Expression Elaboration

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper describes an approach to evaluation of expression-based languages such as XSLT, XQuery, and XPath, in which nodes on the expression tree output by the language parser are converted to lambda expressions in Java, Javascript, or C#, with the aim of doing as much work as possible once only, in advance of the actual expression evaluation.

1. Introduction

Traditionally, when processing a language such as XSLT, XQuery, or XPath, there is a choice of two approaches: interpretation, or code generation.

In its pure form, interpretation works by constructing a parse tree of the source code, and then writing an interpreter that evaluates the constructs on this parse tree, typically in bottom-up fashion: a node on the tree is evaluated by first evaluating its operands (represented as child nodes on the expression tree), and then combining the results according to the semantics of the relevant operator.

In practice it is possible to improve the performance of the interpreter by analyzing and modifying the expression tree before evaluation starts: examples of these processes include resolving references (such as references to variables and functions), inferring types, and optimizations such as loop-lifting (pulling code out of a loop to avoid repeated execution). Declarative languages like XSLT, XQuery and XPath benefit greatly from such optimisations.

By contrast, code generation in its pure form takes the parse tree and converts it into a sequence of machine instructions that are then executed to evaluate the program. Today, to achieve portability, these will normally be instructions for a virtual machine (such as the Java VM) rather than physical hardware.

In practice the two approaches are not quite as distinct as it might appear, and it's certainly possible to use a blend of both. In particular, even when code generation is used, much of the generated code will consist of calls into a run-time library.

Both approaches have been used in the Saxon product. When code-generation was first introduced, it often delivered a performance boost of the order of 25% (though the range was anything from 0% to 50%). However, there was a penalty: compile time costs increased. Given that in many workloads, stylesheets are compiled every time they are executed, this turned out to be a poor trade-off; it is

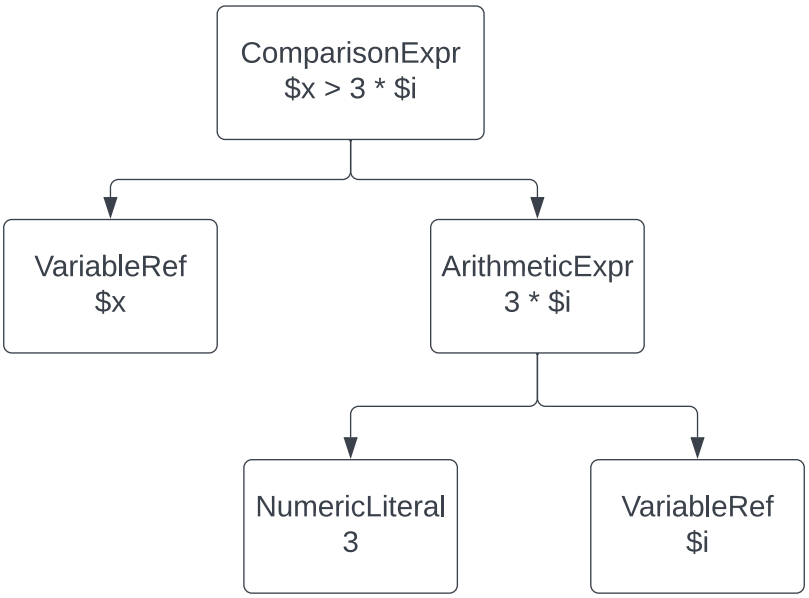
quite common for compilation costs to exceed run-time costs by an order of magnitude.

It's also noticeable that the benefits of generating Java byte-code have declined over time. It's hard to be certain about the causes of this, but we suspect it's primarily because the Java hot-spot compiler has improved over time to the extent that it can often speed up our interpreted code to make it just as fast (or sometimes faster) than the code that we laboriously generate ourselves. That may be partly because the bytecode that Saxon generates is rather different from the bytecode that the Java compiler generates, and the hot-spot JVM (naturally) is optimised for the latter. It may also have something to do with locality of reference: in a modern CPU, the main bottleneck is not the speed of executing instructions, but the speed of moving data from main memory into the CPU cache, which in turn benefits substantially if all the data (and code) needed to execute some function is in nearby storage locations, so they can be transferred to the CPU en bloc, thus improving CPU cache hit rates. Generating code that takes advantage of these low-level effects is a specialized skill, and it's not surprising if the team working on the hot-spot compiler are better at it than we are.

