<transpile from="Java" to="C#" via="XML" with="XSLT"/>

Michael Kay, Saxonica <mike@saxonica.com>

Abstract

This paper describes a project in which XSLT 3.0 was used to convert a substantial body of Java code (around 500K lines of code) to C#. The Java code, as it happens, is the source code of the Saxon XSLT processor, but that's not really relevant: it could have been anything.

Table of Contents

Introduction

For a number of years, Saxonica has developed the Saxon product [1], a Java implementation of the W3C XSLT, XQuery, XPath, and XSD specifications. The product has also been made available on the .NET platform, by converting the bytecode generated by the Java compiler into the equivalent intermediate language (called IL) on .NET. The tool for this conversion was the open-source IKVMC library[2] developed by Jeroen Frijters.

IKVMC was largely a one-man project, and when Jeroen (after many years of faithful service to the community) decided to move on to other things, there was no-one to step into his capable shoes, and the project has languished.

In 2019, Microsoft announced a change of direction for the .NET platform[3]. .NET had diverged into two separate strands of development, known as .NET Framework and .NET Core, and Microsoft announced in effect that .NET Framework would be discontinued, and the future lay with .NET Core. The differences between the two strands need not really concern us here, except to note that IKVMC never supported .NET Core, therefore Saxon didn't run on .NET Core, and therefore we needed to find a different way forward.

The way that we chose was source code conversion from Java to C#. At the time of writing this has been successfully achieved for a large subset of the Saxon product, and work is ongoing to convert the remainder. This paper describes how it was done.

[1] http://www.saxonica.com/
[2] http://www.ikvm.net/
[3] https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/

Let's start by describing the objectives of the project:

- Automated conversion of as much of the source code as possible from Java to C#.
- Repeatable conversion: this is not a one-off conversion to create a fork of the code; we want to continue developing and maintaining the master Java code and port all changes over to C# using the same conversion technology.
- Performance: the performance of the final product on .NET must be at least as good as the existing product. In fact, we would like it to be considerably better, because (for reasons we have never fully understood) some workloads on the current product perform much more slowly than on the Java platform.
- Maintainability: although we don't intend to develop the C# code independently, we will certainly need to debug it, and that means we need to generate human-readable code.
- Adaptability: because the .NET platform is different from the Java platform, some parts of the product need to behave differently. We need to have convenient mechanisms to manage these differences.

I should also stress one non-objective: we were not setting out to provide a tool that could convert any Java program to C# fully automatically. We only needed to convert one (admittedly rather large) program, and this meant that:

1. We only needed to convert Java constructs that Saxon actually uses (which turns out to be quite a small subset of the total Java platform).
2. In the case of constructs that Saxon uses rarely, we could do some manual assistance of the conversion, rather than requiring it fully automatic. Indeed, by Zipf's law, many of the Java constructs that Saxon uses are only used once in the entire product, and in many cases they are used unnecessarily and could easily be rewritten a different way (sometimes beneficially). The main device we have used for this manual assistance is the use of Java annotations in the source code, annotations that are specially recognised as hints by the converter.

Preliminaries

Our initial investigations explored a number of available tools for source code conversion. The only one that looked at all promising was a commercial product, Tangible[4]. We bought a license to evaluate its capabilities, and the exercise taught us a lot about where the difficulties were going to arise. It was immediately apparent that we would have considerable difficulties with Java generics, with anonymous inner classes, and with our extensive use of the Java `CharSequence` interface, which has no direct equivalent in .NET. The exercise also taught us that Tangible, on its own, wasn't up to the job. (Having said that, the conversions performed by Tangible helped us greatly in defining our own rules.)

Our next step was to reduce our dependence on the constructs that were going to prove difficult to convert: especially generics, and the use of `CharSequence`. I have described in more detail how we achieved this in blog postings: [5] [6].

Generics are difficult because although Java and C# use superficially-similar syntax (for example `List<String>`) the semantics are very different. In C# instances are annotated at run-time with the full expanded type, and one can therefore write run-time tests such as x is `List<String>`. Writing `x is List` will return false: `List<String>` is not a subtype of `List`. By contrast, with Java, the type parameters are used only at compile time and are discarded at run time (the process is called *Type Erasure*). This means that on Java, `x instanceof List<String>` is not allowed, while `x instanceof List` returns true.

We decided to reduce the scale of the problem by dropping some of our use of generics from the product. In particular, in Saxon 9.9, two key interfaces, `Sequence` and `SequenceIterator`, were

[4] https://www.tangiblesoftwaresolutions.com
[5] https://dev.saxonica.com/blog/mike/2020/07/string-charsequence-ikvm-and-net.html
[6] https://dev.saxonica.com/blog/mike/2020/01/java-generics-revisited.html

defined with type parameters restricting the type of items contained in an XDM sequence, and we dropped this in Saxon 10.0. The use of type parameters here had always been somewhat unsatisfactory, for two reasons:

Most of the time, the code has to deal with sequences-of-anything: we don't know statically, when we write the Saxon code, what type of input it is going to be dealing with (that depends on the user-written stylesheet). So providing the type parameter (`Sequence<Item>`) simply doesn't add any value.

