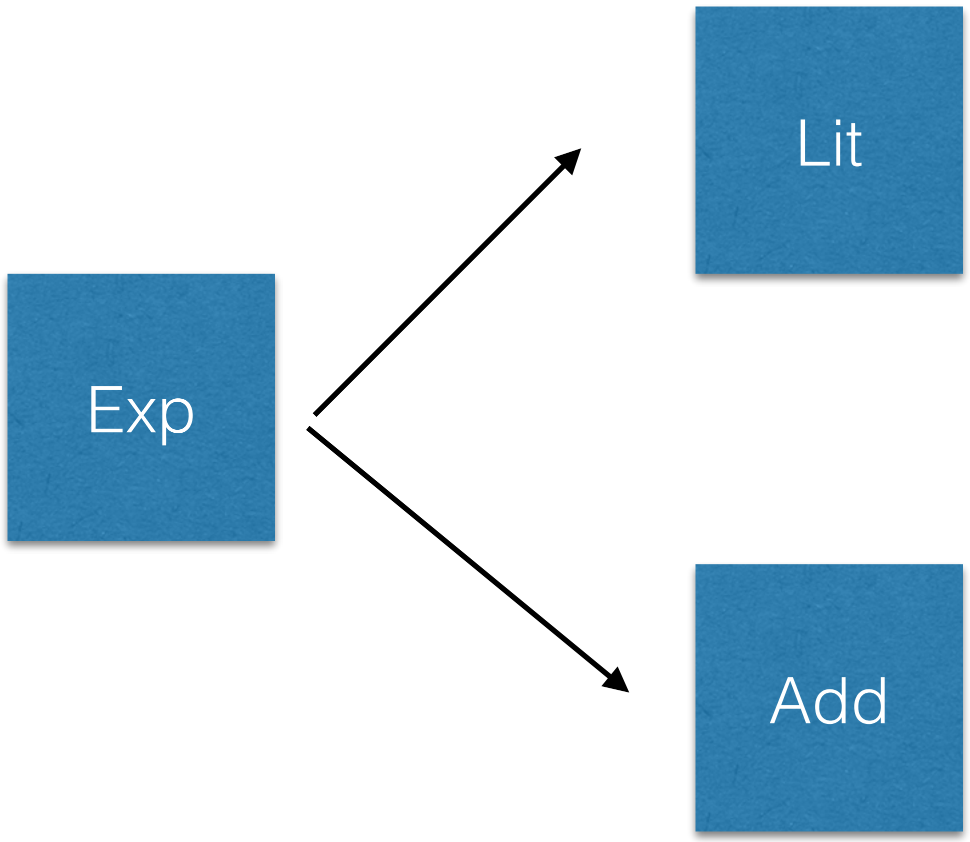
Exploring Pattern Matching in Object Algebras

Yanlin Wang

20150326

Integer Expression Example

eval : Int



eval() = x

eval() = e1.eval + e2.eval

‘Region’ Example [1]

```
eval : (Double, Double) => Boolean
```

$$\begin{tikzcd}
 & \boxed{\text{Univ}} \\
\boxed{\text{reg}} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\text{Circle}} \\
 & \boxed{\text{Union}}
\end{tikzcd}$$

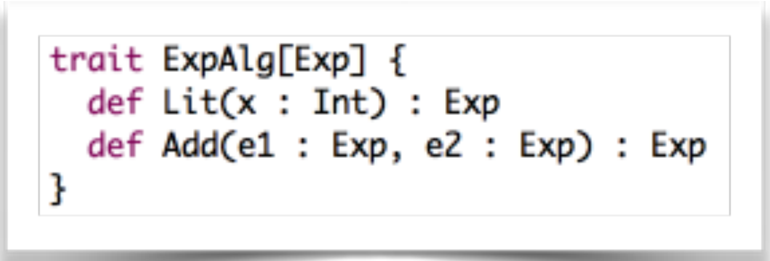
```
eval = (_, _) => true
```

```
eval = (x, y) => x * x + y * y <= r * r
```