In the Javascript version of Saxon (SaxonJS) we developed an alternative approach to expression evaluation, which has proved very successful. We call this "elaboration" (a term borrowed from Algol68, though we don't claim to use it with precisely the same meaning.) This approach relies heavily on the fact that the languages we are dealing with are side-effect free, which gives us a lot more freedom in rearranging how code is executed. We've recently been extending its use to the Java and C# versions of the Saxon product, and this paper reports preliminary results of this work. This paper describes how expression elaboration works. But first we'll look in more detail at how the two existing strategies, the interpreter and the bytecode generator, are implemented, and at their strengths and weaknesses.

2. The Expression Interpreter

At static analysis time, Saxon parses the source XSLT or XQuery code and generates an expression tree: a hierarchic structure of Java objects in which each node represents an expression (or another construct, such as an XSLT instruction, or a clause in a FLWOR expression), with links to its subexpressions. For example, an expression like `$x > 3 * $i` produces a `ComparisonExpr` node, with two child nodes, a `VariableReference` node for `$x`, and an `ArithmeticExp` node for subexpression `3 * $i`; this in turn has two child nodes for its operands (a `NumericLiteral` and a `VariableReference`. Each kind of node is represented by a subclass of the `Expression` class, and has additional fields for relevant properties such as the arithmetic operator, the name of the variable, and the value of the literal.



In principle, every kind of `Expression` overrides the method `Expression.evaluate(Context context)` which takes as its argument the current evaluation context (providing access to details such as the context item, position, and size, the current group, current mode, and so on). Calling this method returns the result of evaluating the expression, which in general is a `Sequence` object.

(The above is a simplification. We actually provide a variety of evaluation methods: `iterate()` for lazy evaluation as an iterator, `process()` for push mode evaluation where the result are written to a serializer rather than being returned to the caller, `evaluateItem()` where the result is known to be a singleton, and so on).

The expression tree originates from parsing of the source XSLT, XPath, or XQuery code, but before we get to evaluate it, it goes through a number of modifications. For example:

- Nodes are added to the tree to represent implicit operations such as type checking, type conversion, and sorting of nodes into document order. To achieve this, type analysis first annotates the tree with information about the expected type of each construct.
- Links are added, for example from a variable reference to the corresponding variable declaration, or from a function call to the function declaration.
- Operational information is added to the tree, for example local variables are allocated a slot number in the stack frame for their containing function or template, so that reading and writing of variables at run-time can use simple numeric addressing, rather than matching of user-oriented variable names as strings.

- Expressions are optimized using local tree rewrites. Some expression kinds are used which can only result from such optimization rewrites: an example might be an `IntegerRangeTest` with three operands representing the expression `V = P to Q`, which tests whether the value of `V` is in the range `P` to `Q` inclusive. Other rewrites generate constructs that could have been written by the user explicitly, for example `P = Q` might be rewritten as `P eq Q` if it is known (from type analysis) that both operands are singletons. The more powerful optimizations change the structure of the tree, for example by moving a subexpression out of a loop where it is safe to do so.

The design of the expression tree has some limitations:

- The same data structure is used during static analysis and at run time. In principle much of the information that's needed for static analysis could be discarded once evaluation starts; evaluation might benefit from a lighter-weight structure designed explicitly for that purpose.
- The data structure is, for most purposes, read-only at run-time. That's necessary for thread safety - if you process several source documents concurrently in a web server using the same stylesheet, they will share the same copy of the expression tree. This means it's not possible to do things like replacing a global variable reference with the value of the variable once the value is known. (Actually, it's not completely read-only. There are some changes that happen when a node in the tree is evaluated for the first time, for example. Such operations need careful attention to thread safety.)