The XDM model for sequences has the property that an item is a sequence of length one. So `Item` implements `Sequence<Item>`. Which means that a subclass of `Item`, such as `DateTimeValue`, implements `Sequence<DateTimeValue>`. Which followed to its logical conclusion mans that a `DateTimeValue` is an `Item<DateTimeValue>`, and a generic item is therefore an `Item<Item>` (or is it an `Item<Item<Item<...>>>`?). Modelling the XDM structure accurately using Java generics proved very difficult, and in the end, it introduced a whole load of complexity without adding much value. Getting rid of it was welcome.

As far as the `CharSequence` interface is concerned, we used this extensively in interfaces where strings are passed around, to enable us to use implementations of strings other than the Java `String` class. For example, the whitespace that often occurs between elements in an XML document is compressed using run-length encoding as a `CompressedWhitespace` object, which implements the `CharSequence` interface, and can therefore be substituted in many cases for a Java `String`.

The use of `CharSequence` isn't perfect for this purpose, however. Firstly, it has the same problem as a Java `String` in that it models a string as a sequence of 16-bit UTF-16 char values, using a surrogate pair to represent Unicode astral codepoints. In XPath, strings need to be codepoint-addressible (at least for the purposes of functions such as `substring()` and `translate()`), and neither `String` nor `CharSequence` meets this requirement. There are also issues concerning comparison across different implementations of the `CharSequence` interface, plus the fact that many commonly used methods in the standard Java class library require the `CharSequence` to be converted to a `String`, which generally involves copying the content. In addition, the `CharSequence` interface doesn't guarantee immutability. For these reasons, we had already introduced another string representation, the `UnicodeString`, which we were using in many corners of the code, notably when processing regular expressions.

C# has no direct equivalent of `CharSequence`: that is, an interface which is implemented by the standard String class, but which also allows for other implementations. The interface `IEnumerable<Char>` comes close, but that doesn't allow for direct addressing to get the *N*th character in a string.

So we decided to scrap our extensive use of `CharSequence` throughout the product, and replace it with our own `UnicodeString` interface – which allows for direct codepoint addressing, rather than `char` addressing with surrogate pairs. There is a performance hit in doing this, because there's a lot of conversion between `String` and `UnicodeString` when data crosses the boundary between Saxon and third-party software (notably the XML parser, but also library routines such as `upperCase()` and `lowerCase()`). However, it's sufficiently small that most users won't notice the difference, and we can mitigate it – for example we have our own UTF-8 Writer used by the Saxon serializer, and it was easy to extend the UTF-8 Writer to accept a `UnicodeString` as input, bypassing the conversion of `UnicodeString` to `String` prior to UTF-8 encoding.

Examples of Converted Code

To set the scene, it might be useful to provide a couple of examples of converted code, illustrating the challenges.

Here's a rather simple method in Java:

```
@Override
    public AtomicSequence atomize() throws XPathException {
        return tree.getTypedValueOfElement(this);
```

```
    }
```

And here is the C# code that we generate:

```
public override net.sf.saxon.om.AtomicSequence atomize() {
    return tree.getTypedValueOfElement(this);
}
```

Nothing very remarkable there, but it actually requires a fair bit of analysis of the Java code to establish that the conversion in this case is fairly trivial. For example:

- The class name `AtomicSequence` has been expanded; this requires analysis of the `import` declarations in the module, and it can't be done without knowing the full set of packages and classes available.
- The `@Override` declaration is optional in Java, but `optional` is mandatory in C#; moreover they don't mean quite the same thing, for example when overriding methods defined in an interface or abstract class.
- The conversion of Java `this` to C# `this` works here, but there are other contexts where it doesn't work.

Now let's take a more complex example. Consider the following Java code:

```
public Map<String, Sequence> getDefaultOptions()  {
    Map<String, Sequence> result = new HashMap<>();
    for (Map.Entry<String, Sequence> entry : defaultValues.entrySet()) {
        result.put(entry.getKey(), entry.getValue());
    }
    return result;
}
```

In C# this becomes (with abbreviated namespace qualifiers, for readability):

