Compiling XQuery to Java bytecodes

Per Bothner
<per@bothner.com>

1. INTRODUCTION

XQuery is new language currently being standardized by the World Wide Web Consortium (W3C). Its application domain is querying, filtering, and generating XML files – or any data matching the XML infoset model. There is a lot of industry and research interest in XQuery: The "database community" is interested in XQuery as a query language for XML databases, and the "document community" is interested in querying collection of documents. The former tend to have a relatively simple and regular structure, while the latter have a more irregular and deeply-nested structure. Most of the implementation effort currently appears to be driven by existing database vendors, which want to improve their XML offerings. This leads to implementation strategies similar to existing relational database implementation, such as optimizating to make uses of indexes, creating a query plan, and result-driven (demand pull rather than data push) execution.

Qexo is an implementation that is unusual in a number of aspects:

- Qexo compiles a query to a general-purpose instruction-set, specifically Java bytecodes (which can be straightforwardly compiled to machine code) rather than a "plan" or other interpreted representation.
- Execution flow more closely follow the "natural" program structure rather than being driven by demand pull.
- Qexo is well integrated in Java, including access to arbitrary Java objects and easily calling methods in Java's extensive class library.
- Qexo supports texual XML files as well as a compact internal DOM. We will touch on extending it to special-format or indexed databases, but currently it is best suited for ad hoc queries of modest files, or bulk processing where execution speed proportional to file size is ok.

Informal Proceedings of the *First International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P),* June 17-18, 2004, Paris, France.

- Qexo is Free GNU Software (open-source), written by an individual, rather than a company or a research group.
- It is based on an existing multi-language framework, with a compiler originally written in 1996 to compile the Scheme functional language.

2. COMPILING TO BYTECODES

Qexo's basic structure is based on the existing Kawa [2] framework, of which Qexo is part. (We use "Qexo" to refer to the XQuery-specific support in Kawa, while "Kawa" refers to the framework as a whole regardless of language.) The Kawa project started in 1996 with compiling the Scheme functional language to Java bytecodes. Over the years it has developed into a more general framework that can compile multiple languages. For each supported language, you can use Kawa in multiple "modes", including interactively typing expressions at a command-line prompt, or compiling a "query" in different modes.

Kawa supports both a compiler and an interactive "interpreter". But the interpreter is very limited as it is only used for the most simple expressions. Most programs are "interpreted" by compiling them to bytecode in an internal byte array, and then a class is compiled on-the-fly using Java `ClassLoader` mechanism. This implementation supports fast interactive response without sacrificing performance.

3. LAZY VS DIRECT EVALUATION

XQuery implementation by database people [3] tend to be written using database techniques, where a query is compiled to a plan, and then result of the query is generated lazily when demanded by the application that made the query. This has the big advantage that you only generate the results and perform the calculation that are needed, and some optimizations fall out by themselves. The disadvantage is that representing the state of a computation requires non-trivial data structures and book-keeping: You need a special-purpose interpreter to execute the plan, thus getting an extra layer of interpretive overhead.

It is instructive to consider non-strict functional programming languages such as Haskell [1], whose specification require lazy demand-driven execution. However, this is quite expensive, so optimizing implementations perform *strictness analysis* [4] to determine when it is safe to convert the demand-driven execution to a direct "eager" execution. The

specification of XQuery permits either implementation style, but experience from Haskell suggests that direct execution will be easier to make efficient, at least when CPU use by the query itself is a major factor.

True, some queries using direct evaluation will take exponentially or even infinitely longer than using lazy evaluation. However, queries that depend on lazy evaluation are not portable and should be re-written. In contrast, I believe most queries can be implemented an order of magnitude faster using direct rather than lazy evaluation. This assumes that most of the execution time is executing the logic of the query itself, rather than in library functions or reading from disk: Lazy evaluation makes more sense if execution is likely to be I/O-bound.

Best performance might need a combination of techniques. You probably want to use lazy evaluation for quantified `some` expressions, selections using numerical predicate, and some functions, such as `fn:exists`. I don't know if anyone has tried such a hybrid approach.