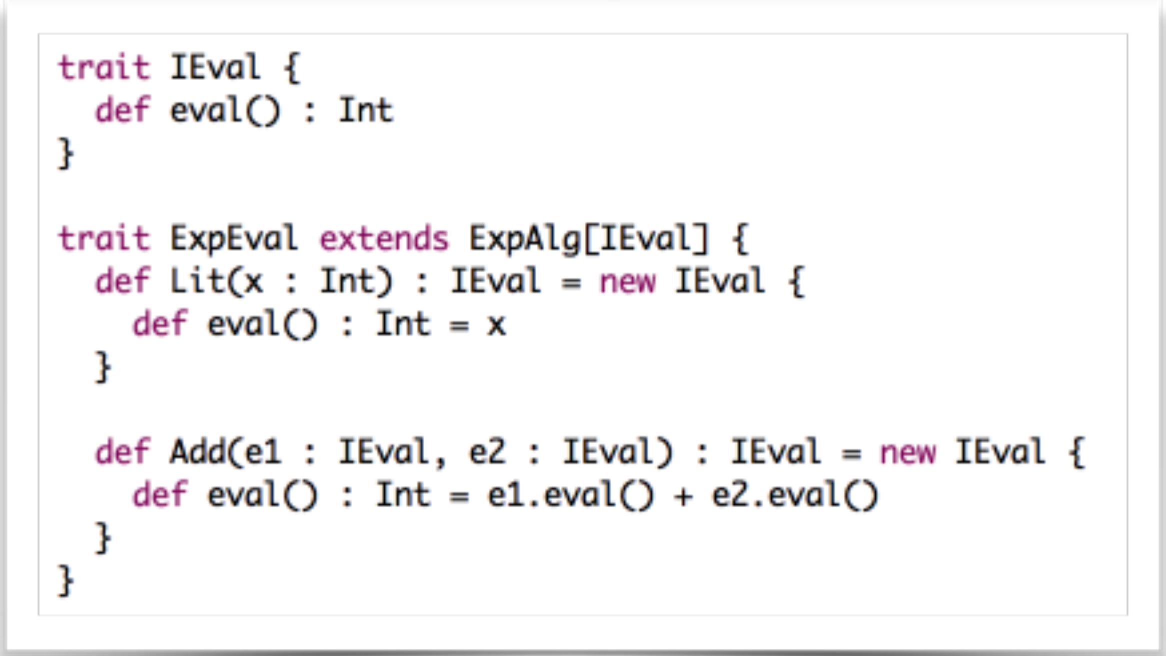
```
eval = (x, y) => reg1.eval(x, y) || reg2.eval(x, y)
```

OA Review

Integer Expressions

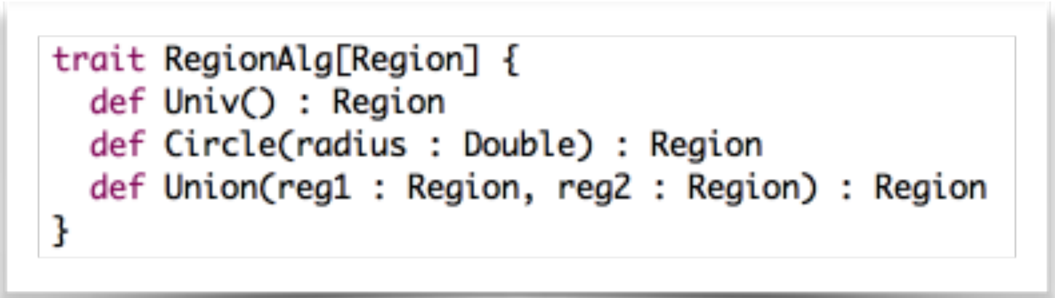


```
trait ExpAlg[Exp] {
  def Lit(x : Int) : Exp
  def Add(e1 : Exp, e2 : Exp) : Exp
}
```