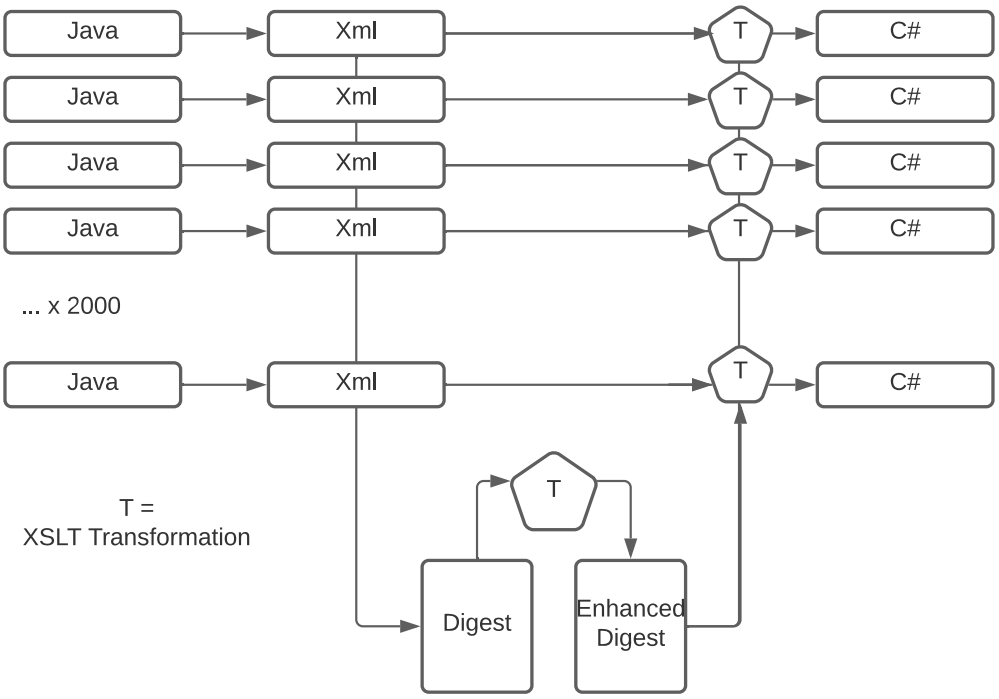
```
public S.C.G.IDictionary<string, n.s.s.o.Sequence> getDefaultOptions() {
    S.C.G.IDictionary<string, n.s.s.o.Sequence> result =
            new S.C.G.Dictionary<string, n.s.s.o.Sequence>();
    foreach (S.C.G.KeyValuePair<string, n.s.s.o.Sequence> entry in defaultValue
        result[entry.Key] = entry.Value;
    }
    return result;
}
```

There's a lot going on here:

- We've replaced the Java `Map` with a C# `Dictionary`, and its `put()` method has been replaced with an indexed assignment;
- The Java iterable `defaultValues.entrySet()` has been replaced with the C# enumerable `defaultValues`;
- The references to `entry.getKey()` and `entry.getValue()` have been replaced with property accessors `entry.Key` and `entry.Value`.
- The replacement of `result.put(key, value)` by `result[key] = value` is fine in this context, but it needs care, because if the return value of the expression is used, the Java code returns the previous value associated with the key, while the C# code returns the new value. The rewrite works here only because the expression appears in a context where its result is discarded.

Architecture of the Converter

The overall structure of the transpiler is shown below:



Let's explain this:

- On the left, we have 2000+ Java modules.
- These are converted to an XML representation by applying the JavaParser, and serializing the resulting parse tree as XML.
- An XSLT transformation takes all the XML files as input and generates a digest of the class and method hierarchy.
- A further XSLT transformation enhances the digest by analyzing which methods override each other.
- Each of the 2000+ XML modules is then converted to C# by applying an XSLT transformation, which takes the enhanced digest file as an additional input.

The first stage of conversion is to parse each Java module and generate an abstract syntax tree, which can be serialized as XML. For this purpose we use the open-source JavaParser product[7].

JavaParser generates the parse tree as a hierarchy of (not very well documented) Java objects. It also includes the capability to serialize this hierarchy as XML. We don't use its out-of-the-box serialization however: we augment it with additional semantic information. JavaParser in fact has two parts (originally developed independently, and still showing evidence of the fact): the parser itself, which is exactly what it says, and the "symbol solver", which is a set of queries that can be executed on the parse tree to obtain additional information. For example, if the raw source code contains the expression `new HashMap<>()`, this will appear in the raw tree as:

```
<value nodeType="ObjectCreationExpr">
    <type nodeType="ClassOrInterfaceType">
       <name nodeType="SimpleName" identifier="HashMap"/>
       <typeArguments/>
    </type>
</value>
```

[7] http://javaparser.org

But with the aid of the symbol solver, it is straightforward to establish that the name `HashMap` refers to the class `java.util.HashMap`, and we output this as an additional attribute on the tree, thus:

```
<value nodeType="ObjectCreationExpr">
    <type nodeType="ClassOrInterfaceType"
          RESOLVED_TYPE="java.util.HashMap">
       <name nodeType="SimpleName" identifier="HashMap"/>
       <typeArguments/>
    </type>
</value>
```

Similarly, the symbol solver is usually able to find the declaration corresponding to a variable reference or method call, and hence to establish the static type of the variable or of the method result. I say usually, because there are cases it gives up on. It struggles, for example, with the types of the arguments to a lambda expression, for example the variable *n* in

```
search.setPredicate(n -> n.name="John")
```

Similarly it has difficulty with static wildcard imports:

```
import static org.w3.dom.Node.*;
```

The other problem with the symbol solver is that it can do a lot of things that aren't mentioned in the documentation: we've found some of these by experiment, or by studying the source code. No doubt there are other gems that remain hidden.

The result of this process is that for each Java module in the product, we generate a corresponding XML file containing its decorated syntax tree.

