case element bib(xs:AnyType) return $x
      default return ()
}
```

This function accepts any sequence, iterates over the items in the sequence, and returns an item when the item is a `bib` element. We use a generic type, `xs:AnyType`, for polymorphism. Type `xs:AnyType` denotes a sequence of items each of which is any simple type, element, or attribute.

Consider a query `getBib($bib0)` over the XML data in Figure 1(b) of the type `Bib` (indicated after "`:`" in the last line). Because `T($bib0) = Bib = element bib(Book*)`, the effective case is the first `case` rule. Thus, the query gives the `bib` element bound to `$bib0`.

We show a naive function inlining for the query `getBib($bib0)` in Figure 2(a). The formal parameter `$a` is replaced by the argument `$bib0`. Then, we inline the body instead of the corresponding function call. After inlining, we have the resulting expression in Figure 2(a), which is an iteration sensitive to local types. Since `T($bib0)`, `T($n)`, and hence `T($x)` are `element bib(Book*)`, the type of the expression is `element bib(Book*)`. In contrast, without function inlining, the type is `xs:AnyType` because the declared return type of `getBib()` is `xs:AnyType`. This shows that function inlining would give more precise type.

However, this naive function inlining is insufficient in that it ignores static optimization with respect to the horizontal structure and the vertical structure that can be found from the associated type information in the environment. For example, Figure 3 depicts parsed type trees that constitute a tree view of the type definitions for the example query. In the trees,

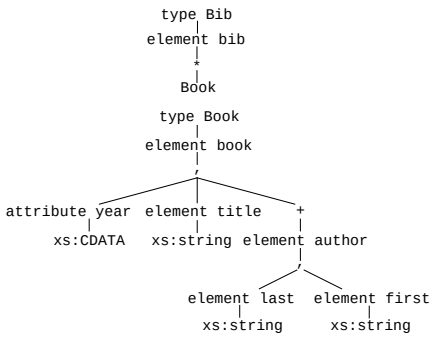


Figure 3: A tree view of the types in Figure 1(a)

each node is a type, a type variable (`Bib, Book`), or a type constructor (`*, +,` comma). If a node is a type, the children of the node is the content of the type. The horizontal and the vertical structures indicate sibling types and descendant types of a type, respectively. For example, the sibling types of the type `element title(xs:string)` are `attribute year(xs:CDATA)` and `element author(···)+`, while its descendant type is `xs:string`. The structural function inlining yields an optimal expression for a given query by means of two kinds of static optimization, which are horizontal and vertical optimizations.

The horizontal optimization specializes the `case` rules of a `typeswitch` expression with respect to the possible types of the operand expression. For example, consider the function call `getBib($bib0)`. Since `T($bib0) = element bib(Book*)`, `T($n)` in the function body is `element bib(Book*)` also. With respect to the type, the horizontal optimization specializes the effective case. That is, instead of the original type `element bib(xs:AnyType)` in the effective case, we take `element bib(Book*)`∧`element bib(xs:AnyType)`, which is `element bib(Book*)`. Thus, the first `case` rule in Figure 2(b) has the exact type `element bib(Book*)` instead of the original type `element bib(xs:AnyType)` in Figure 2(a). (They are boldfaced in the figures.) This illustrates what is horizontal optimization, which is used later. The other rules, if exist, are discarded. Instead, we add a new rule "`default return ()`" for both future optimization and exhaustiveness. Note that the original `default` rule is specialized only when it is the effective case.

The other optimization is vertical optimization on descendant types. Since the function `getBib()` is nonrecursive, we introduce another function:

```
define function s1(xs:AnyType $a) returns xs:AnyType
{
  for $n in $a return
    typeswitch ($n) as $x
      case element title(xs:AnyType)
        return $x, s1(children($x))
      case () return ()
      default return s1(children($x))
}
```

This function retrieves every `title` element regardless of its depth. It is an example of structural recursion [3] that follows the structure of data and consists of horizontal and vertical recursion. The horizontal recursion corresponds to the iteration expression of `s1()` over a given sequence. On the other hand, the vertical recursion corresponds to every recursive call in the `case` rules. It is notable that the vertical recursion is used on the content of the operand variable (i.e., `children($x)` in the first and the last rules). In this way, structural recursion always terminates. The stopping point of the recursion is the second rule for an empty sequence type. Interestingly, the structurally recursive function is applied frequently to nonrecursive XML data. For example, a query `s1($bib)` gives the following result:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming ...</title>
<title>Data on the Web</title>
<title>The Economics ...</title>
```

Now, we describe the vertical optimization, which can be thought of as a cascaded series of function inlining with horizontal optimization. Suppose that a variable `$book` is bound to a `book` element in the example data (Figure 1(b)), and its type is `Book` (Figure 1(a)). Consider the query `s1($book)`. The optimization starts with typing the argument `$book`, and `T($book) = Book = element book(···)`. According to `T($book)`, the function body is optimized by the horizontal optimization. The effective case is the `default` rule of `s1()`'s body, and it is specialized into "`case Book return s1(children($x))`" because we regard the label `default` as `xs:AnyType`. After that, a new rule "`default return ()`" is added for further optimization and exhaustiveness. The resulting expression contains another function call `(2)` with the argument `children($x)` which differs from `$book` as in Box `(1)` of Figure 4. This optimization is applied to the call `(2)` as well.

