Implementing the f string interpolator using Dotty macros

Computer Science Project I - Final Report

Sara Alemanno
EPFL
sara.alemanno@epfl.ch

1. Introduction

Scala 3 is introducing an innovative way of implementing macros. The latter will be run right after the type checking. The aim of this project is to write the f-interpolator macro in Scala 3, taking advantage of Tasty ASTs.

1.1 The f-Interpolator

The f-interpolator macro already exists in Scala 2. This kind of interpolator lets the user prepend `'f'` to any string literal to create formatted strings. Every variable is formatted using format specifiers, that should follow this structure:

```
%[argument_index$$][<][flags][width][.precision]conversion
```

where

- `argument_index` is an integer that refers to the position of the argument in the list of arguments[1].
- `<` is specified when using relative indexing[2].
- `flags` are characters that modify the appearance of the output. It could be text justification, for example.
- `width` corresponds to the number of minimum characters written in the output.
- `precision` restricts the number of characters (behavior depends on the type of the argument).
- `conversion` is a character which indicates how the argument should be formatted.

1.2 Examples

One of the most common examples is the following *(from [str])*:

```
val height = 1.9d
val name = "James"
println(f"$name%s is $height%2.2f meters tall")
```

This will output `James is 1.90 meters tall`.

Another more complex example is the following:

```
f"\textdollar{-1.222222}\%f == \%1$$(5.2f
```

gives `-1.222222 == (1.22)` as

[1] Note that the first argument is referenced by "1", *not* "0" and an argument may be referenced more than once.
[2] Relative indexing : if the format specifier contains a `'<'`, the argument of the previous format specifier is re-used.

- `1$$` refers to the 1st argument,
- Width is 5, which means that we have minimum 5 spaces for the output. Therefore, if `21.22222`, then it will not be cut, but if we have `1`, it will add some spaces and output `  1`.
- Precision is 2, so we have two numbers after the floating point.
- The flag `'('` means that negative numbers are in parenthesis.

When an interpolation parameter is not allowed/correct, an error or a warning is reported and the compilation may be aborted. For example *(from [str])*,

```
val height: Double = 1.9d
scala> f"$height%4d"
```

will report an error as `d` is only allowed for integers and `height` is a `Double`:

```
<console>:9: error: type mismatch;
 found : Double
 required: Int
          f"$height%4d"
             ^
```

1.3 Current implementation

Originally, we used the `Formatter.format` function:

```
implicit class Formatter(c: StringContext) {
   def format(args: Any*): String = ... //macro implementation
}
```

Therefore, any call to `f"$x%d"` was rewritten into `StringContext("", "%d").format(x)` and the macro

- checked the conformance of the type of the argument based on the statically known corresponding parameter of the formatter,
- rewrote it as `"%d".format(x)`.

1.4 New implementation

Every call to the f interpolator will call the macro on

- A `StringContext` that contains all the parts of the string without the arguments, i.e. the variables to format. A new `StringContext` part is computed as soon as we encounter a $ sign (as before).
- A list of all the variables to interpolate (*the arguments*).

For example, `f"$x%d"` will be rewritten as `StringContext("", "%d").f(x)`.

The f interpolator macro gets as inputs `strCtxExpr: Expr[StringContext]` and `argsExpr: Expr[Seq[Any]]` and expands itself as `parts.mkString.format(arguments)`. However, before being expanded, it needs to check the parts and the arguments and add the default formatting.

The most challenging consists in checking that all of the format specifiers are given in the correct order and respect the type of the argument that is interpolated, otherwise an error or a warning should be reported.

2. Implementation

2.1 Theoretical Background *(from [for])*

The conversion parameter is the only one which is mandatory and the most important one. Almost each check will be made based on the value of this parameter. Indeed, given a conversion parameter, we know which values

all the other format specifiers (except relative/argument indexes) are allowed to have and which type the argument being formatted should be.