3. Bytecode Generation

The Enterprise Edition of Saxon attempts to speed up execution by generating JVM bytecode for evaluation of selected parts of the expression tree. Because bytecode generation is itself an expensive process, and may consume large amounts of memory, this is done very selectively. During static analysis, Saxon identifies particular expressions as candidates for bytecode generation. The body of a function or template is always such a candidate, but so are smaller units of code such as the predicate in a filter expression, or the body of an `xsl:for-each` instruction.

This isn't as sophisticated as the JVM hot-spot compiler, which actually monitors how effective its optimizations are, and is capable of reversing them if they prove not to be worthwhile. But it's the same general idea.

There are two interesting questions to ask about bytecode generation: how effective is it, and why?

The answer to the first question is that we see bytecode generation speeding up XSLT and XQuery execution by anything from 0 to 25%; but most cases, sadly, are towards the lower end of that range. It's most effective with simple queries

dominated by evaluation of a simple predicate — but it has to be one that can't be optimized by other techniques such as indexing. For example, in the XMark benchmark suite, query Q11 execution improves from 145ms to 115ms with bytecode generation enabled (around 20%). This query is dominated by the execution of a single predicate of the form `[$vv > 5000 * data(.)]`. Here `$vv` is a local variable generated by the optimizer (as a result of loop-lifting an expression out of the predicate), and `data(.)` involves atomizing a node and converting its string value to an `xs:double`. In fact, string-to-double conversion dominates the query execution time. This is done in a library routine, and there's no opportunity for bytecode generation to speed it up.

By contrast, we found that execution of a large DocBook XSLT transformation improves only from 10.17s to 10.08s as a result of bytecode generation. We've profiled this, and it's very hard to identity significant hot-spots that account for a substantial part of the total execution time.

Where exactly does bytecode generation help? It's surprisingly difficult to answer this question.

It's easy to see where it doesn't help: most of the run-time execution is spent doing things like string-to-number conversion, regular expression processing, navigation of the TinyTree data structure, parsing, and serialization, where the logic is all in library routines that are exactly the same whether invoked by the interpreter or by generated bytecode. So where do we get gains? I think the answer is some combination of the following:

- Reduced navigation of the expression tree. Some of the expressions on the expression tree execute so quickly that finding your way to the expression that needs to be evaluated is as much work as doing the actual evaluation. This is pure overhead in the interpreter.

- Eliminating run-time checks. Even with interpreted code, we go to great lengths to do everything we can at compile time to reduce work done at run-time. For example, if a regular expression or a collation URI is supplied as a string literal, we'll always try to take advantage of the fact. And we do static type analysis to avoid unnecessary run-time type checking. But sometimes it's just not practical. For example, there are many instructions where there's a run-time error if the context item is absent, or if it isn't a node. We might know at compile time that this check isn't needed on a particular path, but with the interpreter, it's simplest to do it anyway. The bytecode generator can be a bit more selective and avoid a few unnecessary instructions.

- Inlining. When we generate code for a predicate like `[$vv > 5000 * data(.)]`, the code all goes in a single generated method. There are no calls from the method that does the comparison to the method that does the arithmetic to the method that does the atomization. Fewer method calls means less

overhead; it also means that the next instruction you want to execute is more likely to already be in the CPU cache.

Now, you might ask, surely the JVM hot-spot compiler can do inlining anyway, so why do we need to do it ourselves? Well, there's a very important difference. In the Saxon interpreter, the methods are highly polymorphic ("megamorphic" is the term used by the JVM experts). That is, we have literally a couple of hundred subclasses of `Expression` to evaluate different kinds of expression, and when `ArithmeticExpression.evaluate()` calls the `evaluate()` method of its two operands, that method call could be despatched to any one of a hundred different implementations of the `evaluate()` method. In that situation, no inlining is possible, except perhaps in the case where one kind of operand (perhaps a numeric literal) is much more common than any other. By contrast, we're generating bytecode for a specific arithmetic expression where we know that the two operands are a literal and an atomizer, and in that situation inlining is eminently possible.

