Changes in Overload Resolution

Edit this page on GitHub

Overload resolution in Scala 3 improves on Scala 2 in three ways. First, it takes all argument lists into account instead of just the first argument list. Second, it can infer parameter types of function values even if they are in the first argument list. Third, default arguments are no longer relevant for prioritization.

Looking Beyond the First Argument List

Overloading resolution now can take argument lists into account when choosing among a set of overloaded alternatives. For example, the following code compiles in Scala 3, while it results in an ambiguous overload error in Scala 2:

```
def f(x: Int)(y: String): Int = 0
def f(x: Int)(y: Int): Int = 0

f(3)("")     // ok
```

The following code compiles as well:

```
def g(x: Int)(y: Int)(z: Int): Int = 0
def g(x: Int)(y: Int)(z: String): Int = 0

g(2)(3)(4)     // ok
g(2)(3)("")    // ok
```

To make this work, the rules for overloading resolution in SLS §6.26.3 are augmented as follows:

> In a situation where a function is applied to more than one argument list, if overloading resolution yields several competing alternatives when `n >= 1` parameter lists are taken into account, then resolution re-tried using `n + 1` argument lists.

This change is motivated by the new language feature extension methods, where emerges the need to do overload resolution based on additional argument blocks.

Parameter Types of Function Values

The handling of function values with missing parameter types has been improved. We can now pass such values in the first argument list of an overloaded application, provided that the remaining parameters suffice for picking a variant of the overloaded function. For example, the following code compiles in Scala 3, while it results in a missing parameter type error in Scala2:

```
def f(x: Int, f2: Int => Int) = f2(x)
def f(x: String, f2: String => String) = f2(x)
f("a", _.toUpperCase)
f(2, _ * 2)
```

To make this work, the rules for overloading resolution in SLS §6.26.3 are modified as follows:

Replace the sentence

> Otherwise, let `S1,…,Sm` be the vector of types obtained by typing each argument with an undefined expected type.

with the following paragraph:

> Otherwise, let `S1,…,Sm` be the vector of known types of all argument types, where the *known type* of an argument `E` is determined as followed:

- If `E` is a function value `(p_1, ..., p_n) ⇒ B` that misses some parameter types, the known type of `E` is `(S_1, ..., S_n) ⇒ ?`, where each `S_i` is the type of parameter `p_i` if it is given, or `?` otherwise. Here `?` stands for a *wildcard type* that is compatible with every other type.
- Otherwise the known type of `E` is the result of typing `E` with an undefined expected type.

A pattern matching closure

```
{ case P1 => B1 ... case P_n => B_n }
```

is treated as if it was expanded to the function value

```
x => x match { case P1 => B1 ... case P_n => B_n }
```

and is therefore also approximated with a `? => ?` type.

Default Arguments Are No longer Relevant for Prioritization

In Scala 2 if among several applicative alternatives one alternative had default arguments, that alternative was dropped from consideration. This has the unfortunate side effect that adding a default to a parameter of a method can render this method invisible in overloaded calls.

Scala 3 drops this distinction. Methods with default parameters are not treated to have lower priority than other methods.

‹ Implici... Match ... ›