Scala 3 Reference / Experimental / Erased Definitions

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

Erased Definitions

Edit this page on GitHub

`erased` is a modifier that expresses that some definition or expression is erased by the compiler instead of being represented in the compiled output. It is not yet part of the Scala language standard. To enable `erased`, turn on the language feature `experimental.erasedDefinitions`. This can be done with a language import

```
import scala.language.experimental.erasedDefinitions
```

or by setting the command line option `-language:experimental.erasedDefinitions`. Erased definitions must be in an experimental scope (see Experimental definitions).

Why erased terms?

Let's describe the motivation behind erased terms with an example. In the following we show a simple state machine which can be in a state `On` or `Off`. The machine can change state from `Off` to `On` with `turnedOn` only if it is currently `Off`. This last constraint is captured with the `IsOff[S]` contextual evidence which only exists for `IsOff[Off]`. For example, not allowing calling `turnedOn` on in an `On` state as we would require an evidence of type `IsOff[On]` that will not be found.

```
sealed trait State
final class On extends State
final class Off extends State

@implicitNotFound("State must be Off")
class IsOff[S <: State]
object IsOff:
  given isOff: IsOff[Off] = new IsOff[Off]

class Machine[S <: State]:
  def turnedOn(using IsOff[S]): Machine[On] = new Machine[On]

val m = new Machine[Off]
```

```
m.turnedOn
m.turnedOn.turnedOn // ERROR

// ^
// State must be Off
```

Note that in the code above the actual context arguments for `IsOff` are never used at runtime; they serve only to establish the right constraints at compile time. As these terms are never used at runtime there is not real need to have them around, but they still need to be present in some form in the generated code to be able to do separate compilation and retain binary compatibility. We introduce *erased terms* to overcome this limitation: we are able to enforce the right constrains on terms at compile time. These terms have no run time semantics and they are completely erased.

How to define erased terms?

Parameters of methods and functions can be declared as erased, placing `erased` in front of a parameter list (like `given`).

```
def methodWithErasedEv(erased ev: Ev): Int = 42

val lambdaWithErasedEv: erased Ev => Int =
  (erased ev: Ev) => 42
```

`erased` parameters will not be usable for computations, though they can be used as arguments to other `erased` parameters.

```
def methodWithErasedInt1(erased i: Int): Int =
  i + 42 // ERROR: can not use i

def methodWithErasedInt2(erased i: Int): Int =
  methodWithErasedInt1(i) // OK
```

Not only parameters can be marked as erased, `val` and `def` can also be marked with `erased`. These will also only be usable as arguments to `erased` parameters.

```
erased val erasedEvidence: Ev = ...
methodWithErasedEv(erasedEvidence)
```

What happens with erased values at runtime?

As `erased` are guaranteed not to be used in computations, they can and will be

As `erased` are guaranteed not to be used in computations, they can and will be erased.

```
// becomes def methodWithErasedEv(): Int at runtime
def methodWithErasedEv(erased ev: Ev): Int = ...

def evidence1: Ev = ...
erased def erasedEvidence2: Ev = ... // does not exist at runtime
erased val erasedEvidence3: Ev = ... // does not exist at runtime

// evidence1 is not evaluated and no value is passed to methodWithErasedEv
methodWithErasedEv(evidence1)
```

State machine with erased evidence example

The following example is an extended implementation of a simple state machine which can be in a state `On` or `Off`. The machine can change state from `Off` to `On` with `turnedOn` only if it is currently `Off`, conversely from `On` to `Off` with `turnedOff` only if it is currently `On`. These last constraint are captured with the `IsOff[S]` and `IsOn[S]` given evidence only exist for `IsOff[Off]` and `IsOn[On]`. For example, not allowing calling `turnedOff` on in an `Off` state as we would require an evidence `IsOn[Off]` that will not be found.