So the key difference is: in the interpreter, one Java method is handling all arithmetic expressions. In the generated bytecode, there's one Java method for each individual arithmetic expression in the stylesheet (provided of course that it's executed often enough to justify the code generation). An individual arithmetic expression knows statically what kinds of operands are; the generic code that handles all arithmetic expressions only finds this out when it gets executed.

- Avoiding boxing and unboxing. One of the consequences of using highly polymorphic methods like `Expression.evaluate()` is that data has to be passed from caller to callee, and back, in a form that satisfies a strongly typed interface. For example, the result of every XPath function call has to be returned as an instance of the class `net.sf.saxon.om.Sequence`. So with an XPath expression like `count($x) + 1`, the chances are that the implementation of `count()` is computing an integer, which has to be wrapped as an `net.sf.saxon.om.Sequence`, merely so that this can be unwrapped again in order to add one to the value. The bytecode generator is able to avoid a lot of this boxing and unboxing.

  How much does this matter? We don't really know. We know that the costs of allocating and garbage collecting short-lived objects are much less than they were in Java's early days, but small costs incurred millions of times do add up.

4. Elaboration

In this section we'll first look at requirements: what are we trying to achieve? Then we'll explain the concept of expression elaboration; and we'll illustrate it with an example.

4.1. Why try something new?

For this project we wanted to try a new technique, called expression elaboration, which I will go on to explain in the next section. But before doing so, I should explain why we were motivated to experiment with new ideas.

The immediate driver was the development of a new product (SaxonCS) targeted at the .NET Core platform. [1] For many years, we (Saxonica) delivered a version of Saxon for the .NET Framework platform, which was built by using the open source IKVM tool to convert the compiled SaxonJ JAR file into a .NET executable, and adding an API layer to integrate it with other facilities of the platform. In 2019, Microsoft announced that they planned to discontinue development of .NET Framework, and concentrate future work on .NET Core. Although the two platforms offer very similar capabilities at the API level, the internal engineering is very different, sufficiently so that IKVM would need a complete rewrite to make it work with .NET Core; which was unlikely to be forthcoming since the main developer of IKVM, Jeroen Frijters, announced that he had no enthusiasm to take the task on. As a result we needed to find a different way of bridging Saxon from the Java platform to .NET, and we did this by writing our own source code transpiler [XML London 2021]. With IKVM (perhaps surprisingly) our Saxon bytecode generation logic worked seamlessly on .NET — as soon as we generate bytecode, IKVM translates it on the fly to .NET 's equivalent. In the new transpiler-based product, this wasn't going to work.

For the Java platform, we're a little disillusioned with bytecode generation anyway, because there's a lot of code to maintain and the benefits, as we've seen, are quite modest. We wanted to see if there might be another way of getting the benefits with lower maintenance cost. Because of our business model where we offer a free open-source product alongside a commercial Enterprise Edition, it's useful to offer features like bytecode generation that provide an easy-to-understand turbo-charger to the base product. So we were reluctant to drop it entirely, but at the same time we wanted to see if we could do better.

On the Javascript product, SaxonJS, which is developed using completely separate source code, we had seen outstanding performance benefits from a technique we called expression elaboration. In fact, the benefits were so clearly apparent to the naked eye that we never took the trouble to make detailed measurements of the actual speed-up. We knew that we were unlikely to achieve the same kind of benefit with the Java product because we were starting with something that was already much more highly tuned; but it looked as if it might give us an alternative to bytecode generation for the SaxonCS version, and perhaps even enable us to drop bytecode generation from SaxonJ.

[1] The terminology has evolved. SaxonCS = Saxon on the .NET platform (primarily for C#); SaxonJ = Saxon on the Java platform; SaxonJS = Saxon on Javascript platforms (Node.js and browsers)

4.2. Expression Elaboration Explained

Expression elaboration starts with exactly the same expression tree that we use for interpretation, but it then splits the work of evaluation into two phases:

- The first time any expression node on the tree is evaluated, we construct a lambda expression, which we then leave on the tree for subsequent use. The name "elaboration" refers to this stage of the process.
- All subsequent evaluations of the expression then merely call this lambda expression, passing the evaluation context as an argument.