In the case of `(2)`, `T(children($x))` is either `element title(xs:string)` or `element author(···)`. Since the built-in function `children()` returns no attributes, `attribute year(xs:CDATA)` is ignored. Box `(2)` in Figure 4 shows the result of the horizontal optimization. The original `case` rules are specialized for each possible type, and the resulting `case` rules introduce two new recursive function calls `(3)` and `(5)`.

Regarding the call `(3)`, the type of children for the type `element title(xs:string)` is `xs:string`. So the `default` rule of `s1()`'s body is the effective case. The resulting `case` rules are given in Box `(3)`. Now, another call `(4)` is inlined. Since simple types such as `xs:string` has no children, the type of children is an empty sequence `()`. Therefore, the rule for `()` matches, and the resulting expression is given in Box `(4)`, which has no function call. Here, the recursion for this branch has finished. Before producing the final expression, the resulting expression of the call `(4)` that

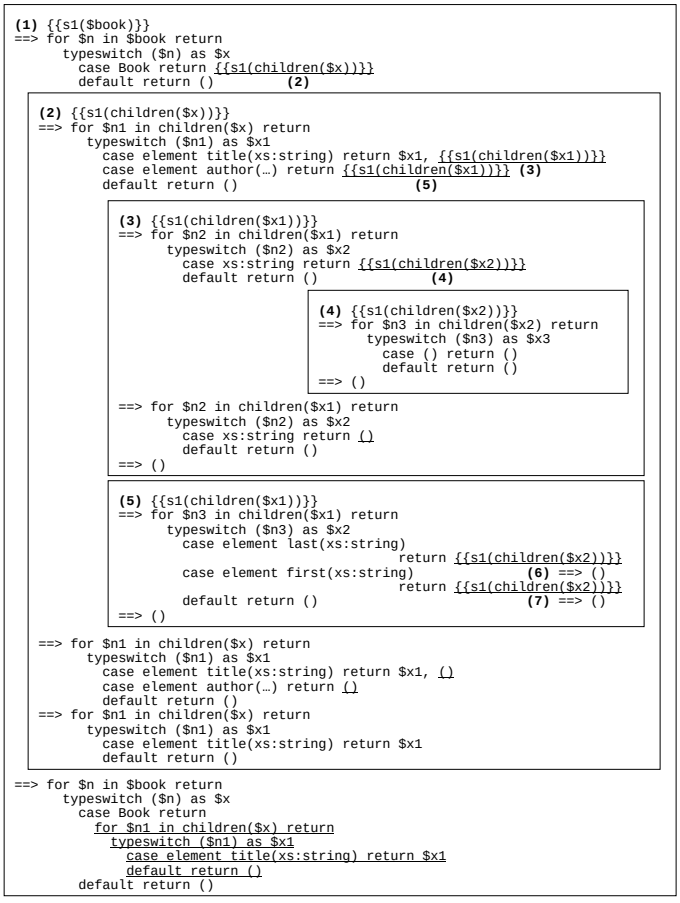


Figure 4: Trace of inlining the query `s1($book)`

returns only empty sequence is simplified into `()`. This empty sequence expression replaces the call `(4)` in Box `(3)`, and the resulting expression of `(3)` is simplified into `()` as well. These steps are depicted beneath Box `(4)` in Box `(3)`. The final expression `()` also replaces the call `(3)` in Box `(2)`.

Next, let us consider the call `(5)`. Inlining call `(5)` introduces two `case` branches, since `T(children($x1))` is either `element last(xs:string)` or `element first(xs:string)`. Each of them contains a recursive call with an argument of type `xs:string`. Inlining this call is identical to that of Box `(3)`, and hence the call becomes `()`. So Box `(5)` shows the expression in which each `case` rule returns `()`, and then the expression is simplified into `()`.

After inlining both `(3)` and `(5)` in the two `case` branches of the call `(2)`, algebraic simplification makes the expression more concise. The resulting expression is given beneath Box `(5)` in Figure 4. Finally, the call `(2)` of Box `(1)` has been inlined. The final expression is the last expression in Box `(1)`, which is a finitely nested iterations sensitive to their local types. The expression provides precise type information (`element title(xs:string)` instead of `xs:AnyType`) as in the case of projection, and it is optimal in that it is evaluated without useless searches. Furthermore, this static optimization can be followed by further optimization such as the cost based optimization.

