Semester Project Report
Implement string interpolator inline unapply

Pietro Vladimir Gorilskij
with
Martin Odersky
supervised by
Nicolas Alexander Stucki
Anatolii Kmetiuk

1 Motivation

Scala is famous for its DSL capabilities which rely in large part on features such as macros, string interpolation, and pattern matching, and their customizability. A very important application of these features is XML, so important that Scala 2 included XML literal syntax, both for creation of XML trees and for pattern matching.

However, in Scala 3, the authors decided that that was a step too far and decided to implement the same functionality using the existing facilities of the language. Thus, the existing XML class hierarchy was extracted into the `scala-xml` package and the `dotty-xml-interpolator` package was created to implement XML literals as a string interpolator to replace Scala 2's special syntax.

In Scala 2, the compiler would check the syntax of XML literals and generate code either for interpolation or pattern matching as appropriate. This ensured that syntax errors would be caught at compile-time and that the implementation would be as performant as possible. Not wanting to give up these advantages, the XML string interpolator in Scala 3 is implemented using macros.

This is where the problem, and consequently the motivation for this project, arises. The particular combination of features needed to implement pattern matching for a string interpolator using macros turned out to cause internal errors in the Scala 3 compiler.

2 The desugaring

Scala does not prescribe an encoding for string interpolators, the compiler only guarantees a certain desugaring, how the desugared code will behave is up to the programmer to implement. There are two contexts where a string interpolator can be used, in statement

position and in pattern position.

For simplicity, I will often refer to both `unapply` and `unapplySeq` methods collectively as `unapply` because the distinction is irrelevant in most cases.

2.1 Statement position

In statement position, in our case when creating an XML tree from an interpolated literal, a statement such as the following

```
val tree = xml"<foo a=$x>$y</foo>"
```

will be desugared into

```
val tree = StringContext("<foo a=", ">", "</foo>").xml(x, y)
```

where we can see that the compiler split the string around the interpolated variables and passed the variables to a method on StringContext with the same name as the interpolator. There are various ways to implement this method, the one we will use will actually be an object with an `apply` method so the code will further be desugared to this

```
val tree = StringContext("<foo a=", ">", "</foo>").xml.apply(x, y)
```

The apply method will have access to the contents of `StringContext` at compile time and to the variables at runtime, thus, the macro backing it will create the necessary tree at compile time and fill in the missing variables when they are available.

```
val tree = xml(StringContext("<foo a=", ">", "</foo>")).apply(x, y)
```

```
val tree = apply(xml(StringContext("<foo a=", ">", "</foo>")))(x, y)
```

2.2 Pattern position

A literal can be used in pattern position to extract values from an XML tree or to pattern match on its structure. The simplest pattern position is the following

```
val xml"<foo a=$x>$y</foo>" = <foo a="x">y</foo>
```

which we expect to define the two variables `x == "x"` and `y == "y"`. This would be desugared to

```
val StringContext("<foo a=", ">", "</foo>").xml(x, y) =
  <foo a="x">y</foo>
```

which isn't legal Scala syntax but is a legal AST, and this is further desugared to

```
val Seq(x, y) = StringContext("<foo a=", ">", "</foo>")
                  .xml
                  .unapplySeq(<foo a="x">y</foo>)
```

assuming the implementation of `unapply` for `Seq` is a compiler builtin.

3 What we want

The `dotty-xml-interpolator` package defines a string interpolator called `xml` to be used both for interpolation and pattern matching. Its intended usage looks like this

Interpolation

```
val tree = xml"<foo>${2 + 2}</foo>"
// tree == <foo>4</foo>
```

Pattern matching

```
val xml"<foo>$x</foo>" = <foo>{7}</foo>
// x == scala.xml.Atom(7)
```

Notice the Scala 2 XML syntax highlighted in orange, I use it here only for the purposes of illustration.

The goal of this project was to implement the pattern matching functionality.

4 What we have

Currently, the `dotty-xml-interpolator` package only implements interpolation. This includes most of the required machinery for compile-time XML literals. Crucially, the package implements parsing and syntax checking for XML.

