XQuery Function Inlining for Optimizing XQuery Queries

Maxim Grinev[1] and Dmitry Lizorkin[2]

[1] Institute for System Programming RAS, Russia,
grinev@ispras.ru
[2] Moscow State University, Russia,
lizorkin@hotbox.ru

Abstract. Function inlining replaces a call to a function with the body of the function. In this paper, we investigate the application of function inlining technique to user-defined functions expressed in XML Query Language (XQuery). XQuery function inlining eliminates function call overhead and allows statically optimizing the inlined function body with the rest of the query. We suggest an inlining algorithm that inlines calls to all non-recursive functions and usually allows completely inlining calls to structurally recursive functions, a class of recursive functions most interesting from practical viewpoint. The suggested algorithm reasonably terminates infinite inlining for recursive functions of any kind that makes the algorithm applicable to any XQuery query. Experimental measurements show the efficiency of the suggested approach for optimizing XQuery queries and for speeding up their evaluation.

1 Introduction

XML is a popular format for data representation. A query language to XML data sources is being developed by the W3 Consortium and is called XQuery [1]. The growing amounts of information stored in XML and the growing number of XML applications make optimization of XQuery queries a crucial task.

Two XQuery queries are considered equivalent if they produce the same result when evaluated. By optimization we mean a machine transformation of a given query into an equivalent one, such that the latter is better than the former with respect to some criteria. In this paper, we consider two such criteria for XQuery queries:

- *The total number of nodes accessed in target XML documents during query evaluation.* Query optimization should be aimed at minimizing the number of nodes required to be accessed. Thus, the optimized query ideally should access only those nodes that affect the query result, and skip nodes that do not affect it.
- *The total number of operations performed over target XML data to produce the query result.* Query optimization should follow the strategy of minimizing the number of operations that are required to be performed during query evaluation.

These criteria are closely related to execution time required for query evaluation: by accessing only the minimally required number of nodes and by executing the minimal number of operations over them, we minimize the running time required to obtain the query result.

In accordance with the XQuery Specification [1], a query is processed in two phases: *static type analysis* phase and *dynamic evaluation* phase. Static type analysis phase deals with the query expression itself and not with input data. Next occurs dynamic evaluation phase. During dynamic evaluation phase the value of the query is computed using input data.

In this paper we consider query optimization techniques that are applied during static type analysis phase, and the optimized query is then passed to dynamic evaluation phase. To produce the optimized query, we use type information available about the query during static analysis phase. This information includes:

- Statically-known schema for target XML documents. Such schemas may be supplied with XML documents and are specified in schema definition language, such as DTD, XML Schema, Relax NG. Statically-known schema allows us to statically obtain the type of the document node that would result from calling the `fn:doc` function within the XQuery query.
- Static typing rules for XQuery operations, formally specified in [2]. These typing rules specify types for XQuery primary expressions like literals, and allow to determine a type of a more complex XQuery expression based on types of its sub-expressions. Typing rules thus allow us to infer a type for any expression, because types can be determined recursively for its sub-expressions, until we reach primary expressions like literals and a function call to `fn:doc`.

This paper focuses on the query optimization technique based on XQuery function inlining. Function inlining is the replacement of a function call with the corresponding function body with arguments of the function call substituted instead of the corresponding function parameters. Function inlining allows reducing the number of nodes required to be accessed in the target XML documents during query evaluation, because function inlining facilitates the application of the other optimization techniques [3] to the inlined function body expression. Once the function body expression is inlined into the query, it becomes independent of the function declaration, and optimization techniques from [3] can freely be applied to the inlined expression, without corrupting the function declaration. Function inlining also eliminates function call overhead and thus reduces the total number of operations required to evaluate a query.

This paper is organized as follows. Section 2 introduces the notion of XQuery functions and discusses their inlining into the query. Section 3 discusses related works on the subject of XQuery function inlining for optimization purposes. Section 4 suggests classification of XQuery functions and analyses the possibility of inlining for optimization purposes for each of the function classes introduced. In Sect. 5 we give a detailed description of the suggested inlining algorithm, illustrate it with examples and prove the algorithm correctness. Section 6 presents

limitations possessed by the inlining algorithm. Results of practical experiments are shown in Sect. 7.

2 XQuery Functions and their Inlining into the Query

XQuery queries can contain different operators, and in particular they can contain *function call*s. A function call consists of a *function name* followed by a list of zero or more *arguments* enclosed in parentheses. A function name identifies the XQuery function. The XQuery function produces a value which is evaluated with respect to the argument list provided in the function call. The value produced by the function becomes the result of the function call.