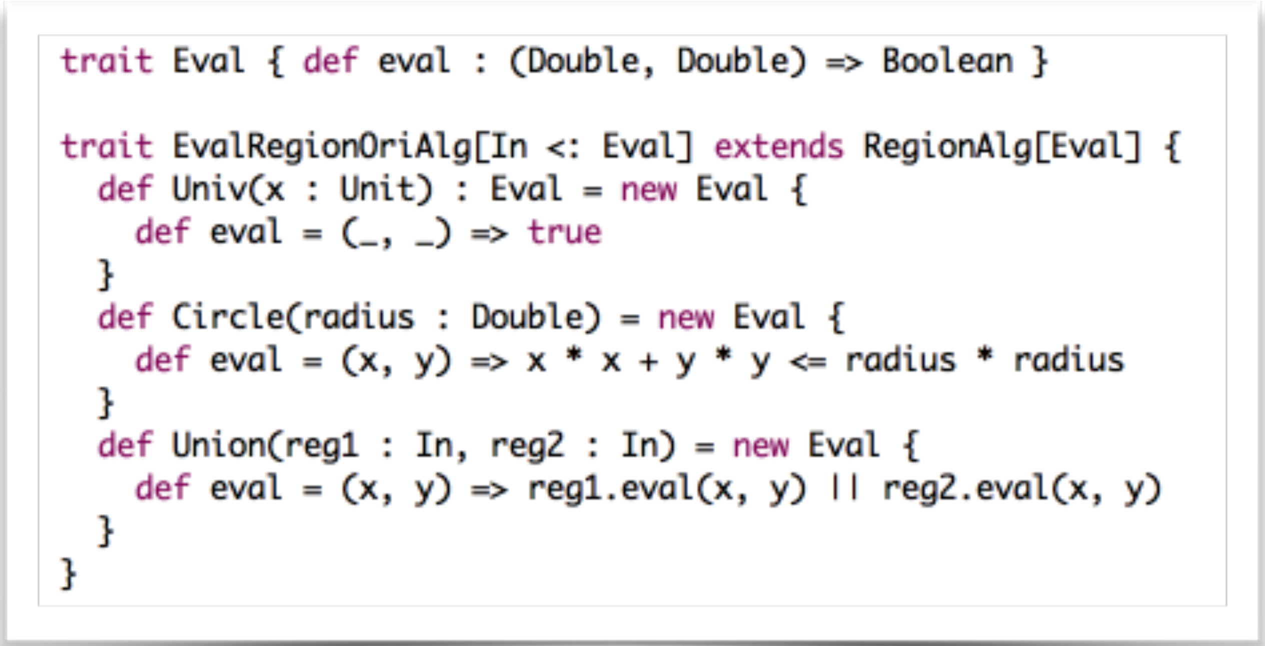


```
trait IEval {
  def eval() : Int
}

trait ExpEval extends ExpAlg[IEval] {
  def Lit(x : Int) : IEval = new IEval {
    def eval() : Int = x
  }

  def Add(e1 : IEval, e2 : IEval) : IEval = new IEval {
    def eval() : Int = e1.eval() + e2.eval()
  }
}
```

Regions



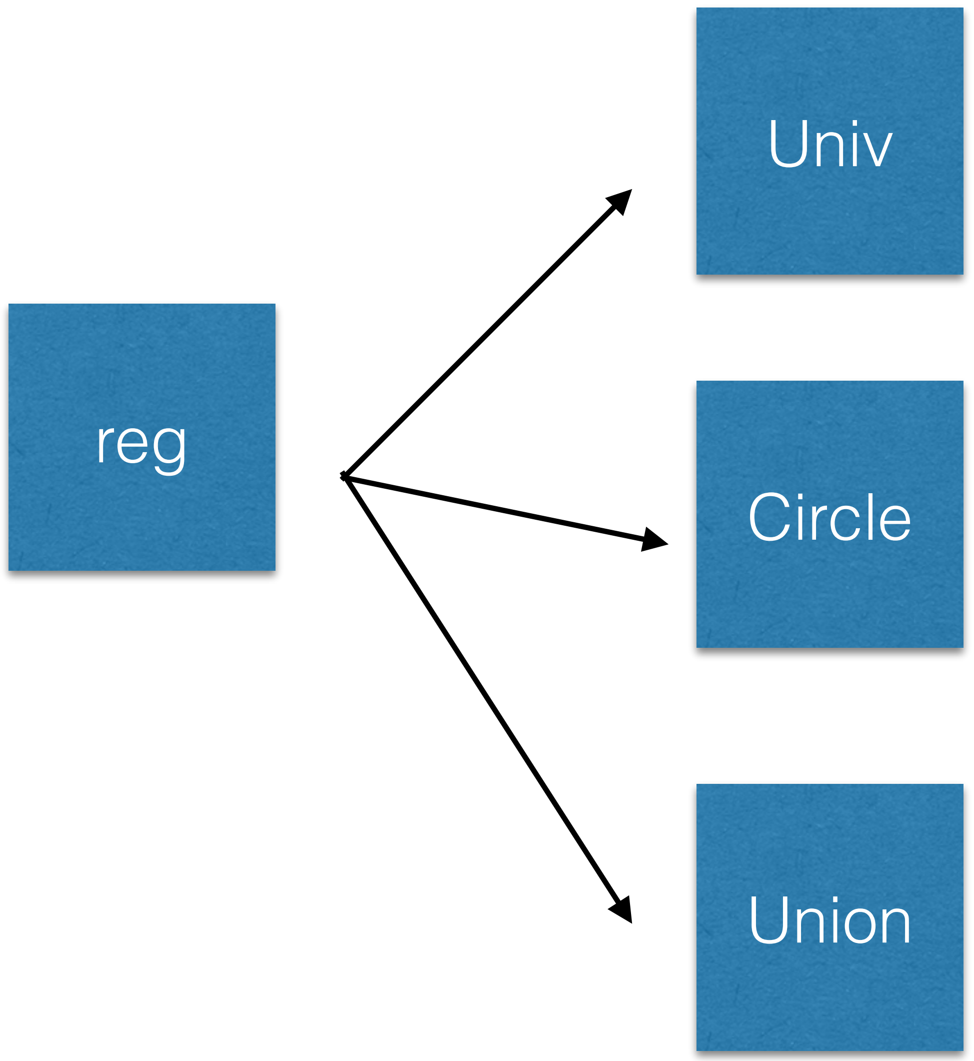
```
trait RegionAlg[Region] {
  def Univ() : Region
  def Circle(radius : Double) : Region
  def Union(reg1 : Region, reg2 : Region) : Region
}
```



```
trait Eval { def eval : (Double, Double) => Boolean }

trait EvalRegionOriAlg[In <: Eval] extends RegionAlg[Eval] {
  def Univ(x : Unit) : Eval = new Eval {
    def eval = (_, _) => true
  }
  def Circle(radius : Double) = new Eval {
    def eval = (x, y) => x * x + y * y <= radius * radius
  }
  def Union(reg1 : In, reg2 : In) = new Eval {
    def eval = (x, y) => reg1.eval(x, y) || reg2.eval(x, y)
  }
}
```