The string interpolator is implemented as follows

```
extension (inline ctx: StringContext)
  transparent inline def xml(inline args: (Scope ?=> Any)*)
                             (using scope: Scope): Any =
    ${ impl('ctx, 'args, 'scope) }
```

If we boil down this definition to just the parts that are relevant for the purposes of this project, we are left with this

```
extension (inline ctx: StringContext)
  inline def xml(inline args: Any*): Any =
    ${ impl('ctx, 'args, 'scope) }
```

As we can see, the package defines an extension method directly on `StringContext`. This works for interpolation but cannot be extended to work for pattern matching due to how that is desugared (Section 2.2).

5 A new encoding

For a string interpolator to work in pattern position, we need to be able to call `StringContext.xml.unapplySeq(...)` which means `StringContext.xml(...)` cannot be a method. Rather, we want to declare `StringContext.xml` as an object with `apply` and `unapplySeq` methods.

However, while one can add an extension method to an existing class in Scala, one cannot add an extension member. We can get around this restriction by declaring `StringContext.xml` as an extension method which returns the aforementioned object. Notice that this object must contain all the information contained by `StringContext` so we could use a wrapper class. However, for simplicity, we use an opaque type called `StrCtx` which will be equivalent to `StringContext` in its declaration scope but will be treated as a separate type outside of it. We can then define `apply` and `unapplySeq` methods on `StrCtx`. But, since `StrCtx` is just `StringContext`, we cannot add methods to it directly, so we have to implement `apply` and `unapplySeq` as extension methods too.

We create an object called `XML` to serve as the declaration scope of `StrCtx` and to handle conversion from `StringContext` to `StrCtx`.

The full implementation looks like this

```
object XML:
  opaque type StrCtx = StringContext
  def apply(ctx: StringContext): StrCtx = ctx

extension (ctx: StringContext)
  def xml: XML.StrCtx = XML(ctx)

extension (inline ctx: XML.StrCtx)
  inline def apply(inline args: Any*): Any =
    ${ implApply('ctx, 'args) }

extension (inline ctx: XML.StrCtx)
  inline def unapplySeq(inline arg: Any): Option[Seq[Any]] =
    ${ implUnapplySeq('ctx, 'arg) }
```

`implApply` and `implUnapplySeq` are the macro implementations that actually perform the required logic.

6 The first issue

The problem arises at the call-site of the `unapplySeq` method. The desugaring described in Section 2.2 is not performed correctly. Let's use the following code for our example, it's not valid XML syntax but our problem occurs before that becomes relevant

```
val xml"$x" = ???
```

Compilation fails with the following error message

```
|  val xml"$x" = ???
|      ^^^^^^^
|      value of type Option[Seq[Any]] does not take parameters
```

after which the compiler crashes with a `NoSuchElementException`.

The first error looks suspicious. `Option[Seq[Any]]` is the return type of our `unapplySeq` method. We expect the value returned by this method to be inspected and, if it is `Some`, for its contents to be assigned to the variable `x`. Instead, it appears that the compiler is attempting to pass arguments to the value or, in other words, to call it.

6.1 inlinedUnapply

Looking through the stack trace, we determine that the problem occurs in the `inlinedUnapply` method of the `Inliner` class. This method is responsible specifically for inlining `unapply` methods in pattern position. Due to how the compiler works, this case requires special handling.

The body of the `unapply` method is inlined normally but instead of being inserted directly into the surrounding scope, it's wrapped in a new `unapply` method belonging to a new anonymous class. An object of this class is then immediately created and used as the pattern instead. The anonymous class is removed by a later stage of the compiler. The reasons for this peculiarity are beyond the scope of this report.

Our example line of code is equivalent to

```
val x = ??? match
  case xml"$x" => x
```

which, after passing through `inlinedUnapply`, should be turned into

```
val x = ??? match
  case {
    final class $anon():
      def unapplySeq(arg: Any): Option[Seq[Any]] =
        inline { XML.StrCtx("", "").unapplySeq(arg) }
    $anon()
  }(x) => x
```

which is not legal Scala syntax but an intuitive text representation of the AST. The fictitious syntax `inline { ... }` is simply used to indicate where actual inlining occurs.