XQuery introduces two kinds of functions:

- A set of built-in functions, described in XQuery 1.0 and XPath 2.0 Functions and Operators [4]. This document formally specifies how the result of each built-in function is computed from its parameters.
- User-defined functions[3]. These are specified in the XQuery *prolog*. Each user-defined function declaration specifies the name of the function, its parameters and an expression called the *function body* that defines how the result of the function is computed from its parameters.

To speed up query evaluation, we suggest to statically perform XQuery *function inlining* for XQuery user-defined functions. Function inlining is the replacement of a function call with the corresponding function body, with arguments of the function call substituted instead of the corresponding function parameters. Function inlining is a well-known optimization technique that is first introduced in compilers for programming languages [5] and is proven to be especially effective to optimize program evaluation.

However, unlike programming languages, XQuery has its own slightly different semantics for function evaluation. This semantics is reflected in XQuery 1.0 and XPath 2.0 Formal Semantics [2] and can involves atomization, subsequence extraction and implicit conversion operations:

- *Atomization* converts an argument to a sequence of atomic values, if atomic values are required by the corresponding function parameter.
- *Subsequence extraction* extracts just the first item from a sequence.
- *Conversion operations* convert an argument value to its expected type specified in the function declaration.

The above 3 operations are applied implicitly to each of the argument when the XQuery function call is evaluated. When we inline a function, we have to insert these 3 operations explicitly, to guarantee the correct semantics of a function call with respect to the XQuery formal semantics.

[3] In the XQuery specification such functions are called both "user-defined" and "user-declared", because the XQuery draft recently replaced the keyword "define" with the keyword "declare". Since the terminology is not currently fixed, in this paper we use the older term "user-defined" to denote such a class of functions

Compared to function inlining for programming languages, the task of function inlining for XQuery is simplified in that user-defined XQuery functions may not be overloaded [1].

2.1 Inlining Example

To illustrate the idea of function inlining for XQuery functions, we consider the following example. Let us introduce a simple user-defined XQuery function which counts text nodes that are descendants of a given element node. XQuery declaration for this function is given in Fig. 1. To calculate descendant text nodes for a given element node `$e`, the function:

1. counts child text nodes of `$e` (with a built-in function `fn:count`);
2. recursively applies to each of child element nodes of `$e` to count their descendant text nodes (with `for-return` expression), recursion terminates when `$e` has no child element nodes;
3. takes the sum of all these results (with the addition operation "`+`" and a built-in function `fn:sum`).

```
declare function count_leaves($e as element()) as xs:integer {
  fn:count($e/text()) + fn:sum(for $c in $e/* return count_leaves($c))
}
```

Fig. 1. Sample XQuery user-defined function which counts descendant text nodes for a given node $e

Now let us consider a sample function call to `count_leaves`, which looks as follows:

```
count_leaves( fn:doc("address-book.xml")/addressBook )
```

We will now discuss how XQuery function inlining can help us to optimize this function call. As it was mentioned in Sect. 1, a schema for a target XML document (`"address-book.xml"` in our case) allows us to obtain the type of the document node that would result from calling the built-in function `fn:doc`. Suppose that the schema for `"address-book.xml"` looks as described in Fig. 2: it is an `addressBook` element definition that contains zero or more `cards`, and each `card` contains a person `name` and an `email` [6].

We see that `count_leaves` is called with the argument whose type is `element addressBook`, and this type information helps us to inline this function call and further optimize it. This process is described in Fig. 3. Since `count_leaves` is a recursive function, several inlinings are performed one after another, and each inlining is expressed in Fig. 3 in its own nested frame.

```
type AddressBook = element addressBook(Card*)
type Card = element card ( element name(xs:string),
                           element email(xs:string) )
```

Fig. 2. Sample schema that describes an address book

Starting in Frame (1) in Fig. 3, we replace the call to `count_leaves` with its function body, with the argument of the function call substituted instead of the formal parameter `$e`. Exploiting type information, we can see that the argument of `fn:count` function is an empty sequence, since `element addressBook` has no child text nodes. Function `fn:count` is a built-in XQuery function and it returns the number of items in the argument; thus we can statically determine that this function call would always evaluate to 0, since an argument supplied is always an empty sequence (i.e. a sequence that contains no items). Variable `$c` in the `for-return` expression iterates over `card` elements (according to schema in Fig. 2). The recursive call to the function `count_leaves` is underlined, to show that it will be further inlined, as illustrated in the nested Frame (2) in Fig. 3.