3.2 Inlining Algorithm

Consider a function $f(p_1, \cdots, p_n)$ with n formal parameters whose body is B and a function call $f(a_1, \cdots, a_n)$ with n arguments. Function inlining can be formalized as follows:

Rule 1 (Function inlining)
$\{\{f(a_1, \cdots, a_n)\}\}$ `==>` B_n where

$$B_i = \begin{cases} \texttt{let } v_i \texttt{ := } a_i \texttt{ return } B_{i-1}[p_i/v_i] \\ \qquad\qquad\qquad\qquad \textit{if } a_i \textit{ is an expression} \\ B_{i-1}[p_i/a_i] \qquad \textit{if } a_i \textit{ is a variable} \\ B \qquad\qquad\qquad\qquad \textit{if } i = 0 \end{cases}$$

In the rule, B_i is the function body obtained after substituting the ith argument for the ith formal parameter, and $B_{i-1}[p_i/v_i]$ is that v_i substitutes for every occurrence of p_i in B_{i-1}. $B_{i-1}[p_i/a_i]$ is similarly defined. Introducing new `let` expressions is a kind of optimization since it prevents multiple evaluations of expression a_i. Otherwise, a_i would be evaluated as many times as it appears in B. (In Figure 4 and later on, we intentionally omit this optimization for brevity.) This rule is used to blindly inline the corresponding function body in place of an arbitrary function call.

Obviously, if we apply the function inlining indiscriminately, it is easy to find cases where inlining a function creates new function calls that can be inlined again, ad infinitum. Therefore, there have been numerous heuristics such as inlining functions called only once and inlining functions containing no function call. Such heuristics are still valid in the context of XQuery.

However, we have observed that some function classes in XQuery would be inlined more systematically under the guidance of type information. Structurally recursive functions are a kind of the function classes to which we can apply the structural function inlining. In order to identify what function class we focus our consideration on, we adopt the syntactic restrictions of the state-of-the-art work on structural recursion [3], which define the common form of structurally recursive function. Many papers including [3, 10, 13] suggest such restriction for structural recursion. Despite the syntactic difference between XQuery and the query language (UnQL) used in [3], the restrictions are applicable to XQuery.

Let the structural parameter be a parameter of any sequence type which is used for structural recursion. The structural function inlining exploits the property that the structural parameter's type changes for each recursive call according to the syntactic restrictions.

The structural function inlining algorithm is presented in Figure 5. Main algorithm *Structural_Function_Inlining* accepts a query expression,

Algorithm Structural_Function_Inlining(e)
input: a query expression e
output: the inlined expression
begin
1. **for** each function call f in e
2. **if** f is a structurally recursive function
3. **then**
4. b = Horizontal_Optimization(f)
5. prevent any name conflict between b and e
6. **endif**
7. replace f with b
8. **endfor**
9. simplify the resulting expression e'
10. output e'
end

Algorithm Horizontal_Optimization(f)
input: a function call f
output: the inlined expression
begin
11. b = the expression obtained by Rule 1
12. **for** each `typeswitch` expression te in b
13. replace te with te' obtained by Rule 2
14. **for** each "`case` t `return` e" in te'
15. e' = Vertical_Optimization(e, t)
16. replace e with e'
17. **endfor**
18. **endfor**
19. output b
end

Algorithm Vertical_Optimization(e, t)
input: a query expression e, the current type t
output: the inlined expression
begin
20. **for** each function call f in e
21. **if** f is a structurally recursive function
22. **then**
23. **if** t is recursive
24. b = Build_Surrogate_Fn(f, t)
25. **else**
26. b = Horizontal_Optimization(f)
27. prevent any name conflict between b and e
28. **endif**
29. **endif**
30. replace f with b
31. **endfor**
32. simplify the resulting expression e'
33. output e'
end

Algorithm Build_Surrogate_Fn(f, t)
input: a function call f, the current type t
output: a new call to the surrogate function
begin
34. **if not**(input pair (f, t) is a previously kept pair)
35. **then**
36. keep input pair (f, t)
37. b = Horizontal_Optimization(f)
38. create a new surrogate function f_s
input argument: same as f but take the structural parameter's type as T(children(t))
function body: b
output type: f_s_t
39. add type definition f_s_t = T(b) to the environment
40. output a new function call to f_s for (f,t)
41. **else**
42. output a new function call to f_s for (f,t)
43. **endif**
end

