XML String Interpolator for Dotty

Semester Project

Yassin Kammoun
EPFL
first.last@epfl.ch

Abstract

XML literals have always been a controversial feature since introduced in the Scala Language Specification. They are nevertheless expected to disappear from Dotty, a research compiler that will become Scala 3. XML string interpolation was designated the best candidate to replace them. A solution in the form of an external library is preferred in order to remove XML from the language specification.

To address this, our project builds XML string interpolation upon Dotty's principled meta programming framework. Interpolation is processed at compile-time using Dotty's new capability to write macros. Our transformation relies on the standard Scala XML library to ensure behaviour equivalence with Scala's XML literals, and thus with Scala programs making use of them.

1. Introduction

The Scala programming language introduced XML Literals in the very early days of its development. The rationale behind this decision was the promising future augured for XML. In that respect, the authors of the language saw fit to make XML parts of the Scala Language Specification.

XML was not as successful as expected, though. Dotty, a research compiler that will become Scala 3, decided to replace XML literals with XML string interpolation. String interpolation is the process of evaluating a string literal containing one or more placeholders, yielding a result in which the placeholders are replaced with their corresponding values.

It is in this context that we present the implementation of an XML string interpolator for Dotty. Our project relies on Dotty's principled meta programming - a new framework for staging and for some forms of macros - to transform interpolated strings within user code. The implementation provides the very same support of XML specification as Scala 2 does but with a few minor differences, essentially related to the interpolation syntax.

Section 2 provides background information on the new macro framework of Dotty. Section 3 introduces the running example that we use throughout the paper to explain the transformation. Section 4 presents the implementation of the string interpolator. Section 5 briefly mentions the main related work from which our project drew on. And section 6 concludes.

2. Background

Dotty comes with principled meta programming, a new framework for staging and for some forms of macros [1]. It is expressed as strongly and statically typed code using two fundamental operations: quotations and splicing. Quotation is expressed as `'{...}` for expressions and as `'[...]` for types. Splicing is expressed as `${...}`. Macros can then be defined through a combination of inline function, quotation, and splicing. For example, the code below presents an inline function `assert` which calls at compile-time a method `assertImpl` with a boolean expression tree as argument. `assertImpl` evaluates the expression and prints it again in an error message if it evaluates to `false`.

```
import scala.quoted._

inline def assert(expr: => Boolean): Unit =
  ${ assertImpl('{ expr }) }

def assertImpl(expr: Expr[Boolean]) = '{
  if (! ${ expr }) throw new AssertionError(
      s"failed assertion: ${ expr.show }"
  )
}
```

3. Example

Before introducing the running example that will illustrate how our implementation works, it seems appropriate to describe how Scala deals with regular string interpolation. To this end, consider the following code:

```
val name = "James"
println(s"Hello, $name")
```

Under the hood, the Scala compiler (and Dotty incidentally) transforms such an expression into a call to the `s` method of `StringContext`, thus turning the previous snippet into the following one:

```
StringContext("Hello, ", "").s(name)
```

Interpolated expressions become arguments passed to the `s` method. The parts that make up the interpolated string, without the expressions that get inserted by interpolation, become arguments of the *constructor* of `StringContext`. For this to work, however, two conditions must be satisfied: 1) the string parts need to be known at compile-time, 2) the interpolated expressions need to have an implementation, i.e. no `???`-like expressions can be used.

Coming back to our XML string interpolator, the latter can be used like any default string interpolator of `StringContext`, while being subject to the same constraints as described above, though. The only difference lies in the method to call, `xml` in our particular case:

```
val text = "bar"
println(xml"<foo>$text</foo>")
```

And like previously, the compiler rewrites the user code so then it becomes, before the interpolation takes place, as follows:

```
StringContext("<foo>", "</foo>").xml(text)
```

It is actually a bit more complicated than that. For reasons that will be explained in the implementation section, the code rewritten by the compiler is actually more verbose and involves some quite a bit advanced features of Dotty.

In that respect, we pick a more involved example of XML string interpolation for the purpose of discussion. We define an XML string interpolation that describes an HTML document which, in turn, embeds another XML string interpolation that describes the body of the document:

```
xml"""
<html xmlns:x="http://www.w3.org/1999/xhtml">
 <x:head>
  <x:title>draft ${1.0}</x:title>
 </x:head>
 ${
   xml"""
   <x:body>
     <x:p face="verdana">Lorem ipsum...</x:p>
     <x:button disabled="false">ok</x:button>
   </x:body>
   """
 }
</html>
"""
```

