<:<

Stefan

Based on Generalized type constraints in Scala

Table of content

Introduction
- Usage
- Meaning

Questions
- Can't we just use type bounds?

Other

How is it useful?

- on Option

```
def flatten[B](implicit ev: A <:< Option[B]):
    Option[B]
```

- on Traversable

```
def toMap[K, V](implicit ev: A <:< (K, V)): Map[K, V]
```

- on Try

```
def flatten[U](implicit ev: T <:< Try[U]): Try[U]
```

What do they have in common?

- implicit parameter list, with a single parameter called ev
- type of this parameter is of the form `Type1 <:< Type2`

Meaning

`Type1 <:< Type2`

Meaning

Make sure that Type1 is a subtype of Type2, or else report an error.

Example 1

```
scala> val oo: Option[Option[Int]] = Some(Some(42))
oo: Option[Option[Int]] = Some(Some(42))

scala> oo.flatten
res1: Option[Int] = Some(42)
```

Example 2

```
scala> val oi: Option[Int] = Some(42)
oi: Option[Int] = Some(42)

scala> oi.flatten
<console>:21: error: Cannot prove that Int <:< Option[B].
       oi.flatten
```

Plain example

```
scala> def tuple[T, U](t: T, u: U) = (t, u)
scala> tuple("Lincoln", 42)
res1: (String, Int) = (Lincoln,42)
```

Example with upper bound

```
scala> def tupleIfSubtype[T <: U, U](t: T, u: U) = (t, u)
scala> tupleIfSubtype("Lincoln", 42)
```

Example with upper bound

```
scala> def tupleIfSubtype[T <: U, U](t: T, u: U) = (t, u)
scala> tupleIfSubtype("Lincoln", 42)

res2: (String, Any) = (Lincoln,42)
```

Puzzler

Why?

type inference solves a constraint system

Puzzler

Why?

type inference solves a constraint system

so given

```
scala> def tupleIfSubtype[T <: U, U](t: T, u: U) = (t, u)
scala> tupleIfSubtype("Lincoln", 42)
res2: (String, Any) = (Lincoln,42)
```

the constraints are satisfied with

- String is a String of course
- Int is a subtype of Any
- String is also a subtype of Any

Other

=:=

<:!<