Function Inlining in XQuery 3.0 Optimization

Leonard Wörteler Michael Grossniklaus Christian Grün Marc H. Scholl

Department of Computer and Information Science, University of Konstanz
P.O. Box 188, 78457 Konstanz, Germany
firstname.lastname@uni-konstanz.de

Abstract

Originally developed as a query language for XML databases, XQuery has evolved into a complete functional programming language. In order to unlock all optimization opportunities, XQuery processors therefore need to combine traditional query optimization with techniques used in optimizing compilers. In this paper, we discuss how the well-known technique of *function inlining* can be applied to XQuery. We present an implementation of function inlining based on the query processor of BaseX, an open-source XML database. Finally, a detailed quantitative evaluation demonstrates that the performance benefits obtained by blending compiler and query optimizer techniques surpass results from any one single technique.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Query processing; H.2.3 [*Database Management*]: Languages—Query languages; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords function inlining, query optimization, XML databases

1. Introduction

XQuery (XML Query Language) is a language designed to retrieve, navigate, and generate XML documents. In its first version, XQuery was mainly a query language for XML data, which extended the XPath language used to navigate XML documents. Its most prominent language construct, the FLWOR expression (**`for`**, **`let`**, **`where`**, **`order by`**, and **`return`**), has been inspired by the SQL `SELECT` statement, even though its semantics are order-preserving. XQuery 1.0 also included *function declarations* and support for *library modules*, making it feasible to use XQuery as a general-purpose programming language. The language specification defined the language itself as well as the standard library to be *purely functional*, *i.e.*, not having any perceivable *side effects* during evaluation of a query. Other examples of this class of programming languages are HASKELL [7] and CLEAN [4].

In XQuery 3.0 [8, 13], *function items*, *i.e.*, functions that can be passed around as values, bound to variables and processed by *higher-order functions*, were added. The concept was first made popular in functional and dynamic languages (*e.g.*, LISP, HASKELL, PYTHON) and later also made its way into static, imperative languages such as C# and JAVA. At the same time, FLWOR expressions were extended significantly. Additional clauses were introduced (*e.g.*, **`window`** [3]) and the previously rigid ordering between clauses (*e.g.*, **`order by`** only directly before **`return`**) was loosened. Together, these extensions to XQuery significantly strengthened non-database uses cases. For example, XQuery is increasingly used as a programming language for building REST services and web applications [11].

Higher-order functions play an important role in these bigger XQuery code bases and their more complex logic. As with any (functional) programming language, higher-order functions can also be used in XQuery to abstract over *control flow*. Additionally, the ability to build arbitrary efficient *functional data structures* [9] using higher-order functions is particularly important in XQuery. While the Candidate Recommendation [12] for the next version of the language contains *map* and *array* data structures, the current standard lacks variety in data structures. A well-optimized compiler for XQuery should support this programming style and rewrite abstracted code to achieve similar performance to non-abstracted code typical for pure query languages.

In this paper, we focus on the well-known optimization technique of *function inlining*. We present how function inlining can be applied to XQuery and show that it has the expected effect when integrated into an XQuery processor. More importantly, however, we demonstrate that the interplay of compiler and query optimizer techniques can yield a performance benefit that surpasses any one of these techniques on their own. Specifically, this paper makes the following contributions to the state of the art.

1. Application of the well-known technique of function inlining to the XQuery 3.0 language (Section 2).
2. Implementation of function inlining for XQuery in a native XML database that is used productively (Section 3).
3. Experimental evaluation that quantifies the benefits of combining function inlining with query optimization (Section 3).

Section 4 discusses related work, whereas concluding remarks are given in Section 5.

2. Function Inlining in XQuery 3.0

Relying heavily on functions and function application to structure programs and to keep their size manageable can lead to significantly slower code. The most obvious additional cost is the function call itself and the associated type checks of arguments and return value. Another (and often much more important) cause of performance loss is the optimizer's inability to "look through" the function call into the function's implementation. Since typical optimization algorithms work by matching and rewriting local patterns in a program's syntax tree, introducing additional levels of abstraction can completely disable crucial optimizations. Finally, even if the optimizer can access the function's implementation through the function call, it cannot safely rewrite the function body using information available

DBPL'15, October 27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3902-5/15/10...$15.00
http://dx.doi.org/10.1145/2815072.2815079

at the call site because that information can be different for other calls to the same function. An optimizer should therefore be able to replace a function call with the function's implementation whenever that is possible and desirable. This rewriting is called *inlining* and is well-known and widely implemented in compilers for any languages having functions or a similar abstraction mechanism.