4. Implementation

To provide a new string interpolator, one normally needs to create an implicit class which adds a method to `StringContext`. Our implementation avoids creating such an implicit class. It uses instead Dotty's extension methods feature. As a result, the entry point of the interpolator is the following:

```
import scala.quoted._

inline def (ctx: => StringContext) xml
    (args: => (given Scope => Any)*) given Scope =
  ${ internal.Macro.impl('ctx, 'args, '{the[Scope]}) }
```

We extend the class `StringContext` with a new method `xml`. This method happens to be `inline`, and combines both quotation and splicing in order for the compiler to consider it as a macro. It takes two parameters: `ctx` - a `StringContext` - from which we retrieve the string parts of the original interpolated string, and `args` which refer to the interpolated expressions. Before explaining the rest of the signature, especially what the `Scope` thing is all about and where it comes from, we need to say a few words about how Scala actually represents XML literals under the hood and how it deals with XML namespaces.

Scala relies on the standard Scala XML library - an independent project - to represent XML literals. This library provides support for the XML literal syntax in Scala programs. To this end, the library defines an abstract syntax tree mapping its nodes to conceptual XML components, among which are comments, elements, entities, but also namespaces. XML namespaces are used for providing uniquely named elements and attributes in an XML document. The standard Scala XML library represents namespaces by instances of `scala.xml.NamespaceBinding`. This (case) class happens to be *recursive* in the sense that it defines a `parent` parameter of the very same type.

The standard Scala XML library uses namespaces to describe the scope of XML elements. The scope of an XML element starts from the namespace of the said element, includes the namespace of its parent, and continues doing so up to the most outer enclosing element's namespace, by default `scala.xml.TopScope`. A situation arises with these XML literals when this is actually not true anymore, i.e. the most outer enclosing element's namespace is not `scala.xml.TopScope`. Consider the following snippet:

```
<foo xmlns:f='scp0'>
  {
    <bar xmlns:b='scp1'/>
  }
</foo>
```

This example shows two Scala XML literals, one embedded into the other. Each literal is an element that defines a namespace. Due to implementation decisions from the authors of the standard Scala XML library, embedding `<bar/>` into `<foo></foo>` has a particular side effect: `<bar/>`'s outer most namespace is not `scala.xml.TopScope`, but rather the namespace of `<foo></foo>` as depicted by the Scala compiler after its *typer* phase:

```
var $tmp: NamespaceBinding = TopScope;
$tmp = new NamespaceBinding("f", "scp0", $tmp);
{
 val $s: NamespaceBinding = $tmp;
 new Elem(null, "foo", Null, $s, false, ({
  val $buf: NodeBuffer = new NodeBuffer();
  $buf.&+(new Text("\n "));
  $buf.&+({
   var $tmp: NamespaceBinding = $s;
   $tmp = new NamespaceBinding("b", "scp1", $tmp);
   {
    val $scope: NamespaceBinding = $tmp;
    newElem(null, "bar", Null, $s, true)
   }
   // ...
```

Considering the same example with our string interpolator makes the task of producing the same side effect quite difficult:

```
xml"""
<foo xmlns:f='scp0'>
${
   xml"<bar xmlns:b='scp1'/>"
 }
</foo>
"""
```

... rewritten as ...

```
StringContext("<foo xmlns:f='scp0'>", "</foo>").xml(
  xml"<bar xmlns:b='scp1'/>"
)
```

... eventually rewritten as ...

```
StringContext("<foo xmlns:f='scp0'>", "</foo>").xml(
  StringContext("<bar xmlns:b='scp1'/>").xml()
)
```

On the one hand, the evaluation of the interpolated string `xml"<bar/>"` needs to take place during the evaluation of `xml"<foo></foo>"`, neither before nor after it. On the other hand, the scope of `xml"<bar/>"` needs to be completed with the namespace of `xml"<foo></foo>"`, or in other words the namespace of `xml"<foo></foo>"` needs to be passed somehow to `xml"<bar/>"` during the evaluation of the latter. The running example introduced in section 3 would also be subject to such an evaluation process since it involves two XML string interpolations, one embedded into the other.

Our implementation responds to this challenge by defining first a contextual abstraction named `Scope` at the very same level as the interpolator's definition, along with an `implicit` value `top` that serves as the default scope with which an interpolation should normally start:

```
type Scope = scala.xml.NamespaceBinding
implicit val top: Scope = scala.xml.TopScope
```

The interpolator then expresses a dependency to a given scope by means of Dotty's inferable parameters feature with the `given` clause. If the evaluation context is bound to a specific scope, that scope is considered during the interpolation. Otherwise, the default scope denoted by `top` is used. Last but not least, the interpolated expressions designated by the variable argument `args` are not only by-name parameters to differ their evaluation, but also contextual arguments of type `given Scope => Any`. This essentially means that the evaluation of such arguments requires some scope that can either be the default one or a context-dependant one.

