SAXONICA.COM
XSLT AND XQUERY PROCESSING

Expression Elaboration

Michael Kay, Saxonica
XML Prague 2022

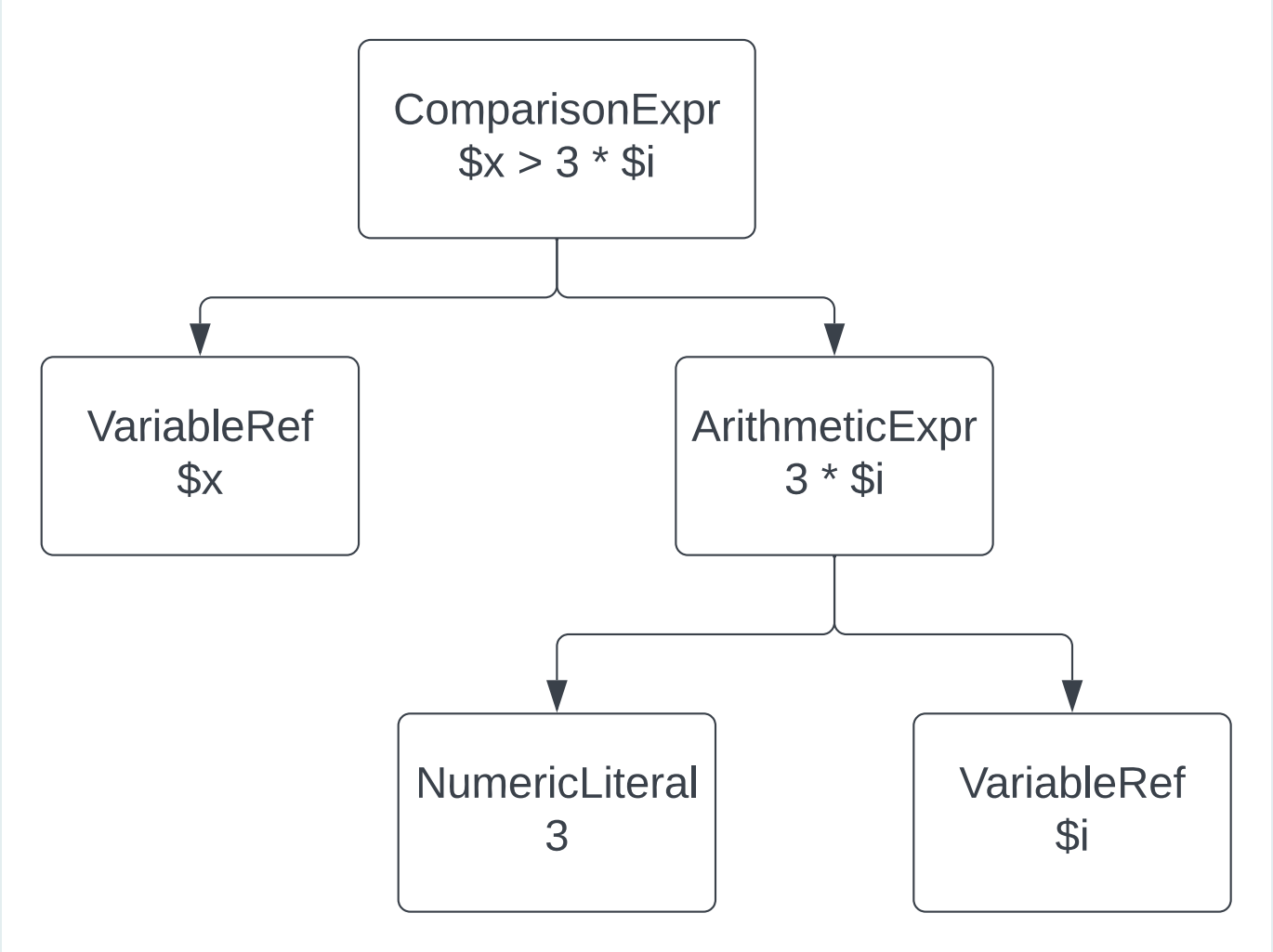
Warning

We learnt a lot from this project.

But it was not 100% successful.

Some viewers may find this distressing.

