Scala 3 Reference / Metaprogramming / Macros

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

Macros

Edit this page on GitHub

> When developing macros enable `-Xcheck-macros` scalac option flag to have extra runtime checks.

Macros: Quotes and Splices

Macros are built on two well-known fundamental operations: quotation and splicing. Quotation is expressed as `'{...}` for expressions and splicing is expressed as `${ ... }`. Additionally, within a quote or a splice we can quote or splice identifiers directly (i.e. `'e` and `$e`). Readers may notice the resemblance of the two aforementioned syntactic schemes with the familiar string interpolation syntax.

```
println(s"Hello, $name, here is the result of 1 + 1 = ${1 + 1}")
```

In string interpolation we *quoted* a string and then we *spliced* into it, two others. The first, `name`, is a reference to a value of type `String`, and the second is an arithmetic expression that will be *evaluated* followed by the splicing of its string representation.

Quotes and splices in this section allow us to treat code in a similar way, effectively supporting macros. The entry point for macros is an inline method with a top-level splice. We call it a top-level because it is the only occasion where we encounter a splice outside a quote (consider as a quote the compilation-unit at the call-site). For example, the code below presents an `inline` method `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(inline expr: Boolean): Unit =
  ${ assertImpl('expr) }
```

```scala
def assertImpl(expr: Expr[Boolean])(using Quotes) = '{
  if !$expr then
    throw AssertionError(s"failed assertion: ${${ showExpr(expr) }}")
}

def showExpr(expr: Expr[Boolean])(using Quotes): Expr[String] =
  '{ [actual implementation later in this document] }
```

If `e` is an expression, then `'{e}` represents the typed abstract syntax tree representing `e`. If `T` is a type, then `Type.of[T]` represents the type structure representing `T`. The precise definitions of "typed abstract syntax tree" or "type-structure" do not matter for now, the terms are used only to give some intuition. Conversely, `${e}` evaluates the expression `e`, which must yield a typed abstract syntax tree or type structure, and embeds the result as an expression (respectively, type) in the enclosing program.

Quotations can have spliced parts in them; in this case the embedded splices are evaluated and embedded as part of the formation of the quotation.

Quotes and splices can also be applied directly to identifiers. An identifier `$x` starting with a `$` that appears inside a quoted expression or type is treated as a splice `${x}`. Analogously, an quoted identifier `'x` that appears inside a splice is treated as a quote `'{x}`. See the Syntax section below for details.

Quotes and splices are duals of each other. For arbitrary expressions `e` we have:

```scala
${'{e}} = e
'{${e}} = e
```

Types for Quotations

The type signatures of quotes and splices can be described using two fundamental types:

- `Expr[T]`: abstract syntax trees representing expressions of type `T`
- `Type[T]`: non erased representation of type `T`.

Quoting takes expressions of type `T` to expressions of type `Expr[T]` and it takes types `T` to expressions of type `Type[T]`. Splicing takes expressions of type `Expr[T]` to expressions of type `T` and it takes expressions of type `Type[T]` to types `T`.

The two types can be defined in package `scala.quoted` as follows:

```scala
package scala.quoted

sealed trait Expr[+T]
sealed trait Type[T]
```

Both `Expr` and `Type` are abstract and sealed, so all constructors for these types are provided by the system. One way to construct values of these types is by quoting, the other is by type-specific lifting operations that will be discussed later on.

The Phase Consistency Principle

A fundamental *phase consistency principle* (PCP) regulates accesses to free variables in quoted and spliced code:

- *For any free variable reference `x`, the number of quoted scopes and the number of spliced scopes between the reference to `x` and the definition of `x` must be equal.*

Here, `this`-references count as free variables. On the other hand, we assume that all imports are fully expanded and that `_root_` is not a free variable. So references to global definitions are allowed everywhere.

The phase consistency principle can be motivated as follows: First, suppose the result of a program `P` is some quoted text `'{ ... x ... }` that refers to a free variable `x` in `P`. This can be represented only by referring to the original variable `x`. Hence, the result of the program will need to persist the program state itself as one of its parts. We don't want to do this, hence this situation should be made illegal. Dually, suppose a top-level part of a program is a spliced text `${ ... x ... }` that refers to a free variable `x` in `P`. This would mean that we refer during *construction* of `P` to a value that is available only during *execution* of `P`. This is of course impossible and therefore needs to be ruled out. Now, the small-step evaluation of a program will reduce quotes and splices in equal measure using the cancellation rules above. But it will neither create nor remove quotes or splices individually. So the PCP ensures that program elaboration will lead to neither of the two unwanted situations described above.

In what concerns the range of features it covers, this form of macros introduces a principled metaprogramming framework that is quite close to the MetaML family of languages. One difference is that MetaML does not have an equivalent of the PCP - quoted code in MetaML *can* access variables in its immediately enclosing environment, with some restrictions and caveats since such accesses involve serialization. However, this does not constitute a fundamental gain in expressiveness.

From `Expr`s to Functions and Back

It is possible to convert any `Expr[T => R]` into `Expr[T] => Expr[R]` and back. These conversions can be implemented as follows:

```
def to[T: Type, R: Type](f: Expr[T] => Expr[R])(using Quotes): Expr[T => R] =
  '{ (x: T) => ${ f('x) } }

def from[T: Type, R: Type](f: Expr[T => R])(using Quotes): Expr[T] => Expr[R] =
  (x: Expr[T]) => '{ $f($x) }
```

Note how the fundamental phase consistency principle works in two different directions here for `f` and `x`. In the method `to`, the reference to `f` is legal because it is quoted, then spliced, whereas the reference to `x` is legal because it is spliced, then quoted.

They can be used as follows:

```
val f1: Expr[Int => String] =
  to((x: Expr[Int]) => '{ $x.toString }) // '{ (x: Int) => x.toString }

val f2: Expr[Int] => Expr[String] =
  from('{ (x: Int) => x.toString }) // (x: Expr[Int]) => '{ ((x: Int) => x.toString)($x) }
f2('{2}) // '{ ((x: Int) => x.toString)(2) }
```

One limitation of `from` is that it does not β-reduce when a lambda is called immediately, as evidenced in the code `{ ((x: Int) => x.toString)(2) }`. In some cases we want to remove the lambda from the code, for this we provide the method `Expr.betaReduce` that turns a tree describing a function into a function mapping trees to trees.

```
object Expr:
  ...
  def betaReduce[...](...)(...): ... = ...
```

The definition of `Expr.betaReduce(f)(x)` is assumed to be functionally the same as `'{($f)($x)}`, however it should optimize this call by returning the result of beta-reducing `f(x)` if `f` is a known lambda expression. `Expr.betaReduce` distributes applications of `Expr` over function arrows:

```
Expr.betaReduce(_): Expr[(T1, ..., Tn) => R] => ((Expr[T1], ..., Expr[Tn]) =>
```

Lifting Types

Types are not directly affected by the phase consistency principle. It is possible to use types defined at any level in any other level. But, if a type is used in a subsequent stage it will need to be lifted to a `Type`. Indeed, the definition of `to` above uses `T` in the next stage, there is a quote but no splice between the parameter binding of `T` and its usage. But the code can be rewritten by adding an explicit binding of a `Type[T]`:

```
def to[T, R](f: Expr[T] => Expr[R])(using t: Type[T])(using Type[R], Quotes):
  '{ (x: t.Underlying) => ${ f('x) } }
```

In this version of `to`, the type of `x` is now the result of inserting the type `Type[T]` and selecting its `Underlying`.

To avoid clutter, the compiler converts any type reference to a type `T` in subsequent phases to `summon[Type[T]].Underlying`.

And to avoid duplication it does it once per type, and creates an alias for that type at the start of the quote.

For instance, the user-level definition of `to`:

```
def to[T, R](f: Expr[T] => Expr[R])(using t: Type[T], r: Type[R])(using Quotes
  '{ (x: T) => ${ f('x) } }
```

would be rewritten to

```
def to[T, R](f: Expr[T] => Expr[R])(using t: Type[T], r: Type[R])(using Quotes
  '{
    type T = t.Underlying
    (x: T) => ${ f('x) }
  }
```

The `summon` query succeeds because there is a given instance of type `Type[T]` available (namely the given parameter corresponding to the context bound `: Type`), and the reference to that value is phase-correct. If that was not the case, the phase inconsistency for `T` would be reported as an error.

Lifting Expressions

Consider the following implementation of a staged interpreter that implements a compiler through staging.

