Scala Type System III

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

11 марта 2021

Structural Types

Disclaimer

Do not use structural types unless you're 100% sure you need them

Structural Types

Structural type is a type whose conformance is defined by its strucrure(a set of contained signatures) only.

Structural Types

Structural type is a type whose conformance is defined by its strucrure(a set of contained signatures) only.

```
1 type withFoo = { def foo(x: Int): String }
2
3 def foo(x: withFoo)
4 def bar(x: { def foo(x: Int): String })
```

Structural Types

- implemented using runtime reflection — watch performance
- *usecase*: combine with `.asInstanceOf` for straightforward reflective method calls

This Types

Singleton types

Singleton type is a type of a single value, to which only this particular value and `null` conform.

Singleton types

Singleton type is a type of a single value, to which only this particular value and `null` conform.

```
1 val hello = "Hello"
2
3 val notHello:  hello.type = "world" // error: type mismatch
4 val alsoHello: hello.type = hello   // OK
5
6
```

Singleton types

Singleton type is a type of a single value, to which only this particular value and `null` conform.

```
1 val hello = "Hello"
2
3 val notHello:  hello.type = "world" // error: type mismatch
4 val alsoHello: hello.type = hello   // OK
5
6 object A
7 def method(): A.type = A
```

Singleton Types

Method chaining

```
1 class A
2   { def method1: this.type = { ...; this } }
3 class B extends A
4   { def method2: this.type = { ...; this } }
5
6 new B.method1.method2 // OK
```

F-bound Polymorphism

Problem

Frequent question

I have a type hierarchy ... how do I declare a supertype method that returns the "current" type?[2]

F-bound Polymorphism

```scala
trait Pet {
  def renamed: Pet
}

abstract class Dog extends Pet

class Cat extends Pet {
  override def renamed: Dog = ??? // OOPS
}
```

F-bound Polymorphism

```scala
trait Pet {
  type T <: Pet
  def renamed: T
}

abstract class Dog extends Pet

class Cat extends Pet {
  override type T = Dog
  override def renamed: Dog = ??? // still OOPS
}
```

F-bound Polymorphism

```scala
trait Pet {
  type T >: this.type <: Pet
  def renamed: T
}

abstract class Dog extends Pet

class Cat extends Pet {
  override type T = Dog   // error: type T has incompatible type
  override def renamed: Dog = ???
}
```

F-bound Polymorphism

```scala
trait Pet {
  type T >: this.type <: Pet
  def renamed: T
}

abstract class Dog extends Pet

class Cat extends Pet {
  override type T = Cat
  override def renamed: Cat = ??? // OK ✓
}
```

Existential Types

Existential Types

Existential type lets up assume **existence** of some type parameter without exposing or providing it explicitly[1]

Existential Types

```
1 trait Foo[T] { def foo: T }
2
3 type Dep[A] = Foo[A]                      // A is required on lhs
4 type Exis   = Foo[A] forSome { type A } // A only in rhs
5 type ExisA  = Foo[_]                      // A is not required at all
6
7 def bar(x: Exis) = x.foo // return type ?
```

Existential Types

```
1 trait Foo[T] { def foo: T }
2
3 type Dep[A] = Foo[A]                              // A is required on lhs
4 type Exis   = Foo[A] forSome { type A <: Int } // A only in rhs
5 type ExisA  = Foo[_ <: Int]                       // A is not required at all
6
7 def bar(x: Exis): Int = x.foo
```

Higher-Kinded Types

Kinds

Kinds are an abstraction over types that implement parametric polymorphism in Scala

Tere are two *kinds* in Scala:

- ***** — a proper type
- $\rightarrow$ — a type constructor

Kinds

Example

```
scala> :kind -v String
String's kind is A { * }
This is a proper type.

scala> :kind -v List
List's kind is F[+A] { * -(+)-> * }
This is a type constructor: a 1st-order-kinded type.
```

Type Constructor

A *type constructor* is a type-level function of a $\rightarrow$ kind, that takes type parameters and applies them to yield a proper type or another type constructor

```
1 type IntList = List[Int]
2 type StringOrInt = Either[String, Int]
3
4 type StringOr??? = Either[String, _] // how?
```

Higher Kinded Types

A *higher kinded type* is a type constructor that accepts or returns another type constructor.

This is analogous to a higher-order function, and denoted as $(* \to *) \to *$.