To complete the picture for any purpose whatsoever, we propose a program example that serves as a basis for illustrating how the compiler deals with all the bricks mentioned in the previous two paragraphs once put together:

```
import scala.quoted._
object P {

  type Scope

  def xml(args: (given Scope => Any)*)
        given Scope: Any =
    ${ xmlImpl('args, '{the[Scope]}) }

  def xmlImpl(args: Expr[Seq[given Scope => Any]],
              scope: Expr[Scope]) {
    val arg: Expr[given Scope => Any] = ???
    arg.apply(scope)
    ???
  }

  implicit val top: Scope = ???

  xml(
    println(
      xml("b")
    )
  )
}
```

Similarly to our string interpolator implementation, this program defines a new type denoting a scope, along with a default one, namely `top`. It also defines an `xml` function expressing a dependency to a scope. Moreover, it takes by-name contextual arguments of type `given Scope => Any`, again, exactly like our implementation. The body of the function shows the last *magic* trick that completes the solving of the original problem of contextual scope: beta-reduction of quoted contextual function. This gives us the capability to apply a given scope to a quoted contextual argument, a quoted (contextual) interpolated expression in the particular case of our interpolator.

As a whole, the previous program is not really interesting; it is rather how the compiler rewrites it at the end of its *frontend* phase that interests us, in particular the part of the two nested calls to `xml` which emphasizes the passing of contextual scope:

```
P.xml(
[{
  def $anonfun(implicit ev$1: P.Scope): Any =
   println(
    P.xml(
    [{
      def $anonfun(implicit ev$2: P.Scope): Any = "b"
      closure($anonfun)
     }
     : (given P.Scope => Any)]:(given P.Scope => Any)*
    )(ev$1)
   )
   closure($anonfun)
 }
: (given P.Scope => Any)]:(given P.Scope => Any)*
)(P.top)
```

The expression passed to the outer call to `xml` is turned into an anonymous function because of pass-by-name parameter passing of the function. This anonymous function takes an implicit parameter: the contextual scope. The scope is then applied to the inner call to `xml`, thus passing the contextual scope to the contextual expression. This example remains rather basic but gives anyhow a good idea of how our implementation passes a scope to a contextual interpolated expression.

This concludes the long but necessary explanation of the `xml` interpolator signature. The body of the interpolator is rather straightforward to understand; it invokes the internal implementation of the interpolation with a call to `internal.Macro.impl`, passing the quoted version of both the parameters and the given scope.

The rest of the section describes how the internal implementation of the macro is able to rewrite the AST of the code that is inside it. It does so in six steps, each being explored in a separate subsection.

4.1 Checking Macro Call Parameters

Before the interpolation process really begins, the parameters of the macro need to be checked according to the constraints that govern `StringContext` introduced in section 3. Recall that the interpolated expressions must have an implementation and the string parts need to be statically known. Assuming that the first interpolated expression of our running example was `${???}` instead of `${1.0}`, the interpolation would result in a compilation error since an implementation is missing.

4.2 Encoding Interpolated Expressions

This phase of the interpolation produces an encoded XML string. The XML within the `StringContext` needs to be parsed at some point. For this to be possible, the parts of the `StringContext` instance are joined together into a single string to form an XML document. However, to be fully complete, the string must also include the interpolated expressions given by the arguments of the `xml` method. Those expressions are encoded as *holes* into that string using the Unicode symbol □ (`0xE000`). Such a hole serves as a marker during parsing to indicate that there is actually an interpolated expression at that current position. Holes differ in length (or in the number of characters used) between each other to distinguish which interpolated expression they refer to. We denote the beginning of a hole with the code `0xE000` and any subsequent character with the code `0xE001`. For example, the original string of our running example would be encoded like this:

```
"""
<html xmlns:x="http://www.w3.org/1999/xhtml">
<x:head>
  <x:title>draft □</x:title>
</x:head>
□□
</html>
"""
```

This encoding phase also creates a mapping between a hole and the corresponding interpolated expression it denotes for later use. This mapping is based on the index of a given interpolated expression in the variable argument sequence `args`. Our implementation then relies on a very simple convention: the first interpolated expression is encoded with a hole of one character, the second expression is encoded with a hole of two characters, and so on.

