Changes in Implicit Resolution

Edit this page on GitHub

This section describes changes to the implicit resolution that apply both to the new `given`s and to the old-style `implicit`s in Scala 3. Implicit resolution uses a new algorithm which caches implicit results more aggressively for performance. There are also some changes that affect implicits on the language level.

1. Types of implicit values and result types of implicit methods must be explicitly declared. Excepted are only values in local blocks where the type may still be inferred:

```
class C {

    val ctx: Context = ...        // ok

    /*!*/ implicit val x = ...    // error: type must be given explicitly

    /*!*/ implicit def y = ...    // error: type must be given explicitly
  }
  val y = {
    implicit val ctx = this.ctx // ok
    ...
  }
```

2. Nesting is now taken into account for selecting an implicit. Consider for instance the following scenario:

```
def f(implicit i: C) = {
    def g(implicit j: C) = {
      implicitly[C]
    }
  }
```

This will now resolve the `implicitly` call to `j`, because `j` is nested more deeply

than `i`. Previously, this would have resulted in an ambiguity error. The previous possibility of an implicit search failure due to *shadowing* (where an implicit is hidden by a nested definition) no longer applies.
3. Package prefixes no longer contribute to the implicit search scope of a type.
Example:

```
package p

  given a: A = A()

  object o:
    given b: B = B()
    type C
```

Both `a` and `b` are visible as implicits at the point of the definition of `type C`. However, a reference to `p.o.C` outside of package `p` will have only `b` in its implicit search scope but not `a`.

In more detail, here are the rules for what constitutes the implicit scope of a type:

Definition: A reference is an *anchor* if it refers to an object, a class, a trait, an abstract type, an opaque type alias, or a match type alias. References to packages and package objects are anchors only under `-source:3.0-migration`. Opaque type aliases count as anchors only outside the scope where their alias is visible.

Definition: The *anchors* of a type *T* is a set of references defined as follows:

1. If *T* is a reference to an anchor, *T* itself plus, if *T* is of the form *P#A*, the anchors of *P*.
2. If *T* is an alias of *U*, the anchors of *U*.
3. If *T* is a reference to a type parameter, the union of the anchors of both of its bounds.
4. If *T* is a singleton reference, the anchors of its underlying type, plus, if *T* is of the form *(P#x).type*, the anchors of *P*.
5. If *T* is the this-type *o.this* of a static object *o*, the anchors of a term reference *o.type* to that object.
6. If *T* is some other type, the union of the anchors of each constituent type of *T*.

Definition: The *implicit scope* of a type *T* is the smallest set *S* of term references such that

1. If *T* is a reference to a class, *S* includes a reference to the companion object of the class, if it exists, as well as the implicit scopes of all of *T*'s parent classes.

2. If *T* is a reference to an object, *S* includes *T* itself as well as the implicit scopes of all of *T*'s parent classes.

3. If *T* is a reference to an opaque type alias named *A*, *S* includes a reference to an object *A* defined in the same scope as the type, if it exists, as well as the implicit scope of *T*'s underlying type or bounds.
4. If *T* is a reference to an abstract type or match type alias named *A*, *S* includes a reference to an object *A* defined in the same scope as the type, if it exists, as well as the implicit scopes of *T*'s given bounds.
5. If *T* is a reference to an anchor of the form *p.A* then *S* also includes all term references on the path *p*.
6. If *T* is some other type, *S* includes the implicit scopes of all anchors of *T*.

4. The treatment of ambiguity errors has changed. If an ambiguity is encountered in some recursive step of an implicit search, the ambiguity is propagated to the caller.

Example: Say you have the following definitions:

```
class A
  class B extends C
  class C
  implicit def a1: A
  implicit def a2: A
  implicit def b(implicit a: A): B
  implicit def c: C
```

and the query `implicitly[C]`.

This query would now be classified as ambiguous. This makes sense, after all there are two possible solutions, `b(a1)` and `b(a2)`, neither of which is better than the other and both of which are better than the third solution, `c`. By contrast, Scala 2 would have rejected the search for `A` as ambiguous, and subsequently have classified the query `b(implicitly[A])` as a normal fail, which means that the alternative `c` would be chosen as solution!

Scala 2's somewhat puzzling behavior with respect to ambiguity has been exploited to implement the analogue of a "negated" search in implicit resolution, where a query `Q1` fails if some other query `Q2` succeeds and `Q1` succeeds if `Q2` fails. With the new cleaned up behavior these techniques no longer work. But there is now a new special type `scala.util.NotGiven` which implements negation directly. For any query type `Q`, `NotGiven[Q]` succeeds if and only if the implicit search for `Q` fails.

5. The treatment of divergence errors has also changed. A divergent implicit is

5. The treatment of divergence errors has also changed. A divergent implicit is treated as a normal failure, after which alternatives are still tried. This also makes sense: Encountering a divergent implicit means that we assume that no finite solution can be found on the corresponding path, but another path can still be tried. By contrast, most (but not all) divergence errors in Scala 2 would terminate the implicit search as a whole.

6. Scala 2 gives a lower level of priority to implicit conversions with call-by-name parameters relative to implicit conversions with call-by-value parameters. Scala 3 drops this distinction. So the following code snippet would be ambiguous in Scala 3:

```
implicit def conv1(x: Int): A = new A(x)
  implicit def conv2(x: => Int): A = new A(x)
  def buzz(y: A) = ???
  buzz(1)   // error: ambiguous
```

7. The rule for picking a *most specific* alternative among a set of overloaded or implicit alternatives is refined to take context parameters into account. All else being equal, an alternative that takes some context parameters is taken to be less specific than an alternative that takes none. If both alternatives take context parameters, we try to choose between them as if they were methods with regular parameters. The following paragraph in the SLS §6.26.3 is affected by this change:

Original version:

> An alternative A is *more specific* than an alternative B if the relative weight of A over B is greater than the relative weight of B over A.

Modified version:

An alternative A is *more specific* than an alternative B if

- the relative weight of A over B is greater than the relative weight of B over A, or
- the relative weights are the same, and A takes no implicit parameters but B does, or
- the relative weights are the same, both A and B take implicit parameters, and A is more specific than B if all implicit parameters in either alternative are replaced by regular parameters.

8. The previous disambiguation of implicits based on inheritance depth is refined to make it transitive. Transitivity is important to guarantee that search outcomes are compilation-order independent. Here's a scenario where the previous rules violated transitivity:

```
class A extends B
  object A { given a ... }
  class B
  object B extends C { given b ... }
  class C { given c }
```

Here `a` is more specific than `b` since the companion class `A` is a subclass of the companion class `B`. Also, `b` is more specific than `c` since `object B` extends class `C`. But `a` is not more specific than `c`. This means if `a, b, c` are all applicable implicits, it makes a difference in what order they are compared. If we compare `b` and `c` first, we keep `b` and drop `c`. Then, comparing `a` with `b` we keep `a`. But if we compare `a` with `c` first, we fail with an ambiguity error.

The new rules are as follows: An implicit `a` defined in `A` is more specific than an implicit `b` defined in `B` if

- `A` extends `B`, or
- `A` is an object and the companion class of `A` extends `B`, or
- `A` and `B` are objects, `B` does not inherit any implicit members from base classes (*), and the companion class of `A` extends the companion class of `B`.

Condition (*) is new. It is necessary to ensure that the defined relation is transitive.

[//]: # todo: expand with precise rules

‹ Chang... Implici... ›

Scaladoc Copyright (c) 2002-2022, LAMP/EPFL