Qexo mostly uses a more direct execution model. The big advantage of this is that the state of the computation can be expressed using the target machine's program counter and execution stack. The execution state maps directly and efficiently to the (virtual) machine program counter and stack. No extra level of interpretation is needed.

4. STREAMING

Eager evaluation does not require that every value in the XQuery semantics is realized as a object at run-time. Qexo tries to stream sequences using an event-driven interface like SAX. Consider the following example.

```
for $i in (10, 20) return ($i+1, $i+2)
```

This can be translated to:

```
void main (Consumer output) {
  temp_1(10, output);
  temp_1(20, output);
}
void temp_1 (Object i, Consumer output) {
  output.writeItem(i + 1);
  output.writeItem(i + 2);
}
```

Kawa's `Consumer` interface is an abstract "data sink", which is conceptually similar to SAX2's `ContentConsumer`, but generalized to forests of general values, as needed by the XQuery data model.

This is much more efficient than a demand-driven (client pull) translation of this query, which has to use two cursors, one for each sequence expression, to track which value to return next. To lazily get the first result, we would first have to request a value from `($i+1, $i+2)`, which causes a request for the first value of `$i` or `(10, 20)`. When the client requests the next result, we need the next value of `($i+1, $i+2)`, using the same value of `$i`. For the next result, there are no more values in the "inner" sequence, so it has to request a new value for `$i`, before re-evaluating `($i+1, $i+2)`. The necessary bookkeeping is substantial for applications that are CPU-bound, but it is probably well worth it if it makes it easier to minimize disk or network accesses.

5. COMPILER OVERVIEW

Qexo compiles an XQuery module as follows:

1. *Parsing.* Qexo has a hand-written recursive-descent parser. It keeps track of line and column numbers.

   The result from the parser is an `Expression`, which is a language-independent abstract syntax tree. Some special XQuery forms, such as FLWOR expressions, are represented as calls to special built-in functions. We'll see examples later.

2. *QName expansion.* Resolving namespace prefixes to namespace URIs must be done after parsing, because a namespace prefix can be used in an element constructor before it is defined by a namespace attribute.

3. *Module import.* Importing library modules causes some complications. At the time of writing the public XQuery drafts are inconsistent. Until these issues are resolved it is premature to say too much about module import.

4. *Name resolution.* Resolve variable references and function calls to their definitions.

5. *Analysis and optimization passes.* There are a number of passes that work on the `Expression` tree. Calls to certain built-in functions (such as basic arithmetic) are re-written to more efficient forms. We do some ad hoc type propagation. We figure out how functions can be compiled into methods or inlined, and how variables are assigned to virtual machine registers or fields. New query optimization passes can be added here.

6. *Code generation.* Qexo generates bytecode by recursively traversing the abstract syntax tree. We can generate bytecode in different modes, depending on how it is to be used and specified options.

7. *Output.* The bytecode can written out to a `class` file. Alternatively, a `ClassLoader` can take the bytecode, as stored in a `byte` array, and directly create a "live" class, without writing out any files.

6. EXPRESSIONS

Qexo, like other XQuery implementation, translates XQuery surface syntax into a simplified "core XQuery". Unlike other implementations, the core representation is not designed for XQuery, but uses a nested tree of language-independent `Expression` objects. Kawa has a small number of sub-classes of the abstract `Expression` class, including ones used for contants, variables reference, anonymous function values, lexical scoping blocks, and function application. Special XQuery forms are represented as calls to built-in functions. For example:

```
<p>sum: {3+4}</p>
```

This is converted into a data structure that has the following structure:

```
ApplyExp[
  function: makeElement,
```

```
  args: {QuoteExp[value: "p"],
         QuoteExp[value: "sum :"],
         ApplyExp[
           function: +,
           args: {QuoteExp[value: 3],
                  QuoteExp[value: 4]}]}]
```

An `ApplyExp` is used for procedure application. Its `function` property specifies the procedure to call; and its `args` property is an array of parameter expressions. A `QuoteExp` wraps a literal Java object, and turns it into a constant expression that always evaluates to that object. The `makeElement` function takes an element tag, followed by zero or more attribute expressions, followed by zero or more expressions for the children.