Optimization
- Where pattern matching is required

```
eval : (Double, Double) => Boolean
```



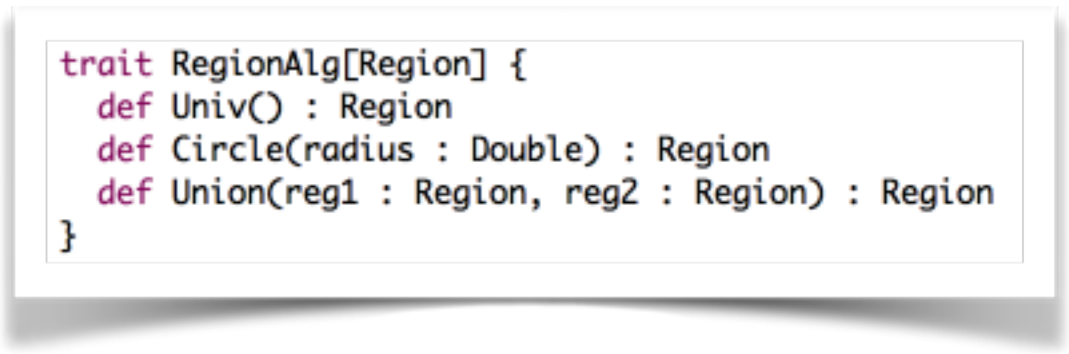
eval = (_, _) => true

```
eval = (x, y) => x * x + y * y <= radius * radius
```

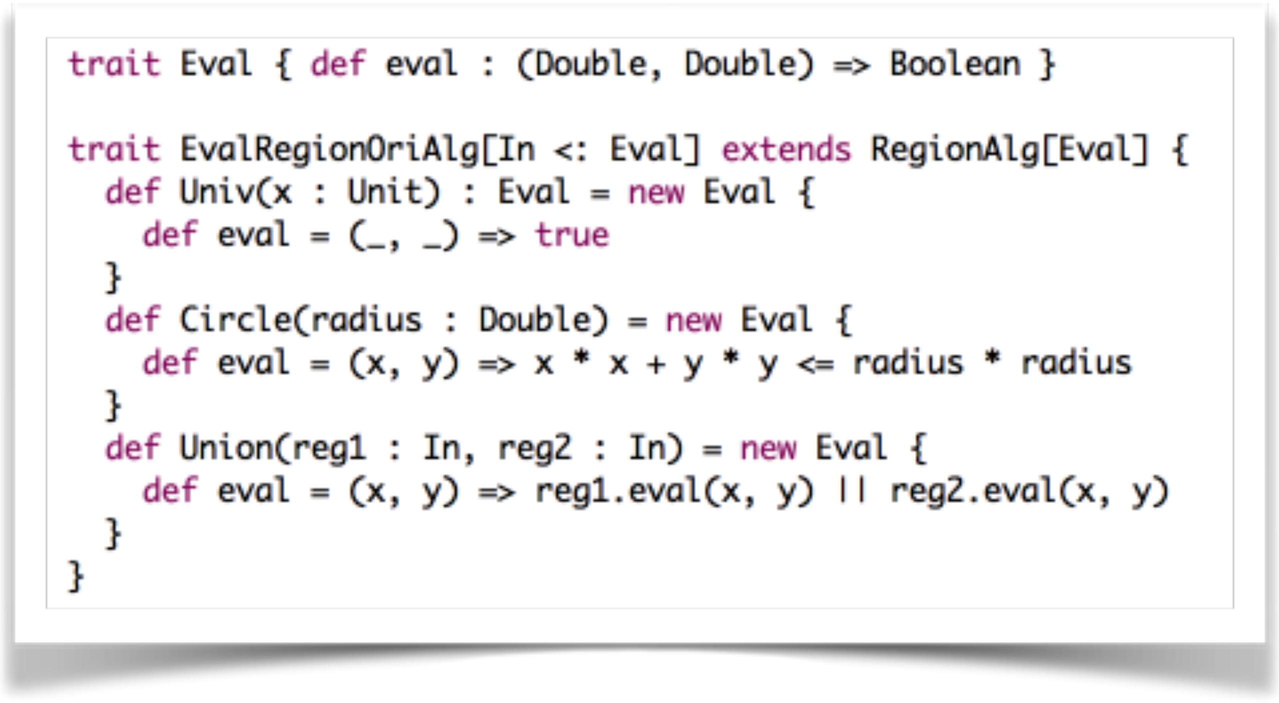
~~`eval = (x, y) => reg1.eval(x, y) || reg2.eval(x, y)`~~