Figure 5: Structural Function Inlining Algorithm

and apply subalgorithm *Horizontal_Optimization* to each function (lines 1-4). *Horizontal_Optimization* prepares the function body to be inlined by means of Rule 1, and the horizontal optimization is conducted by using Rule 2 given below (lines 11-13). Rule 2 specifies how to transform the `typeswitch` expression in the function body with respect to the possible types for its operand expression. Such type information can be inferred from the concrete type of the structural paremeter. Let a `typeswitch` expression T be:

```
typeswitch (E) as v
 case t1 return E1
 case t2 return E2
 ...
 default return El
```

By regarding the `default` rule as `case xs:AnyType return` E_l, the horizontal optimization can be formalized as follows:

Rule 2 (Horizontal optimization)
For the `typeswitch` expression T and a set of possible types $t'_1, \cdots, t'_m$ for T(E), T `==>`

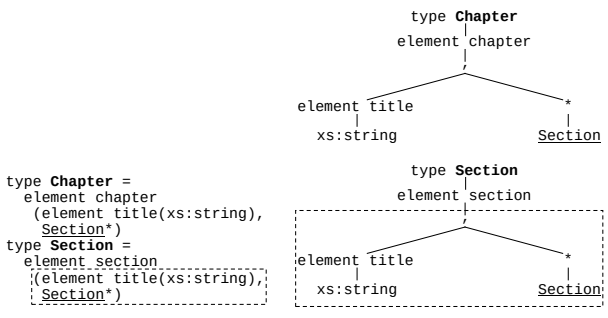
```
typeswitch (E) as v
 case t''1 return E'1
 case t''2 return E'2
 ...
 default return E'n
```

where $\forall_{(1\leq i\leq (n-1))}\exists_{(1\leq j\leq m, 1\leq k\leq l)}(t''_i = t'_j \wedge t_k \bigwedge t'_j <: t_k \bigwedge E'_i = E_k)$ such that t_k and E_k are the associated type and the associated expression, respectively, of the effective case for the possible operand type t'_j. Here, $\bigwedge$ denotes the logical AND, and E'_n in the `default` rule is a designated expression.

The designated expression can be either `error` or `()`. We have chosen `()` as E'_n because this choice permits further simplification as we have seen in Figure 4. According to Rule 2, the horizontal optimization specializes the `case` rules by pruning useless `case` rules and retaining only the effective cases for sibling types at a certain level in the type tree.

The vertical optimization applies this horizontal optimization to many subsequent levels recursively. So the vertical optimization, together with algebraic simplification, eventually prunes a large number of `case` rules to be evaluated fruitlessly. In the algorithm, on producing the horizontally optimized `typeswitch` expression te', *Horizontal_Optimization* invokes *Vertical_Optimization* for each `case` rule of te' (lines 14-15). *Vertical_Optimization* inlines each function call by invoking either *Build_Surrogate_Fn* or *Horizontal_Optimization* (lines 20-31), and simplifies the resultant expression (line 32). This recursive process always terminates due to the syntactic restrictions if the process involves no recursive type. Otherwise, it may not terminate.

To understand the separate treatment of recursive types, let us consider the type definitions in Figure 6. The name after keyword `type` is referred to as the



(a) Type definitions (b) Tree view

Figure 6: An example of the recursive type

type variable, which is divided into three syntactic categories in the concrete type syntax of the XQuery core [11]: attribute group variable, element group variable, and content type variable. The latter two kinds of the type variable are used in Figure 6. The element group variable is boldfaced, and is used to name an element type group. The content type variable is underlined, and is used to denote the content type of elements. The recursive type is always explicitly defined by means of the element group variable and the content type variable, and both type variables are represented as a node in the parsed type trees. Thus, we can easily detect a recursive type by discovering an element group variable containing one or more content type variable identical to itself. `Section` type is an example.

Considering a function call `s1($sec0)` where `$sec0` is bound to an element of type `Section`, the structural function inlining may produce an infinite inlining sequence as in Figure 7(a). Box `(1)` shows that the horizontal optimization produces a function call (call `(2)`) with element `section`'s children whose type is `(element title(xs:string), Section*)` enclosed by a rectangle in Figure 6(a) and 6(b). This function call yields two new function calls `(3)` and `(4)`. The call `(3)` will be simplified to `()` (see Box `(3)` in Figure 4), but the call `(4)` is inlined infinitely. The expressions (enclosed by two rectangles in Figure 7(a)) to be inlined forever are identical.