More complex control structures may have sub-expressions that need to be evaluated out of order. We handle these by wrapping them in an anonymous function, represented by a `LambdaExp`. Consider for example:

```
for $i in (2, 3) return $i+10
```

This is represented by:

```
ApplyExp[
  function: valuesMap,
  args: {
    LambdaExp
      params: {$i},
      body: ApplyExp[
        function: +,
        args: {ReferenceExp[$i], QuoteExp[value: 10]}],
    ApplyExp[
      function: appendValues,
      args: {QuoteExp[2], QuoteExp[3]}]}]
```

The built-in `valuesMap` takes two arguments: a function, and a sequence. It applies the function to each element of the sequences, returning the sequence of the concatenated results. A simple evaluation of this expressions yields the correct result, but does so inefficiently; below we will show some ways Qexo optimizes such expressions.

7. CODE GENERATION

To compile an `Expression`, the Qexo compiler invokes its `compile` method:

```
public abstract void
compile (Compilation comp, Target target);
```

The `Compilation` parameter manages the state of the current compilation, including the current method being generated. When `compile` is invoked on an `Expression`, it will append to the current method bytecode instructions to evaluate the `Expression`. What is the best strategy for doing so, and where to leave the result of the `Expression`, may depend on the expression's context. Kawa uses a simple and effective convention: when an outer expression needs to compile a sub-expression, it passes to the latter's `compile` method a `Target` object that specifies what the sub-expression should do with its result.

The default `Target` expects the result to be pushed onto the JVM stack as an `Object` reference. I.e. if such a target is passed to a `compile` method for an expression, that method is responsible for evaluating the expression and leaving the result on the JVM stack, where the caller can make use of it. If some other `Target` is passed to a `compile` method, then the method must send the result to the given `Target`. The easiest way to do this is to leave the result on the JVM stack, and then call the `Target`'s `compileFromStack` method, which is responsible for moving the result from the JVM stack to the desired target. (For the default `Target` the `compileFromStack` method does nothing, since its caller has left the result where it needs to go.) Thus a `compile` only needs to be able to evaluate a result and leave it on the JVM stack; it can handle other kinds of `Target`s by just calling their `compileFromStack` method. However, it has the *option* of inspecting the passed-in `Target` if that may lead to more efficient code.

For example, the simplest kind of `Target` is an `IgnoreTarget`, which is used when an expression is evaluated for its side-effects, but the result will be ignored. (This isn't useful for XQuery, but it is used by Scheme and other languages.) The `IgnoreTarget`'s `compileFromStack` method just pops the result from the JVM stack and ignores it. If an expression has no side-effects and its `compile` method was passed an `IgnoreTarget` it generates no code.

The `compileFromStack` method of a `ConditionalTarget` is more interesting. It pops off a value, converts it to a boolean value (in a language-dependent manner), and then jumps to either of two labels depending on whether the value is true or false. When Kawa compiles a conditional (`if`) expression, it creates a `ConditionalTarget` for compiling the test expression. This makes it easy to optimize boolean expressions as jumps.

We'll look at `ConsumerTarget` and `SeriesTarget` in the next sections.

8. OPTIMIZING FOR EXPRESSIONS

Much of XQuery's power comes from the "FLWOR" expressions, and compiling them efficiently is a challenge. To avoid materializing the whole `for` clause sequence as an object, Qexo uses a special `Target` when compiling the `for` expression. In the case of a `SeriesTarget` the expression is evaluated in a mode where each item in the resulting sequence calls a given function. In the case of a `FLWOR` expression, the function is the anonymous function representing the `return` clause. Consider the earlier example:

```
for $i in (10, 20) return ($i+1, $i+2)
```

Qexo compiles `(10, 20)` with a `SeriesTarget` that references the anonymous function:

```
function($i) { ($i+1, $i+2) }
```

Compiling `(10, 20)` with a `SeriesTarget` is a matter of compiling first `10` and then `20` with the same `SeriesTarget` and putting the bytecode for the two pieces in sequence. Compiling `10` (or any singleton expression) is then just a matter of evaluating the value and calling the anonymous function.

The `return` clause function is implemented using the "internal subroutine" instructions that Java traditionally uses

for `finally` clauses. This allows direct and efficient access to surrounding variables, and it's an interesting use of a feature of the JVM that is not accessible from the Java source language.