6.2 What actually happens

This would work if it were not for the fact that we declared `unapplySeq` as an extension method, but since it is, the code that gets inlined is actually the following

```
unapplySeq(XML.StrCtx("", ""))(arg)
```

The difference being in the value that is applied to the parameter list `(arg)`. This value resides in variable `fun`. For a non-extension method, `fun` is simply a class member, namely `XML.StrCtx("", "").unapplySeq` while for an extension method it's a partially applied function, namely `unapplySeq(XML.StrCtx("", ""))`. This has repercussions on how it's handled.

To generate the wrapping function, the compiler reads the signature of `unapplySeq`, which, since it's an extension method, is `StrCtx => Any => Option[Seq[Any]]`. It then passes all the

required arguments to fully apply it, namely `(ctx: StrCtx)(any: Any)`. But this ignores the partial application, the value we actually have has type `Any => Option[Seq[Any]]` and only expects one parameter list. In effect, the compiler attempts to pass the `ctx` parameter twice. This is what causes the compiler to report the "does not take parameters" error. It is also what causes the `NoSuchElementException` as the compiler tries to access the second parameter list expected by the method (which does not exist) causing it to try to get the head of an empty list.

The code responsible for this mistaken assumption is the following

```
val unapplySym = newSymbol(
  cls,
  sym.name.toTermName,
  Synthetic | Method,
  unapplyInfo,
  coord = sym.coord
).entered
val unapply = DefDef(unapplySym, argss =>
  inlineCall(
    fun.appliedToArgss(argss)
      .withSpan(unapp.span)
  )(using ctx.withOwner(unapplySym))
)
```

`DefDef` creates the new `unapplySeq` method inside the anonymous class (represented by `cls`). It determines which arguments should be taken and forwarded to the original method based on its first parameter, `unapplySym`, which in turn derives this information from its own fourth parameter, `unapplyInfo`. The code that calculates `unapplyInfo` is

```
val targs = fun match
  case TypeApply(_, targs) => targs
  case _ => Nil

val unapplyInfo = sym.info match
  case info: PolyType => info.instantiate(targs.map(_.tpe))
  case info => info
```

`sym` is the `unapplySeq` symbol and `sym.info` is its full type signature. We can see some special handling for polymorphic methods but we'll get back to that.

In our case, our `unapplySeq` method is not polymorphic so it will not make use of the `targs` variable. It will match the second case in the second match statement resulting in `unapplyInfo` being equal to `sym.info`. Remember that `sym.info` is the type of the `unapplySeq` method not applied to any parameter lists, namely `XML.StrCtx => Any => Option[Seq[Any]]` which is what eventually causes the crash.

7 The first fix

The way I tried to fix this was to add two more special cases to the calculation of `unapplyInfo`. These cases make sure that if `fun` is partially applied, then we do not include the first parameter list in `unapplyInfo` making sure the expected type and actual type of the value are the same. Due to the previously existing special case for polymorphic methods, these new special cases have to be duplicated and are thus factored out into a separate method (`methodTpeInfo`).

```
def methodTpeInfo(info: Type) =
  info match
    case MethodTpe(_, _, rt: PolyType) =>
      rt.instantiate(targs.map(_.tpe))
    case MethodTpe(_, _, rt) if sym.flags.is(ExtensionMethod) =>
      rt
    case info => info

val unapplyInfo = sym.info match
  case info: PolyType =>
    methodTpeInfo(info.instantiate(targs.map(_.tpe)))
  case info =>
    methodTpeInfo(info)
```

`sym.info` can either be a `PolyType`, a polymorphic method expecting a list of type argruments first, or a `MethodType`, a non-polymorphic method. In the first case we apply the necessary type arguments.