The structural function inlining algorithm should cope with this problem. Our solution is to detect any recursive type, define a new surrogate function for the infinite inlining sequence, and define the surrogate function's new output type. This solution is depicted in Figure 7(b). In Box `(1)`, the first horizontal optimization results in a new function call `(2)` (lines 1-4, 11-13), and *Vertical_Optimization* is invoked with a pair of arguments, the resulting expression and the type `Section` (lines 14-15). Since the type is recursive, *Build_Surrogate_Fn* is invoked instead of *Horizontal_Optimization* (lines 23-26). *Build_Surrogate_Fn* *(1)* checks the input pair and keeps it (lines34-36), *(2)* invokes *Horizontal_Optimization* to build the inlined function body (enclosed by the first rectangle in Box `(2)`) for the call `(2)` (line 37), *(3)* adds a new surrogate function with a fresh name (`s1'` in Box `(2)`) with same parameters, the inlined function body, and a new output type (line 38), *(4)* adds type definition (enclosed by the second rectangle in Box `(2)`) for the output type (line 39), and finally *(5)* returns a new function call (line 40) boldfaced at the bottom of Box `(1)`. It is noticeable that the second invocation of *Build_Surrogate* with the same arguments as the first invocation simply returns a new function call to the surrogate function defined by the first invocation (lines 34, 42), and so the inline process ends. This is denoted at the right side of call `(4)` in Box `(2)` of Figure 7(b). Notably, the type of the structural parameter of function `s1'` is precisely type `Section`'s content type, and the output type `s1'_t` is equivalent to `element title(xs:string)+` as expected.

Finally, the following theorem states that the structural function inlining produces an optimal expression with respect to type information. We define the cost of evaluating a query Q over a sequence s denoted by $\text{cost}(Q, s)$, which means total number of nodes (defined in the XQuery data model [12]) that are accessed in the evaluation of Q. Suppose that a structurally recursive query Q is transformed into Q^T by the structural function inlining with respect to type information T.

Theorem 1 *For any query Q' equivalent to Q and any sequence s conforming to T,* $\text{cost}(Q^T, s) \leq \text{cost}(Q', s)$.

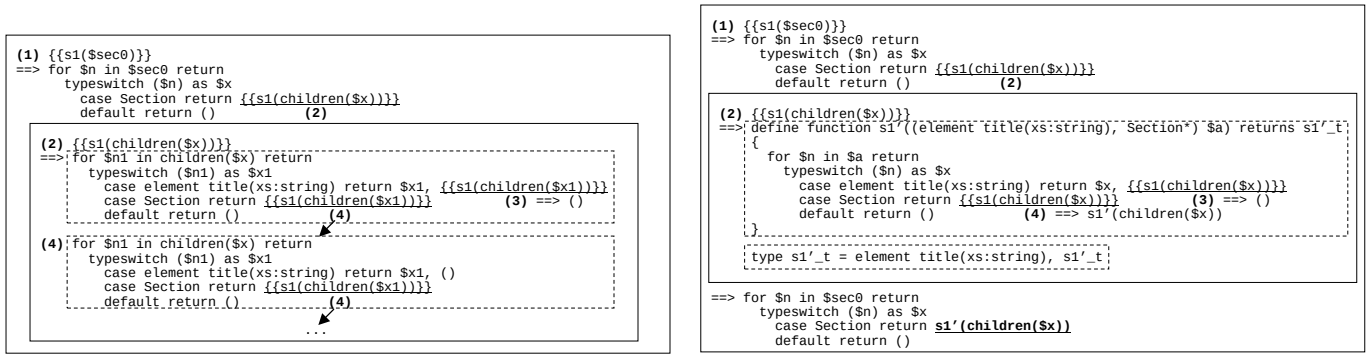
Proof Omitted due to space limitation. □

4 Application of the Structural Function Inlining

In this section, we describe how to apply the structural function inlining to structurally recursive queries in XQuery. For each of the three representative types of the structurally recursive query, we present the current approach of the XQuery core, new approaches that exploit the structural function inlining, and some discussion.

Recursive navigation. The XQuery core's approach to support recursive navigation is based on the built-in `descendant-or-self()` function and the internal typing function `recfactor()` as we have already seen in Section 2.

Instead, our approach maps a recursive navigation into a function call to a structurally recursive function by means of the translation method presented in [3] for a regular path expression. For example, we can think of a query `//title` as a nondeterministic finite automaton depicted in Figure 8, and define two structurally recursive functions from the automaton.



(a) Infinite inlining

(b) Building a surrogate function

Figure 7: Infinite inlining and our approach

The two functions are also in Figure 8. Each function and each function call correspond to a state and a transition to a state, respectively. For the transition, its symbol is represented as the associated type of the `case` rule that contains the corresponding function call. Thus, the default rule in function `s1()` calls `s1()` itself, and this call corresponds to the transition with symbol `*` from `s1` to itself. In addition, if a transition leads to a terminal state, the current element should be returned as well. So the first `case` rule in `s1()` returns the current element bound to `$x` and calls `s2()` that corresponds to the terminal state `s2`. It is notable that this `case` rule should call `s1()` as well since `title` is matched also with symbol `*`. Function `s2()` simply returns `()`, because state `s2` has no transition.