Do pattern matching on reg1, reg2:
If (reg1 is a Univ) || (reg2 is a Univ)
Then true
Else reg1.eval(x, y) || reg2.eval(x, y)

But



```scala
trait RegionAlg[Region] {
  def Univ() : Region
  def Circle(radius : Double) : Region
  def Union(reg1 : Region, reg2 : Region) : Region
}
```

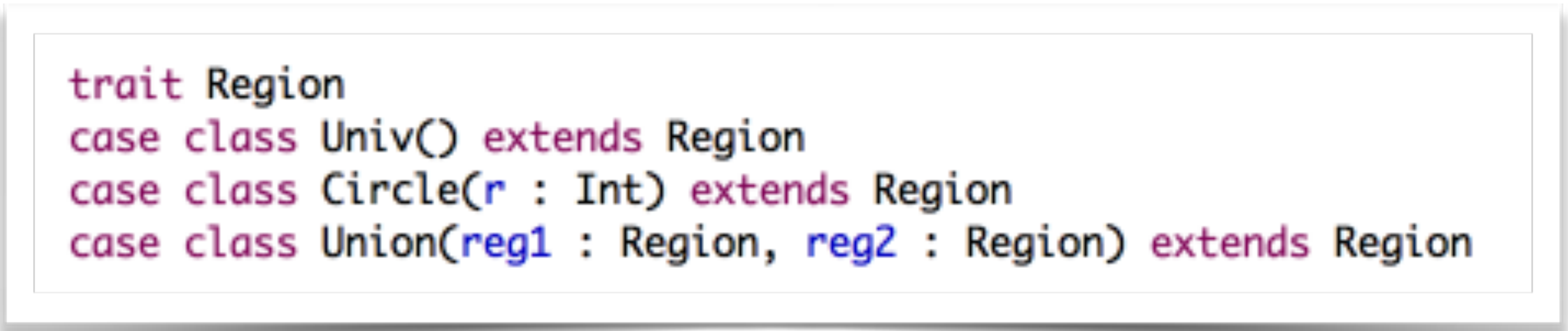


```scala
trait Eval { def eval : (Double, Double) => Boolean }

trait EvalRegionOriAlg[In <: Eval] extends RegionAlg[Eval] {
  def Univ(x : Unit) : Eval = new Eval {
    def eval = (_, _) => true
  }
  def Circle(radius : Double) = new Eval {
    def eval = (x, y) => x * x + y * y <= radius * radius
  }
  def Union(reg1 : In, reg2 : In) = new Eval {
    def eval = (x, y) => reg1.eval(x, y) || reg2.eval(x, y)
  }
}
```