In Frame (2), we inline the function call to `count_leaves` in the same manner as in Frame (1). Again, we can statically determine that a function call to `fn:count` would always evaluate to 0, since the static type of `$c` is `element card` which has no child text nodes. Considering static type information, we can further notice that the `for-return` expression in Frame (2) iterates just over two elements: `name` and `email`, and thus the whole expression can be replaced by the direct sequence of two items. Both items are function calls to function `count_leaves`, however with different arguments. Both function calls are underlined in Fig. 3 to show that they will be further inlined, in Frames (3) and (4) respectively.

In Frame (3), after we inline the function call, static type information again helps us to further optimize the expression. Namely, `fn:count` would always evaluate to 1, since `element name` defined in Fig. 2 has exactly one child text node. Similar considerations allow us to determine that `for-return` expression on Frame (3) would always iterate over an empty sequence, because `element name` cannot have child element nodes. Thus the result of the whole `for-return` expression in Frame (3) would be an empty sequence, and consequently the result of `fn:sum` in Frame (3) would always be 0. Function inlining and type analysis allow us to statically determine that the function call in Frame (3) would always evaluate to 1.

The same considerations fully apply to Frame (4), because `element email` also has exactly one child text node and no child element nodes.

In the bottom of Frame (2) we substitute the result obtained in Frames (3) and (4) instead of previously underlined function calls to `count_leaves`. Expression that results from optimization is shown in the bottom of Frame (1). This expression fully reflects the intuitive notion of the expected result: the

```
(1) count_leaves(fn:doc("address-book.xml")/addressBook)
==>
fn:count(fn:doc("address-book.xml")/addressBook/text()) +
 fn:sum(for $c in fn:doc("address-book.xml")/addressBook/*
        return count_leaves($c) )
==>
fn:count( () ) +
 fn:sum(for $c in fn:doc("address-book.xml")/addressBook/*
        return count_leaves($c) )
==>
fn:sum(for $c in fn:doc("address-book.xml")/addressBook/*
       return count_leaves($c) )

   (2) count_leaves($c)
   ==>
   fn:count($c/text()) +
    fn:sum(for $c2 in $c/* return count_leaves($c2))
   ==>
   fn:count( () ) +
    fn:sum(for $c2 in $c/* return count_leaves($c2))
   ==>
   fn:sum(for $c2 in $c/* return count_leaves($c2))
   ==>
   fn:sum( ( count_leaves($c/*/self::name),
             count_leaves($c/*/self::email) ) )

      (3) count_leaves($c/*/self::name)
      ==>
      fn:count($c/*/self::name/text()) +
       fn:sum(for $c3 in $c/*/self::name/*
              return count_leaves($c3))
      ==>
      1 + fn:sum(for $c3 in () return count_leaves($c3))
      ==>
      1 + fn:sum( () )
      ==>
      1

      (4) count_leaves($c/*/self::email)
      ==> ... ==>
      1

   ==>
   fn:sum( (1, 1) )
   ==>
   2

==>
fn:sum(for $c in fn:doc("address-book.xml")/addressBook/*
       return 2)
```

Fig. 3. XQuery function inlining example

`addressBook` has twice as text nodes as there are `card` child elements in the `addressBook` element.

It is worth noting that this optimization helped us to significantly reduce the number of nodes required to be accessed in the target XML document `"address-book.xml"` during dynamic evaluation phase. Before inlining, the whole content of the document had to be scanned. Now, dynamic evaluation phase requires only the document element `addressBook` and its child `card` elements to be accessed. Also the less number of operations is required to evaluate the optimized expression, because we were able to statically determine results of many operations. The optimization considered can make evaluation faster, as practically proven by experimental results reflected in Sect. 7.

3 Related Works

Structural Function Inlining Technique suggested in [7] greatly influenced our work. For a query class called structurally recursive queries, that paper proposes the optimization technique that actively involves function inlining. However, that approach is not suitable for queries that are not structurally recursive, and [7] does not suggest any mechanisms to determine whether a given query is structurally recursive or not. Also, the algorithm proposed in [7] is strongly based on the `typeswitch` operator, which limits the class of recursion termination conditions to conditions on argument *types*, not on their *values*. Although applicable, Structural Function Inlining Technique produces a poor optimization for XQuery recursive functions whose recursion termination conditions involves argument values, even when this function is structurally recursive.

