Scala 3 Reference / Contextual Abstractions / Multiversal Equality

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

Multiversal Equality

Edit this page on GitHub

Previously, Scala had universal equality: Two values of any types could be compared with each other with `==` and `!=`. This came from the fact that `==` and `!=` are implemented in terms of Java's `equals` method, which can also compare values of any two reference types.

Universal equality is convenient. But it is also dangerous since it undermines type safety. For instance, let's assume one is left after some refactoring with an erroneous program where a value `y` has type `S` instead of the correct type `T`.

```
val x = ... // of type T
val y = ... // of type S, but should be T
x == y      // typechecks, will always yield false
```

If `y` gets compared to other values of type `T`, the program will still typecheck, since values of all types can be compared with each other. But it will probably give unexpected results and fail at runtime.

Multiversal equality is an opt-in way to make universal equality safer. It uses a binary type class `scala.CanEqual` to indicate that values of two given types can be compared with each other. The example above would not typecheck if `S` or `T` was a class that derives `CanEqual`, e.g.

```
class T derives CanEqual
```

Alternatively, one can also provide a `CanEqual` given instance directly, like this:

```
given CanEqual[T, T] = CanEqual.derived
```

This definition effectively says that values of type `T` can (only) be compared to other values of type `T` when using `==` or `!=`. The definition affects type checking but it has no significance for runtime behavior, since `==` always maps to `equals` and `!=` always maps to the negation of `equals`. The right-hand side `CanEqual.derived` of the definition is a value that has any `CanEqual` instance as its type. Here is the definition of class `CanEqual` and its companion object:

```
package scala
import annotation.implicitNotFound

@implicitNotFound("Values of types ${L} and ${R} cannot be compared with == or
sealed trait CanEqual[-L, -R]

object CanEqual:
  object derived extends CanEqual[Any, Any]
```

One can have several `CanEqual` given instances for a type. For example, the four definitions below make values of type `A` and type `B` comparable with each other, but not comparable to anything else:

```
given CanEqual[A, A] = CanEqual.derived
given CanEqual[B, B] = CanEqual.derived
given CanEqual[A, B] = CanEqual.derived
given CanEqual[B, A] = CanEqual.derived
```

The `scala.CanEqual` object defines a number of `CanEqual` given instances that together define a rule book for what standard types can be compared (more details below).

There is also a "fallback" instance named `canEqualAny` that allows comparisons over all types that do not themselves have a `CanEqual` given. `canEqualAny` is defined as follows:

```
def canEqualAny[L, R]: CanEqual[L, R] = CanEqual.derived
```

Even though `canEqualAny` is not declared as `given`, the compiler will still construct an `canEqualAny` instance as answer to an implicit search for the type `CanEqual[L, R]`, unless `L` or `R` have `CanEqual` instances defined on them, or the language feature `strictEquality` is enabled.

The primary motivation for having `canEqualAny` is backwards compatibility. If this is of no concern, one can disable `canEqualAny` by enabling the language feature `strictEquality`. As for all language features this can be either done with an import

```
import scala.language.strictEquality
```

or with a command line option `-language:strictEquality`.

Deriving CanEqual Instances

Instead of defining `CanEqual` instances directly, it is often more convenient to derive them. Example:

```
class Box[T](x: T) derives CanEqual
```

By the usual rules of type class derivation, this generates the following `CanEqual` instance in the companion object of `Box`:

```
given [T, U](using CanEqual[T, U]): CanEqual[Box[T], Box[U]] =
  CanEqual.derived
```

That is, two boxes are comparable with `==` or `!=` if their elements are. Examples:

```
new Box(1) == new Box(1L)   // ok since there is an instance for
`CanEqual[Int, Long]`
new Box(1) == new Box("a")  // error: can't compare
new Box(1) == 1             // error: can't compare
```

Precise Rules for Equality Checking

The precise rules for equality checking are as follows.

If the `strictEquality` feature is enabled then a comparison using `x == y` or `x != y` between values `x: T` and `y: U` is legal if there is a `given` of type `CanEqual[T, U]`.

In the default case where the `strictEquality` feature is not enabled the comparison is also legal if

1. `T` and `U` are the same, or
2. one of `T`, `U` is a subtype of the *lifted* version of the other type, or
3. neither `T` nor `U` have a *reflexive* `CanEqual` instance.

Explanations:

- *lifting* a type `S` means replacing all references to abstract types in covariant positions of `S` by their upper bound, and replacing all refinement types in covariant positions of `S` by their parent.

- a type `T` has a *reflexive* `CanEqual` instance if the implicit search for `CanEqual[T, T]` succeeds.

Predefined CanEqual Instances

The `CanEqual` object defines instances for comparing

- the primitive types `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`,
- `java.lang.Number`, `java.lang.Boolean`, and `java.lang.Character`,
- `scala.collection.Seq`, and `scala.collection.Set`.