  That's a convenient way to explain it, but in practice when an expression is elaborated, this usually involves elaborating its subexpressions, and so on down to the bottom of the tree. So typically, the first time a user-written function or template is called, the body of the function is elaborated into a lambda expression, which invokes further lambda expressions held in its closure, and so on recursively; in the typical case the original expression tree then plays no further part.

Lambda expressions have become ubiquitous in nearly all modern programming languages, and the syntax and semantics are similar across Java, C#, and Javascript.

4.3. A simple example

Let's look at one particular instruction, called "negate". This implements the unary minus operator: it corresponds to an XPath expression such as `-$x`. [2] In SaxonJ, the code to evaluate a negate instruction in the interpreter looks like this:

```
@Override
public NumericValue evaluateItem(XPathContext context) throws
XPathException {
    NumericValue v1 = (NumericValue)
getBaseExpression().evaluateItem(context);
    if (v1 == null) {
        return backwardsCompatible ? DoubleValue.NaN : null;
    }
    return v1.negate();
}
```

Some observations:

[2] According to the XPath grammar, `-1` is a negate expression applied to a literal; but we sort that out during static analysis, so this will always appear as a constant at run-time. Unary minus operators are rarely used with operands other than numeric literals, but we've chosen them as our example because they are so simple.

- The method `evaluateItem()` takes the XPathContext as a parameter. There's a lot of information in this object, but the only thing we do is pass it on when evaluating the single operand (accessed as `getBaseExpression()`)
- The logic essentially does four things:
  - Evaluate the operand.
  - Cast the result to a `NumericValue` (we know this cast is safe, because static type analysis will have generated a guard expression on the expression tree to check or convert the value in cases where it is necessary).
  - if the value of the operand is null (representing an empty sequence) return either `NaN` or `null` depending on whether XPath 1.0 backwards compatibility is in force
  - call the `negate()` method on the `NumericValue`.

Now see what happens when we elaborate this instruction:

```
@Override
public ItemEvaluator elaborateForItem() {
    final NegateExpression exp = (NegateExpression)getExpression();
    final ItemEvaluator argEval =
makeElaborator(exp.getBaseExpression()).elaborateForItem();
    final boolean maybeEmpty =
Cardinality.allowsZero(exp.getBaseExpression().getCardinality());
    final boolean backwardsCompatible = exp.isBackwardsCompatible();
    if (maybeEmpty) {
        if (backwardsCompatible) {
            return context -> {
                NumericValue v1 = (NumericValue) argEval.eval(context);
                return v1 == null ? DoubleValue.NaN : v1.negate();
            };
        } else {
            return context -> {
                NumericValue v1 = (NumericValue) argEval.eval(context);
                return v1 == null ? null : v1.negate();
            };
        }
    } else {
        return context -> ((NumericValue)
argEval.eval(context)).negate();
    }
}
```

What's going on here? Remember that the method elaborateForItem() is called the first time a particular negate instruction is evaluated. It does the following:

- Gets the operand expression in the expression tree (`getBaseExpression()`)
- Elaborates the operand expression, returning a lambda function

- Examines the expression tree to see (a) whether the result of the operand may be an empty sequence, and (b) whether evaluation is in XPath 1.0 backwards compatibility mode
- Returns one of three different lambda functions, depending on these input conditions. The resulting function performs no run-time check for backwards compatibility, and no check for the operand being null unless this is actually a known possibility.

  If you're not familiar with lambda expressions in Java, there are three in this sample, all taking the form `params -> (expression | "{" statements "}")`. This corresponds to the lambda calculus notation `λ params : expr`, but neither Java nor any other of the mainstream programming languages was prepared to take the plunge of using Greek letters in the concrete syntax. The syntax denotes an anonymous function that takes a `context` object as its argument, and returns the result of evaluating the supplied expression (or statements) which typically depend both on the explicit `context` argument supplied by the caller, and on variables (such as `argEval`) that are in scope at the point where the lambda expression appears: the values of these variables are carried along with the function itself and are referred to as the function's *closure*.

