The Expression Problem

Grzegorz Balcerek

24-10-2015

Software entities

(classes, modules, functions, etc.)

should be open for extension,

but closed for modification.

Robert Martin, The Open-Close Principle

```scala
trait Shape {
  def area: Double
}

class Rect(a: Double, b: Double)
extends Shape {
  override def area = a*b
}

object Test extends App {
  val s1: Shape = new Rect(2,4)
  println( s1.area )
}
```

```scala
sealed trait Shape
case class Circle(r: Double) extends Shape
case class Rect(a: Double,
                b: Double) extends Shape

object Area {
  def area(s: Shape) = s match {
    case Circle(r) => Math.PI*r*r
    case Rect(a,b) => a*b
  }
}
```

```
object Test extends App {

  val s1: Shape = new Circle(1)
  val s2: Shape = new Rect(2,4)

  import Area._

  println( area(s1) )
  println( area(s2) )

}
```

The Expression Problem

is a new name for an old problem.

The goal is to define a datatype by cases,

where one can add new cases to the datatype

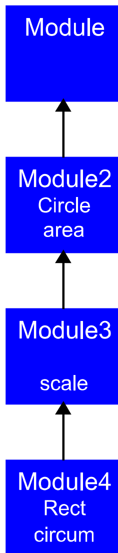
and new functions over the datatype,

without recompiling existing code,

and while retaining static type safety

(e.g., no casts).

Philip Wadler, 12 November 1998


Module
Module2
Circle
area
Module3
scale
Module4
Rect
circum

```
trait Module {
  type shape
}
```

```
trait Module2 extends Module {

  trait Shape {
    def area: Double
  }

  type shape <: Shape

  class Circle(val r: Double)
  extends Shape {
    def area = Math.PI*r*r
  }

}
```

```scala
object Module2Test extends App with Module2 {

  println("Module2Test")

  type shape = Shape

  val s1: shape = new Circle(1)
  val s2: shape = new Circle(3)

  println(s1.area)
  println(s2.area)

}
```

```
trait Module3 extends Module2 {
  trait Shape extends super.Shape {
    def scale(n: Double): shape
  }
  type shape <: Shape

  class Circle(r: Double)
  extends super.Circle(r) with Shape {
    def scale(n: Double) = Circle(n*r)
  }
  def Circle(r: Double): shape

  def scale(s: shape, n: Double) = s.scale(n)
}
```

```scala
trait Module3Final extends Module3 {

  type shape = Shape

  def Circle(r: Double): shape =
    new Circle(r)

}
```

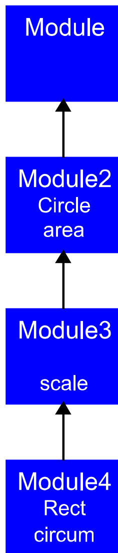
```
object Module3Test extends App
with Module3Final {

  println("Module3Test")

  val s1: shape = Circle(3)
  val s2: shape = s1.scale(4)
  val s3: shape = scale(s1,4)

  println(s1.area)
  println(s2.area)
  println(s3.area)
}
```

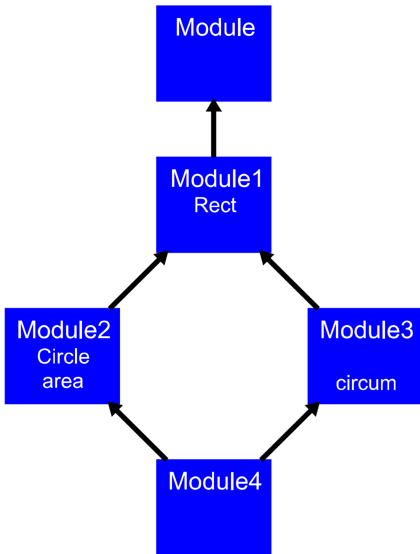
```
trait Module4 extends Module3 {

  trait Shape extends super.Shape {
    def circum: Double
  }

  type shape <: Shape

  class Circle(r: Double)
  extends super.Circle(r) with Shape {
    def circum = Math.PI*2*r
  }
```