```
import scala.quoted.*

enum Exp:
  case Num(n: Int)
  case Plus(e1: Exp, e2: Exp)
  case Var(x: String)
  case Let(x: String, e: Exp, in: Exp)

import Exp.*
```

The interpreted language consists of numbers `Num`, addition `Plus`, and variables `Var` which are bound by `Let`. Here are two sample expressions in the language:

```
val exp = Plus(Plus(Num(2), Var("x")), Num(4))
val letExp = Let("x", Num(3), exp)
```

Here's a compiler that maps an expression given in the interpreted language to quoted Scala code of type `Expr[Int]`. The compiler takes an environment that maps variable names to Scala `Expr` s.

```
import scala.quoted.*

def compile(e: Exp, env: Map[String, Expr[Int]])(using Quotes): Expr[Int] =
  e match
    case Num(n) =>
      Expr(n)
    case Plus(e1, e2) =>
      '{ ${ compile(e1, env) } + ${ compile(e2, env) } }
    case Var(x) =>
      env(x)
    case Let(x, e, body) =>
      '{ val y = ${ compile(e, env) }; ${ compile(body, env + (x -> 'y)) } }
```

Running `compile(letExp, Map())` would yield the following Scala code:

```
'{ val y = 3; (2 + y) + 4 }
```

The body of the first clause, `case Num(n) ⇒ Expr(n)`, looks suspicious. `n` is declared as an `Int`, yet it is converted to an `Expr[Int]` with `Expr()`. Shouldn't `n` be quoted? In fact this would not work since replacing `n` by `'n` in the clause would not be phase correct.

The `Expr.apply` method is defined in package `quoted`:

```
package quoted

object Expr:
  ...
  def apply[T: ToExpr](x: T)(using Quotes): Expr[T] =
    summon[ToExpr[T]].toExpr(x)
```

This method says that values of types implementing the `ToExpr` type class can be converted to `Expr` values using `Expr.apply`.

Scala 3 comes with given instances of `ToExpr` for several types including `Boolean`, `String`, and all primitive number types. For example, `Int` values can be converted to `Expr[Int]` values by wrapping the value in a `Literal` tree node. This makes use of the underlying tree representation in the compiler for efficiency. But the `ToExpr` instances are nevertheless not *magic* in the sense that they could all be defined in a user program without knowing anything about the representation of `Expr` trees. For instance, here is a possible instance of `ToExpr[Boolean]`:

```
given ToExpr[Boolean] with
  def toExpr(b: Boolean) =
    if b then '{ true } else '{ false }
```

Once we can lift bits, we can work our way up. For instance, here is a possible implementation of `ToExpr[Int]` that does not use the underlying tree machinery:

```
given ToExpr[Int] with
  def toExpr(n: Int) = n match
    case Int.MinValue    => '{ Int.MinValue }
    case _ if n < 0      => '{ - ${ toExpr(-n) } }
    case 0               => '{ 0 }
    case _ if n % 2 == 0 => '{ ${ toExpr(n / 2) } * 2 }
    case _               => '{ ${ toExpr(n / 2) } * 2 + 1 }
```

Since `ToExpr` is a type class, its instances can be conditional. For example, a `List` is liftable if its element type is:

```
given [T: ToExpr : Type]: ToExpr[List[T]] with
  def toExpr(xs: List[T]) = xs match
    case head :: tail => '{ ${ Expr(head) } :: ${ toExpr(tail) } }
    case Nil          => '{ Nil: List[T] }
```

In the end, `ToExpr` resembles very much a serialization framework. Like the latter it can be derived systematically for all collections, case classes and enums. Note also that the synthesis of *type-tag* values of type `Type[T]` is essentially the type-level analogue of lifting.

Using lifting, we can now give the missing definition of `showExpr` in the introductory example:

```
def showExpr[T](expr: Expr[T])(using Quotes): Expr[String] =
  val code: String = expr.show
  Expr(code)
```

That is, the `showExpr` method converts its `Expr` argument to a string (`code`), and lifts the result back to an `Expr[String]` using `Expr.apply`.

Lifting Types

The previous section has shown that the metaprogramming framework has to be able to take a type `T` and convert it to a type tree of type `Type[T]` that can be reified. This means that all free variables of the type tree refer to types and values defined in the current stage.

For a reference to a global class, this is easy: Just issue the fully qualified name of the class. Members of reifiable types are handled by just reifying the containing type together with the member name. But what to do for references to type parameters or local type definitions that are not defined in the current stage? Here, we cannot construct the `Type[T]` tree directly, so we need to get it from a recursive implicit search. For instance, to implement

```
summon[Type[List[T]]]
```

where `T` is not defined in the current stage, we construct the type constructor of `List` applied to the splice of the result of searching for a given instance for `Type[T]`:

```
Type.of[ List[ summon[Type[T]].Underlying ] ]
```

This is exactly the algorithm that Scala 2 uses to search for type tags. In fact Scala 2's type tag feature can be understood as a more ad-hoc version of `quoted.Type`. As was the case for type tags, the implicit search for a `quoted.Type` is handled by the compiler, using the algorithm sketched above.

Relationship with `inline`

Seen by itself, principled metaprogramming looks more like a framework for runtime metaprogramming than one for compile-time metaprogramming with macros. But combined with Scala 3's `inline` feature it can be turned into a compile-time system. The idea is that macro elaboration can be understood as a combination of a macro library and a quoted program. For instance, here's the `assert` macro again together with a program that calls `assert`.

```scala
object Macros:

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

  def assertImpl(expr: Expr[Boolean])(using Quotes) =
    val failMsg: Expr[String] = Expr("failed assertion: " + expr.show)
    '{ if !($expr) then throw new AssertionError($failMsg) }

@main def program =
  val x = 1
  Macros.assert(x != 0)
```

Inlining the `assert` function would give the following program:

```scala
@main def program =
  val x = 1
  ${ Macros.assertImpl('{ x != 0}) }
```

The example is only phase correct because `Macros` is a global value and as such not subject to phase consistency checking. Conceptually that's a bit unsatisfactory. If the PCP is so fundamental, it should be applicable without the global value exception. But in the example as given this does not hold since both `assert` and `program` call `assertImpl` with a splice but no quote.

However, one could argue that the example is really missing an important aspect: The macro library has to be compiled in a phase prior to the program using it, but in the code above, macro and program are defined together. A more accurate view of macros would be to have the user program be in a phase after the macro definitions, reflecting the fact that macros have to be defined and compiled before they are used. Hence, conceptually the program part should be treated by the compiler as if it was quoted:

```
@main def program = '{
  val x = 1
  ${ Macros.assertImpl('{ x != 0 }) }
}
```

If `program` is treated as a quoted expression, the call to `Macro.assertImpl` becomes phase correct even if macro library and program are conceptualized as local definitions.

But what about the call from `assert` to `assertImpl`? Here, we need a tweak of the typing rules. An inline function such as `assert` that contains a splice operation outside an enclosing quote is called a *macro*. Macros are supposed to be expanded in a subsequent phase, i.e. in a quoted context. Therefore, they are also type checked as if they were in a quoted context. For instance, the definition of `assert` is typechecked as if it appeared inside quotes. This makes the call from `assert` to `assertImpl` phase-correct, even if we assume that both definitions are local.

The `inline` modifier is used to declare a `val` that is either a constant or is a parameter that will be a constant when instantiated. This aspect is also important for macro expansion.

To get values out of expressions containing constants `Expr` provides the method `value` (or `valueOrError`). This will convert the `Expr[T]` into a `Some[T]` (or `T`) when the expression contains value. Otherwise it will return `None` (or emit an error). To avoid having incidental val bindings generated by the inlining of the `def` it is recommended to use an inline parameter. To illustrate this, consider an implementation of the `power` function that makes use of a statically known exponent:

```
inline def power(x: Double, inline n: Int) = ${ powerCode('x, 'n) }

private def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double
  n.value match
    case Some(m) => powerCode(x, m)
    case None => '{ Math.pow($x, $n.toDouble) }

private def powerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else if n == 1 then x
  else if n % 2 == 0 then '{ val y = $x * $x; ${ powerCode('y, n / 2) } }
  else '{ $x * ${ powerCode(x, n - 1) } }
```

Scope Extrusion

Quotes and splices are duals as far as the PCP is concerned. But there is an additional restriction that needs to be imposed on splices to guarantee soundness: code in splices must be free of side effects. The restriction prevents code like this:

```
var x: Expr[T] = ...
'{ (y: T) => ${ x = 'y; 1 } }
```

This code, if it was accepted, would *extrude* a reference to a quoted variable `y` from its scope. This would subsequently allow access to a variable outside the scope where it is defined, which is likely problematic. The code is clearly phase consistent, so we cannot use PCP to rule it out. Instead, we postulate a future effect system that can guarantee that splices are pure. In the absence of such a system we simply demand that spliced expressions are pure by convention, and allow for undefined compiler behavior if they are not. This is analogous to the status of pattern guards in Scala, which are also required, but not verified, to be pure.

Multi-Stage Programming introduces one additional method where you can expand code at runtime with a method `run`. There is also a problem with that invocation of `run` in splices. Consider the following expression:

```
'{ (x: Int) => ${ run('x); 1 } }
```

This is again phase correct, but will lead us into trouble. Indeed, evaluating the splice will reduce the expression `run('x)` to `x`. But then the result

```
'{ (x: Int) => ${ x; 1 } }
```

is no longer phase correct. To prevent this soundness hole it seems easiest to classify `run` as a side-effecting operation. It would thus be prevented from appearing in splices. In a base language with side effects we would have to do this anyway: Since `run` runs arbitrary code it can always produce a side effect if the code it runs produces one.

Example Expansion

Assume we have two methods, one `map` that takes an `Expr[Array[T]]` and a function `f` and one `sum` that performs a sum by delegating to `map`.

```scala
object Macros:

  def map[T](arr: Expr[Array[T]], f: Expr[T] => Expr[Unit])
            (using Type[T], Quotes): Expr[Unit] = '{
    var i: Int = 0
    while i < ($arr).length do
      val element: T = ($arr)(i)
      ${f('element)}
      i += 1
  }

  def sum(arr: Expr[Array[Int]])(using Quotes): Expr[Int] = '{
    var sum = 0
    ${ map(arr, x => '{sum += $x}) }
    sum
  }

  inline def sum_m(arr: Array[Int]): Int = ${sum('arr)}

end Macros
```

A call to `sum_m(Array(1,2,3))` will first inline `sum_m`:

```scala
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
${_root_.Macros.sum('arr)}
```

then it will splice `sum`:

```scala
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)

var sum = 0
${ map('arr, x => '{sum += $x}) }
sum
```

then it will inline `map`:

```scala
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)

var sum = 0
val f = x => '{sum += $x}
${ _root_.Macros.map('arr, 'f)(Type.of[Int])}
sum
```

then it will expand and splice inside quotes `map`:

```scala
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)

var sum = 0
val f = x => '{sum += $x}
var i: Int = 0
while i < arr.length do
  val element: Int = (arr)(i)
  sum += element
  i += 1
sum
```

Finally cleanups and dead code elimination:

```scala
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
var sum = 0
var i: Int = 0
while i < arr.length do
  val element: Int = arr(i)
  sum += element
  i += 1
sum
```

Find implicits within a macro

Similarly to the `summonFrom` construct, it is possible to make implicit search available in a quote context. For this we simply provide `scala.quoted.Expr.summon`:

```scala
import scala.collection.immutable.{ TreeSet, HashSet }
inline def setFor[T]: Set[T] = ${ setForExpr[T] }

def setForExpr[T: Type](using Quotes): Expr[Set[T]] =
  Expr.summon[Ordering[T]] match
    case Some(ord) => '{ new TreeSet[T]()($ord) }
    case _ => '{ new HashSet[T] }
```

Relationship with Transparent Inline

Inline documents inlining. The code below introduces a transparent inline method that can calculate either a value of type `Int` or a value of type `String`.

```scala
transparent inline def defaultOf(inline str: String) =
  ${ defaultOfImpl('str) }

def defaultOfImpl(strExpr: Expr[String])(using Quotes): Expr[Any] =
  strExpr.valueOrError match
    case "int" => '{1}
```

```scala
    case "string" => '{"a"}

// in a separate file
val a: Int = defaultOf("int")
val b: String = defaultOf("string")
```

Defining a macro and using it in a single project

It is possible to define macros and use them in the same project as long as the implementation of the macros does not have run-time dependencies on code in the file where it is used. It might still have compile-time dependencies on types and quoted code that refers to the use-site file.

To provide this functionality Scala 3 provides a transparent compilation mode where files that try to expand a macro but fail because the macro has not been compiled yet are suspended. If there are any suspended files when the compilation ends, the compiler will automatically restart compilation of the suspended files using the output of the previous (partial) compilation as macro classpath. In case all files are suspended due to cyclic dependencies the compilation will fail with an error.

Pattern matching on quoted expressions

It is possible to deconstruct or extract values out of `Expr` using pattern matching.

`scala.quoted` contains objects that can help extracting values from `Expr`.

- `scala.quoted.Expr` / `scala.quoted.Exprs`: matches an expression of a value (or list of values) and returns the value (or list of values).
- `scala.quoted.Const` / `scala.quoted.Consts`: Same as `Expr` / `Exprs` but only works on primitive values.
- `scala.quoted.Varargs`: matches an explicit sequence of expressions and returns them. These sequences are useful to get individual `Expr[T]` out of a varargs expression of type `Expr[Seq[T]]`.

These could be used in the following way to optimize any call to `sum` that has statically known values.

```scala
inline def sum(inline args: Int*): Int = ${ sumExpr('args) }
private def sumExpr(argsExpr: Expr[Seq[Int]])(using Quotes): Expr[Int] =
  argsExpr match
    case Varargs(args @ Exprs(argValues)) =>
      // args is of type Seq[Expr[Int]]
```

```scala
      // argValues is of type Seq[Int]
      Expr(argValues.sum) // precompute result of sum
    case Varargs(argExprs) => // argExprs is of type Seq[Expr[Int]]
      val staticSum: Int = argExprs.map(_.value.getOrElse(0)).sum
      val dynamicSum: Seq[Expr[Int]] = argExprs.filter(_.value.isEmpty)
      dynamicSum.foldLeft(Expr(staticSum))((acc, arg) => '{ $acc + $arg })
    case _ =>
      '{ $argsExpr.sum }
```

Quoted patterns

Quoted pattens allow deconstructing complex code that contains a precise structure, types or methods. Patterns `'{ ... }` can be placed in any location where Scala expects a pattern.

For example

```scala
optimize {
   sum(sum(1, a, 2), 3, b)
} // should be optimized to 6 + a + b
```

