Generalized type constraints

Stefan

Based on Generalized type constraints in Scala

Table of content

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

Simplified implementation

```scala
trait <::<[-From, +To]

object <::< {
  implicit def conforms[A]: A <::< A = new <::<[A,A]{}
}
```

Simplified implementation

```
trait <::<[-From, +To]

object <::< {
  implicit def conforms[A]: A <::< A = new <::<[A,A]{}
}
```

e.g.

```
def tupleIfSubtype[T, U](t: T, u: U)(implicit ev: T <::<
    U) = (t, u)

tupleIfSubtype(new Apple(), new
    Fruit())(<::<.conforms[Fruit])
tupleIfSubtype(new Apple(), new
    Fruit())(<::<.conforms[Apple])
```

Why does it work?

- Variance of the type parameters of `trait <:<[-From, +To]` is similar to a function: require less, provide more: $F \Rightarrow T$
- $\rightarrow$ compiler tries to find a type `<:<[A, A]` that conforms to e.g. `<:<[Apple, Fruit]`

What constraints must be satisfied?

- $F \Rightarrow T$ is required, what are the constraints to pass
- $A \Rightarrow A$?

What constraints must be satisfied?

- $F \Rightarrow T$ is required, what are the constraints to pass
- $A \Rightarrow A$?
- Due to variance `trait <:<[-F, +T]`:
  - A must be a supertype of F: `A >: F` or `F <: A`
  - A must be a subtype of T: `A <: T`

What constraints must be satisfied?

- $F \Rightarrow T$ is required, what are the constraints to pass
- $A \Rightarrow A$?
- Due to variance `trait <::<[-F, +T]`:
  - A must be a supertype of F: `A >: F` or `F <: A`
  - A must be a subtype of T: `A <: T`
- $\rightarrow$ `F <: A <: T`
- $\rightarrow$ `F <: T`

Real implementation

- adds a nice error message:
  ```
  @implicitNotFound(msg = "Cannot prove that ${From}
  <:< ${To}.")
  ```
- extends function:
  ```
  class <:<[-From, +To] extends (From =>To)
  ```
- uses a singleton value:
  ```
  val singleton_<:< =
  new <:<[Any,Any] { def apply(x: Any): Any = x }
  ```
- and typecast it:
  ```
  singleton_<:<.asInstanceOf[A <:< A]
  ```
- evidence also used as conversion *From* ⇒ *To*, e.g. in Option:
  ```
  def flatten[B](implicit ev: A <:< Option[B]):
  Option[B] = if (isEmpty)None else ev(this.get)
  ```

Other

- alternative: implicit class or type class $\rightarrow$ more overhead
- exact type `=:=[From, To]` vs `<:<[-From, +To]`

Other

- alternative: implicit class or type class → more overhead
- exact type `=:=[From, To]` vs `<:<[-From, +To]`
- not subtype <:!<

```
def unexpected : Nothing = sys.error("Unexpected
    invocation")
@scala.annotation.implicitNotFound("${A} must not be
    a subtype of ${B}")
trait <:!<[A, B] extends Serializable
implicit def nsub[A, B] : A <:!< B = new <:!<[A, B] {}
implicit def nsubAmbig1[A, B >: A] : A <:!< B =
    unexpected
implicit def nsubAmbig2[A, B >: A] : A <:!< B =
    unexpected
```