There are five categories of conversion:

1. **General**: refers to any argument type: `h, H, s, S` and to Booleans : `b, B`.
2. **Character**: refers to the type of any Unicode character, namely Character, Byte, Short and Integer : `c, C`.
3. **Numeric**:
   (a) **Integral**: refers to any integral type: Byte, Short, Integer, Long and BigInteger : `d, o, x, X`.
   (b) **Floating point**: refers to floating-point types, namely Float, Double and BigDecimal : `e, E, f, g, G, a, A`.
4. **Date/Time**: refers to date or time types, namely Long, Calendar and Date: `t, T` always followed by a suffix `H, I, k, l, M, S, L, N, p, z, Z, s, Q, B, b, h, A, a, C, Y, y, j, m, d, e, R, T, r, D, F, c` that specifies which format the date should have: it could be only the year, the seconds, etc.
5. **Special characters**:
   (a) % is represented as `%%` when using the f interpolator, whereas normally a simple `%` is enough.
   (b) A new line is represented as `%n` when using the f interpolator, whereas normally, we use `\n`.

Some types of parameters cannot be used depending on the chosen conversion:

- `width` parameter is accepted unless we use it with `"%n"`.
- `precision` is only allowed for `e, E, f, g` and `G` conversion.
- `flags` parameters are a bit more complex :
  - general: `'-'` is allowed and `'#` is only allowed if the argument is `Formattable`. No other flag is allowed.
  - character: only `'-'` flag is accepted.
  - `'d'`: every flag, except `'#'` is allowed.
  - `'o', 'x', 'X': '-', '#', '0'` are always allowed and `'+', ' ', '('` are only allowed if the argument is a BigInteger.
  - floating-point: every flag is accepted, except `', ', '('` that are only accepted for conversion `e, E, f, g, G`.
  - date/time: only `'-'` flag is accepted.

2.2 Implementation Goal

Implementation aims at checking the formatting specifiers and their matching argument to interpolate. This is done at compile time, inside the f interpolator macro. There are many kinds of errors/warnings that can be reported, as listed below:

- *There are no parts* [error]: if the `parts` of the `StringContext` are empty.
- *Too few/many arguments* [error] : this happens when we have too many/few arguments compared to the number of formatting string parts there are in the `StringContext`.
- *Argument index out of range* [error]: if the `index` parameter given by the user is not in the range of the number of actual arguments.
- *Index is not this arg* [warning]: if the argument `index` is not the one corresponding to the actual argument.

- *Argument index ignored if `<` flag is present* [warning]: when relative indexing is used and we still specify the value of an argument, then the latter value will be ignored and the value of the argument for the previous format will be taken into account.
- *No last arg* [error]: if relative indexing is used but there is not any argument before.
- *Illegal flag '[flag]'* [error]: if the user uses an unknown `flag` or use it in a non-allowed way.
- *Only '[flag]' allowed for [conversion]* [error]: if a non-allowed flag is used, whereas only one in particular is allowed.
- *'[flag]' not allowed for [conversion]* [error]: if we use a flag that should be allowed given the conversion type, but is not given the specific conversion character. For example, if a flag is allowed for integral conversions, but not for the `d` conversion.
- *Only '[flag]' for BigInt conversion to o, x, X* [error]: if a given flag is specified, but should only be used for BigInt argument type.
- *Width not allowed* [error]: if the user uses a `width` for the line separator.
- *Precision not allowed* [error]: if the user gives a `precision` that the compiler does not understand and cannot apply in the formatting part.
- *Illegal conversion character* [error]: if the user uses an unknown formatting string.
- *Date/time conversion must have two characters* [error]: if there is only `'t'` as formatting parameter, without any required suffix.
- *Missing conversion operator in '[formatting substring]'; use %% for literal %, %n for new line* [error]: if we are missing the `conversion` parameter, i.e. there is not even an illegal conversion character.
- *Conversions must follow a splice; use %% for literal %, %n for new line* [error]: if the formatting substring has correct specifiers, but no argument is bound/referenced or if the formatting substring is too far from the argument, as `f"$drandom-leading-junk%d"`.
- *type mismatch; found : String required: java.util.Formattable* [error]: if we use the `'#' flag` with a general type that is not a subtype of `java.util.Formattable`.
- *'[char]' doesn't seem to be a date or time conversion* [error]: if the suffix of the date (char) formatting parameter does not exist.
- *type mismatch; found : [type1] required: [type2]* [error]: if we have an argument type type1 which is not compatible with the `conversion` parameter, that only accepts type1. *Assumption: we ignore the implicit conversion.*

The aim of the implementation is to catch and report all of these cases with their corresponding message and position inside the file that is being compiled.