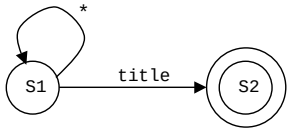
We can fuse the two functions by *(1)* simplifying the body of `s2()` into `()` and then *(2)* inlining it in lieu of any function call to `s2()` by Rule 1. This simplification yields a generic mapping for recursive navigation, and the example query becomes:

```
define function s1(xs:AnyType $a) returns xs:AnyType
{
  for $n in $a return
    typeswitch ($n) as $x
      case element title(xs:AnyType)
        return $x, s1(children($x))
      case () return ()
      default return s1(children($x))
}

s1($roots)
```

The function call `s1($roots)` produces the expected results (a sequence of `title` elements). This mapping is generic in that we can map any other recursive navigation query in the same way. For example, if "`//`a" is given, the first `case` rule is changed to "`case element` a`(xs:AnyType) return` $\cdots$."

Suppose `T($roots)` is `Book` in Figure 1(a). The same trace as given in Figure 4 results from the application. Thus, we have the following optimized expression for the query `//title` and the expression's type




```
define function s1(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return
  typeswitch ($n) as $x
   case element title(xs:AnyType)
    return $x, s1(children($x)), s2(children($x))
   case () return ()
   default return s1(children($x))
}
define function s2(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return ()
}
```

Figure 8: Mapping of the recursive navigation query

is `element title(xs:string)*`:

```
for $n in $roots return
 typeswitch ($n) as $x1
  case Book return
   for $n1 in children($x) return
    typeswitch ($n1) as $x1
     case element title(xs:string) return $x1
     default return ()
  default return ()
```

We note that this expression will traverse only the paths to `title` elements during evaluation, and never traverse the other irrelevant paths (for example, to `last` or `first` elements). Moreover, using the type information, the resulting expression can be further simplified [10, 11, 13]. When the query is a subexpression of a larger expression, the effect of this further simplification would be even more dramatic. Another important advantage of our approach is its typing, which is very similar to that of projection [10, 11, 13].

Recursive filtering. There is no valid approach in the XQuery core to support recursive filtering, and even the current mapping for filtering is obsolete [2, 11]. We propose two possible approaches. The first approach is for a special but most common case of recursive filtering, and the other is for the general case.

Even though a recursive filtering takes a single parameter which can be any expression, the most com-

mon argument is the path expression. The first approach maps a recursive filtering query whose argument is a path expression into a set of structurally recursive functions by means of the translation we have used for recursive navigation.

If `filter($sec0//title|$sec0//title/text())` is given, for instance, we can construct a nondeterministic finite automaton and define functions from the automaton as in Figure 9. The automaton can be constructed and simplified by the well-known methods in the automata area, and the functions are defined and simplified in the same way as we have explained in "Recursive navigation" section. The two functions in Figure 9 are simplified versions of the originals. Due to the shallow copy semantics [2] of the recursive filtering, we construct a new element with the known name and copy attributes instead of simply returning the current element node as in the first `case` rule of `s1()` (boldfaced in Figure 9). Thus, the mapped query `s1($sec0)` will produce the expected result, and will be optimized by the structural function inlining. Figure 10 is the trace when `T($sec0)` is `Section` in Figure 6. We omit the explanation of the trace since it is almost the same as that of Figure 7(b). The resulting expression is optimized with respect to the type information. In addition, `$sec0`'s type `Section` and the surrogate function's type `s1'_t` mean that the type of the example query is `element title(xs:string)+`.

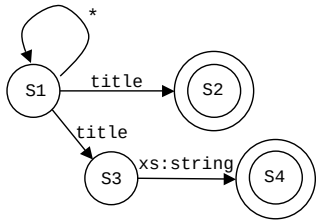
However, we cannot use the first approach when the argument is any expression other than the path expression. In this case, as the second approach, we should define a more generic structurally recursive function. If a given query is `filter(getBib($bib0))` in which `getBib($bib0)` is the first example query used in Section 3.1, the query is mapped as follows:

```
define function member(xs:AnyTree $x, xs:AnyType $y) returns xs:boolean
{
 not(empty(for $v in $y return
  if $v==$x then $v else ()))
}
define function filter(xs:AnyType $x, xs:AnyType $y) returns xs:AnyType
{
 for $n in $x return
  typeswitch ($n) as $n1
   case (Comment|PI|xs:AnyAttribute|xs:AnySimpleType) return
    if (member($n1,$y)) then $n1 else ()
   default return
    if (member($n1,$y)) then <{name($n1)}>
                              {$n1/@*, filter(children($n1),$y)}
                             </{name($n1)}>
                        else filter(children($n1),$y)
}

let $target := getBib($bib0) return filter($roots, $target)
```

The structural function inlining works fine with this mapped expression as well. To our knowledge, these two approaches are the first solution to the open issues on the recursive filtering [2, 10, 11].