In principle we could now write an XSLT transformation that serializes this syntax tree using C# syntax. But there's another step first. In some cases we can't generate the C# one file at a time: we need some global information. For example, if a C# method is to be overridden in a subclass, it needs to be flagged with the `virtual` modifier. Similarly, overriding methods need to be flagged as `override`. We therefore need to construct a map of the entire class hierarchy, working out which methods are overridden and which are overrides.

So the second phase of processing is to scan the entire collection of XML documents and generate a digest file (itself an XML document, naturally) which acts as an index of classes, interfaces, and methods, and which represents the class hierarchy of the application. Then (our third phase) we do a transformation on the digest file which augments it with decisions about which methods are overriding and which are virtual.

Now finally we can perform the XML-to-C# phase, implemented as an XSLT transformation applied to each of the XML documents generated in phase one, but with the digest file available as additional information.

The C# is then ready to be compiled and executed.

Difficulties

In this section we outline some of the features of the Java language where conversion posed particular challenges, and explain briefly how these were tackled.

It's worth noting that there are broadly three classes of solution for each of these difficulties:

- Create an automated conversion that handles the Java construct and converts it to equivalent C#. Note that although this is an automatic conversion, it doesn't necessarily have to handle every edge case, in the way that a productised converter might be expected to do. In particular, it doesn't have to handle edge cases that the Saxon code doesn't rely on: for example the converted code doesn't have to handle `null` as an input in exactly the same way as the original Java, if Saxon never supplies `null` as the input.

- Convert with the aid of annotations manually added to the Java code. We'll see examples of some of these annotations later.

- Eliminate use of the problematic construct from the Java code, replacing it with something that can be more easily converted. A trivial example of this relates to the use of names. Java allows a field and a method in a class to have the same name; C# does not. Simple solution: manually rename fields where necessary so that no Saxon class ever has a field and a method with the same name. (Very often, imposing such a rule actually improves the Java code.)

The following sections describe some of the difficulties, in no particular order.

Dependencies

Java has a class `java.util.HashMap` (which Saxon uses extensively). C# does not have a class with this name. It does have a rather similar class `System.Dictionary`, but there are differences in behavior.

Broadly speaking, there are three ways we deal with dependencies:

- *Rewriting*. Here the converter (specifically, the XML-to-C# transformation stylesheet) has logic to rename references to the class `java.util.HashMap` so they instead refer to `System.Collections.Generic.Dictionary`, and to convert calls on the methods of `java.util.HashMap` so they instead call the corresponding methods of `System.Dictionary`. We've already seen an example of this above. Sometimes there is no direct equivalent for a particular method, in which case we instead generate a call on a helper method that emulates the required functionality. (`System.Collections.Generic.Dictionary`, for example, has no direct equivalent to the `get()` method on `java.util.HashMap`, largely because it cannot use `null` as a return value when the required key is absent.)

  The converter uses rewriting for the vast majority of calls on commonly used classes and methods. There's more detail on how this is done below.

- *Emulation*. Here we implement (in C#) a class that emulates the behaviour of the Java class – or at least, those parts of the behaviour that Saxon relies on. An example where we do this is `java.util.Properties`, which has no direct equivalent in C#, but which is easily implemented using dictionaries. Saxon doesn't use the complicated methods for importing and exporting `Properties` objects, so we don't need to emulate those.

- *Avoidance*. Here we simply eliminate the dependency. For example, the Java product will accept input from either a push (SAX) or pull (StAX) parser. On C# we will only support a single XML parser, the one from Microsoft. This is a pull parser, so we eliminate all the Saxon code that's specific to SAX support. This is non-trivial, of course, because the relevant code is widely scattered around the product. But once found, it's usually easy to get rid of it using preprocessor directives in the Java (`//#if CSHARP==false`). I should perhaps have mentioned that there's a "phase 0" in our conversion pipeline, which is to apply these preprocessor directives.

In cases where dependencies are handled by rewriting, there are two parts to this. Firstly, we have a simple mapping of class names. This includes both system classes and Saxon-specific classes. Here are a few of them:

```
<xsl:variable name="specialTypes"
        as="map(xs:string, xs:string)"
        select="map{
  'boolean':           'bool',
  'java.io.BufferedInputStream':
                       'System.IO.BufferedStream',
  'java.io.BufferedOutputStream':
                       'System.IO.BufferedStream',
  'java.io.BufferedReader':
```

```
                        'Saxon.Impl.Helpers.BufferedReader',
  'java.lang.ArithmeticException':
                        'System.ArithmeticException',
  'java.lang.ArrayIndexOutOfBoundsException':
                        'System.IndexOutOfRangeException',
  'java.lang.Boolean': 'System.Boolean',
  'java.lang.Byte':    'System.Byte',
      ...
  'java.math.BigDecimal':
                        'Singulink.Numerics.BigDecimal',
      ...
  'java.util.ArrayList':
                        'System.Collections.Generic.List',
  'java.util.Collection':
                        'System.Collections.Generic.ICollection',
  'java.util.Comparator':
                        'System.Collections.Generic.Comparer',
      ...
  'net.sf.saxon.ma.trie.ImmutableHashTrieMap':
                        'System.Collections.Immutable.ImmutableDictionary',
  'net.sf.saxon.ma.trie.ImmutableMap':
                        'System.Collections.Immutable.ImmutableDictionary',
  'net.sf.saxon.ma.trie.ImmutableList':
                        'System.Collections.Immutable.ImmutableList',
  'net.sf.saxon.ma.trie.TrieKVP':
                        'System.Collections.Generic.KeyValuePair',
       ...
  'net.sf.saxon.s9api.Message':
                        'Saxon.Api.Message',
  'net.sf.saxon.s9api.QName':
                        'Saxon.Api.QName',
  'net.sf.saxon.s9api.SequenceType':
                        'Saxon.Api.XdmSequenceType',
      ...
}"/>
```