Implementing inlining in XQuery 3.0 is much more complex than before because of the introduction of *function items*. While many languages have both anonymous functions and inlining optimizers, they are typically either *strongly typed* (which makes identifying recursive anonymous functions easier) or provide only basic support of inlining, which leaves out many opportunities because they cannot be proven as safe. In this section, we introduce the different types of functions supported by XQuery 3.0. For each type, we discuss how and under which conditions it can be inlined.

2.1 Static Functions

Static functions that can be addressed by name are the most common type of functions in programming languages. In XQuery 3.0, they are defined by the **`declare function`** statement and are the most straightforward to inline. Since XQuery does not support *ad-hoc polymorphism*, each function call is associated with exactly one function implementation.

As in other programming languages, functions in XQuery can be *recursive*, *i.e.*, there is a chain of function calls that starts inside the function's body and leads to the same function being called again. Both *self-recursive* and *mutually recursive* calls can occur and need to be detected when inlining functions, as failure to do so will result in an infinite inlining loop. In order to avoid non-termination, we apply the approach of Peyton Jones and Marlow [10] to XQuery. Following this approach, we create the *dependency graph* of the program. A function is recursive iff it is part of a cycle in the dependency graph. In order to obtain an acyclic graph, a subset of function calls that are not inlined, so-called *loop breakers*, is chosen. Many heuristics for choosing loop breakers have been proposed [10], *e.g.*, breaking the most cycles, having code structure that would not profit from inlining, and code size. Currently, we use a simple scheme that favors small functions with few calls for inlining.

2.2 Function Items

Inlining XQuery *function items* is more challenging. As the function actually called at a call site is only determined at run-time, it is often not possible to decide what to inline. Additionally, recursion is harder to detect than in the case of static functions. While function items are anonymous and can thus not call themselves by name from inside their function body, they can take a self-reference (even nested in other data) as a parameter and call themselves through that. Such recursion is not only hard to detect syntactically, but impossible to decide at compile time (*cf.* halting problem). Consider for example the following XQuery expression.

```
let $f :=
  function($f, $n) {
    if($n <= 0) then 1 else $n * $f($f, $n - 1) }
return $f($f, 42)
```

Even though the function item bound to `$f` is by nature non-recursive and only uses values given to it as arguments, the computation uses recursion. Therefore, applying the rules described in Section 2.1 would once again lead to non-termination. A simple approach to preventing such inlining loops is the use of heuristics to decide whether a function item can safely be inlined or not. For example, a function item that does not contain any function call cannot lead to an infinite loop because inlining it reduces the overall number of function calls in the program, so the inliner stops when no calls are left. It is also safe to inline function items that do not take arguments that could contain function items. Finally, function items that do not use their arguments are another obviously non-recursive class of functions. While using heuristics is simple and correct, it is easy to construct cases that are not optimized even though they are completely safe. These missed optimization opportunities weigh particularly heavy in cases where function items are used to encode data structures, as doing so yields highly nested closures with non-trivial call structures. For example, the three-element list `[1, 2, 3]` is represented in Scott encoding[1] by the following nested expression.

```
function($n1, $c1) { $c1( 1,
  function($n2, $c2) { $c2( 2,
    function($n3, $c3) { $c3( 3,
      function ($n4, $c4) { $n4() }) }) }) }
```

We found that shifting the focus from the enclosing function items to the calls in their bodies is an approach that works better in practice than relying on heuristics to decide on the inlining of function items. If a loop occurs while inlining functions, there must be some function call expression that is (possibly after some intermediate steps) inlined into itself, *i.e.*, replaced by the body of the function that it originated from in the source code. Therefore, such dynamic cycles can be avoided by tagging each function call at compile time, attaching all of the source code locations of functions it was lexically enclosed by. These tags are then updated whenever the expression containing the call is inlined into another function. Using this scheme, together with other heuristics such as limits on the size of the function body, the same inlining approach can be used for both static and dynamic functions.

