Compile-time operations

The `scala.compiletime` Package

The `scala.compiletime` package contains helper definitions that provide support for compile-time operations over values. They are described in the following.

`constValue` and `constValueOpt`

`constValue` is a function that produces the constant value represented by a type.

```
import scala.compiletime.constValue
import scala.compiletime.ops.int.S

transparent inline def toIntC[N]: Int =
  inline constValue[N] match
    case 0         => 0
    case _: S[n1] => 1 + toIntC[n1]

inline val ctwo = toIntC[2]
```

`constValueOpt` is the same as `constValue`, however returning an `Option[T]` enabling us to handle situations where a value is not present. Note that `S` is the type of the successor of some singleton type. For example the type `S[1]` is the singleton type `2`.

`erasedValue`

So far we have seen inline methods that take terms (tuples and integers) as parameters. What if we want to base case distinctions on types instead? For instance, one would like to be able to write a function `defaultValue`, that, given a type `T`, returns optionally the default value of `T`, if it exists. We can already express this

using rewrite match expressions and a simple helper function, `scala.compiletime.erasedValue`, which is defined as follows:

```
def erasedValue[T]: T
```

The `erasedValue` function *pretends* to return a value of its type argument `T`. Calling this function will always result in a compile-time error unless the call is removed from the code while inlining.

Using `erasedValue`, we can then define `defaultValue` as follows:

```
import scala.compiletime.erasedValue

transparent inline def defaultValue[T] =
  inline erasedValue[T] match
    case _: Byte    => Some(0: Byte)
    case _: Char    => Some(0: Char)
    case _: Short   => Some(0: Short)
    case _: Int     => Some(0)
    case _: Long    => Some(0L)
    case _: Float   => Some(0.0f)
    case _: Double  => Some(0.0d)
    case _: Boolean => Some(false)
    case _: Unit    => Some(())
    case _          => None
```

Then:

```
val dInt: Some[Int] = defaultValue[Int]
val dDouble: Some[Double] = defaultValue[Double]
val dBoolean: Some[Boolean] = defaultValue[Boolean]
val dAny: None.type = defaultValue[Any]
```

As another example, consider the type-level version of `toInt` below: given a *type* representing a Peano number, return the integer *value* corresponding to it. Consider the definitions of numbers as in the *Inline Match* section above. Here is how `toIntT` can be defined:

```
transparent inline def toIntT[N <: Nat]: Int =
  inline scala.compiletime.erasedValue[N] match
    case _: Zero.type => 0
    case _: Succ[n] => toIntT[n] + 1

inline val two = toIntT[Succ[Succ[Zero.type]]]
```

`erasedValue` is an `erased` method so it cannot be used and has no runtime behavior. Since `toIntT` performs static checks over the static type of `N` we can safely use it to scrutinize its return type (`S[S[Z]]` in this case).

`error`

The `error` method is used to produce user-defined compile errors during inline expansion. It has the following signature:

```
inline def error(inline msg: String): Nothing
```

If an inline expansion results in a call `error(msgStr)` the compiler produces an error message containing the given `msgStr` .

```
import scala.compiletime.{error, code}

inline def fail() =
  error("failed for a reason")

fail() // error: failed for a reason
```

or

```
inline def fail(p1: => Any) =
  error(code"failed on: $p1")

fail(identity("foo")) // error: failed on: identity("foo")
```

The `scala.compiletime.ops` package

The `scala.compiletime.ops` package contains types that provide support for primitive operations on singleton types. For example, `scala.compiletime.ops.int.*` provides support for multiplying two singleton `Int` types, and `scala.compiletime.ops.boolean.&&` for the conjunction of two `Boolean` types. When all arguments to a type in `scala.compiletime.ops` are singleton types, the compiler can evaluate the result of the operation.

```
import scala.compiletime.ops.int.*
import scala.compiletime.ops.boolean.*

val conjunction: true && true = true
val multiplication: 3 * 5 = 15
```

Many of these singleton operation types are meant to be used infix (as in SLS §3.2.10).