Note that there are cases where we replace system classes with Saxon-supplied classes, and there are also cases where we do the reverse: the extract above illustrates that we can replace Saxon's immutable map implementation with the standard immutable map in .NET. In the case of `BigDecimal`, we rewrite the code to use a third-party library[8] with similar functionality to the built-in Java class.

The other part of the rewrite process is to handle method calls. We rely here on knowing the target class of the method, and we typically handle the rewrite with a template rule like this (long namespace names abbreviated for space reasons: `S.N` = `Singulink.Numerics`, `S.I.H` = `Saxon.Impl.Helpers`)

```
<xsl:template match="*[@RESOLVED_TYPE = 'java.math.BigDecimal']"
      priority="20" mode="methods">
  <xsl:sequence select="f:applyFormat(., map{
     'add#1':       '(%scope%+%args%)',
     'subtract#1':  '(%scope%-%args%)',
     'multiply#1':  '(%scope%*%args%)',
     'divide#1':    'S.N.BigDecimal.Divide(%scope%, %args%, 18)',
     'divide#2':    'S.N.BigDecimal.Divide(%scope%, %args%)',
     'divide#3':    'S.N.BigDecimal.Divide(%scope%, %args%)',
```

[8] https://github.com/Singulink/Singulink.Numerics.BigDecimal

```
      'negate#0':    '-%scope%',
      'mod#1':       'S.I.H.BigDecimalUtils.Mod(%scope%, %args%)',
      'signum#0':    '%scope%.Sign',
      'remainder#1': 'S.I.H.BigDecimalUtils.Remainder(%scope%, %args%)',
      'divideToIntegralValue#1':
                     'S.I.H.BigDecimalUtils.Idiv(%scope%, %args%)',
      'divideAndRemainder#1':
                     'S.I.H.BigDecimalUtils.DivideAndRemainder(%scope%, %args%)'
      'valueOf#1':   'Saxon.Impl.Helpers.BigDecimalUtils.ValueOf(%args%)',
      'intValue#0':  '((int)%scope%)',
      'longValue#0': '((long)%scope%)',
      'doubleValue#0':
                     '((double)%scope%)',
      'floatValue#0':'((float)%scope%)',
      'longValueExact#0':
                     'S.I.H.BigIntegerUtils.LongValueExact(%scope%)',
      'setScale#1':  '%scope%', (:no-op, values are normalized:)
      'setScale#2':  '%scope%', (:no-op, values are normalized:)
      'stripTrailingZeros#0':
                     '%scope%', (:no-op, values are normalized:)
      'toBigInteger#0':
                     '((System.Numerics.BigInteger)%scope%)',
      '*':           '%scope%.%Name%(%args%)'
  })"/>

</xsl:template>
```

This is a template rule in mode `methods`, a mode that is only used to process `MethodCall` expressions, so we don't need to repeat this in the match pattern. This particular rule handles all calls where the target class is `java.math.BigDecimal`. It delegates the processing to a function `f:apply-Format()` which is given as input a set of sub-rules supplied as a map in a custom microsyntax. Given the name and arity of the method call, this function looks up the applicable sub-rule, and interprets it: for example `value1.add(value2)` translates to `(value1+value2)` (C# allows user-defined overloading of operators such as "+"). Some methods such as `mod()` are converted into calls on a static helper method (written in C#) in class `Saxon.Impl.Helpers.BigDecimalUtils`.

Most of the product's dependencies have proved easy to tackle using one or more of these mechanisms. We were able to use rewriting more often than I expected – for example it's used to replace the dependency on Java's BigDecimal class with a third-party library, Singulink.Numerics.BigDecimal. It's worth showing the XSLT code that drives this:

Iterators and Iterables

There is a close correspondence between the Java interface `Iterable` and C#'s `IEnumerable`; and similarly between Java's `Iterator` and C# `IEnumerator`. In both cases the interface is closely tied up with the ability to write a "for each" loop. If we're going to be able to translate this Java:

```
for (Attribute att : attributes) {...}
```

into this C#:

```
foreach(Attribute att in attributes) {...}
```

then the variable `attributes`, which is an `Iterable` in Java, had better become an `IEnumerable` in C#. We can handle that by rewriting class names; and we can also rewrite the method `attributes.iterator()` as `attributes.GetEnumerable()` so that it satisfies the C# interface. What now gets tricky is that Java's `Iterator` has two methods `hasNext()` and `next()` which don't correspond neatly to C# `IEnumerator`, which has `MoveNext()` and `Current`.

