Scala 3 Reference / Other New Features / TypeTest

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

# TypeTest

Edit this page on GitHub

## TypeTest

When pattern matching there are two situations where a runtime type test must be performed. The first case is an explicit type test using the ascription pattern notation.

```
(x: X) match
  case y: Y =>
```

The second case is when an extractor takes an argument that is not a subtype of the scrutinee type.

```
(x: X) match
  case y @ Y(n) =>

object Y:
  def unapply(x: Y): Some[Int] = ...
```

In both cases, a class test will be performed at runtime. But when the type test is on an abstract type (type parameter or type member), the test cannot be performed because the type is erased at runtime.

A `TypeTest` can be provided to make this test possible.

```
package scala.reflect

trait TypeTest[-S, T]:
  def unapply(s: S): Option[s.type & T]
```

It provides an extractor that returns its argument typed as a `T` if the argument is a `T`. It can be used to encode a type test.

```
def f[X, Y](x: X)(using tt: TypeTest[X, Y]): Option[Y] = x match
def f[X, Y](x: X)(using tt: TypeTest[X, Y]): Option[Y] = x match
  case tt(x @ Y(1)) => Some(x)


  case tt(x) => Some(x)
  case _ => None
```

To avoid the syntactic overhead the compiler will look for a type test automatically if it detects that the type test is on abstract types. This means that `x: Y` is transformed to `tt(x)` and `x @ Y(_)` to `tt(x @ Y(_))` if there is a contextual `TypeTest[X, Y]` in scope. The previous code is equivalent to

```
def f[X, Y](x: X)(using TypeTest[X, Y]): Option[Y] = x match
  case x @ Y(1) => Some(x)
  case x: Y => Some(x)
  case _ => None
```

We could create a type test at call site where the type test can be performed with runtime class tests directly as follows

```
val tt: TypeTest[Any, String] =
  new TypeTest[Any, String]:
    def unapply(s: Any): Option[s.type & String] = s match
      case s: String => Some(s)
      case _ => None

f[AnyRef, String]("acb")(using tt)
```

The compiler will synthesize a new instance of a type test if none is found in scope as:

```
new TypeTest[A, B]:
  def unapply(s: A): Option[s.type & B] = s match
    case s: B => Some(s)
    case _ => None
```

If the type tests cannot be done there will be an unchecked warning that will be raised on the `case s: B =>` test.

The most common `TypeTest` instances are the ones that take any parameters (i.e. `TypeTest[Any, T]`). To make it possible to use such instances directly in context bounds we provide the alias

```
package scala.reflect

type Typeable[T] = TypeTest[Any, T]
```

This alias can be used as

```
def f[T: Typeable]: Boolean =
  "abc" match
    case x: T => true
    case _ => false

f[String] // true
f[Int] // false
```

# TypeTest and ClassTag

`TypeTest` is a replacement for functionality provided previously by `ClassTag.unapply`. Using `ClassTag` instances was unsound since classtags can check only the class component of a type. `TypeTest` fixes that unsoundness. `ClassTag` type tests are still supported but a warning will be emitted after 3.0.

# Example

Given the following abstract definition of Peano numbers that provides two given instances of types `TypeTest[Nat, Zero]` and `TypeTest[Nat, Succ]`

```
import scala.reflect.*

trait Peano:
  type Nat
  type Zero <: Nat
  type Succ <: Nat

  def safeDiv(m: Nat, n: Succ): (Nat, Nat)

  val Zero: Zero

  val Succ: SuccExtractor
  trait SuccExtractor:
    def apply(nat: Nat): Succ
    def unapply(succ: Succ): Some[Nat]

  given typeTestOfZero: TypeTest[Nat, Zero]
  given typeTestOfSucc: TypeTest[Nat, Succ]
```

together with an implementation of Peano numbers based on type `Int`

```
object PeanoInt extends Peano:
  type Nat  = Int
  type Zero = Int
  type Succ = Int

  def safeDiv(m: Nat, n: Succ): (Nat, Nat) = (m / n, m % n)

  val Zero: Zero = 0

  val Succ: SuccExtractor = new:
    def apply(nat: Nat): Succ = nat + 1
    def unapply(succ: Succ) = Some(succ - 1)

  def typeTestOfZero: TypeTest[Nat, Zero] = new:
    def unapply(x: Nat): Option[x.type & Zero] =
      if x == 0 then Some(x) else None

  def typeTestOfSucc: TypeTest[Nat, Succ] = new:
    def unapply(x: Nat): Option[x.type & Succ] =
      if x > 0 then Some(x) else None
```

it is possible to write the following program

```
@main def test =
  import PeanoInt.*

  def divOpt(m: Nat, n: Nat): Option[(Nat, Nat)] =
    n match
      case Zero => None
      case s @ Succ(_) => Some(safeDiv(m, s))

  val two = Succ(Succ(Zero))
  val five = Succ(Succ(Succ(two)))

  println(divOpt(five, two))  // prints "Some((2,1))"
  println(divOpt(two, five))  // prints "Some((0,2))"
  println(divOpt(two, Zero))  // prints "None"
```

Note that without the `TypeTest[Nat, Succ]` the pattern `Succ.unapply(nat: Succ)` would be unchecked.

‹ Safe Ini... Experi... ›

Copyright (c) 2002-2022, LAMP/EPFL

Scaladoc