So comparing the interpreted code with the generated lambda function, what have we achieved?

- We've eliminated the code that navigates the expression tree at run-time to locate the operand expression. Instead, the elaborated operand expression is present in the closure of the generated function, as variable `argEval`.
- We don't check at run-time for null values unless they can actually occur.
- The run-time logic doesn't need to consider whether backwards compatibility is in force or not: this decision has been "baked in".
- This is only saving us a few instructions; but negating a number is only one instruction, so in relative terms, we've cut out a lot of overhead.

I'm not going to show the code for bytecode generation of this expression, but I'll show what the generated bytecode looks like (with added comments for explanation). This bytecode is produced when compiling the XQuery function `declare function f:negate($x as xs:double) as xs:double {-$x};`

```
// load the first argument (the XPathContext)
ALOAD 1
// Get the stack frame holding local variables
INVOKEINTERFACE net/sf/saxon/expr/XPathContext.getStackFrame ();
INVOKEVIRTUAL net/sf/saxon/expr/StackFrame.getStackFrameValues ();
// Load the value of the variable at slot 0 on the stack frame
ICONST_0
```

```
    AALOAD
    // The value is in general a Sequence; call head() to get its first
and only item
    INVOKEINTERFACE net/sf/saxon/om/Sequence.head ();
    // Cast this to type NumericValue
    CHECKCAST net/sf/saxon/value/NumericValue
    // Invoke NumericValue.negate()
    INVOKEVIRTUAL net/sf/saxon/value/NumericValue.negate ();
    // Wrap the result in a SingletonIterator
    INVOKESTATIC net/sf/saxon/tree/iter/SingletonIterator.makeIterator
(Lnet/sf/saxon/om/Item;);
    // Return the iterator as the result of the XQuery function
    ARETURN
```

The only really signficant difference from the elaboration case is that bytecode for the operand expression is generated inline, rather than being invoked separately.

All three approaches (interpreter, compiler, elaborator) end up calling the library routine `NumericValue.negate()` to do the real work. This is a polymorphic method with different implementations for integers, decimals, double, and floats. In all three cases the JVM hotspot optimizer has the opportunity to optimize the call by inlining, but it's only likely to do so in practice if one of these types occurs much more frequently than the others.

It's possible that as a result of Saxon's static analysis the elaborator already knows what the type of the numeric value will be. With bytecode generation, we can easily pass this information to the Java compiler by casting to the relevant type instead of to the generic type NumericValue (though in fact, we fail to take advantage of this opportunity). With the interpreter, this isn't possible, because a single method is handling all Negate expressions in the query or stylesheet, and they will typically be handling different types of operand. For the elaborated case, we could do it in principle, by generating different lambda functions for the four cases, plus one for the case where the type is statically unknown. However, the complexity multiplies exponentially -- instead of generating one of three possible lambda functions, we would be generating one of 15, and it would need strong justification to attempt this.

4.4. Push mode, Pull mode

In the example above, the interpreter used a method `evaluateItem()` to evaluate the `negate` expression (and its operand). We use that method where the expression result will always be a singleton item (or perhaps an empty sequence). Other methods are used where an expression can return an arbitrary sequence. Elaboration, similarly, can generate code that uses different modes of evaluation.

At the top level, we have two ways of evaluating an expression: pull mode and push mode.

- In pull mode, the `iterate` method returns the result of the expression (which in the general case is a sequence) as an iterator over the items in the sequence. This means we are doing lazy evaluation, which is an important technique in all functional languages — it means that in many cases, evaluation of an expression can finish before the operands are fully evaluated, because enough information is available to establish the result.

  The `evaluateItem()` method seen in the example above is a short-cut method provided for convenience when an expression always returns a singleton result.
- In push mode, the results of the expression are not returned to the caller, but are written to a result stream, which will often be the final serialized result of a transformation. The advantage of push mode is that there is no need to hold the entire result document in memory, it can be written out "on the fly".