Specifically, `hasNext()` is stateless, and can be called any number of times, while `MoveNext()` is state-changing and can only be called once. However, "sane" code that uses an iterator always makes one call on `hasNext()` followed by one call on `next()`, and that sequence translates directly to a call on `MoveNext()` followed by a call on `Current`. So the converter assumes that the code will follow this discipline – and if we find code that doesn't, then we have to change it[9].

Inner classes

Java effectively has three kinds of inner class: named static inner classes, named instance-level inner classes, and anonymous classes. Only the first of these has a direct equivalent in C#.

Saxon makes extensive use of all three kinds of inner class. The converter makes a strenuous effort to convert all of them to static named inner classes, but this doesn't always succeed. In some cases it can't succeed, because there are things that static named inner classes aren't allowed to do.

Sometimes the conversion can be made to work with the help of hints supplied as Java annotations. For example we might see the following annotation on a method that instantiates an anonymous inner class:

```
@CSharpInnerClass(outer=false,
        extra={"net.sf.saxon.expr.XPathContext context",
               "net.sf.saxon.om.Function function"})
```

This indicates to the converter that in the generated static inner class, there is no need to pass a reference to the outer `this` class (because it's not used), but there is a need to pass the values of variables `context` and `function` from the outer class to the inner class. (Annotations, like anything else in the Java source code, are parsed by JavaParser and made visible in the syntax tree.)

Overriding, Covariance, Contravariance

As we've already mentioned, C# requires any method that is overridden to be declared `virtual`, and any method that overrides another to be declared with the modifier `override`. We handle this by analyzing the class hierarchy and recording the analysis in the digest XML file, which is available to the stylesheet that generates the C# code.

In addition, Java allows an overriding method to have a covariant return type: if `Expression.evaluate()` returns `Sequence`, then `Arithmetic.evaluate()` can return `AtomicValue`, given that `AtomicValue` is a subclass of Sequence. C# doesn't allow covariant return types until version 9.0 of the language, and we decided this was a new promised feature that we would be unwise to rely on. Instead:

- when we're analyzing the class hierarchy, we detect any use of covariance, and change the overriding method to use the same return type as its base method;
- when we're analyzing the class hierarchy, we detect any use of covariance, and change the overriding method to use the same return type as its base method when we find a call to a method that's been overridden with a covariant return type, we insert a cast so the expected type remains as it was.

Java allows interfaces to define default implementations for methods; C# does not. The transpiler handles default method implementations by copying them into each subclass. This of course can lead to a lot of code duplication, so we have eliminated some of the cases where we were using default methods unnecessarily.

Generics

I've already mentioned that we identified early on that generics would be a problem, and one of the steps we took was to reduce unnecessary and unproductive use of generic types. In fact, we have almost

[9] A benefit of having the parsed Java code in XML format is that it's easy to do queries to search for code that needs to be inspected.

totally eliminated all use of generics in Saxon-defined classes, which was the major problem. That leaves generics in system-defined classes (notably the collection classes such as `List<T>`) which we can't easily manage without.

In fact, most uses of these classes translate from Java to C# without trouble. But there are still a few difficulties:

Diamond Operators

Java allows you to write `List<X> list = new ArrayList<>()` (referred to as a diamond operator, though it's not technically an operator). In C# it has to be `new ArrayList<X>()`. So we need to work out what X is – essentially by applying the same type inferencing rules that the Java compiler applies. The way we do this is by recognising common cases: object instantiation on the right-hand side of an assignment, in a return clause, in an argument to a non-polymorphic method, etc. The logic is quite complex, and it catches perhaps 95% of cases. The remainder are handled by changing the Java code: either by introducing a variable, or by adding the type redundantly within the diamond.

XSLT template rules really come into their own here. We handle about a dozen patterns where the type of the parameter can be inferred, and each of these is represented by a template rule. As we get smarter or discover more cases, we can simply add more template rules. Here's an example of one of the rules:

```
<xsl:template match="*[@nodeType='ReturnStmt']
        [ancestor::member[1]/type/@RESOLVED_TYPE]/*">
   <xsl:variable name="type"
                 select="ancestor::member[1]/type/@RESOLVED_TYPE"
                 as="xs:string"/>
   <xsl:value-of select="f:extract-type-arguments($type)"/>
</xsl:template>
```

This rule detects a diamond operator appearing in a return statement (the rule appears in a module with default mode `diamond`, which is only used to process expressions that have already been recognised as containing a diamond operator). It finds the ancestor method declaration (`ancestor::member[1]`), determines the declared type of the method result, and inserts that into the C# code as the type parameter in place of the diamond operator.