This method enables powerful abstractions with almost no overhead because all abstracted-over expressions can be handed into the abstracting function as function items that are then inserted back by the inliner. For example, the following function `local:lf` (left fold) iterates over a sequence `$seq` of items and updates an accumulating parameter `$acc` that is initialized by a value `$start` using the combining function `$f`.

```
declare function local:lf($seq, $start, $f) {
  let $go :=
    function($curr-seq, $acc, $go) {
      if(empty($curr-seq)) then $acc
      else $go(tail($curr-seq),
               $f($acc, head($curr-seq)), $go) }
  return $go($seq, $start, $go)
};
```

The call `local:lf(1 to $n,1,function($a,$b){$a*$b})`, which calculates the factorial of `$n`, can be inlined to yield the following efficient expression.

```
let $go :=
  function($curr-seq, $acc, $go) {
    if(empty($curr-seq)) then $acc
    else $go(tail($curr-seq),
             $acc * head($curr-seq), $go) }
return $go(1 to $n, 1, $go)
```

2.3 Closures

A special case of non-static functions are *closures*, *i.e.*, inline functions that contain references to non-local variables. These expressions cannot be statically compiled to function items because the values of the closed-over variables are not yet available. Our XQuery compiler thus creates a CLOSURE object that gathers the closed-over values at run-time and creates a function item.

While this structure does not seem to lend itself to inlining well at first, we can make two observations. First, closures can be recursive

[1] First appears in a set of unpublished lecture notes by Dana Scott, cited by Curry *et al.* [5, p. 504].

at run-time, but inlining them cannot lead to an infinite recursion. On the one hand, they cannot close over themselves as XQuery does not have recursive **let** expressions. On the other hand, the code for inlining variables they could be bound to takes care not to duplicate the costs of evaluating the closure and creating the function item. This also guarantees that a closure can never end up as argument to an inlinable call to itself. Second, if a CLOSURE expression is inlined as callee into a function call expression, it can thus be inlined directly without additional checks, because all parameters as well as closed-over variables must be in scope.

If, however, the values of all closed-over variables become known at one point during compilation, *i.e.*, if the closure contains only mappings from local variables to values, it can then be rewritten into a function item at compile time. This observation can be generalized by interpreting the closed-over variables as a set of **let** bindings that create a mapping from variables in the outer scope to those in the closure's function body. The closure `function($j) { $i * $j }` would thus be interpreted as follows.

```
let $i′ := $i return function($j) { $i′ * $j }
```
closed-over variables — resulting *function item*

With this model it becomes easier to reason about the treatment of non-local variables. While the bindings in a closure initially always have a reference to a variable from the outer scope on the right-hand side, variable inlining may replace it with an arbitrarily complex expression if it determines that the transformation is beneficial. A binding may be moved into the function item's body if it does not depend on the outer scope any more. This is trivially true for fully evaluated values, but may also hold for other more complex and expensive expressions. Expressions in the non-local bindings are evaluated once when the function item is created. If a binding is moved into the function, it is evaluated each time the function is called. It is therefore not advantageous to move expensive expressions if the function item is called more than once. As soon as there are no non-local bindings left in the closure, it can be compiled to a function item at compile time.

Apart from values, which can always be moved into the function, another important case to optimize is that of nested closures. When an outer closure closes over another inner one, any references to the outer scope in this expression must be in the inner closure's closed-over expressions. These are evaluated when the inner closure is, which in turn is exactly once per evaluation of the outer closure into a function item. It is therefore safe to bind those closed-over expressions to temporary variables in the outer closure's bindings and then copy the result over into the inner one. This can be seen in the following query. Colored boxes highlight the nested closures.

```
for $i in 1 to 10
let $f :=
  let $g := let $i′ := $i return
            function($y) { $i′ * $y } return
  function($x) { $g($x * $x) }
return $f($f(42))
```

Since `$g` closes over `$i`, it cannot be compiled to an item. Therefore, `$f` has to close over `$g` and none of the functions can be inlined. However, using the transformation discussed above, we can introduce a new variable `$i″` to capture `$i` inside `$f`.

```
for $i in 1 to 10
let $f :=
  let $i″ := $i,
      $g := let $i′ := $i″ return
            function($y) { $i′ * $y } return
  function($x) { $g($x * $x) }
return $f($f(42))
```