```scala
def sum(args: Int*): Int = args.sum
inline def optimize(inline arg: Int): Int = ${ optimizeExpr('arg) }
private def optimizeExpr(body: Expr[Int])(using Quotes): Expr[Int] =
  body match
    // Match a call to sum without any arguments
    case '{ sum() } => Expr(0)
    // Match a call to sum with an argument $n of type Int.
    // n will be the Expr[Int] representing the argument.
    case '{ sum($n) } => n
    // Match a call to sum and extracts all its args in an `Expr[Seq[Int]]`
    case '{ sum(${Varargs(args)}: _*) } => sumExpr(args)
    case body => body

private def sumExpr(args1: Seq[Expr[Int]])(using Quotes): Expr[Int] =
  def flatSumArgs(arg: Expr[Int]): Seq[Expr[Int]] = arg match
    case '{ sum(${Varargs(subArgs)}: _*) } => subArgs.flatMap(flatSumArgs)
    case arg => Seq(arg)
  val args2 = args1.flatMap(flatSumArgs)
  val staticSum: Int = args2.map(_.value.getOrElse(0)).sum
  val dynamicSum: Seq[Expr[Int]] = args2.filter(_.value.isEmpty)
  dynamicSum.foldLeft(Expr(staticSum))((acc, arg) => '{ $acc + $arg })
```

Recovering precise types using patterns

Sometimes it is necessary to get a more precise type for an expression. This can be achieved using the following pattern match.

```
def f(expr: Expr[Any])(using Quotes) = expr match
  case '{ $x: t } =>
    // If the pattern match succeeds, then there is
    // some type `t` such that
    // - `x` is bound to a variable of type `Expr[t]`
    // - `t` is bound to a new type `t` and a given
    // instance `Type[t]` is provided for it
    // That is, we have `x: Expr[t]` and `given Type[t]`,
    // for some (unknown) type `t`.
```

This might be used to then perform an implicit search as in:

```
extension (inline sc: StringContext)
  inline def showMe(inline args: Any*): String = ${ showMeExpr('sc, 'args) }

private def showMeExpr(sc: Expr[StringContext], argsExpr: Expr[Seq[Any]])(using
  import quotes.reflect.report
  argsExpr match
    case Varargs(argExprs) =>
      val argShowedExprs = argExprs.map {
        case '{ $arg: tp } =>
          Expr.summon[Show[tp]] match
            case Some(showExpr) =>
              '{ $showExpr.show($arg) }
            case None =>
              report.error(s"could not find implicit for ${Type.show[Show[tp]]
      }
      val newArgsExpr = Varargs(argShowedExprs)
      '{ $sc.s($newArgsExpr: _*) }
    case _ =>
      // `new StringContext(...).showMeExpr(args: _*)` not an explicit `showMeE
      report.error(s"Args must be explicit", argsExpr)
      '{???}

trait Show[-T]:
  def show(x: T): String

// in a different file
given Show[Boolean] with
  def show(b: Boolean) = "boolean!"

println(showMe"${true}")
```

Open code patterns

Quoted pattern matching also provides higher-order patterns to match open terms. If a quoted term contains a definition, then the rest of the quote can refer to this definition.

```
'{
  val x: Int = 4
  x * x
}
```

To match such a term we need to match the definition and the rest of the code, but we need to explicitly state that the rest of the code may refer to this definition.

```
case '{ val y: Int = $x; $body(y): Int } =>
```

Here `$x` will match any closed expression while `$body(y)` will match an expression that is closed under `y`. Then the subexpression of type `Expr[Int]` is bound to `body` as an `Expr[Int => Int]`. The extra argument represents the references to `y`. Usually this expression is used in combination with `Expr.betaReduce` to replace the extra argument.

```
inline def eval(inline e: Int): Int = ${ evalExpr('e) }

private def evalExpr(e: Expr[Int])(using Quotes): Expr[Int] = e match
  case '{ val y: Int = $x; $body(y): Int } =>
    // body: Expr[Int => Int] where the argument represents
    // references to y
    evalExpr(Expr.betaReduce('{$body(${evalExpr(x)})}))
  case '{ ($x: Int) * ($y: Int) } =>
    (x.value, y.value) match
      case (Some(a), Some(b)) => Expr(a * b)
      case _ => e
  case _ => e
```

```
eval { // expands to the code: (16: Int)
  val x: Int = 4
  x * x
}
```

We can also close over several bindings using `$b(a1, a2, ..., an)`. To match an actual application we can use braces on the function part `${b}(a1, a2, ..., an)`.

More details

More details