4.3 Parsing Encoded XML String

The encoded XML string is then parsed. We implemented a parser using the Scala Standard Parser Combinator Library, formerly part of the Scala standard library. It introduces a dependency to our project, but makes nevertheless very easy to write a parser for a context-free grammar when using Scala's parser combinators. One needs only to write a method for every production. Our parser is then a straightforward mapping of the productions of the XML grammar as defined in the Scala Language Specification [2]. As a matter of fact, the production

```
EmptyElemTag ::= < Name {S Attribute} [S] />
```

is implemented as follows in our parser:

```
object Parser extends JavaTokenParsers {

// other definitions here

private def EmptyElemTag =
  "<" ~> Name ~ (S ~> Attribute).* <~ S.? <~ "/>"
}
```

The parsing eventually returns an abstract syntax tree internal to our implementation, but which is nevertheless closely mirroring the abstract syntax tree the standard Scala XML library is built upon. In case the parsing fails, the interpolation reports an error which consists of a meaningful error message indicating why and where parsing the XML document failed.

4.4 Validating Parsed XML

Provided that parsing went successfully, this phase validates the parsed XML. The validation makes sure of two things: 1) there is no element with duplicate attributes, 2) there is no element with mismatched tags. Given the fact that those verifications are of semantic order, they need to be done post-parsing, especially to get the position of semantically erroneous nodes which is only available after parsing.

Our running example happens to be a *valid* XML document, valid with respect to the aforementioned verifications. This would not be true if for example the `<p>` element turned out to define the attribute `font` twice. Such a situation would lead to a compilation error reporting the definition of duplicate attributes. Same idea for the mismatched tags verification. If, for example, the closing tag of the `<head>` element happened to be misspelled, a compilation error would also occur.

4.5 Type Checking Interpolated Attribute Values

Due to the implementation of the attribute construct in the standard Scala XML library, some precautions need to be taken for interpolated attribute values with respect to their type. Interpolated expressions used as values for attributes need to typecheck under the following typing rules:

- if it is for a namespace attribute, the interpolated value should have type `String`,
- if it is for a regular attribute, the interpolated value should either have type `String`, or either `Seq[Node]` or `Option[Seq[Node]]` from the standard Scala XML library.

This phase thus typechecks interpolated attribute values on the basis of those typing rules. The interpolation fails and typing errors are reported if any of the rules are violated.

From our running example, if the value of the attribute `disabled` of the element `<x:button>` happened to be the interpolated expression `${false}` instead of the string literal `"false"`, the interpolator would report a typing error since this expression has type `Boolean`.

4.6 Transforming The User Code

The last phase of the interpolation process transforms the nodes of the parsing tree into instances from the corresponding classes of the standard Scala XML library. The instanciations of those classes are quoted and eventually used by the compiler to rewrite the AST of the user code during macro expansion. For example, the text `Lorem ipsum...` would eventually be rewritten with an instantiation of `scala.xml.Text`, through a call to a method `expandText` as shown in the following example.

```
package internal
import scala.quoted._

object Expander {

  // other definitions here

  private def expandText(text: Text):
        Expr[scala.xml.Text] = '{
    new _root_.scala.xml.Text(${text.text.toExpr})
  }
}
```

This instantiation is quoted since Dotty's macros need to manipulate quoted expressions so the expansion phase rewrites the user code. Similar methods are defined for other kind of nodes, nodes in the sense of our internal abstract syntax tree.

5. Related work

Our work is greatly inspired by a prototype of XML string interpolator for Scala [3]. The two implementations only differ in the underlying macro framework they rely on. Our interpolator is built upon Dotty's principled meta programming, an integral feature of the language. On the other hand, the aforementioned prototype is based on the experimental macros of Scala which are substantially different. Binary compatibility is therefore not possible between Scala and Dotty programs involving XML string interpolation, but source compatibility is since both implementations transform interpolated strings to class instances from the standard Scala XML library.

6. Conclusion

Our project was able to build an XML string interpolator for Dotty. It shows itself as a good candidate to replace Scala XML literals in Dotty as an external library. We use Dotty's principled meta programming to provide our interpolator in the form of a macro. Our implementation makes full use of Dotty's features and embodies the expressiveness and the elegance of its contextual abstractions.

Overall, our transformation produces semantically equivalent code to Scala XML literals. Source compatibility between Scala and Dotty programs is possible. Compiling a Scala program involving XML literals with Dotty would require the development of a tool transforming those literals into interpolated XML strings, though. Last but not least, our project misses a performance evaluation which could be deferred to a future work. Apart from that, the interpolator is operational and ready to use.

References

[1] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. A practical unification of multi-stage programming and macros. *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2018.

[2] Scala Language Specification Version 2.12. Chapter 10: XML Expressions and Patterns. URL: `https://www.scala-lang.org/files/archive/spec/2.12/10-xml-expressions-and-patterns.html`.

[3] Denys Shabalin and Allan Renucci. String interpolator for Scala to replace built-in xml syntax. URL: `https://github.com/densh/scala-xml-quote`.