Qexo currently does need to reify the sequence in the case of more complex `for` clauses. Optimizing the general case is not yet done, but it is fairly easy to do at the cost of allocating an inner class instance. Consider for example:

```
for $x in f($arg) return use($x)
```

It can be compiled to the following:

```
void main(Consumer out) {
  f(arg, new Consumer {
           void writeItem(Object x) {
             use(x, out);
           };
         }
}
```

The idea is that each time `f` yields an item, it calls the `writeItem` of its passed-in `Consumer`. That happens to be the unnamed inner class shown above, where the body of the `writeItem` method is the compilation of the FLWOR's `return` expression. This calls the `use` function, passing it the outer (original) `Consumer`. This mechanism can handle general `for` expressions without materializing the `for` sequence, and with little overhead.

9. COMPILING FUNCTIONS

An XQuery function is compiled to a Java method whose name is generated from the function name. A query body is treated as a zero-argument function which we here call `main`.

Each XQuery formal parameter results in a corresponding formal parameter in the generated method. In addition there is a compiler-generated `out` parameter, which has type `Consumer`. The result of the function is written to this `Consumer`; hence the generated method's return type is `void`.

Here is a simple example function:

```
declare function my-func ($delta, $x) {
  if ($delta=0)
  then $x
  else ($x+$delta, $x-delta)
}
```

This is compiled to:

```
void myFunc(Object delta, Object x,
            Consumer out) {
  if (NumEqual(x, 0))
    out.writeItem(x);
  else {
    out.writeItem(NumAdd(x, delta));
    out.writeItem(NumSub(x, delta));
  }
}
```

`NumEqual`, `NumAdd`, and `NumSub` are static methods in the runtime library; with appropriate type declarations Kawa can generate more specific code or inlined arithmetic.

Generating code like this is straight-forward. Kawa creates a `ConsumerTarget` that contains the name (actually virtual register number) of the `out` temporary, and passes this `ConsumerTarget` instance to the `compile` method for the function's body. The same `ConsumerTarget` instance gets passed on when compiling the conditional and sequence sub-expressions.

The Qexo environment creates the initial `Consumer` that it passes to the `main` function. What kind of `Consumer` to pass depends on how the query is invoked. By default Qexo writes out the result of a query to the standard output stream using XQuery serialization. To do that, it allocates an instance of a `Consumer` subclass such that methods like `writeItem` call appropriate output functions.

Compiling a function call is simple. The actual parameters are compiled with a default `Target`, leaving the result on the JVM stack. If the target for the function call as a whole is a `ConsumerTarget`, we just pass the current `CallContext` and `Consumer` to the method as the context parameter. Otherwise, the compiler generates code to collect the output from the function (which writes to a `Consumer`) into a sequence object. A `TreeList` helper class makes this simple and reasonably efficient.

More efficient function calls can be done with global analysis, which can cause functions to be inlined or use an optimized calling convention. Kawa does some of this, but the current focus is aimed at Scheme, where nested and anonymous functions are more of a priority.

9.1 Tail calls

A *tail-call* is a function call that is the last expression executed in a function body. It is desirable to optimize tail-calls so that they can execute without growing the call frame stack. This allows many recursive functions to execute on large data sets (such as long sequences) without running out of stack space. Unfortunately, the Java virtual machine does not optimize tail-calls, so a directly mapping of XQuery function calls to Java methods invocation will not optimize tail-calls.

The solution is to split up a function call into three parts:

1. Evaluate the argument expressions, and leave the result in a well-known location. Leave a reference to the function we want to call in another well-known location.
2. Return from the method that implements the calling function, which releases its stack frame.
3. A generic driver calls the function, as specified in the second well-known location, using the previously saved argument values.

For "well-known locations" we could use static fields, but that would not work for multiple threads. Instead, we use non-static fields of `CallContext` class, and use a separate `CallContext` instance for each thread. The current thread's `CallContext` is accessible as a `ThreadLocal` variable, but

for performance we pass it along on each function call as an implicit parameter.