2.3 Implementation Details

The inlined macro is :

```
inline def f(sc: => StringContext)(args: Any*): String = ${ interpolate('sc, 'args) }
```

We need to implement the `interpolate` function, which receives two arguments:

- `strCtxExpr: Expr[StringContext]`
- `argsExpr: Expr[Seq[Any]]`

This function analyzes the content of the parts of the `StringContext` and the type of the arguments inside `argsExpr`:

- Check if the formatting specifiers inside the `StringContext` parts are allowed with the conversion character.

- Check that the type of the argument bound with the format specifiers is allowed with the conversion character, i.e. is a subtype of at least one of the types which are allowed.

The function applies blacklisting to its parameters, i.e. if anything is not allowed, an error/warning is reported. It was implemented in ten steps, which are described below.

2.3.1 Step 1 : Extract the parts and the arguments

`Expr` are useful, because they allow to extract the type of what they contain, to extract their position, to extract the source file which they are into, which is helpful when reporting errors and warnings. On the whole, we need:

- The expression containing the list of arguments to report errors and warnings about the whole list of arguments,
- The expression containing each single argument, i.e. a list `List[Expr[Any]]` to report errors and warnings on a single argument and to extract the type of them,
- The expression containing the `StringContext` to report errors and warning on the whole `StringContext`,
- The expression containing each single part, i.e. a list `List[Expr[String]]` to report errors and warnings on a single part,
- The list of parts, i.e. a `List[String]` that contains all the parts as String so that we can analyze their content and find the format specifiers their contain.

The expressions containing the `StringContext` and the list of arguments are given as parameters to the function `interpolate`. Therefore, we only need to extract the parts and arguments as lists of expressions and to extract the parts as list of `String`.

To get a list of expressions containing literals from an expression containing a `StringContext`, we need to pick the underlying argument of the unsealed expression and return its content as a list. To get its content, we need to pattern match and the underlying argument could either be a `StringContext` or a `new StringContext`. If it is neither of them, we simply return a `QuoteError` for simplicity, and to avoid throwing an Exception, which would be too expensive.

```
strCtxExpr.unseal.underlyingArgument match {
  // StringContext(parts1)
  case Apply(Select(Select(_, "StringContext") | Ident("StringContext"), "apply"),
    List(parts1)) => parts1.seal.cast[Seq[String]] match {
      case ExprSeq(parts2) => parts2.toList
      case _ => QuoteError("Expected statically known String Context", strCtxExpr)
    }
  // new StringContext(parts1)
  case Apply(Select(New(TypeIdent("StringContext")), _), List(parts1)) =>
    parts1.seal.cast[Seq[String]] match {
      case ExprSeq(parts2) => parts2.toList
      case _ => QuoteError("Expected statically known String Context", strCtxExpr)
    }
  case _ => QuoteError("Expected statically known String Context", strCtxExpr)
}
```

The same process is used for the arguments to transform an expression containing a sequence into a list containing expressions:

```
argsExpr.unseal.underlyingArgument match {
  case Typed(Repeated(args, _), _) => args.map(_.seal)
  case tree => QuoteError("Expected statically known argument list", argsExpr)
}
```

To extract the parts as list of `String`, we use a simple pattern match on each expression and return the literal value which is contained in the `Expr`:

```
expression match {
    case Const(string : String) => string
    case _ => QuoteError("Expected statically known literal", expression)
}
```

2.3.2 Step 2 : Create a reporter

As we intent to report a huge amount of compilation errors and warnings, the help of a reporter `trait` was essential.