Instances are defined so that every one of these types has a *reflexive* `CanEqual` instance, and the following holds:

- Primitive numeric types can be compared with each other.
- Primitive numeric types can be compared with subtypes of `java.lang.Number` (and *vice versa*).
- `Boolean` can be compared with `java.lang.Boolean` (and *vice versa*).
- `Char` can be compared with `java.lang.Character` (and *vice versa*).
- Two sequences (of arbitrary subtypes of `scala.collection.Seq`) can be compared with each other if their element types can be compared. The two sequence types need not be the same.
- Two sets (of arbitrary subtypes of `scala.collection.Set`) can be compared with each other if their element types can be compared. The two set types need not be the same.
- Any subtype of `AnyRef` can be compared with `Null` (and *vice versa*).

Why Two Type Parameters?

One particular feature of the `CanEqual` type is that it takes *two* type parameters, representing the types of the two items to be compared. By contrast, conventional implementations of an equality type class take only a single type parameter which represents the common type of *both* operands. One type parameter is simpler than two, so why go through the additional complication? The reason has to do with the fact that, rather than coming up with a type class where no operation existed before, we are dealing with a refinement of pre-existing, universal equality. It is best illustrated through an example.

Say you want to come up with a safe version of the `contains` method on `List[T]`. The original definition of `contains` in the standard library was:

```
class List[+T]:
  ...
  def contains(x: Any): Boolean
```

That uses universal equality in an unsafe way since it permits arguments of any type to be compared with the list's elements. The "obvious" alternative definition

```
def contains(x: T): Boolean
```

does not work, since it refers to the covariant parameter `T` in a nonvariant context. The only variance-correct way to use the type parameter `T` in `contains` is as a lower bound:

```
def contains[U >: T](x: U): Boolean
```

This generic version of `contains` is the one used in the current (Scala 2.13) version of `List`. It looks different but it admits exactly the same applications as the `contains(x: Any)` definition we started with. However, we can make it more useful (i.e. restrictive) by adding a `CanEqual` parameter:

```
def contains[U >: T](x: U)(using CanEqual[T, U]): Boolean // (1)
```

This version of `contains` is equality-safe! More precisely, given `x: T`, `xs: List[T]` and `y: U`, then `xs.contains(y)` is type-correct if and only if `x == y` is type-correct.

Unfortunately, the crucial ability to "lift" equality type checking from simple equality and pattern matching to arbitrary user-defined operations gets lost if we restrict ourselves to an equality class with a single type parameter. Consider the following signature of `contains` with a hypothetical `CanEqual1[T]` type class:

```
def contains[U >: T](x: U)(using CanEqual1[U]): Boolean   // (2)
```

This version could be applied just as widely as the original `contains(x: Any)` method, since the `CanEqual1[Any]` fallback is always available! So we have gained nothing. What got lost in the transition to a single parameter type class was the original rule that `CanEqual[A, B]` is available only if neither `A` nor `B` have a reflexive `CanEqual` instance. That rule simply cannot be expressed if there is a single type parameter for `CanEqual`.

The situation is different under `-language:strictEquality`. In that case, the `CanEqual[Any, Any]` or `CanEqual1[Any]` instances would never be available, and the single and two-parameter versions would indeed coincide for most practical purposes.

But assuming `-language:strictEquality` immediately and everywhere poses migration problems which might well be unsurmountable. Consider again `contains`, which is in the standard library. Parameterizing it with the `CanEqual` type class as in (1) is an immediate win since it rules out non-sensical applications while still allowing all sensible ones. So it can be done almost at any time, modulo binary compatibility concerns. On the other hand, parameterizing `contains` with `CanEqual1` as in (2) would make `contains` unusable for all types that have not yet declared a `CanEqual1` instance, including all types coming from Java. This is clearly unacceptable. It would lead to a situation where, rather than migrating existing libraries to use safe equality, the only upgrade path is to have parallel libraries, with the new version only catering to types deriving `CanEqual1` and the old version dealing with everything else. Such a split of the ecosystem would be very problematic, which means the cure is likely to be worse than the disease.

For these reasons, it looks like a two-parameter type class is the only way forward because it can take the existing ecosystem where it is and migrate it towards a future where more and more code uses safe equality.

In applications where `-language:strictEquality` is the default one could also introduce a one-parameter type alias such as

```
type Eq[-T] = CanEqual[T, T]
```

Operations needing safe equality could then use this alias instead of the two-parameter `CanEqual` class. But it would only work under `-language:strictEquality`, since otherwise the universal `Eq[Any]` instance would be available everywhere.

More on multiversal equality is found in a blog post and a GitHub issue.

‹ How to... Contex... ›

Contributors to this page

pikinier20 anatoliykmetyuk BarkingBad julienrf odersky michelou asakaev robstoll dvirf1 solnaranu

Scaladoc