Expression Trees



Evaluation using an Interpreter

```
ComparisonExpr.evaluate(context) {
    Value v0 = getOperand(0).evaluate(context);
    Value v1 = getOperand(1),evaluate(context);
    return comparator.compare(v0, v1) < 0;
}

VariableReference.evaluate(context)  {
   return context.getLocalVariables().getValue(slotNumber);
}

ArithmeticExpr.evaluate(context) {
    Value v0 = getOperand(0).evaluate(context);
    Value v1 = getOperand(1),evaluate(context);
    return calculator.calc(v0, v1);
}

Literal.evaluate(context) {
    return getValue();
}
```

ByteCode for {-$x}

```
// load the first argument (the XPathContext)
ALOAD 1
// Get the stack frame holding local variables
INVOKEINTERFACE net/sf/saxon/expr/XPathContext.getStackFrame ();
INVOKEVIRTUAL n/s/s/expr/StackFrame.getStackFrameValues ();
// Load the value of the variable at slot 0 on the stack frame
ICONST_0
AALOAD
// Call head() to get its first and only item
INVOKEINTERFACE n/s/s/om/Sequence.head ();
// Cast this to type NumericValue
CHECKCAST n/s/s/value/NumericValue
// Invoke NumericValue.negate()
INVOKEVIRTUAL n/s/s/value/NumericValue.negate();
// Wrap the result in a SingletonIterator
INVOKESTATIC
n/s/s/tree/iter/SingletonIterator.makeIterator(...);
// Return the iterator as the result of the XQuery function
ARETURN
```

Disadvantage of Interpretation

- Half the time is spent deciding what to do, rather than actually doing it
- Navigating the expression tree is a significant cost
- Very highly polymorphic code
  - reduces potential for JIT optimisations
  - leads to boxing/unboxing costs
- Poor "locality of reference"
  - meaning poor CPU cache hit rate

Disadvantage of ByteCode Generation

- Generating the code is expensive, unless done very selectively
- Memory consumption / limits / security issues etc
- Debugging is a nightmare
  - maintainability
- Platform-dependent
- Performance benefits are modest

Project Background: Saxon on .NET

- Until Saxon 10, the code was bridged from Java to .NET using IKVMC
  - bytecode just worked!
- IKVMC doesn't work with .NET Core
- So from Saxon 11, we transpile source Java to source C#
  - using XSLT, of course
  - bytecode stops working

So, we thought we'd try out expression elaboration.

We'd used it on SaxonJS, very successfully.

How would it perform with Java and C#?

Expression Elaboration

- The first time an expression is evaluated, construct a lambda function {context → result}
- On subsequent calls, invoke the lambda function
- Pre-compute everything possible on the first time through, putting the results in the closure of the lambda function

Example: Unary Minus

```
@Override
public Evaluator elaborate() {
  final NegateExpression exp = (NegateExpression)getExpression();
  final Evaluator argEval = makeEvaluator(exp.getBaseExpression());
  final boolean maybeEmpty = exp.getBaseExpression().allowsEmpty();
  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();
  }
}
```

Results (Elaborator vs Interpreter)

- JavaScript
  - ~5x faster
- C#
  - 10%-25% improvement
- Java
  - 1%-3% improvement

Results (Elaborator vs Interpreter)

- JavaScript
  - ~5x faster
- C#
  - 10%-25% improvement
- Java
  - 1%-3% improvement

Why the differences between platforms?

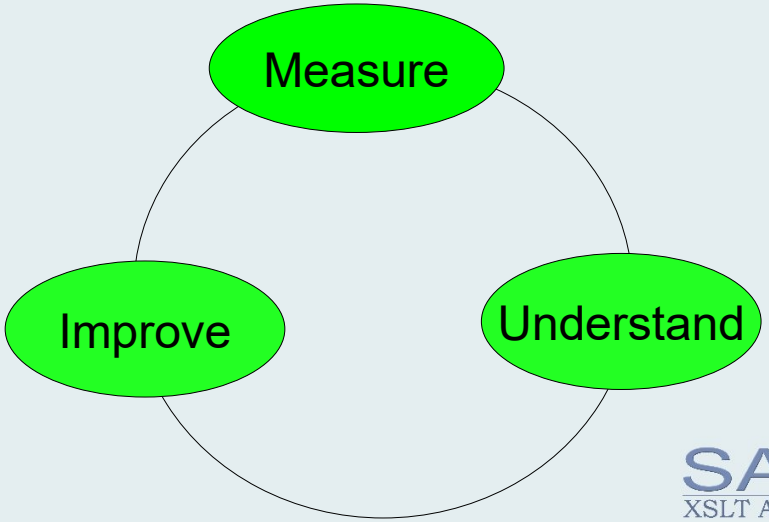
- No obvious explanation
- Elaboration would deliver much higher benefits if we hadn't already tuned the interpreter so much
- Presumably: differences in low-level JIT optimization of lambda expressions and their closures

Measurement

- To spot a 5x difference:
  - use the naked eye
- To spot a 10% difference:
  - take some simple measurements
- To spot a 1% difference:
  - Do some very careful benchmarking
  - Need to run for hours on a machine with carefully controlled configuration

Serendipity

- When you measure things carefully, you discover things you weren't looking for.
- We've achieved at least 10% speed-up in areas unrelated to the focus of the project.



Conclusions

- For C#, elaboration is a sufficient improvement to be worth implementing
- It's not good enough on Java that we can get rid of bytecode generation
- We've learnt a lot about benchmarking
- We've put some of that to good use