In previous sections we indicated that each function gets an implicit `Consumer` parameter. That is not quite right. The implicit parameter is actually a `CallContext`, which has a field pointing to the current `Consumer`. So the `myFunc` function above actually is compiled thus:

```
void myFunc(Object delta, Object x,
            CallContext ctx) {
    Consumer out = ctx.consumer;
    ...
  }
}
```

9.2 Procedure values

Qexo creates a field for each XQuery function, which contains a `Procedure` object for referencing the function as a value. Being able to use a function as a value is essential for functional languages, such as Scheme, but it isn't strictly needed for XQuery. However, Qexo uses function values to implement for tail-call elimination. discussed above. Also, function-specific optimizations (discussed below) are implemented using special methods of the `Procedure`.

9.3 Inlining

Kawa provides two hooks that the compiler can use to optimize or customise a function call. When the compiler processes a function call, it checks if the called function is a known procedure, and if so if the procedure implements either of `CanInline` or `Inlineable` interfaces.

If a `Procedure` implements the `CanInline` interface, the compiler calls its `inline` method at tree rewriting time, passing in the `Expression` for the function call. The `inline` returns a new `Expression` that replaces the original.

If the procedure is a pure function and the arguments are constants, it can replace the call by the result value. More commonly, it will know the argument types at this point so it can replace the call by a type-specific variant. For example, it may replace a call to a generic function such as addition with an invocation of a known Java method. Another example is the `invoke` library function which takes an object expression, a string expression that names a method, and other parameters. The `inline` method of `invoke` attempts to resolve it to a call to a specific method, which can be compiled much more efficiently.

The `Inlineable` interface is used during code generation. Before generating bytecode for a function call, Kawa checks if the called function is known and implements `Inlineable`, If so, instead of generating general-purpose bytecode to evaluate the arguments and then call the function, Kawa calls the procedure's `compile` method which then is responsible for code generation. This allows instruction-level customization. For example, if the operands to the addition operator are primitive (non-object) 32-bit integers, the `compile` method can emit a single `iadd` instruction to add them.

The `compile` can also do special control flow. For example, the `ValuesMap` class is used to represent a `for` expression. It calls a function (normally an anonymous known "lamba expression" representing the `return` clause), once for each item in a sequence, and concatenates the results. Its `compile` method attempts to inline the call to an efficient loop.

10. NODE REPRESENTATION

There are a number of ways one would represent a node in Java. The obvious way is to use W3C DOM's standard `Node` interface, but this requires one object per node, and (unless you're quite clever) lots of pointers. This is expensive in terms of space, construction time, locality, and GC traversal overhead. Qexo represents a node as a pair consisting of an object that extends `AbstractSequence` plus a 32-bit integer "position". The integer identifies a particular node or position; it is a magic cookie that only has meaning in the context of its owning `AbstractSequence`. Since the position is a resource that is managed by the `AbstractSequence`, there is no problem with a database containing more than 2^{32} nodes, as long as clients only need to reference 2^{32} at a time.

`AbstractSequence` is an abstract class, which is used for many purposes: nodes, sequences, Scheme lists and vectors. The `NodeTree` sub-class is used for nodes. It stores an entire document or document fragment in two arrays: a character array, and an `Object` array. "Pointers" between nodes are relative indexes stored in the character array using one or two 16-bit characters. The representation uses a "buffer gap" which allows efficient insertion and deletion of nodes near the gap. This representation is very compact, easy to append to, and supports efficient navigation (though some tuning of the basic design may be worthwhile). A position cookie is just an index into the character array. This works fine for read-only nodes. For modifiable nodes and documents we use an indirection table: the indexes in the indirection array are used for the magic cookies, while the values of the array are indexes into the `NodeTree`'s character array.

The XMark "standard" 100MB test file (116 million bytes) is read by Qexo into an array of 104 million 16-bit Java characters, plus a 200-element object array of pointers to shared element and attribute names. It took a little over a minute to read the file, on a 1GHz PowerBook with 512MB of memory. Simple XPath selections using this representation run very quickly. In contrast, Saxon 7.9.1 needed about twice as big a heap, and took almost 8 times as long, largely due to increased paging. (The "user" process time was only 50% more with Saxon.)