As the given evidences of `turnedOn` and `turnedOff` are not used in the bodies of those functions we can mark them as `erased`. This will remove the evidence parameters at runtime, but we would still evaluate the `isOn` and `isOff` givens that were found as arguments. As `isOn` and `isOff` are not used except as `erased` arguments, we can mark them as `erased`, hence removing the evaluation of the `isOn` and `isOff` evidences.

```
import scala.annotation.implicitNotFound

sealed trait State
final class On extends State
final class Off extends State

@implicitNotFound("State must be Off")
class IsOff[S <: State]
object IsOff:
  // will not be called at runtime for turnedOn, the
  // compiler will only require that this evidence exists
  given IsOff[Off] = new IsOff[Off]
```

```
@implicitNotFound("State must be On")
class IsOn[S <: State]
object IsOn:

  // will not exist at runtime, the compiler will only
  // require that this evidence exists at compile time
  erased given IsOn[On] = new IsOn[On]

class Machine[S <: State] private ():
  // ev will disappear from both functions
  def turnedOn(using erased ev: IsOff[S]): Machine[On] = new Machine[On]
  def turnedOff(using erased ev: IsOn[S]): Machine[Off] = new Machine[Off]

object Machine:
  def newMachine(): Machine[Off] = new Machine[Off]

@main def test =
  val m = Machine.newMachine()
  m.turnedOn
  m.turnedOn.turnedOff

  // m.turnedOff
  // ^
  // State must be On

  // m.turnedOn.turnedOn
  // ^
  // State must be Off
```

Note that in Inline we discussed `erasedValue` and inline matches. `erasedValue` is implemented with `erased`, so the state machine above can be encoded as follows:

```
import scala.compiletime.*

sealed trait State
final class On extends State
final class Off extends State

class Machine[S <: State]:
  transparent inline def turnOn(): Machine[On] =
    inline erasedValue[S] match
      case _: Off => new Machine[On]
      case _: On  => error("Turning on an already turned on machine")

  transparent inline def turnOff(): Machine[Off] =
    inline erasedValue[S] match
      case _: On  => new Machine[Off]
      case _: Off => error("Turning off an already turned off machine")

object Machine:
```

```
object Machine:
  def newMachine(): Machine[Off] =
    println("newMachine")

    new Machine[Off]
end Machine

@main def test =
  val m = Machine.newMachine()
  m.turnOn()
  m.turnOn().turnOff()
  m.turnOn().turnOn() // error: Turning on an already turned on machine
```

Erased Classes

`erased` can also be used as a modifier for a class. An erased class is intended to be used only in erased definitions. If the type of a val definition or parameter is a (possibly aliased, refined, or instantiated) erased class, the definition is assumed to be `erased` itself. Likewise, a method with an erased class return type is assumed to be `erased` itself. Since given instances expand to vals and defs, they are also assumed to be erased if the type they produce is an erased class. Finally function types with erased classes as arguments turn into erased function types.

Example:

```
erased class CanRead

val x: CanRead = ...        // `x` is turned into an erased val
val y: CanRead => Int = ... // the function is turned into an erased function
def f(x: CanRead) = ...     // `f` takes an erased parameter
def g(): CanRead = ...      // `g` is turned into an erased def
given CanRead = ...         // the anonymous given is assumed to be erased
```

The code above expands to

```
erased class CanRead

erased val x: CanRead = ...
val y: (erased CanRead) => Int = ...
def f(erased x: CanRead) = ...
erased def g(): CanRead = ...
erased given CanRead = ...
```

After erasure, it is checked that no references to values of erased classes remain and that no instances of erased classes are created. So the following would be an error:

```
val err: Any = CanRead() // error: illegal reference to erased class CanRead
```

Here, the type of `err` is `Any`, so `err` is not considered erased. Yet its initializing value is a reference to the erased class `CanRead`.

More Details

‹ CanTh... Erased ... ›

Scaladoc

Copyright (c) 2002-2022, LAMP/EPFL