Transparent Traits

Traits are used in two roles:

1. As mixins for other classes and traits
2. As types of vals, defs, or parameters

Some traits are used primarily in the first role, and we usually do not want to see them in inferred types. An example is the `Product` trait that the compiler adds as a mixin trait to every case class or case object. In Scala 2, this parent trait sometimes makes inferred types more complicated than they should be. Example:

```
trait Kind
case object Var extends Kind
case object Val extends Kind
val x = Set(if condition then Val else Var)
```

Here, the inferred type of `x` is `Set[Kind & Product & Serializable]` whereas one would have hoped it to be `Set[Kind]`. The reasoning for this particular type to be inferred is as follows:

- The type of the conditional above is the union type `Val | Var`.
- A union type is widened in type inference to the least supertype that is not a union type. In the example, this type is `Kind & Product & Serializable` since all three traits are traits of both `Val` and `Var`. So that type becomes the inferred element type of the set.

Scala 3 allows one to mark a mixin trait as `transparent`, which means that it can be suppressed in type inference. Here's an example that follows the lines of the code above, but now with a new transparent trait `S` instead of `Product`:

```
transparent trait S
trait Kind
```

```scala
object Var extends Kind, S
object Val extends Kind, S
val x = Set(if condition then Val else Var)
```

Now `x` has inferred type `Set[Kind]`. The common transparent trait `S` does not appear in the inferred type.

Transparent Traits

The traits `scala.Product`, `java.io.Serializable` and `java.lang.Comparable` are treated automatically as transparent. Other traits are turned into transparent traits using the modifier `transparent`. Scala 2 traits can also be made transparent by adding a `@transparentTrait` annotation. This annotation is defined in `scala.annotation`. It will be deprecated and phased out once Scala 2/3 interoperability is no longer needed.

Typically, transparent traits are traits that influence the implementation of inheriting classes and traits that are not usually used as types by themselves. Two examples from the standard collection library are:

- `IterableOps`, which provides method implementations for an `Iterable`.
- `StrictOptimizedSeqOps`, which optimises some of these implementations for sequences with efficient indexing.

Generally, any trait that is extended recursively is a good candidate to be declared transparent.

Rules for Inference

Transparent traits can be given as explicit types as usual. But they are often elided when types are inferred. Roughly, the rules for type inference say that transparent traits are dropped from intersections where possible.

The precise rules are as follows:

- When inferring a type of a type variable, or the type of a val, or the return type of a def,
- where that type is not higher-kinded,
- and where `B` is its known upper bound or `Any` if none exists:
- If the type inferred so far is of the form `T1 & ... & Tn` where `n ≥ 1`, replace the maximal number of transparent `Ti`s by `Any`, while ensuring that the resulting type is still a subtype of the bound `B`.
- However, do not perform this widening if all transparent traits `Ti` can get

replaced in that way.

The last clause ensures that a single transparent trait instance such as `Product` is not widened to `Any`. Transparent trait instances are only dropped when they appear in conjunction with some other type.

< Trait P... Univer... >