However, the XQuery core's current type system types the inlined function body less precisely. This is due to the typing rules for the element construction with a computed tag name (e.g., `<{name($n1)}>`), the attribute construction with a computed attribute name, and the built-in `name()` function whose return




```
define function s1(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return
  typeswitch ($n) as $x
   case element title(xs:AnyType)
    return <title>
             {$x/@*, s1(children($x)), s3{children($x))}
           </title>
   case () return ()
   default return s1(children($x))
}
define function s3(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return
  typeswitch ($n) as $x
   case xs:string
    return $x
   default return ()
}
```

Figure 9: Mapping of the recursive filtering query

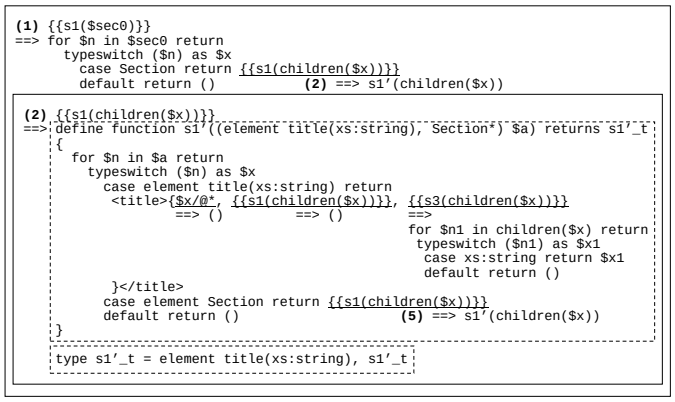
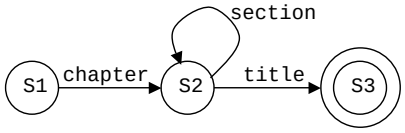


Figure 10: Trace of inlining the query `s1($sec0)`




```
define function s1(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return
  typeswitch ($n) as $x
   case element chapter(xs:AnyType) return s2(children($x))
   default return ()
}
define function s2(xs:AnyType $a) returns xs:AnyType
{
 for $n in $a return
  typeswitch ($n) as $x
   case element title(xs:AnyType) return $x
   case element section(xs:AnyType) return s2(children($x))
   default return ()
}
```

Figure 11: Mapping of `chapter/section*/title`

type is simply `xs:QName` [11]. These rules ignore available type information. The typing rules should be improved to deal with precise type expressions as in the previous version of the XQuery core's type system at *http://www.w3.org/TR/2001/WD-query-algebra-20010215*. With the improvement, the function body is well-typed.

Regular path expression. Since XQuery does not support regular path expressions, the user must express regular path expressions by defining user-defined structurally recursive functions. By means of the translation method in [3], one can easily express any regular path expression in XQuery. For example, `chapter/section*/title` is expressed as a finite automaton and hence structurally recursive functions in Figure 11. The functions in the figure are simplified versions of the originals in the same manner as we have described earlier in this section. Using the functions, the example query becomes a function call to `s1()` with an argument. Like the recursive navigation and the recursive filtering, the structural function inlining provides an inlined and optimized expression, and the XQuery core's type system types it properly.

5 Experiments

We conducted several experiments to show the quantitative effect of our proposed approach with respect to the XQuery core's current approach, using real-life and synthetic datasets. The characteristics of the real-life and the synthetic datasets are summarized in Table 1. Real-life datasets are Shakespeare's plays [8] in XML, which are hen_iv_1.xml and hen_iv_2.xml. In addition, to test the efficiency of the expressions resulting from our approach with regard to various data sizes, we combined some plays of Shakespeare into bigger ones: four_tragedy.xml consists of four tragedies (Hamlet, Macbeth, Othello, and Lear), and shakspeare.xml is an XML file containing all Shakespeare's plays. The other datasets are geneneated by IBM's XML Generator [9]. Types in Figure 1(a) and Figure 6(a) are used for bib1-4.xml datasets and chapter1-4.xml datasets, respectively, which are selected from types in XQuery use cases [5].

All the experiments were conducted on Pentium III-866MHz PC running MS-Windows 2000 with 256 MBytes of memory. To parse XML datasets, we used IBM's XML4J parser.

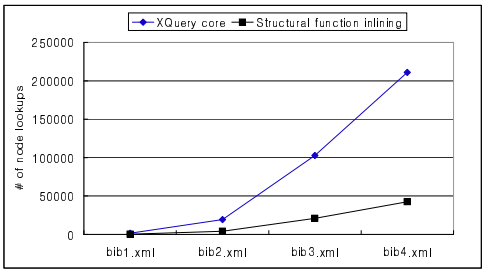
We evaluated two queries `//persona` over the real-life datasets and `//title` over the synthetic datasets. The expression resulting from the latter query by the XQuery core is presented in "Mapping" section of Section 2, and the expressions resulting by the structural function inlining are partially shown in Figure 4 and 7(b). In the experiments, we measured the number of node lookups to evaluate each expression on parsed DOM trees, since the XQuery expressions are evaluated through the tree traversal [2].