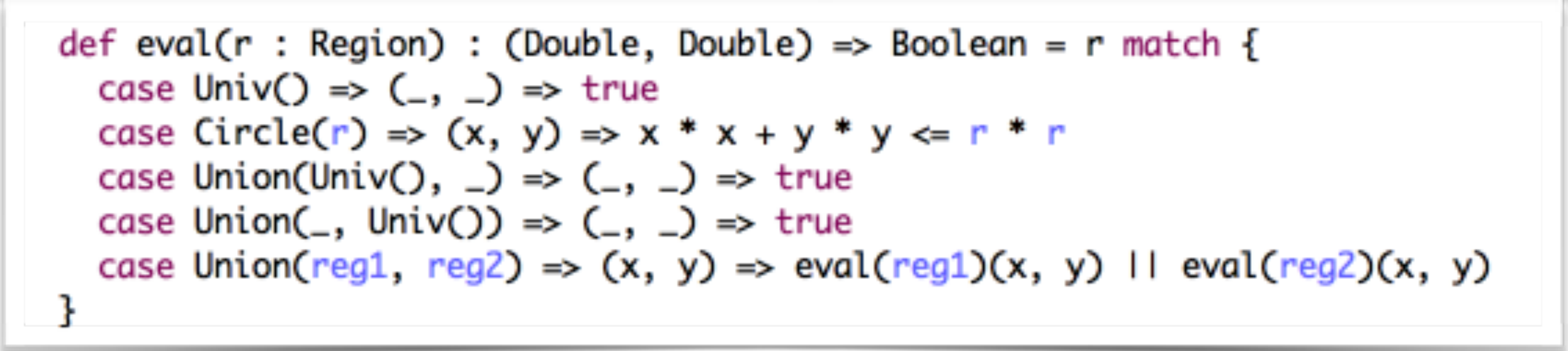
the problem is:
not able pattern match on reg1, reg2.

The only thing we know about reg1, reg2 is :
they have the ‘eval’ method,
but we don’t know what kinds of regions they
are.

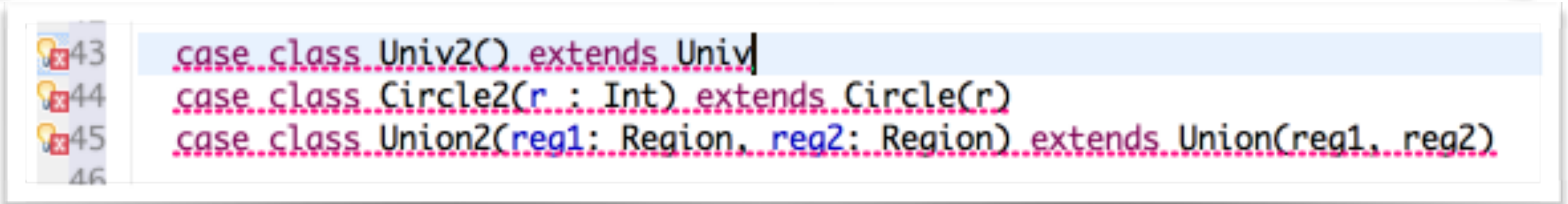
Attempt: case classes



```scala
trait Region
case class Univ() extends Region
case class Circle(r : Int) extends Region
case class Union(reg1 : Region, reg2 : Region) extends Region
```



```scala
def eval(r : Region) : (Double, Double) => Boolean = r match {
  case Univ() => (_, _) => true
  case Circle(r) => (x, y) => x * x + y * y <= r * r
  case Union(Univ(), _) => (_, _) => true
  case Union(_, Univ()) => (_, _) => true
  case Union(reg1, reg2) => (x, y) => eval(reg1)(x, y) || eval(reg2)(x, y)
}
```