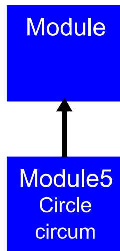
```
  class Rect(val a: Double, val b: Double)
  extends Shape {
    def area = a*b
    def scale(n: Double): shape =
      Rect(n*a,n*b)
    def circum = 2*(a+b)
  }
  def Rect(a: Double, b: Double): shape

}
```

```scala
trait Module4Final extends Module4 {

  type shape = Shape

  def Circle(r: Double): shape =
    new Circle(r)

  def Rect(a: Double, b: Double): shape =
    new Rect(a,b)

}
```

```scala
object Module4Test
extends App with Module4Final {
  println("Module4Test")
  val s1: shape = Circle(1)
  val s2: shape = Rect(2,4)
  val s3: shape = s1.scale(2)
  val s4: shape = s2.scale(2)
  val s5: shape = scale(s2,2)
  println((s1.area, s1.circum))
  println((s2.area, s2.circum))
  println((s3.area, s3.circum))
  println((s4.area, s4.circum))
  println((s5.area, s5.circum))
}
```


Module
Module2
Circle
area
Module3
scale
Module4
Rect
circum


Module
Module1
Rect
Module2
Circle
area
Module3
circum
Module4


Module
Module5
Circle
circum

```
trait Module {
  type shape
}
```

```
trait RectModule extends Module {
  trait Rect {
    def a: Double
    def b: Double
  }
  def Rect(a: Double, b: Double): shape
}

trait Module1 extends RectModule
```

```
trait AreaModule {
  trait Area { def area: Double }
}

trait CircleModule extends Module {
  trait Circle { def r: Double }
  def Circle(r: Double): shape
}
```

```
trait CircleAreaModule extends AreaModule
                                 with CircleModule {
  trait CircleArea extends Circle with Area {
    def area = Math.PI*r*r
  }
}

trait RectAreaModule extends AreaModule
                               with RectModule {
  trait RectArea extends Rect with Area {
    def area = a*b
  }
}
```

```
trait Module2 extends Module1
with CircleAreaModule
with RectAreaModule {

  trait Shape extends Area

  trait Circle extends Shape
                 with CircleArea

  trait Rect extends Shape
               with super.Rect
               with RectArea
}
```

```
trait CircumModule {
  trait Circum {
    def circum: Double
  }
}

trait RectCircumModule extends CircumModule
                          with RectModule {
  trait RectCircum extends Circum
                      with Rect {
    def circum = 2*(a+b)
  }
}
```

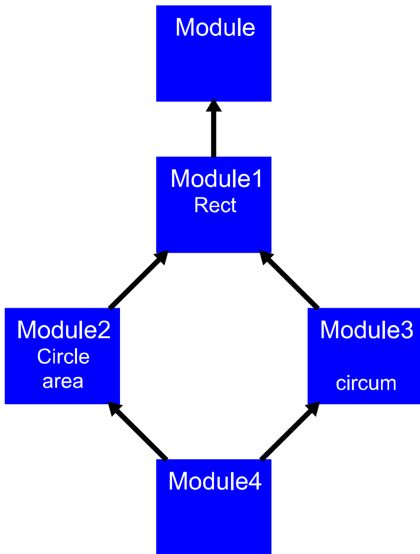
```
trait Module3 extends Module1
with RectCircumModule {

  trait Shape extends Circum

  trait Rect extends Shape with RectCircum

}
```