Since type aliases have the same precedence rules as their term-level equivalents, the operations compose with the expected precedence rules:

```
import scala.compiletime.ops.int.*
val x: 1 + 2 * 3 = 7
```

The operation types are located in packages named after the type of the left-hand side parameter: for instance, `scala.compiletime.ops.int.+` represents addition of two numbers, while `scala.compiletime.ops.string.+` represents string concatenation. To use both and distinguish the two types from each other, a match type can dispatch to the correct implementation:

```
import scala.compiletime.ops.*

import scala.annotation.infix

type +[X <: Int | String, Y <: Int | String] = (X, Y) match
  case (Int, Int) => int.+[X, Y]
  case (String, String) => string.+[X, Y]

val concat: "a" + "b" = "ab"
val addition: 1 + 1 = 2
```

Summoning Implicits Selectively

It is foreseen that many areas of typelevel programming can be done with rewrite methods instead of implicits. But sometimes implicits are unavoidable. The problem so far was that the Prolog-like programming style of implicit search becomes viral: Once some construct depends on implicit search it has to be written as a logic program itself. Consider for instance the problem of creating a `TreeSet[T]` or a `HashSet[T]` depending on whether `T` has an `Ordering` or not. We can create a set of implicit definitions like this:

```
trait SetFor[T, S <: Set[T]]

class LowPriority:
  implicit def hashSetFor[T]: SetFor[T, HashSet[T]] = ...

object SetsFor extends LowPriority:
  implicit def treeSetFor[T: Ordering]: SetFor[T, TreeSet[T]] = ...
```

Clearly, this is not pretty. Besides all the usual indirection of implicit search, we face the problem of rule prioritization where we have to ensure that `treeSetFor` takes priority over `hashSetFor` if the element type has an ordering. This is solved (clumsily) by putting `hashSetFor` in a superclass `LowPriority` of the object `SetsFor` where `treeSetFor` is defined. Maybe the boilerplate would still be acceptable if the crufty code could be contained. However, this is not the case. Every user of the abstraction has to be parameterized itself with a `SetFor` implicit. Considering the simple task *"I want a `TreeSet[T]` if `T` has an ordering and a `HashSet[T]` otherwise"*, this seems like a lot of ceremony.

There are some proposals to improve the situation in specific areas, for instance by allowing more elaborate schemes to specify priorities. But they all keep the viral nature of implicit search programs based on logic programming.

By contrast, the new `summonFrom` construct makes implicit search available in a functional context. To solve the problem of creating the right set, one would use it as follows:

```
import scala.compiletime.summonFrom

inline def setFor[T]: Set[T] = summonFrom {
  case ord: Ordering[T] => new TreeSet[T]()(using ord)
  case _                => new HashSet[T]
}
```

A `summonFrom` call takes a pattern matching closure as argument. All patterns in the closure are type ascriptions of the form `identifier : Type`.

Patterns are tried in sequence. The first case with a pattern `x: T` such that an implicit value of type `T` can be summoned is chosen.

Alternatively, one can also use a pattern-bound given instance, which avoids the explicit using clause. For instance, `setFor` could also be formulated as follows:

```
import scala.compiletime.summonFrom

inline def setFor[T]: Set[T] = summonFrom {
  case given Ordering[T] => new TreeSet[T]
  case _                 => new HashSet[T]
}
```

`summonFrom` applications must be reduced at compile time.

Consequently, if we summon an `Ordering[String]` the code above will return a new instance of `TreeSet[String]`.

```
summon[Ordering[String]]

println(setFor[String].getClass) // prints class scala.collection.immutable.Tre
```

Note `summonFrom` applications can raise ambiguity errors. Consider the following code with two givens in scope of type `A`. The pattern match in `f` will raise an ambiguity error of `f` is applied.

```
class A
given a1: A = new A
given a2: A = new A

inline def f: Any = summonFrom {
  case given _: A => ???  // error: ambiguous givens
}
```

summonInline

The shorthand `summonInline` provides a simple way to write a `summon` that is delayed until the call is inlined. Unlike `summonFrom`, `summonInline` also yields the implicit-not-found error, if a given instance of the summoned type is not found.

```
import scala.compiletime.summonInline
import scala.annotation.implicitNotFound

@implicitNotFound("Missing One")
trait Missing1

@implicitNotFound("Missing Two")
trait Missing2

trait NotMissing
given NotMissing = ???

transparent inline def summonInlineCheck[T <: Int](inline t : T) : Any =
  inline t match
    case 1 => summonInline[Missing1]
    case 2 => summonInline[Missing2]
    case _ => summonInline[NotMissing]

val missing1 = summonInlineCheck(1) // error: Missing One
val missing2 = summonInlineCheck(2) // error: Missing Two
val notMissing : NotMissing = summonInlineCheck(3)
```

Reference

For more information about compile-time operations, see PR #4768, which explains how `summonFrom`'s predecessor (implicit matches) can be used for typelevel programming and code specialization and PR #7201 which explains the new `summonFrom` syntax.

‹ Inline

Macros ›