Our approach, which we suggest in this paper, pays more attention on when to stop recursive function inlining. Our algorithm is applicable to every user-defined XQuery function, not just structurally recursive one. Instead of limiting the class of functions, as it was done in [7], our algorithm is applicable to every XQuery function: the algorithm makes the decision not to inline some function call if this inlining is unlikely to produce a better query or threats to fall into a recursive infinite loop. Our algorithm is able to handle any recursive termination condition, not just the `typeswitch` operator, and uses static type information available to produce a better query via function inlining.

4 Classification of XQuery User-defined Functions

This section gives a simple classification of XQuery user-defined functions from the viewpoint of inlining. The idea behind this classification is to introduce classes of XQuery user-defined functions for which function inlining is possible, and classes of functions for which function inlining is impossible. Three function classes are introduced, each in its own of the three following subsection. The description of each function class contains a discussion on whether function inlining is possible as an optimization technique for an XQuery function of that class. Obviously, every non-recursive function can always be inlined [5]; that is why

classification listed below is given for recursive XQuery user-defined functions only.

4.1 Structurally Recursive Functions

Structurally recursive functions are functions that follow the structure of XML data. For structurally recursive functions, recursion is used to walk an XML tree being processed: the function body processes a given node, and recursive calls are used to process its descendant nodes.

Structurally recursive functions constitute the a practical subset of XQuery user-defined recursive functions. In particular, all user-defined recursive functions introduced in XQuery Use Cases [8] are structurally recursive.

By definition, structurally recursive user-defined functions have an important feature: each of the following function calls is supplied with different types of arguments. This feature makes structurally recursive functions very attractive from the viewpoint of their inlining into the query, because we can actively use type information for further simplification of each function body inlined and for recursion termination.

XQuery user-defined function in Fig. 1 is structurally recursive. As discussed in Sect. 2, information about type declarations (like the one shown in Fig. 2) can be effectively used to inline function calls to structurally recursive functions to produce optimized queries.

4.2 Recursive Functions that Iterate over Sequences

Another practical class of XQuery user-defined recursive functions are those which iterate over sequences. For a user-defined function that iterates over a sequence, the function body generally processes only some items of the given sequence, and a recursive call is applied to process the rest items.

An example of an XQuery user-defined recursive function that iterates over a sequence, is shown in Fig. 4. This function simply reverses the order of items in a given sequence[4].

```
declare function reverse_sequence($s as node()*) as node()* {
 if (empty($s)) then ()
 else ( reverse_sequence($s[position()>1]), $s[1] )
}
```

Fig. 4. User-defined XQuery function which reverses the given sequence

[4] There is such a built-in function called `fn:reverse` specified in XQuery 1.0 and XPath 2.0 Functions and Operators [4]. However, we introduce a user-defined function with the same semantics to keep our examples concise

By nature, recursive user-defined functions that iterate over sequences have the following feature: each of the following function calls is supplied with the same types of arguments – the same sequences to be processed. Of course, the sequence being processed generally gets shorter with each of the following function calls, but this cannot usually be utilized during static analysis phase. The idea can be illustrated by the following sample function call:

```
reverse_sequence( doc("address-book.xml")/addressBook/card )
```

If we recall the type definition for an address book in Fig. 2, this function call returns `cards` in reverse order. But if we attempt to recursively inline such a function call, we would see that the argument always has the type `Card*`. This information does not allow us to terminate inlining because we do not possess static information about the number of `cards` that would be in the target document. We are unable to make any reasonable simplifications in the function body inlined as well.

Actually, it is not reasonable to inline function calls to XQuery user-defined functions that iterate over sequences, because information available during static phase is usually not sufficient to make this inlining justified.

4.3 Recursive Functions over Simple Types

Another practical subset of XQuery user-defined recursive functions are functions applied for calculations over simple types, usually numbers. As an illustration of such a class of functions, we can mention the calculation of mathematical factorial, implemented in Fig. 5.

```
declare function factorial($n as xs:integer) as xs:integer {
 if ($n=0 or $n=1) then 1
 else $n * factorial($n - 1)
}
```

Fig. 5. User-defined XQuery function which calculates mathematical factorial

Static information about types cannot help us to reasonably inline recursive functions over simple types, because information about argument values is generally required for effective inlining and optimization, and information about values can generally be possessed during dynamic evaluation only. Figure 5 illustrates this observation: condition on argument value (not type) is used for terminating recursion, and operations on values (not types) are applied to perform a recursive call.