The aim of this `trait` is to avoid the computation of the position of each expression that is needed to report an error or a warning inside the function, but do it automatically inside the reporter. Therefore, the errors and warnings only require at most `message : String, index : Int` and `offset: Int` and at least `message: String`. The reporter will find the needed position from the `Expr` it knows (arguments, list of expressions containing the arguments, `StringContext`, list of expressions containing the parts).

There are two advantages: the positions are only computed inside the reporter. We only need the indexes and the offset, the source file is computed only once per file.

The reporter is created inside the main `interpolate` function and passed as argument to the helper one. Consequently, after steps 1 and 2, we have two `interpolate` functions : the main function that extract the parts and arguments, creates the reporter and calls the helper function and the helper function that checks the parameters and expands the macro.

```
// main interpolate function
private def interpolate(strCtxExpr: Expr[StringContext], argsExpr: Expr[Seq[Any]])
(implicit reflect: Reflection): Expr[String] = {
    import reflect._
    val sourceFile = strCtxExpr.unseal.pos.sourceFile

    val partsExpr = getPartsExprs(strCtxExpr) // Extract the parts as a list of Expr of String
    val args = getArgsExprs(argsExpr) // Extract the parts as a list of Expr of Any

    // Creation of the reporter
    val reporter = new Reporter{
      private[this] var reported = false
      private[this] var oldReported = false

      // Reports error/warning of size 1 linked with a part of the StringContext.
      def partError(message : String, index : Int, offset : Int) : Unit = {
        reported = true
        val positionStart = partsExpr(index).unseal.pos.start + offset
        reflect.error(message, sourceFile, positionStart, positionStart)
      }
      def partWarning(message : String, index : Int, offset : Int) : Unit = {
        reported = true
        val positionStart = partsExpr(index).unseal.pos.start + offset
        reflect.warning(message, sourceFile, positionStart, positionStart)
      }

      // Reports error linked with an argument to format.
      def argError(message : String, index : Int) : Unit = {
        reported = true
```

```
      reflect.error(message, args(index).unseal.pos)
    }

    // Reports error linked with the list of arguments or the StringContext.
    def strCtxError(message : String) : Unit = {
      reported = true
      val positionStart = strCtxExpr.unseal.pos.start
      reflect.error(message, sourceFile, positionStart, positionStart)
    }
    def argsError(message : String) : Unit = {
      reported = true
      reflect.error(message, argsExpr.unseal.pos)
    }

    // Claims whether an error or a warning has been reported.
    def hasReported() : Boolean = {
      reported
    }

    // Stores the old value of the reported and reset it to false.
    def resetReported() : Unit = {
      oldReported = reported
      reported = false
    }

    // Restores the value of the reported boolean that has been reset.
    def restoreReported() : Unit = {
      reported = oldReported
    }
  }

  // Call to the helper function
  interpolate(partsExpr, args, argsExpr, reporter)
}
```

2.3.3 Step 3 : Implementation checks

In the helper function, as we have the extracted parameters and the reporter, we can finally start implementing the interpolation checks. This was done in three steps :

- Check whether the size of the argument matches with the number of formatting parts (see subsection 2.3.4). If this is not the case, compilation is aborted.
- Check that the argument index, relative indexing, flags, width and precision are correct and are allowed with the given conversion parameter (see subsection 2.3.8),
- If no error/warning has been reported,
  - If no argument is specified or indexed, report an error (*conversions must follow a splice; use %% for literal %, %n for newline*).
  - If an argument is specified, check whether the conversion parameter corresponds to its type (see subsection 2.3.9).

2.3.4 Step 4 : Check the sizes

The following equality must always hold : `number of parts = number of argument + 1`. Indeed, the first elements of parts does not have any formatting substring. There is always one more part than arguments. For example:

- `"$2%d"` will not be rewritten as `StringContext("%d").f(2)`, but rather as `StringContext("", "%d").f(2)` with an empty `String` as first part.
- Even if some argument are referenced, the size of parts does not increase, i.e. `StringContext("%d is even as it is a multiple of %1$$d.").f(2)` will not be rewritten as `StringContext("", "%d is even as it is a multiple of ", "%1$$d.").f(2)` but rather as `StringContext("", "%d is even as it is a multiple of %1$$d.").f(2)`, which still makes the equality hold.