Notice that in `methodTpeInfo`, in the second case of the match statement, we do not need to check if the method is an extension method because only an extension `unapply` method can be a `MethodType` containing a `PolyType`.

After implementing the fix from Section 7, non-polymorphic extension inline `unapply` methods worked perfectly, both called directly and through the desugaring of string interpolator patterns.

8 The second issue

I began testing polymorphic `unapply` methods too and that's where I ran into issues. Some variations of polymorphic methods worked as expected while others crashed the compiler, others still exhibited incorrect behavior. Take these three examples for instance

Example 1

```
object Obj
extension (obj: Obj.type)
  inline def unapply[U](x: U): Option[U] = Some(x)

val Obj(x) = 1
```

works as expected.

Example 2

```
object Obj
extension [T] (obj: Obj.type)
  inline def unapply(x: T): Option[T] = Some(x)

val Obj(x) = 1
```

causes the compiler to crash with an `IndexOutOfBoundsException`.

Example 3

```
object Obj
extension [T] (obj: Obj.type)
  inline def unapply[U](x: T): Option[T] = Some(x)

val Obj(x) = 1
```

doesn't crash but incorrectly reports a type mismatch error

```
   |    val Obj(x) = 1
   |            ^
   |            Found:    (x : Any)
   |            Required: Int
```

8.1 Polymorphic extension methods

Before going further, let's clarify what a polymorphic extension `unapply` method can look like. There are two places where type parameters can be added. They can come after the `extension` keyword, which makes the extension itself polymorphic and allows things like adding an extension method to multiple types at once, or they can come after the method name making just the method itself polymorphic. I will refer to these as the first and second type parameter lists. In general, it looks like this

```
extension [T1, ...] (obj: Obj.type)
  inline unapply[U1, ...](a: A, ...): Option[...] = Some(...)
```

where `T1, ...` is the first list and `U1, ...` is the second list. Both type parameter lists are optional and can have arbitrary numbers of type parameters. The value parameters of the `unapply` method can use type parameters from both lists.

As previously mentioned, extension methods are desugared to curried functions. It is worth taking a look at the type signatures of these functions to better understand what is going on. The full type signature of our hypothetical `unapply` method is

```
[T1, ...] => (Obj.type) => [U1, ...] => (A, ...) => Option[Any]
```

Such a type signature is not (yet) legal Scala syntax, polymorphic signatures cannot have type parameters appearing after value parameters. However, this is the real type of the value at compile time before types are erased. Indeed, one can call this function directly in code as follows

```
unapply[T1, ...](Obj)[U1, ...](a, ...)
```

where the type and value parameter lists have to be passed exactly in that order.

8.2 What is actually happening

Looking at our examples, it is clear that the compiler is not treating the two type parameter lists correctly.

In Example 2, the type of the `unapply` function is `[T] => Obj.type => T => Option[T]`. The `fun` argument that `inlinedUnapply` receives is `unapply[Int](Obj)` which is represented as `Apply(TypeApply(sym.info, Int), Obj)` where `sym.info` is a standin for the full type of `unapply`. Taking a look at our `targs` calculation, we see that this will not match the first case, thus `targs` will be `Nil`. When we try to instantiate `sym.info` using `targs`, `sym.info` expects one type parameter list while `targs` provides none, resulting in an out of bounds access. We failed to extract the first type parameter list.

Example 3 is interesting as it does not cause the compiler to crash but rather results in incorrect behavior. What is occurring is that since the parameter `x` has type `T` and we pass the integer `1` to it, `T` is unified with `Int`. The other type parameter `U` is unconstrained and is thus taken to be the most general type, `Any`. In this case the type of `unapply` is `[T] => Obj.type => [U] => T => Option[T]`. `fun` is `unapply[Int](Obj)[Any]`, represented as `TypeApply(..., List(Any))` since `Any` is the last parameter that was passed. This does match the first case in the calculation of `targs` so `targs` will be `List(Any)`. This time, whin we instantiate `sym.info`, we pass in exactly as many type arguments as expected. However, the first type parameter list `sym.info` expects is `[T]` but we pass it `List(Any)`, meaning `T` will be instantiated to `Any`. Following this logic, the compiler will generate a wrapping function that takes a parameter `x: Any` and passes it to `fun` but `fun` expects a parameter of type `Int` and this is where we get our compilation error, expected `Int` but got `Any`. In this case we extract the second list of type arguments from `fun` and attempt to use it to instantiate the first list.