The nature of XQuery user-defined functions over simple types opposes to their function calls being inlined when argument values cannot be determined during static phase.

5 Inlining Algorithm

As we could conclude from the classification given in the previous section, inlining algorithm should be able to implicitly determine the class of the user-defined XQuery function, a function call to which is currently processed, because functions from different classes should be given different treatment. We developed the inlining algorithm which possesses this feature, and remains simple for its practical implementation.

Before discussing the inlining algorithm, we introduce a notion of *type depth*, which is required for the algorithm and is described in Subsect. 5.1. After that, the inlining algorithm itself is described in Subsect. 5.2. The algorithm is illustrated by examples in Subsect. 5.3. Subsection 5.4 gives the proof of the inlining algorithm correctness.

5.1 Type Depth

With each schema type T we associate its *depth* which we denote as $\mu(T)$. Type depth is a non-negative integer and is calculated as follows:

- For all atomic types, text nodes and an empty sequence, their depth is 0.
- For attribute nodes, comment nodes and processing-instruction nodes, their depth is 1.
- For an element node, its depth is defined recursively as the depth of its child nodes plus 1:

$$T = element(*) \Longrightarrow$$
$$\mu(T) = \mu(T/child :: node()) + 1$$

  If element schema type contains recursive schema definitions, the element depth is considered $+\infty$ (positive infinity).
- For a document node, its depth is the depth of its document element plus 1.
- For a type that is a choice among several alternatives, its depth is the maximal depth for all alternatives:

$$\mu(T_1|T_2|\ldots|T_n) = \max(\mu(T_1), \mu(T_2), \ldots, \mu(T_n))$$

  The same treatment applies for a type that is a sequence:

$$\mu((T_1, T_2, \ldots, T_n)) = \max(\mu(T_1), \mu(T_2), \ldots, \mu(T_n))$$

- For a type that contains occurrence indicator, its depth is the depth of the type standing under that occurrence indicator:

$$\mu(T?) = \mu(T)$$
$$\mu(T*) = \mu(T)$$
$$\mu(T+) = \mu(T)$$

- For `xs:anyType`, its depth is considered $+\infty$ (positive infinity).

5.2 Algorithm Description

Throughout its work, the algorithm stores information about XQuery user-defined function previously inlined. For each previously inlined function $f(x_1, x_2, \ldots, x_n)$, this information includes:

- The name of the function: f
- Type depth for each of its arguments for the previously inlined function call to f: $m_{prev,1}, m_{prev,2}, \ldots, m_{prev_n}$
- A mask: a vector of length n consisting of 0-s and 1-s. A number in each position denotes whether the depth of the corresponding argument decreased with respect to the previous inlining of f.

The inlining algorithm consists of the following steps:

1. Analyze the query prolog and store information about all user-defined XQuery functions.
2. Expression *Expr* to be further processed is the *QueryBody*.
3. *Expr* is traversed from outer query operations to inner operations, in the tree style. For each function call $f(expr_1, expr_2, \ldots, expr_n)$ encountered:
4. If f is not a user-defined XQuery function, keep this function call unchanged and resume step 3 for each of $expr_1, expr_2, \ldots, expr_n$.
5. Otherwise, f is a user-defined function. Let us consider its signature to be

```
declare function f(x1 as T1,...,xn as Tn) as resT
 { body_expr }
```

6. Perform inlining for each argument: $expr_1, expr_2, \ldots, expr_n$. Let us denote these expressions after inlining as $expr'_1, expr'_2, \ldots, expr'_n$.
7. If function f is not stored by the algorithm as previously inlined in the outer expression, inline this function call, as specified in Fig. 6. Store information about f, type depth for each of its arguments

$$\mu(T(expr'_1)), \mu(T(expr'_2)), \ldots, \mu(T(expr'_n))$$

and a mask of 1s: $(1, 1, \ldots, 1)$. Go to step 3 to process the inlined *body_expr*.
8. Otherwise, function call to f was previously inlined in the outer XQuery expression, and we have information about the previous arguments type depths

$$\mu_{prev}[1], \mu_{prev}[2], \ldots, \mu_{prev}[n]$$

and the previous mask:

$$mask_{prev}[1], mask_{prev}[2], \ldots, mask_{prev}[n]$$

9. Calculate current type depths:

$$\mu_{curr}[i] = \mu(T(expr'_i)), i = \overline{1, n}$$

and the delta between the corresponding previous and current type depths:

$$\triangle\mu[i] = \max(\mu_{prev}[i] - \mu_{curr}[i], 0), i = \overline{1,n}$$

The new mask is calculated as follows:

$$mask_{curr}[i] = sgn(\triangle\mu[i] \cdot mask_{prev}[i]), i = \overline{1,n}$$