To handle large persistent or remote databases, Qexo would need a new class derived from `AbstractSequence`. This class would handle caching and communication with the database. It would manage position integers which could be datebase keys or other proxies for the actual database nodes. That is not to imply this would be a trivial task: There are some places in Qexo that assume `NodeTree`, and they would have to be generalized. Making use of indexes would require teaching the Qexo optimizer about them. Updates and transactions will bring in a whole new set of issues.

For convenience Kawa provides a set of wrapper classes that implement the W3C DOM interfaces. For example the class `KNode` implements the `org.w3c.dom.Node` interface. This is an object that has two fields: a reference to an

`AbstractSequence` container, and a 32-bit integer position. A `KNode` does not carry node identity, and can be quite transitory. It is used when a node needs to be represented as an `Object`.

11. EXTENSIONS

Qexo has some non-standard extended features. Here are some of the more interesting ones.

11.1 Calling Java Methods

Qexo (following many XSLT implementations) uses special namespaces to name Java classes. For example:

```
declare namespace JInt = "class:java.lang.Integer";
JInt:toHexString(255)
```

This invokes the static `toHexString(int)` method in the Java class `java.lang.Integer`, evaluating to the string `"ff"`. You can also invoke non-static methods (passing the `this` receiver as the first parameter), or construct new objects (using `new` as the method name). The compiler picks the best matching method using the available type information.

As a further convenience, you can just use a classname directly as a prefix, assuming there is no matching in-scope namespace, and the class is in the compile-time classpath. For example:

```
java.lang.Object:toString(3+4)
```

This calls the `toString` method of the object representing 7, yielding the string `"7"`.

11.2 Servlets

Kawa has built-in suppport for automatically compiling an XQuery query (or other Kawa-supported language) to a servlet. A *servlet* is a kind of Java class that is executed in an appropriate web server in response to HTTP requests. The result of the query becomes the HTTP response. Here is a trivial but valid servlet:

```
<html><body>
  <p><b>Hello</b> world!</p>
</body></html>
```

Qexo has standard functions for querying the HTTP request and setting HTTP response parameters. There is also a helper class that automatically compiles an XQuery source file to a servlet class whenever the source is updated. (Kawa just caches the compiled class internally. Because Kawa's compiler is so fast, there is little point saving the compiled class on disk, the way JSP does it.) This provides a simple low-overhead way of writing "web applications".

11.3 Interactive console

You can interactively type commands at a "console":

```
(: 1 :) declare variable $ten { 10 };
(: 2 :) declare function scale ($x) { $ten * $x };
(: 3 :) scale(3)
30
(: 4 :) declare variable $hundred { scale($ten) }
(: 5 :) scale($hundred)
1000
```

The command prompt includes line numbers, to help with error messages, and is in the form of a comment, to aid cut-and-paste. Declarations add names to the console state, while expressions are evaluated and their result printed. Semi-colons after declarations are optional.

12. STATUS

Qexo implements most but not all of the November 2003 XQuery draft. Many of the standard functions are missing, as is support for the `order by` clauses, and schema validation. There is poor or missing support for some of the standard data types. I hope to add these in the next few months. There is some ad hoc and incomplete static typing. A full implementation of static typing will be added later.

For more information, including download and usage instructions, see the Qexo web site (http://www.qexo.org), or the general Kawa site (http://www.gnu.org/software/kawa), or email the author and implementor at `<per@bothner.com>`.

The Kawa compiler and Scheme runtime has for years been successfully used in both research and production environments, by a small but enthusiastic user group. Early XQuery adopters are doing the same with Qexo. The goal of Qexo is an efficient, self-contained, and complete XQuery implementation, which because of its open-source nature can be tailored to different needs and environments.

13. REFERENCES

[1] Simon Peyton Jones (ed). *Haskell 98 Language and Libraries* The Revised Report. Cambridge University Press. 2003. See also the Haskell web site (http://www.haskell.org).

[2] Per Bothner. *Kawa: Compiling Scheme to Java* Lisp Users Conference (Berkeley). 1998.

[3] Florescu et al (BEA). *The BEA/XQRL Streaming XQuery Processor* Proceedings of the 29th VLDB Conference. 2003.

[4] Philip Wadler and R.J.M. Hughes. *Projections for Strictness Analysis* In *Functional programming languages and computer architecture*. Springer-Verlag. 1987.