Looking closer at the calculation of `targs`, we notice that it only ever extracts the second parameter list from `fun`, if it is present. This is because it was written to deal with non-extension methods which can only ever have one type parameter list (the second one).

However, we use the same `targs` to instantiate both lists. This means that if the `unapply` method is an extension method with a first type parameter list, it will be instantiated with the arguments passed to the second list (or with no arguments if there is no second list). This is not the behavior we want and leads to crashes and miscompilations.

Incidentally, Example 1 works because the `unapply` method doesn't have a first type parameter list so `targs` is used to instantiate the second list, which is correct.

9 The second fix

To fix the issue, we explicitly treat the two type parameter lists separately, we call them `targs1` and `targs2`. We compute them as follows

```
val targs1 = fun match
  case TypeApply(Apply(TypeApply(_, targs), _), _) =>
    targs.map(_.tpe)
  case Apply(TypeApply(_, targs), _) =>
    targs.map(_.tpe)
  case _ => Nil

val targs2 = fun match
  case TypeApply(_, targs) => targs.map(_.tpe)
  case _ => Nil
```

Obviously, these match statements are redundant and can be combined but I present them here separately for clarity. If `unapply` has a first type parameter list, it is always the innermost `TypeApply`, nested inside an `Apply` and potentially a second `TypeApply` for the second type parameter list.

If `unapply` has a second type parameter list, `targs2` is the last application (top level `TypeApply`). If it doesn't, the top level is an `Apply` and `targs2` is set to `Nil`.

We then change how we use these variables using `targs1` to instantiate the first list, if it exists, and `targs2` to instantiate the second list inside our helper method `methodTpeInfo`.

```
def methodTpeInfo(info: Type) =
  info match
    case MethodTpe(_, _, rt: PolyType) =>
      rt.instantiate(targs2)
    case MethodTpe(_, _, rt) if sym.flags.is(ExtensionMethod) =>
      rt
    case info => info

val unapplyInfo = sym.info match
  case info: PolyType =>
    methodTpeInfo(info.instantiate(targs1))
  case info =>
    methodTpeInfo(info)
```

Finally, we put all this code in one branch of an if statement with the condition `sym.info.flags.is(ExtensionMethod)`, checking if the `unapply` method we're dealing with is an extension method. If it is not an extension method, we do not need to deal with partial application, so our else branch retains the original code

```
val targs = fun match
  case TypeApply(_, targs) => targs.map(_.tpe)
  case _ => Nil

val unapplyInfo = sym.info match
  case info: PolyType => info.instantiate(targs)
  case info => info
```

After implementing this fix, all variations of polymorphic, non-polymorphic, regular and extension inline `unapply` methods work as expected.

10 The result

Finally, I was able to complete the implementation of the `dotty-xml-interpolator` package by changing the definition of the string interpolator as described in Section 5 and implementing the logic for `unapplySeq` to match the behavior of Scala 2 XML literals. This implementation is fairly basic amounting to a parallel traversal of the tree generated by the interpolator and the tree provided at runtime by the user where gaps (interpolated variables) in the first tree are filled by values from the second tree.

11 A different fix & Future work

In the course of the implementation, I found an alternative solution that passes all current test cases but needs further investigation. Namely, instead of using `sym.info` and adding special cases to calculate the type we expect `fun` to have, it is possible to simply get this type directly through `fun.tpe.widen`. The call to `widen` is necessary in case `fun.tpe` is a singleton type represented by a `TermRef`, essentially a reference to the type of a class member without any of the actual type information.

It is unclear whether `widen` would cause problems under certain specific corner cases. This needs to be investigated and either the code changed or test cases added to prevent this mistake from being made in the future.