10. If $\forall i = \overline{1,n} : mask_{curr}[i] = 0$, this function call to f should not be inlined, because the current mask shows that the type depth decreased for none of the arguments, compared to the previous function call to f.
11. Otherwise, inline this function call, as specified in Fig. 6. Store information about f, current type depths

$$\mu_{curr}[1], \mu_{curr}[2], \ldots, \mu_{curr}[n]$$

and the current mask:

$$mask_{curr}[1], mask_{curr}[2], \ldots, mask_{curr}[n]$$

Go to step 3 to process the inlined *body_expr*.

```
let x1 := Convert_to_T1( Extract( AtomizeAtomic_for_T1( expr1 ) ) ),
    x2 := Convert_to_T2( Extract( AtomizeAtomic_for_T2( expr2 ) ) ),
    ...
    xn := Convert_to_Tn( Extract( AtomizeAtomic_for_Tn( exprn ) ) ),
return
  body_expr
```

where:

- `AtomizedAtomic_for_Tk($x) = fn:data($x)` if `Tk` is the subtype of `xs:anyAtomic*`, and the identity function otherwise.
- `Extract($x) = fn:subsequence($x, 1, 1)` if compatibility with XPath 1.0 is set to true for specific XQuery implementation, and the identity function otherwise.
- `Convert_to_Tk($x) = $x cast as Tk` if `Tk` is the subtype of `xs:anySimpleType`, and the identity function otherwise.

Fig. 6. Formal form of the expression to replace the function call being inlined

5.3 Examples of the Algorithm Application

In this subsection, we consider the treatment the suggested inlining algorithm provides to user-defined XQuery functions from each function class introduced in Sect. 4. We also give a more formal consideration to the inlining example discussed in Subsect. 2.1.

The inlining algorithm was developed to automatically identify structurally recursive user-defined XQuery functions and inline such function calls as many times as it is required to fully walk an XML tree being processed. A sample function discussed in Subsect. 2.1 is obviously a structurally recursive one. If we recall Fig. 3 once again and now consider it from the viewpoint of the inlining algorithm, it is worth noting the following:

- In Frame (1), function `count_leaves` is called with the argument whose type depth is 3:
$$\mu(addressBook) = \mu(card) + 1 = 2 + 1 = 3$$
- In Frame (2), the type depth of the argument is 2:
$$\mu(card) = \max(\mu(name), \mu(email)) + 1 = 1 + 1 = 2$$
- In Frames (3) and (4), the type depth of the argument is 1:
$$\mu(name) = \mu(xs : string) + 1 = 0 + 1 = 1$$
$$\mu(email) = \mu(xs : string) + 1 = 0 + 1 = 1$$

These values encourage the algorithm to proceed inlining calls to recursive function `count_leaves`, because type depth decreases for each of the subsequent recursive function calls (Frames (3) and (4) are independent of each other for the algorithm). Based on static type information, inlined expressions can be significantly simplified, to finally form the resulting expression shown at the bottom of Frame (1) in Fig. 3.

For XQuery user-defined functions that iterate over sequences (Subsect. 4.2), type depth of their arguments remain stable throughout subsequent recursive function calls. This condition instructs the inlining algorithm to terminate, and this algorithm behavior fully corresponds to the conclusion formulated in Subsect. 4.2: it is not reasonable to inline XQuery user-defined functions that iterate over sequences.

Similar treatment is given by the algorithm to XQuery user-defined recursive functions over simple types (introduced in Subsect. 4.3). The type depth of the argument always remains 0 for such functions, and this instructs the algorithm to terminate inlining, because inlining is quite likely to be fruitless here, as noted in Subsect. 4.3.

5.4 Proof of the Algorithm Correctness

Theorem 1. *For any XQuery query Q, the suggested function inlining algorithm always terminates and produces the equivalent query Q_I*

Proof. We prove the theorem in two steps. As the first step, we prove that our inlining algorithm produces the equivalent query. As the second step, we prove that the algorithm successfully terminates for every XQuery query provided.