Now `$g` does not depend on `$i` and can thus be inlined into `$f`. It can then be inlined into its call and there is only one closure left.

```
for $i in 1 to 10
let $f :=
  let $i″ := $i return function($x) {
    let $y := $x * $x return $i″ * $y
  }
return $f($f(42))
```

Chains of closures as described here occur most often when higher-order functions are used to abstract out programming patterns into reusable library functions. The parts of the code that differ between uses are passed in as function items and called by the library code. In the application code those often have to be closures in order to be able to use data from the surrounding scope. When multiple of these layers of abstraction are built on top of each other, this directly leads to closures capturing each other and, thus, nesting that needs to be resolved.

3. Experimental Evaluation

The inlining techniques presented in the previous section were implemented in the open-source native XML database BaseX[2] and have been in productive use since version 7.8. In this section, we will use this implementation to characterize the performance benefits that can be obtained from function inlining in XQuery 3.0. In particular, we study the reduction of execution time both in a synthetic and a practical setting. Finally, we analyze the effect that an optimizing compiler can achieve in combination with a query optimizer. All results reported in this section have been measured on a personal computer (Intel i7-3740QM CPU, 8 GB RAM) using the Oracle 64-bit Java VM 1.8.0_b20-b26 on a 64-bit Microsoft Windows Professional 8.1. Running times are averaged over 10 runs for the queries in Section 3.1 and 200 runs for the others.

3.1 Newton's Method for Square Roots

In order to understand the performance benefits that can potentially be achieved, we study XQuery 3.0 function inlining in a synthetic setting. As a program, we chose an implementation of Newton's method to calculate *square roots* of floating-point numbers. Since the code is written using three levels of abstraction, each implemented as a static higher-order function that takes a function item as argument, it lends itself to aggressive inlining of dynamic functions and closures. Applying all described optimizations to this query, the resulting code is reduced to a single recursive function.

We measured the execution times of two versions of this query, the original one and one optimized using the approach presented in this paper, in four different system configurations. We compare versions 9.5 and 9.6 of the optimized Saxon Enterprise Edition to BaseX 8.0 with the unnesting of nested closures activated and deactivated. Figure 1 presents an overview of the results. Since we do not have access to Saxon's optimizer framework, we measured optimized execution times using the query string output by BaseX's optimizer. While the results between the two Saxon versions do not change significantly, the optimized query is faster by almost a factor of four. In BaseX, the query also runs faster with optimization activated, but the difference is more pronounced without closure flattening.

3.2 FunctX Library

For an initial quantification of the performance benefits that can be expected in a practical setting, we turn to FunctX[3], a popular and

[2] http://www.basex.org

[3] http://www.xqueryfunctions.com

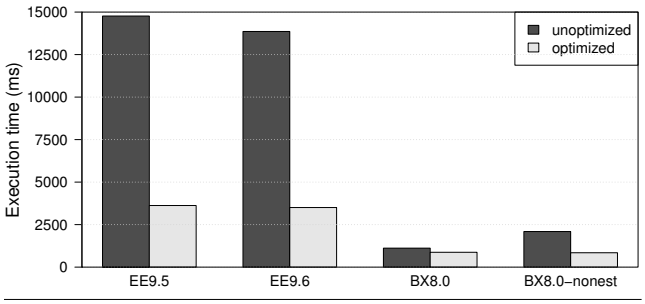

Figure 1. Performance comparison of the square root function.

widely-used XQuery library of utility functions. The XQuery code below uses the `functx:date()` and `functx:days-in-month()` functions to compute the total number of days of all years from the year 1 to the year 2015.

```
fn:sum(
  for $year in 1 to 2015
  for $month in 1 to 12
  let $first := functx:date($year, $month, 1)
  return functx:days-in-month($first)
)
```

Executing this XQuery program without function inlining takes on average 92.5 ms, whereas optimizing it as described in this paper yields a 41% reduction of the average runtime to 54.5 ms. Clearly, this is a very initial result and a more comprehensive study would be required to fully characterize all performance benefits. Nevertheless, even this first result can help to contextualize the results obtained in the synthetic setting.