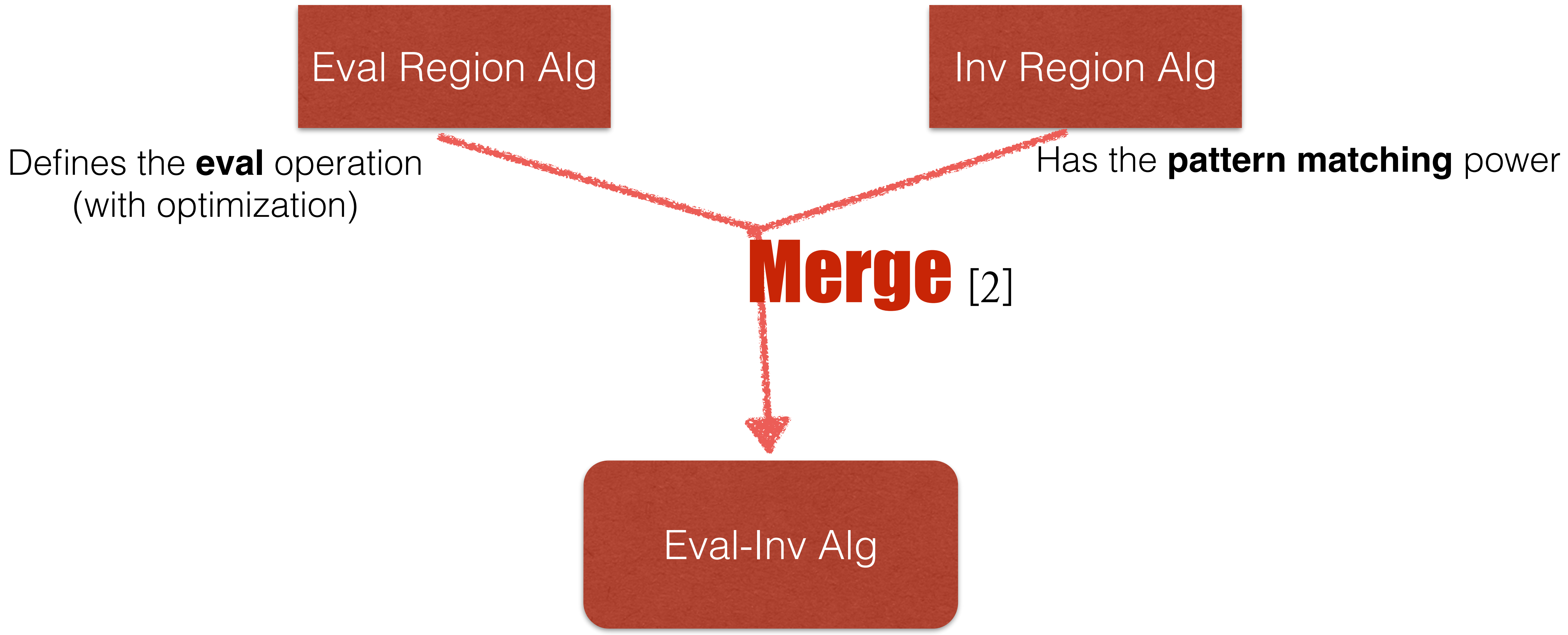


```scala
case class Univ2() extends Univ
case class Circle2(r : Int) extends Circle(r)
case class Union2(reg1: Region, reg2: Region) extends Union(reg1, reg2)
```

- Pattern matching on case classes could be non-exhaustive. May cause runtime error.

- In Scala, case-to-case inheritance is prohibited

Idea


Eval Region Alg
Defines the **eval** operation (with optimization)
Inv Region Alg
Has the **pattern matching** power
Merge [2]
Eval-Inv Alg

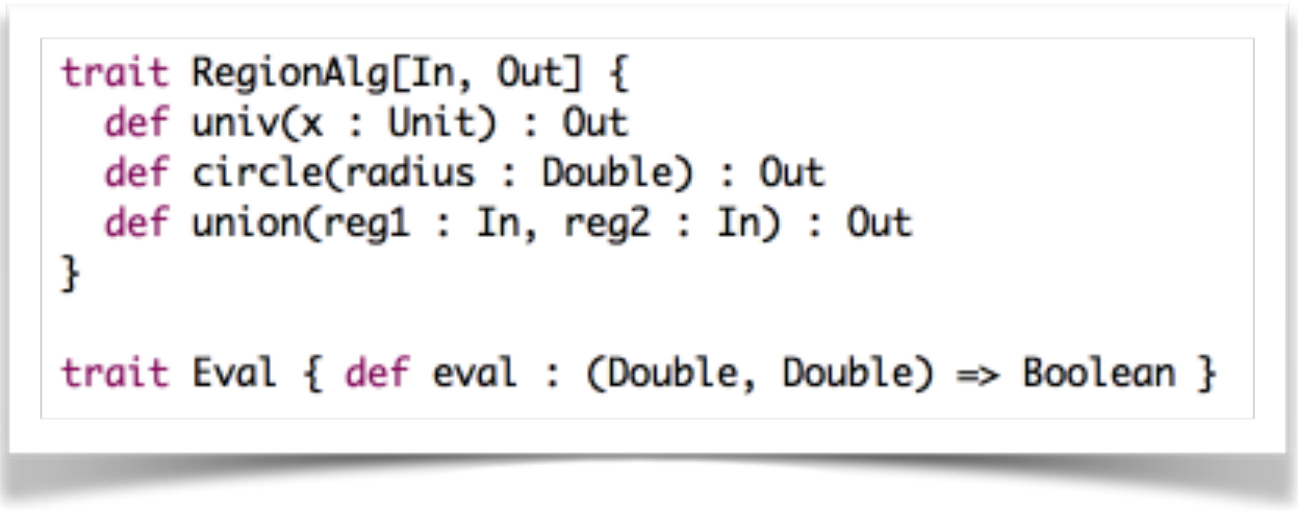
Use pattern patching in the eval operation