Higher Kinded Types

Example

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

val x: Functor[Option] = ???
x.map(Some(123)) { x => x.toString } // :Option[String]
```

Partial Application of HKT

If higher kinded types behave similarly to higher order functions, can they also be partially applied?

```
1 type StringOr = Either[_, String] // this is an existential type
```

Partial Application of HKT

```
1 val efs: Functor[Either[_, String]] = ???
2                // ^ error: Either[_, String] takes no type parameters, expected: one
```

Partial Application of HKT

```scala
val efs = new Functor[???] {
    override def map[A, B](fa: Either[Int, A])(f: A => B): Either[Int, B] = ???
}
```

Partial Application of HKT

```
1 type IntOrA[A] = Either[Int, A]
2 val efs = new Functor[IntOrA] {
3     override def map[A, B](fa: Either[Int, A])(f: A => B): Either[Int, B] = ???
4 }
```

Partial Application of HKT

```scala
val efs = new Functor[({type lam[Y] = Either[Int, Y]})#lam] {
    override def map[A, B](fa: Either[Int, A])(f: A => B): Either[Int, B] = ???
}
```

Kind Projector

Type lambda syntax in Scala 2 is verbose. Kind projector plugin[4] solves this issue for HKT-heavy applications.

Kind Projector

Type lambda syntax in Scala 2 is verbose. Kind projector plugin[4] solves this issue for HKT-heavy applications.

```
1 Tuple2[*, Double]         // equivalent to: type R[A] = Tuple2[A, Double]
2 Either[Int, +*]           // equivalent to: type R[+A] = Either[Int, A]
3 Function2[-*, Long, +*]   // equivalent to: type R[-A, +B] = Function2[A, Long, B]
4 EitherT[*[_], Int, *]     // equivalent to: type R[F[_], B] = EitherT[F, Int, B]
```

AUX Pattern

AUX Pattern

scalac error

illegal dependent method type: parameter appears in the type of another parameter in the same section or an earlier one

Scala tells us that we can't use the dependent type in the same section, we can use it in the **next** parameters block or as a **return type** only.
This is where AUX pattern[5] comes into play

AUX Pattern

```scala
trait Foo[A] {
  type B
  def value: B
}

def foo[T](t: T)(implicit f: Foo[T]): f.B = f.value                // OK
def foo[T](t: T)(implicit f: Foo[T], m: Monoid[f.B]): f.B = m.zero // error
```

AUX Pattern

```scala
trait Foo[A] {
  type B
  def value: B
}

def foo[T](t: T)(implicit f: Foo[T]): f.B = f.value
def foo[T, B0](t: T)(implicit f: Foo[T] {type B = B0}, m: Monoid[B0]): f.B = m.zero // OK
```

AUX Pattern

```scala
trait Foo[A] {
  type B
  def value: B
}
object Foo {
  type Aux[A0, B0] = Foo[A0] { type B = B0  }
}

def foo[T, R](t: T)(implicit f: Foo.Aux[T, R], m: Monoid[R]): R = m.zero
```

Magnet Pattern

Magnet Pattern

Problem statement

How to define multiple method overloads with only difference in parametrized types?

Magnet Pattern

Problem statement

How to define multiple method overloads with only difference in parametrized types?

```
1 def complete(future: Future[HttpResponse]): Int
2 def complete(future: Future[StatusCode]): Int
```

Magnet Pattern

Define a magnet trait:

```
1 sealed trait FutureMagnet {
2   type Result
3   def apply() : Result
4 }
5
6 def completeFuture(magnet: FutureMagnet):magnet.Result = magnet()
```

Magnet Pattern

```
implicit def fromHttpResponseFuture(future: Future[HttpResponse]) =
  new CompletionMagnet {
    type Result = Int
    def apply(): Result = ... // implementation using future
  }
implicit def fromStatusCodeFuture(future: Future[StatusCode]) =
  new CompletionMagnet {
    type Result = Int
    def apply(): Result = ... // implementation using future
  }
```

[1] https://pjrt.medium.com/existential-types-in-scala-6321f19c4a57

[2] https://tpolecat.github.io/2015/04/29/f-bounds.html

[3] https://kubuszok.com/compiled/kinds-of-types-in-scala

[4] https://github.com/typelevel/kind-projector

[5] https://gigiigig.github.io/posts/2015/09/13/aux-pattern.html

[6] http://blog.madhukaraphatak.com/scala-magnet-pattern/