Dataset	Elements	Distinct tags	Recursive
hen_iv_1.xml	4825	21	N
hev_iv_2.xml	5326	21	N
four_tragedy.xml	22786	21	N
Shakespeare.xml	179653	21	N
bib1.xml	993	6	N
bib2.xml	11697	6	N
bib3.xml	61695	6	N
bib4.xml	126554	6	N
chapter1.xml	1555	7	Y
chapter2.xml	14965	7	Y
chapter3.xml	54618	7	Y
chapter4.xml	138728	7	Y

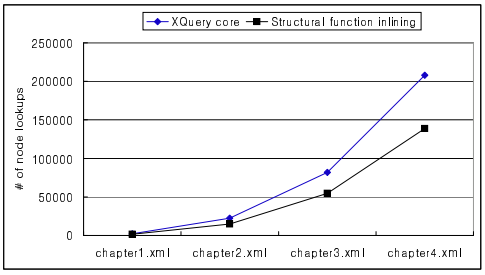
Table 1: XML Datasets

Dataset	XQuery core	Structural function inlining
hen_iv_1.xml	8847	38
hen_iv_2.xml	9715	66
four_tragedy.xml	41405	148
shakespeare.xml	327095	1561

Table 2: Experimental results for the real-life datasets



(a) Results for type `Bib`



(b) Results for type `Chapter`

Figure 12: Experimental results for the synthetic datasets

As shown in Table 2 and Figure 12, our structural function lining significantly reduces the number of node lookups in evaluating the queries, whereas the XQuery core's approach traverses whole DOM tree exhaustively. The total number of node lookups by the XQuery core's approach is 1,036,516, while that by our approach is 279,308. Therefore, our approach is 3.7 times more efficient than the XQuery core's approach. Moreover, the result of four_tragedy.xml dataset in Table 2 shows that the number of node lookups by our approach is 279.8 times smaller than that by the XQuery core's approach. Figure 12(b) shows the results of an unusual case that most of data is relevant to the evaluation of the query according to the type of the data, and hence the performance of our approach is only 1.5 times better than that of the XQuery core's approach. We note that the number of node lookups by the XQuery core's approach is greater than the number of elements in an XML document due to PCDATA (`xs:string`) nodes scattered in a DOM tree.

6 Conclusions

While XQuery has many excellent features, it has suffered from the difficulty of typing and optimizing polymorphic recursive functions. Since structurally recursive queries rely on such functions, structurally recursive queries are neither well-typed nor well-optimized so far. We have addressed this problem in this paper.

Our major contributions are a new technique referred to as the structural function inlining and a new approach to the problem of typing and optimizing structurally recursive queries. We have presented how the technique works, how to cope with technical obstacles such as the infinite inlining, and how to apply the technique to structurally recursive queries. The approach we have presented translates a structurally recursive query into a well-typed optimal expression. This is illustrated with various examples, and our experiments show that the number of node lookups by our approach is on the average 3.7 times and up to 279.8 times smaller than that by the XQuery core's current approach in evaluating structurally recursive queries.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[2] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. Working Draft, http://www.w3.org/TR/2001/WD-xquery-20011220, 20 December 2001.

[3] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9:76–110, 2000.

[4] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML query requirements. Working Draft, http://www.w3.org/TR/xmlquery-req, 15 February 2001.

[5] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML query use cases. Working Draft, http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220, 20 December 2001.

[6] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 529–542, August 1993.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, June 1999.

[8] R. Cover. The XML cover pages. http://www.oasis-open.org/cover/xml.html.

[9] A. L. Diaz and D. Lovell. XML generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[10] P. Fankhauser. XQuery formal semantics: State and challenges. *SIGMOD Record*, 30(3):14–19, September 2001.

[11] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. The XQuery 1.0 formal semantics. Working Draft, http://www.w3.org/TR/2001/WD-query-semantics-20010607, 7 June 2001.

[12] M. Fernandez, J. Marsh, and M. Nagy. XQuery 1.0 and (XPath) 2.0 data model. Working Draft, http://www.w3.org/TR/2001/WD-query-datamodel-20011220, 20 December 2001.

[13] M. Fernandez, J. Simeon, and P. Wadler. A semimonad for semi-structured data. In *Proceedings of the 8th International Conference on Database Theory*, pages 263–300, January 2001.

[14] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, February 1998.

[15] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. In *Proceedings of the 3rd International Workshop on the Web and Databases*, pages 226–244, 2000.