Wildcards

The Java wildcard constructs `<? extends T>` and `<? super T>` have no direct equivalent in C#. The way we handle these depends on where they are used. The default action of the converter is just to replace them with `<T>`, which often works. But in class and method declarations we generate a C# `where` clause to constrain the type bounds, so

```
public class GroundedValueAsIterable<T extends Item>
     implements Iterable<T> {...}
```

becomes

```
public class GroundedValueAsIterable<T> : IEnumerable<T>
     where T : Item {...}
```

One issue we face is that the default type `Object` in Java is less all-embracing than the object type in C#: the former does not include primitive types such as int or double, the latter does. This means that where the required type is `Object`, the supplied value can be null; but this is not so in C#, because primitive types do not allow a null. This permeates the design of collection classes. Often the solution is to constrain the C# class to handle reference types only, using the clause `where T : class`.

Lambda Expressions and Delegates

Lambda expressions in Java translate quite easily to lambda expressions in C#: apart from the use of a different arrow symbol, the rules are very similar.

I've already mentioned that the JavaParser symbol solver struggles a bit with type inference inside lambda expressions, and we sometimes need to provide a bit of assistance by declaring types explicitly.

The main problem, however, is that Java is much more flexible than C# about where lambda expressions are allowed to appear. To take an example, we have a method `NodeInfo.iterateAxis(Axis, NodeTest)`. On the Java side, `NodeTest` is a functional interface, which means the caller can either supply a lambda expression such as `node -> node.getURI() == null`, or they can supply an instance of a class that implements the `NodeTest` interface, for example `new LocalNameTest("foo")`. In C# `NodeTest` must either be defined as a delegate, in which case the caller must supply a lambda expression and not an implementing class, or it can be defined as a regular interface, in which case they can supply an implementing class but not a lambda expression.

To solve this, in most cases we've kept it as an interface, but supplied an implementation of the interface that accepts a lambda expression. So if you want to use a lambda expression here, you have to write `NodeTestLambda.of(node -> node.getURI() == null)`. Which is convoluted, but works.

Exceptions

The most obvious difference here between Java and C# is that C# does not have checked exceptions. Most of the time, all this means is that we can drop the `throws` clause from method declarations.

`Try/catch` clauses generally translate without trouble. A `try` clause that declares resources needs a little more care but we hardly use these in Saxon. The syntax for a `catch` that lists multiple exceptions is a little different, but the conversion rule is straightforward enough.

The main problem is deciding on the hierarchy of exception classes. If the Java code tries to catch `NumberFormatException`, how should we convert it? What exception will the C# code be throwing in the same situation?

To be honest, I think we probably need further work in this area. Although we're passing 95% of test cases already, I think we'll find that quite a few of the remaining 5% are negative tests where correct catching of exceptions plays a role, and we'll need to give this more careful attention.

XSLT Considerations

In this section I'll try to draw out some observations about the XSLT implementation.

Like most XSLT code, it has been developed incrementally: rules are added as the need for them is discovered. This is one of the strengths of XSLT as an implementation language for this kind of task: the program can grow very organically, with little need for structural refactoring. At the same time, uncontrolled growth can easily result in a lack of structure. How many modes should there be, and how do we decide? How should the code be split into modules? How should template rule priorities be allocated?

Again, like most XSLT applications, it's not just template rules: there are also quite a few functions. And as in other programming languages, the set of functions you end up with, and their internal complexity and external API, can grow rather arbitrarily.

It's worth looking a little bit at the nature of the XML we're dealing with. Here's a sample:

```
   <member nodeType="MethodDeclaration">
 <body nodeType="BlockStmt">
  <statements>
   <statement nodeType="ReturnStmt">
    <expression nodeType="BinaryExpr"
                operator="PLUS">
     <left nodeType="MethodCallExpr"
           RETURN="double"
```

```
            RESOLVED_TYPE="net.sf.saxon.expr.Expression">
       <name nodeType="SimpleName" identifier="getCost"/>
       <scope nodeType="MethodCallExpr"
              RETURN="net.sf.saxon.expr.Expression"
              DECLARING_TYPE="net.sf.saxon.expr.BinaryExpression">
         <name nodeType="SimpleName" identifier="getLhsExpression"/>
       </scope>
      </left>
      <right nodeType="BinaryExpr" operator="DIVIDE">
       <left nodeType="MethodCallExpr" RETURN="double"
             RESOLVED_TYPE="net.sf.saxon.expr.Expression">
        <name nodeType="SimpleName" identifier="getCost"/>
         <scope nodeType="MethodCallExpr"
                RETURN="net.sf.saxon.expr.Expression"
                DECLARING_TYPE="net.sf.saxon.expr.BinaryExpression">
          <name nodeType="SimpleName"
                identifier="getRhsExpression"/>
         </scope>
       </left>
       <right nodeType="IntegerLiteralExpr"
              value="2"/>
      </right>
     </expression>
    </statement>
   </statements>
  </body>
  <type nodeType="PrimitiveType"
        type="DOUBLE"
        RESOLVED_TYPE="double"/>
  <modifiers>
   <modifier nodeType="Modifier"
             keyword="PUBLIC"/>
  </modifiers>
  <annotations>
   <annotation nodeType="MarkerAnnotationExpr">
    <name nodeType="Name" identifier="Override"/>
   </annotation>
  </annotations>
</member>
```