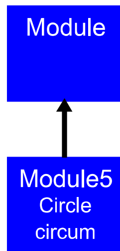
```
trait CircleCircumModule extends CircumModule
                          with CircleModule {
  trait CircleCircum extends Circum
                        with Circle {
    def circum = Math.PI*2*r
  }
}
```

```
trait Module4 extends Module2 with Module3
with CircleCircumModule {

  trait Shape
  extends super[Module2].Shape
  with super[Module3].Shape

  trait Rect extends Shape
  with super[Module2].Rect
  with super[Module3].Rect

  trait Circle extends Shape
  with super.Circle with CircleCircum
}
```

```scala
trait Module4Final extends Module4 {

  type shape = Shape

  class Circle(val r: Double)
    extends super.Circle
  def Circle(r: Double): shape =
    new Circle(r)

  class Rect(val a: Double, val b: Double)
    extends super.Rect
  def Rect(a: Double, b: Double): shape =
    new Rect(a,b)
}
```

```scala
object Test4 extends App with Module4Final {

  println("Test4")

  val s1: shape = Circle(1)
  val s2: shape = Rect(2,4)

  println(s1.area)
  println(s2.area)
  println(s1.circum)
  println(s2.circum)

}
```

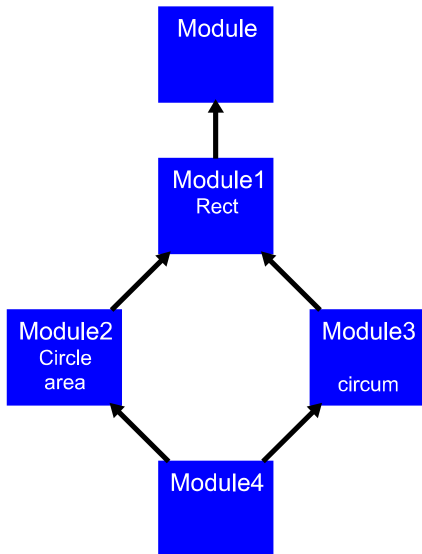

Module
Module1
Rect
Module2
Circle
area
Module3
circum
Module4

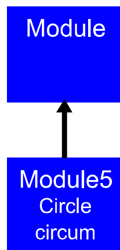

Module
Module5
Circle
circum

```scala
trait Module5 extends Module
with CircleCircumModule {

  trait Shape extends Circum

  trait Circle extends Shape
  with CircleCircum

}
```

```scala
trait Module5Final extends Module5 {

  type shape = Shape

  class Circle(val r: Double)
    extends super.Circle
  def Circle(r: Double): shape =
    new Circle(r)

}
```

```
object Test5 extends App with Module5Final {
  println("Test5")
  val s1: shape = Circle(1)
  println(s1.circum)
}
```


Module
Module1
Rect
Module2
Circle
area
Module3
circum
Module4


Module
Module5
Circle
circum

```scala
trait Area[S] {
  def area(s: S): Double
}

object Area {

  def area[S](s: S)(implicit as: Area[S]) =
    as.area(s)

}
```

```scala
case class Rect(a: Double, b: Double)

object Rect {

  implicit object RectArea
  extends Area[Rect] {
    def area(r: Rect) = r.a * r.b
  }

}
```

```
object Test1 extends App {
  import Area._
  val s1 = Rect(2,4)
  println( area(s1) )
}
```

```
trait Scale[S] {
  def scale(s: S, n: Double): S
}

object Scale {
  def scale[S:Scale](s: S, n: Double) =
    implicitly[Scale[S]].scale(s,n)

  implicit object RectScale
  extends Scale[Rect] {
    def scale(r: Rect, n: Double) =
      Rect(r.a*n, r.b*n)
  }
}
```

```scala
case class Circle(r: Double)

object Circle {

  implicit object CircleArea
  extends Area[Circle] {

    def area(c: Circle) = Math.PI*c.r*c.r

  }
}
```

```
object Test2 extends App {

  import Area._, Scale._

  val s1 = Rect(2,4)
  val s2 = scale(s1,2)
  val s3 = Circle(1)

  println( area(s1) )
  println( area(s2) )
  println( area(s3) )

}
```

```scala
object Test3 extends App {
  import Scale._
  val s1 = Circle(1)
  println( scale(s1,2) ) // compilation error
}
```