TOUR OF SCALA

COMPOUND TYPES

Sometimes it is necessary to express that the type of an object is a subtype of several other types. In Scala this can be expressed with the help of *compound types*, which are intersections of object types.

Suppose we have two traits `Cloneable` and `Resetable`:

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = {
    super.clone().asInstanceOf[Cloneable]
  }
}
trait Resetable {
  def reset: Unit
}
```

Now suppose we want to write a function `cloneAndReset` which takes an object, clones it and resets the original object:

```
def cloneAndReset(obj: ?): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

The question arises what the type of the parameter `obj` is. If it's `Cloneable` then the object can be `clone`d, but not `reset`; if it's `Resetable` we can `reset` it, but there is no `clone` operation. To avoid type casts in such a situation, we can specify the type of `obj` to be both `Cloneable` and `Resetable`. This compound type is written like this in Scala: `Cloneable with Resetable`.

Here's the updated function:

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  //...
}
```

Compound types can consist of several object types and they may have a single refinement which can be used to narrow the signature of existing object members. The general form is: `A with B with C ... { refinement }`

An example for the use of refinements is given on the page about class composition with mixins.

← previous next →

Contributors to this page:


ckipp01

mlachkar

ashawley

dwijnand

rkoeninger

heathermiller