This represents the Java code

```
@Override
public double getCost() {
    return getLhsExpression().getCost()
           + getRhsExpression().getCost() / 2;
}
```

It's interesting to look at the values used (a) for the element name (e.g. `body`, `left`, `right`, `expression`, `statement`), and (b) for the `nodeType` attribute (e.g. `ReturnStmt`, `BinaryExpr`, `SimpleName`). Generally, the `nodeType` attribute says what kind of thing the element represents, and the element name indicates what role it plays relative to the parent. (Reminiscent of SGML architectural forms, perhaps?)

As an aside, the same dichotomy is present in the design of Saxon's SEF file, which represents a compiled stylesheet, but there we do it the other way around: if an integer literal is used as the right hand side of an addition, the JavaParser format expresses this as `<right nodeType="IntegerLiteral">`, whereas the SEF format expresses it as `<IntegerLiteral role="right">`. Of course, neither design is intrinsically better (though the SEF choice works better with XSD validation, since

XSD likes the content model of an element to depend only on the element name, not the value of one of its attributes). But the choice does mean that most of our template rules in the transpiler are matching on the `nodeType` attribute, not on the element name, and this perhaps makes the rules a bit more complicated.

Performance hasn't been a concern. I'm pleased to be able to report that of the various phases of processing, the phases written in XSLT are an order of magnitude faster than the phase written in Java; which means that there's no point worrying about speeding the XSLT up. This is despite the fact that (as the above example demonstrates) the XML representation of the code is about 10 times the size of the Java representation.

(Actually, the Java code is 29Mb, the XML is 120Mb, and the generated C# is 18Mb. The C# is smaller than the Java mainly because we drop all comments, and also because the Java total includes modules we don't (yet) convert, for example a lot of code dealing with SAX parsers, localisation, and optional extras such as the XQJ API and SQL extension functions).

But I would like to think that one reason performance hasn't been a concern is that the code was sensibly written. We've got about 200 template rules here, most of them with quite complicated match patterns, and we wouldn't want to be evaluating every match pattern for every element that's processed. In fact, a lot of the time we're doing three levels of matching:

- If we find that we're processing a method call (which is rather common), we have a single template rule in the top-level mode that matches `*[@nodeType='MethodCallExpr']`.
- This template rule then does `<xsl:apply-templates select="." mode="Method-Call"/>`, which searches for a more specific template rule, but only needs to search the set of rules for handling method calls, because they are all in this mode.

  To make the code manageable and maintainable, we put all the template rules for a mode in the same module, and use the XSLT 3.0 construct `default-mode="M"` to reduce the risk of accidentally omitting a `mode` attribute on a template rule or `xsl:apply-templates` instruction.
- Most of the template rules for method calls are structured as one rule per target class; as described earlier, this uses a microsyntax for defining the formatting of each possible method, using XSLT maps.

So it's not a flat set of hundreds of rules; we've used modes (and the microsyntax) to create a hierarchic decision tree. This both improves performance, and keeps the rules simpler and more manageable. It also makes debugging considerably easier: as with any XSLT stylesheet, working out which rules are firing to handle each input element can be difficult, but the splitting of rules into modes certainly helps.

(A little known Saxon trick here is the `saxon:trace` attribute on `xsl:mode`, which allows tracing of template rule selection on a per-mode basis).

Conclusions

Firstly, we've confirmed the viability of using XSLT for transformation of abstract syntax trees derived from parsing of complex grammars, on quite a signficant scale. The nature of XSLT as a rule-based pattern matching language makes it ideally suited for such tasks. It's hard to imagine how the pattern matching code would look if it were written in a language such as Java; it would certainly be harder to maintain[10].

At the same time (and perhaps not quite so relevant to this particular audience, but significant nonetheless) we've demonstrated a pragmatic approach to transpilation. Without writing a tool that can automatically perform 100% conversion of any Java program, we've written a set of rules that works well on the subset of the Java language (and class library) that we're actually using, and more importantly,

[10]While writing the paper, I discovered (without surprise) that transpilation using XSLT has been done before: see https://www.ijcrt.org/papers/IJCRT2005043.pdf . That paper, however, gives little detail of how the conversion is done, and appears only to tackle trivial code fragments.

done so in a way that allows manual intervention to handle the parts that can't (cost-effectively) be automated, without sacrificing repeatability - that is, the ability to re-run the conversion every time the Java code changes. And by writing the rules in XSLT, we've created a transpiler that is readily capable of extension to cover features that we chose to leave out first time around.

I take satisfaction in the quality of the generated C# code. It's human readable, and it appears to be efficient. This is achieved partly by the policy of not worrying too much about edge cases. By doing our own customised conversion rather than writing a product that has to handle anyone's Java code, we can be pragmatic about exactly how faithfully the C# code needs to be equivalent to the original Java in edge cases.