Violating this equality will lead to an error. The implementation of `checkSize`, function that checks the equality is pretty straight forward:

- If the parts are empty, we report an error on the `StringContext`: *there are no parts*.
- If there are less arguments that parts, i.e. `number of parts + 1 > number of arguments`, we report an error on the last argument: *too few arguments for interpolated string*.
- If there are more arguments that parts, i.e. `number of parts + 1 < number of arguments`, we report an error on the first argument in the list that "overflows": *too many arguments for interpolated string*.

2.3.5 Step 5 : Add the default format

Before checking the formatting parameters, we need to make sure each part, except the first one has a formatting substring, with at least a conversion character. If the user does not specify any conversion format, the default conversion used is `"%s"`. We need to add this default format manually. To do so, we prepend `"%s"` to the part of the `StringContext` if the part is not the first one and it does not start with a `"%"`. A part that does not start with a `"%"` may contain a `"%"` further in it. In such cases, we cannot only add the default format and carry on. This kind of formatting will lead to an error : *conversions must follow a splice; use %% for literal %, %n for newline*.

2.3.6 Step 6 : Look for all the formatting substrings inside a part

A part may have many formatting substrings. Indeed, some may reference the argument. We need to check each substring formatting parameters and hence, need a function to find all of them. We assumed that every `"%"` in the part implies a new formatting substring.

Therefore, the `checkPart` function, which checks whether the format specifiers of a part of the `StringContext` are correct and allowed with the given conversion character and returns the conversion character bound with the type of the argument, needs to be recursive. As pointed out in the simplified implementation below, it uses function `getFormattingSubstring` to check whether a new formatting substring is found inside the part. This function checks if a `%` is in the part and returns its index inside an `Option`, returns `None` if nothing is found.

```
def checkPart(part : String, start : Int, argument : Option[(Int, Expr[Any])],
maxArgumentIndex : Option[Int]) : List[(Option[(Type, Int)], Char, List[(Char, Int)])] = {
  val hasFormattingSubstring = getFormattingSubstring(part, part.size, start)
  if (hasFormattingSubstring.nonEmpty) {
    val formattingStart = hasFormattingSubstring.get
    val (hasArgumentIndex, argumentIndex, flags, hasWidth, width, hasPrecision, precision,
        hasRelative, relativeIndex, conversion) = getFormatSpecifiers(part, argIndex, argIndex + 1,
        false, formattingStart) // see step 7
    if (!reporter.hasReported()){
      val conversionWithType = checkFormatSpecifiers(argIndex + 1, hasArgumentIndex, argumentIndex,
```

```
      Some(argIndex + 1), maxArgumentIndex, hasRelative, hasWidth, width, hasPrecision, precision,
      flags, conversion, Some(arg.unseal.tpe), part) // see step 8
      conversionWithType :: checkPart(part, conversion + 1, argument, maxArgumentIndex) // recursion
    } else checkPart(part, conversion + 1, argument, maxArgumentIndex) // recursion
    /* if the format specifiers are wrong for that formatting substring, report an error and
    check the next formatting substrings */
  } else Nil // end of the recursion when no more formatting substrings are found
}
```

2.3.7 Step 7 : Get the formatting parameters

Once we have every formatting substrings that we need to check, we need all the different parameters per formatting substring, i.e. the flags, the argument index, etc. To do so, we need to scan the whole substring:

- If we find a digit, it could be an argument index or the width.
- If we find a `'$'`, we have an argument index, except if no digit appears before. In that case, an error is reported (*Missing conversion operator...*).
- If we find a `'<'`, we have relative indexing.
- Then, we look for flags : `'-', '', '+', ' ', '0', ',', '('` (there may be more than one).
- We check again for width.
- We look for precision, i.e. for a `'.'` followed by digit. If no digit follows, we report an error (*Missing conversion operator...*).
- Finally, we have the conversion. If we don't have find one, we report an error (*Missing conversion operator...*).

Function `getFormatSpecifiers` does so and returns every parameter bound with its index in the formatting substring.