1. Inlining algorithm produces the equivalent XQuery query Q_I, because every single function inlining produces the equivalent query. This leads from the fact that function inlining is performed in full accordance with dynamic evaluation of XQuery functions defined in XQuery Formal Semantics[2], i.e. the function call is rewritten as it would be evaluated.
2. We will now prove that the inlining algorithm successfully terminates for every XQuery query provided.
   Let us consider an arbitrary XQuery query Q. Suppose that this query contains n functions $f_1, f_2, \ldots, f_n$ defined in its prolog. Every such function f_i (for $i = \overline{1,n}$) has a_i parameters and contains c_i function calls to $f_1, f_2, \ldots, f_n$ in its body.
   Suppose that our query Q considered involves m XML documents, and for j-s document (for $j = \overline{1,m}$), the depth for its document element in accordance with document schema is d_j. Each d_j is either a natural number or positive infinity if document schema is recursive.
   Finally, let us suppose that the query Q contains e element constructors and c function calls to $f_1, f_2, \ldots, f_n$ in its query-body.
   With these symbols taken, the number of levels of inlining does not exceed

   $$ {a_1}^d \cdot {a_2}^d \cdot \ldots \cdot {a_n}^d $$

   with

   $$ d = \max_{i=\overline{1,n}, d_i \neq \infty} d_i + e $$

   This statement leads from the fact that the longest chain of function calls (i.e. when one function calls another one and so on in a chain) contains no more than n different functions $(f_1, f_2, \ldots, f_n)$, and each of these will be inlined for no more that ${a_j}^d$ times, because for every following inlining the type depth must decrease by at least 1 for at least single argument in comparison with the previous inlining, and the initial depth for each argument does not exceed d.
   Thus, if we define

   $$ c_{max} = \max_{i=\overline{1,n}} c_i $$

   the total number of functions inlined cannot exceed

   $$ c \cdot {c_{max}}^{{a_1}^d \cdot {a_2}^d \cdot \ldots \cdot {a_n}^d} $$

   This leads from the reason that we start with c function calls to $f_1, f_2, \ldots, f_n$, and every following level of inlining multiplies this number of function calls by at maximum c_{max}. Since the number of such levels cannot exceed ${a_1}^d \cdot {a_2}^d \cdot \ldots \cdot {a_n}^d$ as previously noted, this finally proves the statement that the suggested algorithm of function inlining always terminates.

Note 1. In practice, the suggested inlining algorithm allows reducing the number of function calls in the query body and often even to eliminate function calls from query body completely, because function inlining facilitates the application of the other static query optimization methods, like the ones described in [9], [3].

6 Limitations of the Inlining Algorithm

The inlining algorithm suggested in this paper does not allow completely inlining a structurally recursive XQuery user-defined function if it iterates over a type whose schema is itself recursive. This limitation of the inlining algorithm leads from the definition of the type depth, which is involved into the stop-condition of the algorithm. Indeed, for a type with a recursive schema, its depth is infinite. Even if a recursive XQuery user-defined function descends over such a type, its statically calculated depth remains infinite because of recursive schema, and thus the inlining algorithm cannot assert that the depth decreases. By having only static information (i.e. the schema of the document), we cannot obtain a more reliable information about the behavior of a recursive function. The suggested inlining algorithm avoids falling into an infinite loop in the case of recursive schema, at the price of keeping function calls as is.

The situation can be illustrated by the following example. Consider the schema of a document in Fig. 7. (The idea of this schema definition is taken from Relax NG Tutorial [6]). It contains a recursive definition, which states that `bold` and `italic` elements are allowed to be arbitrarily nested into one another and mixed with the remaining text.

```
type doc = element doc(markedText)
type markedText = ( xs:string |
                    element bold(markedText) |
                    element italic(markedText) )*
```

Fig. 7. Sample recursive schema definition

Let us consider XQuery function `count_leaves` already discussed in Subsect. 2.1, applied to `doc` element from Fig. 7. Although this function is structurally recursive, it is clear that we cannot statically determine the number of inlinings for such a function call, because we do not know the maximal depth the document data might have, since the schema states that `bold` and `italic` elements may be nested into one another as many times as possible. We thus should avoid falling into an infinite loop during inlining, and this fully corresponds to the description of inlining algorithm suggested in this paper. Inlining of this function call to `count_leaves` terminates, because the argument type depth is infinite and it remains infinite for a subsequent recursive function call.

