Type parameters versus member types in Scala

Asked 15 years, 10 months ago Modified 7 years, 7 months ago Viewed 3k times

10

I'd like to know how do the member types work in Scala, and how should I associate types.

One approach is to make the associated type a type parameter. The advantages of this approach is that I can prescribe the variance of the type, and I can be sure that a subtype doesn't change the type. The disadvantages are, that I cannot infer the type parameter from the type in a function.

The second approach is to make the associated type a member of the second type, which has the problem that I can't prescribe bounds on the subtypes' associated types and therefore, I can't use the type in function parameters (when x : X, X#T might not be in any relation with x.T)

A concrete example would be:

I have a trait for DFAs (could be without the type parameter)

```
trait DFA[S] { /* S is the type of the symbols in the alphabet */
  trait State { def next(x : S); }
  /* final type Sigma = S */
}
```

and I want to create a function for running this DFA over an input sequence, and I want

- the function must take anything `<% Seq[alphabet-type-of-the-dfa]` as input sequence type
- the function caller needn't specify the type parameters, all must be inferred
- I'd like the function to be called with the concrete DFA type (but if there is a solution where the function would not have a type parameter for the DFA, it's OK)
- the alphabet types must be unconstrained (ie. there must be a DFA for Char as well as for a yet unknown user-defined class)
- the DFAs with different alphabet types are not subtypes

I tried this:

```
def runDFA[S, D <: DFA[S], SQ <% Seq[S]](d : D)(seq : SQ) = ....
```

this works, except the type S is not inferred here, so I have to write the whole type parameter list on each call site.

```
def runDFA[D <: DFA[S] forSome { type S }, SQ <% Seq[D#Sigma]]( ... same as
above
```

this didn't work (invalid circular reference to type D??? (what is it?))

I also deleted the type parameter, created an abstract type Sigma and tried binding that type in the concrete classes. runDFA would look like

```
def runDFA[D <: DFA, SQ <% Seq[D#Sigma]]( ... same as above
```

but this inevitably runs into problems like "type mismatch: expected `dfa.Sigma`, got `D#Sigma`"

Any ideas? Pointers?

Edit:

As the answers indicate there is no simple way of doing this, could somebody elaborate more on why is it impossible and what would have to be changed so it worked?

The reasons I want runDFA ro be a free function (not a method) is that I want other similar functions, like automaton minimization, regular language operations, NFA-to-DFA conversions, language factorization etc. and having all of this inside one class is just against almost any principle of OO design.

scala types type-inference

Share

Improve this question

Follow

edited Feb 27, 2009 at 0:46

asked Feb 26, 2009 at 20:40

jpalecek
47.7k ● 7 ● 103 ● 146

Wow, I thought you meant this: huygens-fokker.org/scala – MusiGenesis Jul 26, 2009 at 19:19

3 Answers

Sorted by: Highest score (default)

First off, you don't need the parameterisation SQ <% Seq[S]. Write the method parameter as Seq[S]. If SQ <% Seq[S] then any instance of it is implicitly convertable

3

to Seq[S] (that's what <% means), so when passed as Seq[S] the compiler will automatically insert the conversion.

Additionally, what Jorge said about type parameters on D and making it a method on DFA hold. Because of the way inner classes work in Scala I would *strongly advise* putting runDFA on DFA. Until the path dependent typing stuff works, dealing with inner classes of some external class can be a bit of a pain.

So now you have

```
trait DFA[S]{
  ...

  def runDFA(seq : Seq[S]) = ...
}
```

And runDFA is all of a sudden rather easy to infer type parameters for: It doesn't have any.

Share
Improve this answer
Follow

edited Feb 27, 2009 at 0:34
jpalecek
47.7k ● 7 ● 103 ● 146

answered Feb 26, 2009 at 22:53
DRMacIver
2,305 ● 1 ● 17 ● 17

3

Scala's type inference sometimes leaves much to be desired.

Is there any reason why you can't have the method inside your DFA trait?

```
def run[SQ <% Seq[S]](seq: SQ)
```

If you don't need the D param later, you can also try defining your method without it:

```
def runDFA[S, SQ <% Seq[S]](d: DFA[S])(seq: SQ) = ...
```

Share Improve this answer Follow

answered Feb 26, 2009 at 22:23
Jorge Ortiz
4,722 ● 1 ● 22 ● 22

0

Some useful info on how the two differs :

From the the shapeless guide:

Without type parameters you cannot make dependent types , for example

```scala
trait Generic[A] {
  type Repr
  def to(value: A): Repr
  def from(value: Repr): A
}

import shapeless.Generic
def getRepr[A](value: A)(implicit gen: Generic[A]) =
  gen.to(value)
```

Here the type returned by `to` depends on the input type `A` (because the supplied implicit depends on `A`):

```scala
case class Vec(x: Int, y: Int)
case class Rect(origin: Vec, size: Vec)
getRepr(Vec(1, 2))
// res1: shapeless.::[Int,shapeless.::[Int,shapeless.HNil]] = 1 :: 2 ::
     HNil
getRepr(Rect(Vec(0, 0), Vec(5, 5)))
// res2: shapeless.::[Vec,shapeless.::[Vec,shapeless.HNil]] = Vec(0,0)
     :: Vec(5,5) :: HNil
```

without type members this would be impossible :

```scala
trait Generic2[A, Repr]
def getRepr2[A, R](value: A)(implicit generic: Generic2[A, R]): R =
  ???
```

> We would have had to pass the desired value of Repr to getRepr as a type parameter, effec vely making getRepr useless. The intui ve take-away from this is that type parameters are useful as “inputs” and type members are useful as “outputs”.

please see the shapeless guide for details.

Share

Improve this answer

Follow

edited Apr 27, 2017 at 9:59

answered Apr 27, 2017 at 9:53

jhegedus

20.6k ● 18 ● 102 ● 174