Note that this function is very long but could not be more factorized/optimized as each specifier position depends from the other.

2.3.8 Step 8 : Check the formatting parameters

Once we know which formatting parameters are used and where they are in the part, if no error has been reported before, we can check whether they are allowed with respect to the conversion character. A function was written for each type of conversion : general, integral, floating point, character, date/time and special characters. Errors are reported when a particular parameter is not allowed (see section 2.1 for more details).

As an example, the `checkTimeConversion` checks the specifiers for a time conversion, i.e. a conversion character `'t'` with suffix:

```
def checkTimeConversion(partIndex : Int, part : String, conversionIndex : Int,
flags : List[(Char, Int)], hasPrecision : Boolean, precision : Int) = {
  def checkTime(part : String, partIndex : Int, conversionIndex : Int) : Unit = {
    if (conversionIndex + 1 >= part.size)
      // date/time conversions are always 't' with a suffix
      reporter.partError("Date/time conversion must have two characters", partIndex, conversionIndex)
    else {
      /* only some characters are available as suffixes, we report an error if a not allowed one is
      used */
      part.charAt(conversionIndex + 1) match {
        case 'H' | 'I' | 'k' | 'l' | 'M' | 'S' | 'L' | 'N' | 'p' | 'z' | 'Z' | 's' | 'Q' => //times
        case 'B' | 'b' | 'h' | 'A' | 'a' | 'C' | 'Y' | 'y' | 'j' | 'm' | 'd' | 'e' => //dates
        case 'R' | 'T' | 'r' | 'D' | 'F' | 'c' => //dates and times
```

```
      case c => reporter.partError("'" + c + "' doesn't seem to be a date or time conversion",
      partIndex, conversionIndex + 1)
    }
  }
 }

 val notAllowedFlagOnCondition = for (flag <- List('#', '+', ' ', '0', ',', '(')) yield (flag,
 true, "Only '-' allowed for date/time conversions")

 // reports "Only '-' allowed for date/time conversions" if any flag other that '-' is specified
 checkUniqueFlags(partIndex, flags, notAllowedFlagOnCondition : _*)

 // reports "precision not allowed" if precision is specified
 checkNotAllowedParameter(hasPrecision, partIndex, precision, "precision")

 checkTime(part, partIndex, conversionIndex)
}
```

Note that we also need to check the argument index, which does not depend on the conversion, but rather on the argument bound with the part that being checked. The following properties must hold :

- `argument index = index of the argument within the list of argument + 1` : if this does not hold, a warning is reported: *Index is not this arg*.
- `0 < argument index <= size of the list of argument` : if this does not hold, an error is reported: *Argument index out of range*.
- If argument index and relative indexing are used at the same time, a warning is reported as the argument index will be ignored: *Argument index ignored if '<' flag is present*.

If no error has been reported during this part but no argument is bound with the given part or indexed, an error is reported at the end of this part, as we do not have any actual argument to format. If no error has been reported during this part, we return every argument bound with its conversion character as list.

2.3.9 Step 9 : Type checking

Type checking happens after steps 1 to 8. We do apply it, if and only if the latter steps did not report any error/warning. Otherwise, we skip it. Type checking function receives as parameter the conversion character bound its corresponding argument. It basically checks that the type of the argument is a subtype of at least one of the types which are allowed for the given conversion character (see section 2.1 for more details). If this is not the case, then an error is reported: *type mismatch; found : [type found] required: [required]*.

Note that we do not type check indexed arguments. If the type of the referenced argument does not match with the conversion character, the error will be reported in the `Formatter.format` function.

