Blog ↩ (/blog)

Type lambdas and kind projector

Share on Twitter (https://twitter.com/share) G+ Partag

Posted 05 Dec 2016 **by** Danielle Ashley

Scala developers might have heard of “type lambdas”, a fairly horrendous-looking construction that sometimes appears in code using higher-kinded types. What is a type lambda, why would we ever want to use one, and even better, how can we avoid having to write them? In this blog post we tackle these questions.

The need for type lambdas usually arises when dealing with higher-kinded types. These are type constructors that take other types (or even other type constructor) as parameter. A familiar example is `List`. `List` by itself is not a type. You need to provide a parameter as `Int` to create a type such as `List[Int]`. We call `List` a type constructor. When we declare a type variable that is a type constructor, we write `F[_]` to indicate that `F` needs to be provided with a type to construct a concrete type.

Type aliases

Scala developers will be familiar with declaring type aliases:

```scala
type L = List[(Option[(Int,Double)])]
```

We can use `L` in any place where we can use the unwieldy expression on the right side.

We can also also declare type aliases that take parameters:

```scala
type T[A] = Option[Map[Int, A]]

val t: T[String] = Some(Map(1 -> "abc", 2 -> "xyz"))
```

Often type aliases are simply used as a convenience, but sometimes their use is required as in the following example.

Let's define a type parameterised on another type constructor (we'll call it `Functor` to link to a real-world example, but could be anything else of the same kind—don't let the name confuse you). We'll see what parameters we are allowed to use with it. Remember that it is expecting a type constructor with one parameter:

```
trait Functor[F[_]]
type F1 = Functor[Option] // OK
type F2 = Functor[List]   // OK
type F3 = Functor[Map]    // !!
// error: Map takes two type parameters, expected: one
//        type fo = Functor[Map]
//                          ^
```

The compiler error message indicates the problem: Map takes two type parameters (`Map[K,V]`) while the type parameter to `Functor` expects one. Type aliases are often used to 'partially apply' a type constructor and so to 'adapt' the kind of the type to be used:

```
type IntKeyMap[A] = Map[Int, A]

type F3 = Functor[IntKeyMap] // OK
```

`IntKeyMap` now takes a single type parameter, and the compiler is happy with that. This works fine, but can we achieve the same goal without having to declare an alias? We could try to mirror the syntax of partially-applied value-level functions, with the underscore syntax, as in:

```
val cube = Math.pow(_: Double, 3) // cube: Double => Double
cube(2) // 8
```

But this syntax doesn't do the same thing with types:

```
type F4 = Functor[Map[Int, _]]
// error: Map[Int, _] takes no type parameters, expected: one
//        type F4 = Functor[Map[Int, _]]
//                             ^
```

Scala uses the underscore in different (one could say inconsistent) ways depending on the context. In this case (in the right hand side of the type alias definition) what is implied is not partial application at all, but rather "I don't care what this type is". This is known as an existential type if you want to read up further.

There is, in fact, currently no direct syntax for partial application of type constructors in Scala.

Type lambdas

We can solve this problem of partially applying types by using a type lambda. Let's return to our example:

```
({ type T[A] = Map[Int, A] })#T
```

The heart of the expression above appears to be exactly the same as declaring a type alias, but can be used inline:

```
type F5 = Functor[({ type T[A] = Map[Int, A] })#T] // OK
```

We can read the type lambda syntax as: declaring an anonymous type, inside of which we define the desired type alias, and then accessing its type member with the `#` syntax.

It's easy to argue that the construction above is more offensive to the eye than using an extra line to declare a type alias the traditional way. Indeed we recommend using a type alias whenever possible to keep your code clean and readable. However, sometimes type lambdas are unavoidable. Consider the following rather abstract example:

```
def foo[A[_, _], B](functor: Functor[A[B, ?]]) // won't compile
```

This is not valid Scala, but it is the quickest way to convey the intention. Imagine that the `?` behaves like the partial type constructor application we mentioned earlier, leaving the `functor` argument as a single unspecified type parameter.

Can we resolve the situation using a separate type alias?

```
type AB[C] = ... // what should we put here?

def foo[A[_,_], B](functor: Functor[AB])
```

The answer is no, because at the time at which we define `AB`, we don't have `A` and `B` available. Attempts to 'pass them in' as parameters like this:

```
type T[A, B, C] = ...
```

defeat the purpose because they alter the type arity. We needed an arity of 1 to pass into `foo` but now we've just increased it to 3, which is clearly not going to go down with the compiler.

Alternative encodings

What are the possible ways of implementing our `Functor` example?

Use a type lambda

We can use the type lambda after all:

```
def foo[A[_,_],B](functor: Functor[({type AB[C] = A[B,C]})#AB])
```

This works because the types `A` and `B` are available in the scope when we define `AB`.

Declare a surrounding class

If we prefer not to use type lambdas we can split the definition of `foo` in two:

```
class Foo[A[_,_],B] {
  type AB[C] = A[B, C]
  def apply(functor: Functor[AB]) = ...
}
def foo[A[_,_],B] = new Foo[A,B]
```

Like type lambdas, this technique allows us to define `AB` once `A` and `B` are already known.

However, this approach is verbose and causes allocations at run time, whereas the type lambda exists only within the compiler.

Curried type constructors

A third hypothetical solution, that is not currently possible but would fix this issue cleanly, is to use curried type constructors. These are similar to the partially applied type constructors we hypothesised earlier.

Just as we can have multiple argument lists for methods:

```
def fill(n: Int)(elem: Double) = ...
val fill10 = fill(10) _ // fill10: Double => List[Double]
fill10(5.1)
// List(5.1, 5.1, 5.1, 5.1, 5.1, 5.1, 5.1, 5.1, 5.1, 5.1)
```

so we could, in principle, have the same at the type level:

```
type AB[A, B][C] = A[B, C]
```

We could then 'partially apply' this (with the first argument list only: `AB[A, B]`), leaving behind the arity-1 type constructor we require:

```
type AB[A, B][C] = A[B, C] // (not valid syntax yet!)

def foo[A[_, _], B](functor: Functor[AB[A,B]])
```

While this approach is currently fantasy, there have been rumours about the introduction of curried or partially applied types in Dotty.

Kind projector

While we wait for curried type constructors to become part of the language, we can find another solution in the kind projector (https://github.com/non/kind-projector) compiler plugin.

Kind projector provides a clearer syntax for type lambdas. For example, we can implement our functor from above as follows:

```
type F = Functor[Map[Int, ?]] // now works!

def foo[A[_, _], B](functor: Functor[A[B, ?]]) // now works!
```

With this we get as close as we can to our initial aim of writing types as if we were partially applying type constructors. The only difference is that we use `?` to do it instead of `_`, which already has too many uses in Scala.

During compilation, kind projector translates type expressions containing `?` into regular type lambdas, giving us the same semantics with a large gain in readability. Kind projector doesn't work in all cases, but for the most part it makes our lives a lot simpler!

Conclusions

Type lambdas are an ugly but necessary concept in Scala for the time being. We can usually use a type alias to avoid having to write a type lambda. In many cases that we cannot use an alias, the kind projector compiler plugin will usually solve the problem.

For more discussion of type lambdas, see this blog post (https://blog.adilakhter.com/2015/02/18/applying-scalas-type-lambda/) by Adil Akhter.

For more information about kind projector, see the project page ((https://github.com/non/kind-projector)) on Github.

Comments

Please review our Community Guidelines (/terms#comments) before posting a comment. We encourage discussion in good faith, but do not allow combative, exclusionary, or harassing behaviour. If you have any questions, contact us (/contact)!