Both modes are supported by elaboration: for any given expression on the tree, we can generate either a pull function, or a push function, or both.

For pull mode, the function that we generate takes a single argument, the object holding the dynamic context, and it returns an iterator over the expression results. For push mode, we generate a function that takes two arguments, the dynamic context object and the destination to be written to; the function returns no result, but instead has a side-effect of writing to the destination.

Most instructions only support one mode of execution directly: for example an element constructor supports push mode, while an arithmetic expression or path expression supports pull mode. If the opposite mode is needed, it can be easily achieved using a wrapping function that converts the results. But some instructions - notably "flow of control" instructions such as conditional expressions, iteration instructions (xsl:for-each in XSLT, for expressions in XPath/XQuery), and function calls, support both modes natively.

5. Results

So, what benefits are we seeing from expression elaboration?

The results given here are provisional, for two reasons: firstly, implementation is incomplete (we've only implemented elaboration in SaxonJ for a selection of commonly used expressions), and secondly, measuring the effect is not easy.

At this point I need to acknowledge the contribution of Chris Newland, who has been working with us to improve our ability to benchmark the Saxon software and assess the impact of changes. Benchmarking Java applications is a very skilled task, and it's very easy to come to incorrect conclusions if you cut corners. Getting repeatable results (where today's figures come out the same as yesterday's) is challenging: don't try it on your laptop, where temperature variations or a low battery can cause the CPU speed to be throttled. Getting good reliable data needs a controlled stable machine configuration, and benchmark runs that take

hours rather than minutes. And even where the results are consistent, that's no guarantee that you will draw the right conclusions from the data.

We've been putting a lot of work into measurement on the Java platform, but I mentioned that part of the motivation was to see what we could achieve on .NET, where bytecode generation isn't an option. Our benchmarking activities on .NET have been far less thorough, but the early indications are very positive: for example the XMark query Q11 came down from 1192ms with the interpreter to 858ms with elaboration — a 28% improvement, better than we get with bytecode generation on the Java product. Both figures have an error margin of around ±5%. Other queries also showed a benefit, though not usually as great as this: 10% is more typical. With improvements of this order, we can probably declare victory and dispense with the effort of doing more accurate measurement.

On Java, so far, we're seeing much smaller improvements. For XMark Q11, for example, elaboration brings the timing down from 107ms to 104ms. Other queries show similar results: the improvement, if it exists, is hardly measurable, and is certainly a lot less than we get with bytecode generation. Needless to say, this is disappointing.

Of course, there is a beneficial side-effect: when you put this much effort into instrumentation, you discover all sorts of opportunities for performance improvements that you weren't actually looking for, and following up on some of these opportunities has probably distracted us from the task we set out to accomplish. But they're out of scope for this paper.

We're still exploring why the benefit on Java is so small, and whether there's anything we can do to improve matters. We've found that some of the switches that Java provides to control the behaviour of the hot-spot compiler can make a significant difference, but in the real world that's not very useful knowledge since very few Saxon users in the field are likely to take advantage of it. And many of the Saxon users who do care deeply about performance probably have applications in which Saxon is just one of many components. But observing how these switches affect the results does give us clues about how the lambda functions we're generating are treated by the hot-spot optimizer.

Perhaps the key finding (though a provisional one that we need to confirm) is that the hot-spot optimizer is taking no notice of the values in the closure of a lambda expression: just because a boolean variable in the closure is false, doesn't mean that the hot-spot compiler is eliminating a run-time code branch that depends on that value. That's because it's not optimizing for a particular expression in the query or stylesheet (say, a particular filter predicate), rather it's optimising for a statistical average of all filter predicates in the stylesheet. The net result is that with both elaboration and interpretation, the ability of the hot-spot optimizer to work its magic is inhibited by the fact that the calls we are making to evaluate subexpressions are so heavily polymorphic.

It seems fairly clear that there's some significant difference in the way the Java and C# optimizers handle lambda expressions that cause the technique to show greater benefits in C# than on Java. But so far, we haven't been able to pin down exactly what it is.