2.3.10 Step 10 : Macro expansion

Last but not least, we need to convert `StringContext(p1, p2, p3, ...)` to `p1 + p2 + p3`, where + means concatenation, and call the `format(a1, a2, a3, ...)` function on it, as in the current implementation (1.3). To do so, we use the `mkString` function on the list of `String` and this is the macro that will be inlined :

```
\texttt{'\{(\$\{parts.mkString.toExpr\}).format(\$\{argsExpr\}: \_*)\}}.
```

3. Testing

There are two kinds of tests:

1. Positive tests that test some combination for all the types of conversion.

2. Negative tests that checks some illegal combinations : some were found on [neg] and the other were randomly generated using a choice of characters that could help generating sensitive formatting substrings. Note that normally the program stops compiling as soon as an inlined macro reports an error. Therefore, to be able to check all the errors without creating one file per error, we created a new reporter that, instead or calling `context.error/warning` stores the type of error : part, argument, `StringContext`, args, the position of the element in the list (-1 if it is not an element of a list), the offset position within that element and the error/warning message. The fivetuples contain:
   - 1st element : `true` if error, `false` if warning
   - 2nd element : `0` if part, `1` if arg, `2` if strCtx, `3` if args
   - 3rd element : index in the list if arg or part, `-1` otherwise
   - 4th element : offset, `0` if strCtx, args or arg
   - 5th element : error/warning message

   The reporter will have the following error/warning functions:

```
def partError(message : String, index : Int, offset : Int) : Unit = {
    reported = true
    errors += '{ Tuple5(true, 0, $index, $offset, $message) }
}
def partWarning(message : String, index : Int, offset : Int) : Unit = {
    reported = true
    errors += '{ Tuple5(false, 0, $index, $offset, $message) }
}

def argError(message : String, index : Int) : Unit = {
    reported = true
    errors += '{ Tuple5(true, 1, $index, 0, $message) }
}

def strCtxError(message : String) : Unit = {
    reported = true
    errors += '{ Tuple5(true, 2, -1, 0, $message) }
}
def argsError(message : String) : Unit = {
    reported = true
    errors += '{ Tuple5(true, 3, -1, 0, $message) }
}
```

   The list of errors can then be checked using an assertion.

   For example, `assertEquals(f"$s%b", List((true, 1, 0, 0, "type mismatch;\n found : String \n required: Boolean")))`: it should report an error on the first argument in the list, with message `"type mismatch..."`.

4. Possible Extensions

Some ideas of extensions/ameliorations could be:

- Conversion parameter is mandatory. However, it is the parameter that we check at the very end. We could already abort compilation if it does not type check with the corresponding argument to avoid useless checks that will to an error at the end as no argument is given. This does not respect the actual working of the f interpolator but it could be way more efficient.

- From the type of the argument, we could deduce the conversion character. An idea could be for the user to let the conversion parameter empty and let the compiler deduce it. Of course, some conversion character modify the output, but this could easily be replaced by a specific flag if needed. Therefore, we avoid all the argument errors due to type checking.

5. Remarks

When implementing the macro, I faced two main problems:

- I had to scan the whole part to get the parameters of formatting, but this is ineluctable if we want to get all the format specifiers.
- There are many repetitions, but this is unavoidable as we had to return different kinds of errors/warnings with different messages for each occasion.

5.1 Bugs that still need fixes

- When no argument is specified and we report a part error on the first part, i.e. the part with index 0, the position is one index before where it is supposed to be. For example, `f"%.2%"` shows an error on the `'%'` position, rather than on the `'.'` position. I have written tests for that purpose, but the code, reports correctly an error on part with index 0, with offset 1, which is the reason for which I decided not to correct this. This was also the case for *`f"%.2%", f"%.2n", f"%<s", f"%<c", f"%<tT", f"%1$$d", f"blablablabla %% %.2d", f"b%c.%2ii%iin", f"blablablabla %.2b %%", f"b`*`22%2.c<%{", ...`
- When a warning is reported, no result is output, as if compilation was aborted, which should not be the case. This was the case for *`f"${8}%d ${9}%1$$d", f"$s%s$s%s %1$$<s", f"$s%s $s%1$$s", ...`*

References

Class formatter. `https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.htmldetail`. Accessed: 2019-04-30.

Negative tests. `https://github.com/lampepfl/dotty/blob/master/tests/untried/neg/stringinterpolation_macro-neg.scala`. Accessed: 2019-04-30.

String interpolation. `https://docs.scala-lang.org/overviews/core/string-interpolation.html`. Accessed: 2019-04-30.