For this example, the behavior of the inlining algorithm fully corresponds to the intuitive point of view as well. Inlining the function call to `count_leaves` cannot help us to statically optimize the query in this situation, because document schema in Fig. 7 contains text nodes (described with `xs:string`) at any level of hierarchy and the number of such levels in the document is statically unknown.

We suppose that even for a type whose schema is itself recursive it may be sometimes reasonable to inline a structurally recursive XQuery function that iterates over that type. However, our algorithm does not allow doing this, because it is based on the idea of type depth, which is infinite for a type with a recursive schema definition.

7 Experiments

We implemented the suggested inlining algorithm in our XML DBMS Sedna [10], being developed by the Institute for System Programming, Russian Academy of Science. We conducted several experiments to consider the qualificative effect of XQuery function inlining for query optimization.

Table 1 summarizes the benchmark statistics for the example which we earlier considered in Subsect. 2.1: counting descendant text nodes of an `addressBook` element. Node counting is performed either in the form of the function call to a structurally recursive function `count_leaves` (this case corresponds to the column "Before optimization" in Table 1), or in the form of the expression that results from inlining and optimizing this function call (this case corresponds to the column "After optimization" in Table 1). We measured the corresponding query evaluation time by considering `addressBook` elements with different number of `Card` elements in them (the number of `Card` elements is reflected in the left column of Table 1).

It is worth noting that the execution time in both columns in Table 1 depends linearly on the number of `Card` elements in the address book. For this example, the query after optimization can be evaluated by average 11 times faster than before optimization.

We performed more experiments on different practical XQuery queries and XML documents. The experimental results obtained show that XQuery function inlining and optimization can make query evaluation up to 76 times faster, and the running time of the inlining algorithm is negligible compared to time to evaluate complex queries.

8 Conclusion

In this paper we considered the application of function inlining technique to XML Query Language (XQuery) for optimizing XQuery queries. Two optimization criteria and type information available for optimization purposes during static analysis phase were discussed. The definition of XQuery user-defined function and the peculiarities of its inlining into the query were considered.

Table 1. Benchmarks for the example considered in Subsect. 2.1

Number of card elements in the addressBook	Execution time, seconds Before optimization	After optimization
1000	1.30	0.1
4000	5.18	0.41
16000	18.73	1.30
64000	75.11	5.31
256000	309.55	21.23
1024000	1425.96	129.18

We suggested the classification of XQuery functions from the viewpoint of their inlining, and analyzed the suitability of function inlining for optimization purposes for each of the function classes introduced. The inlining algorithm was suggested and illustrated by examples. The formal proof of the algorithm correctness was given and the limitations of the algorithm were considered.

Experiments conducted showed the efficiency of suggested inlining algorithm for optimizing XQuery queries and the important qualificative effect in speeding up query evaluation.

References

1. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft, http://www.w3.org/TR/2003/WD-xquery-20031112/, 12 November 2003.
2. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, http://www.w3.org/TR/2004/WD-xquery-semantics-20040220/, 20 February 2004.
3. M. Grinev. XQuery Optimization Based on Rewriting. Ph.D. Thesis Overview, http://www.ispras.ru/ grinev/mypapers/phd-short.pdf, 2002
4. A. Malhotra, J. Melton and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, http://www.w3.org/TR/2003/WD-xpath-functions-20031112/, 12 November 2003.
5. D. F. Bacon, S. L. Graham and O. J. Sharp. Compiler Transformations for High-Performance Computing. ACM Computing Surveys, http://citeseer.nj.nec.com/bacon93compiler.html, December 1994.
6. J. Clark and M. Makoto. RELAX NG Tutorial. Committee Specification, http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html, 3 December 2001.
7. Chang-Won Park, Jun-Ki Min and Chin-Wan Chung. Structural Function Inlining Technique for Structurally Recursive XML Queries. 28^{th} International Conference on Very Large Data Bases, August 2002, Hong Kong, China. http://www.vldb.org/conf/2002/S04P01.pdf
8. D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori and J. Robie. XML Query Use Cases. W3C Working Draft, http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/, 12 November 2003.

9. P. Fankhauser. XQuery Formal Semantics: State and Challenges. ACM SIGMOD, http://www.acm.org/sigmod/record/issues/0109/SPECIAL/fankhauser2.pdf, September 2001.
10. Sedna – Native XML DBMS. Institute for System Programming RAS, http://modis.ispras.ru/Development/sedna.htm.