Opaque Type Aliases: More Details

Syntax

```
Modifier          ::=  ...
                    |  'opaque'
```

`opaque` is a soft modifier. It can still be used as a normal identifier when it is not in front of a definition keyword.

Opaque type aliases must be members of classes, traits, or objects, or they are defined at the top-level. They cannot be defined in local blocks.

Type Checking

The general form of a (monomorphic) opaque type alias is

```
opaque type T >: L <: U = R
```

where the lower bound `L` and the upper bound `U` may be missing, in which case they are assumed to be `scala.Nothing` and `scala.Any`, respectively. If bounds are given, it is checked that the right-hand side `R` conforms to them, i.e. `L <: R` and `R <: U`. F-bounds are not supported for opaque type aliases: `T` is not allowed to appear in `L` or `U`.

Inside the scope of the alias definition, the alias is transparent: `T` is treated as a normal alias of `R`. Outside its scope, the alias is treated as the abstract type

```
type T >: L <: U
```

A special case arises if the opaque type alias is defined in an object. Example:

```
object o:
  opaque type T = R
```

In this case we have inside the object (also for non-opaque types) that `o.T` is equal to `T` or its expanded form `o.this.T`. Equality is understood here as mutual subtyping, i.e. `o.T <: o.this.T` and `o.this.T <: T`. Furthermore, we have by the rules of opaque type aliases that `o.this.T` equals `R`. The two equalities compose. That is, inside `o`, it is also known that `o.T` is equal to `R`. This means the following code type-checks:

```
object o:
  opaque type T = Int
  val x: Int = id(2)
def id(x: o.T): o.T = x
```

Opaque type aliases cannot be `private` and cannot be overridden in subclasses.

Type Parameters of Opaque Types

Opaque type aliases can have a single type parameter list. The following aliases are well-formed

```
opaque type F[T] = (T, T)
opaque type G = [T] =>> List[T]
```

but the following are not:

```
opaque type BadF[T] = [U] =>> (T, U)
opaque type BadG = [T] =>> [U] => (T, U)
```

Translation of Equality

Comparing two values of opaque type with `==` or `≠` normally uses universal equality, unless another overloaded `==` or `≠` operator is defined for the type. To avoid boxing, the operation is mapped after type checking to the (in-)equality operator defined on the underlying type. For instance,

```
opaque type T = Int

  ...
  val x: T
  val y: T
  x == y    // uses Int equality for the comparison.
```

Top-level Opaque Types

An opaque type alias on the top-level is transparent in all other top-level definitions in the sourcefile where it appears, but is opaque in nested objects and classes and in all other source files. Example:

```
// in test1.scala
opaque type A = String
val x: A = "abc"

object obj:
  val y: A = "abc"  // error: found: "abc", required: A

// in test2.scala
def z: String = x   // error: found: A, required: String
```

This behavior becomes clear if one recalls that top-level definitions are placed in their own synthetic object. For instance, the code in `test1.scala` would expand to

```
object test1$package:
  opaque type A = String
  val x: A = "abc"

object obj:
  val y: A = "abc"  // error: cannot assign "abc" to opaque type alias A
```

The opaque type alias `A` is transparent in its scope, which includes the definition of `x`, but not the definitions of `obj` and `y`.

Relationship to SIP 35

Opaque types in Scala 3 are an evolution from what is described in Scala SIP 35.

The differences compared to the state described in this SIP are:

1. Opaque type aliases cannot be defined anymore in local statement sequences.
2. The scope where an opaque type alias is visible is now the whole scope where it is defined, instead of just a companion object.
3. The notion of a companion object for opaque type aliases has been dropped.
4. Opaque type aliases can have bounds.
5. The notion of type equality involving opaque type aliases has been clarified. It was strengthened with respect to the previous implementation of SIP 35.