3.3 Interplay of Compiler and Query Optimizer

In order to demonstrate the combined effect of optimizing compiler and query optimizer, we use the following query that computes the relative population of each country in the CIA World Factbook[4] w.r.t. to the population of India.

```
declare function local:f($w, $c) {
  let $in := $w/country[@name="India"]/@population
  return ($in, $c/@population div $in)
};
let $world := doc('factbook')/mondial
for $country in $world/country
return local:f($world, $country)
```

In this form, the query suffers from two inefficiencies. First, there is the overhead of repeated function calls to `local:f($w, $c)`. Second, the query optimizer is not able to utilize value indexes, since all attribute accesses are done inside the function body, relative to an element that is unknown at compile time. Applying the techniques presented in this paper, the above query is rewritten to the following, BaseX-specific, representation.

```
let $in := db:attribute("factbook", "India")
  /self::name/parent::country/@population
for $country in doc("factbook")/mondial/country
return ($in, $country/@population div $in)
```

Now that the compiler has completely inlined the function, the query optimizer can again deal with a single FLWOR expression and choose an index-based access plan. This combined optimization leads to a performance improvement of several orders of magnitude. Whereas the unoptimized query takes on average 199.3 ms to complete, the optimized has an average execution time of 4.1 ms.

[4] `http://files.basex.org/xml/factbook.xml`

4. Related Work

Inlining as an optimization technique is well studied in a wide range of application domains. Most relevant to XQuery is again the functional programming community, since the stronger guarantees of this class of languages allow for more aggressive optimization. Peyton Jones and Marlow describe their approach taken in the GHC Haskell compiler [10] in great detail. The challenges that a more weakly and dynamically typed language (like XQuery) poses can be studied in Baker's work on inlining recursive functions in Scheme [2]. An even more general framework for inlining on lower-level intermediate code is discussed in a paper by Ayers *et al.* [1]. Since XQuery 3.0 itself is a rather young language, its current implementations are not yet optimized to their full potential, and we know of no publications documenting their optimization strategies. An approach for XQuery 1.0 is shown by Grinev and Lizorkin [6] for the Sedna processor.

5. Conclusion

We argue for a combination of optimizing compiler and query optimization techniques in order to efficiently execute complex XQuery programs. In this paper, we revisited the well-known technique of function inlining and showed how it can be applied in the case of XQuery 3.0. Our performance evaluation demonstrated the significant benefits that can be obtained from function inlining alone, but also in concert with traditional query optimization techniques. The work presented in this paper is a first step towards hybrid optimization techniques, which we believe can also benefit other computationally complete query languages, such as PL/SQL.

References

[1] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive Inlining. In *Proc. Intl. Conf. on Programming Language Design and Implementation (PLDI)*, pages 134–145, 1997.

[2] H. G. Baker. Inlining Semantics for Subroutines Which Are Recursive. *SIGPLAN Not.*, 27(12):39–46, 1992.

[3] I. Botan, D. Kossmann, P. M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius. Extending XQuery with Window Functions. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 75–86, 2007.

[4] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. CLEAN: A Language for Functional Graph Writing. In *Proc. Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, pages 364–384, 1987.

[5] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland Publishing Company, Amsterdam, 1972.

[6] M. N. Grinev and D. Lizorkin. XQuery Function Inlining for Optimizing XQuery Queries. In *ADBIS (Local Proceedings)*, 2004.

[7] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, S. Peyton Jones, M. Reeve, D. Wise, and J. Young. Report on the Programming Language Haskell, a Non-strict Purely Functional Language (Version 1.0). Technical Report YALEU/DCS/TR777, Yale University, 1990.

[8] M. Kay. *XPath and XQuery Functions and Operators 3.0*. World Wide Web Consortium, Dec. 2011.

[9] C. Okasaki. Red-Black Trees in a Functional Setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[10] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12:393–434, 7 2002.

[11] A. Retter. RESTful XQuery: Standardised XQuery 3.0 Annotations for REST. In *Proc. of XML Prague*, pages 91–123, 2012.

[12] J. Robie and M. Dyck. *XQuery 3.1: An XML Query Language*. World Wide Web Consortium, Dec. 2014.

[13] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. *XQuery 3.0: An XML Query Language*. World Wide Web Consortium, Dec. 2011.