Solution

~~RegionAlg[E]~~

RegionAlg[In, Out]

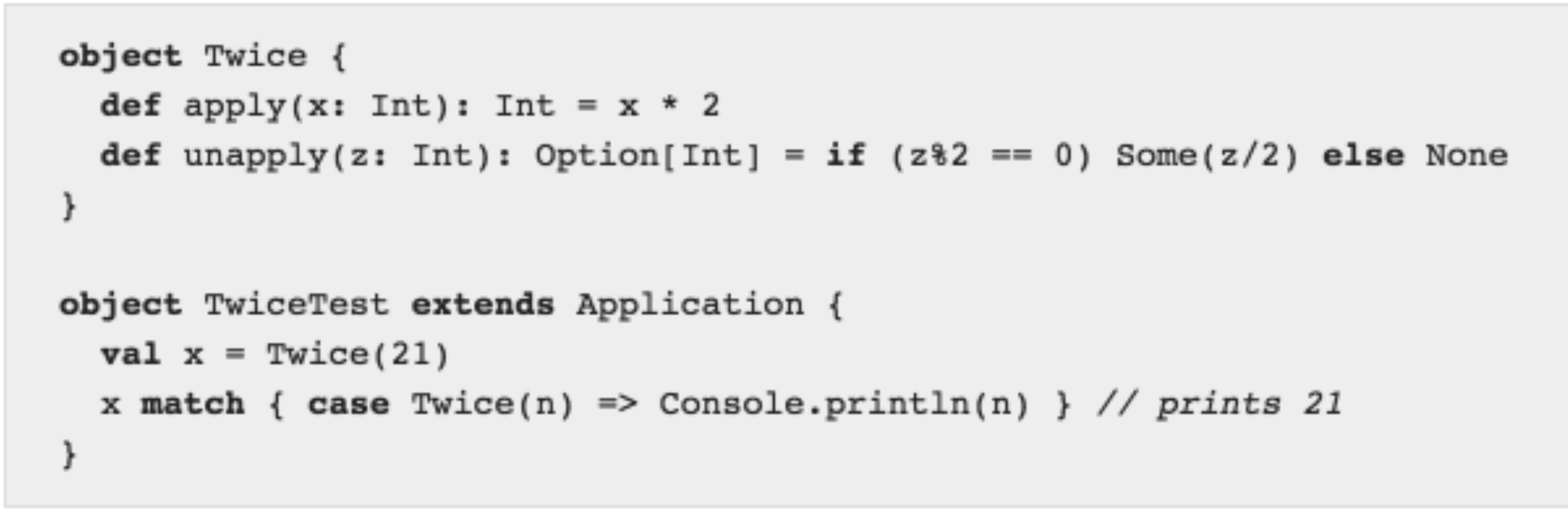
Separate types in argument & return positions



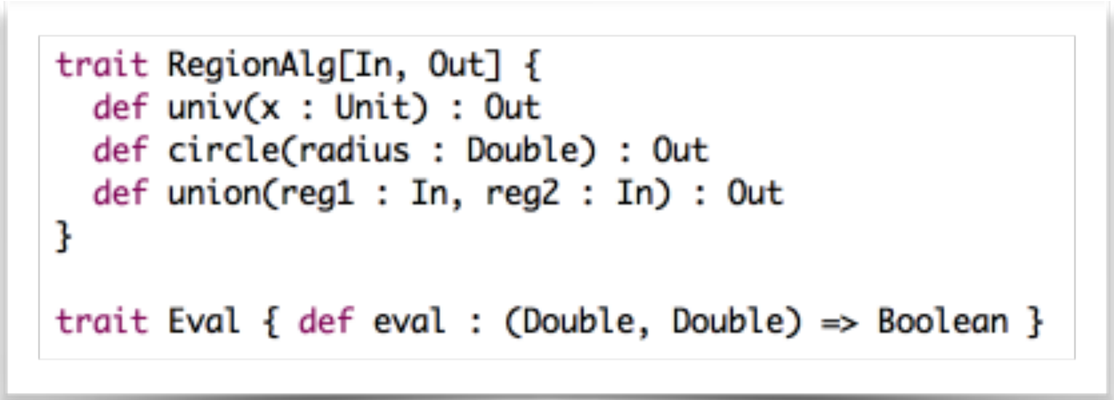
```
trait RegionAlg[In, Out] {
  def univ(x : Unit) : Out
  def circle(radius : Double) : Out
  def union(reg1 : In, reg2 : In) : Out
}

trait Eval { def eval : (Double, Double) => Boolean }
```

Scala ***Option***s Example:[3]

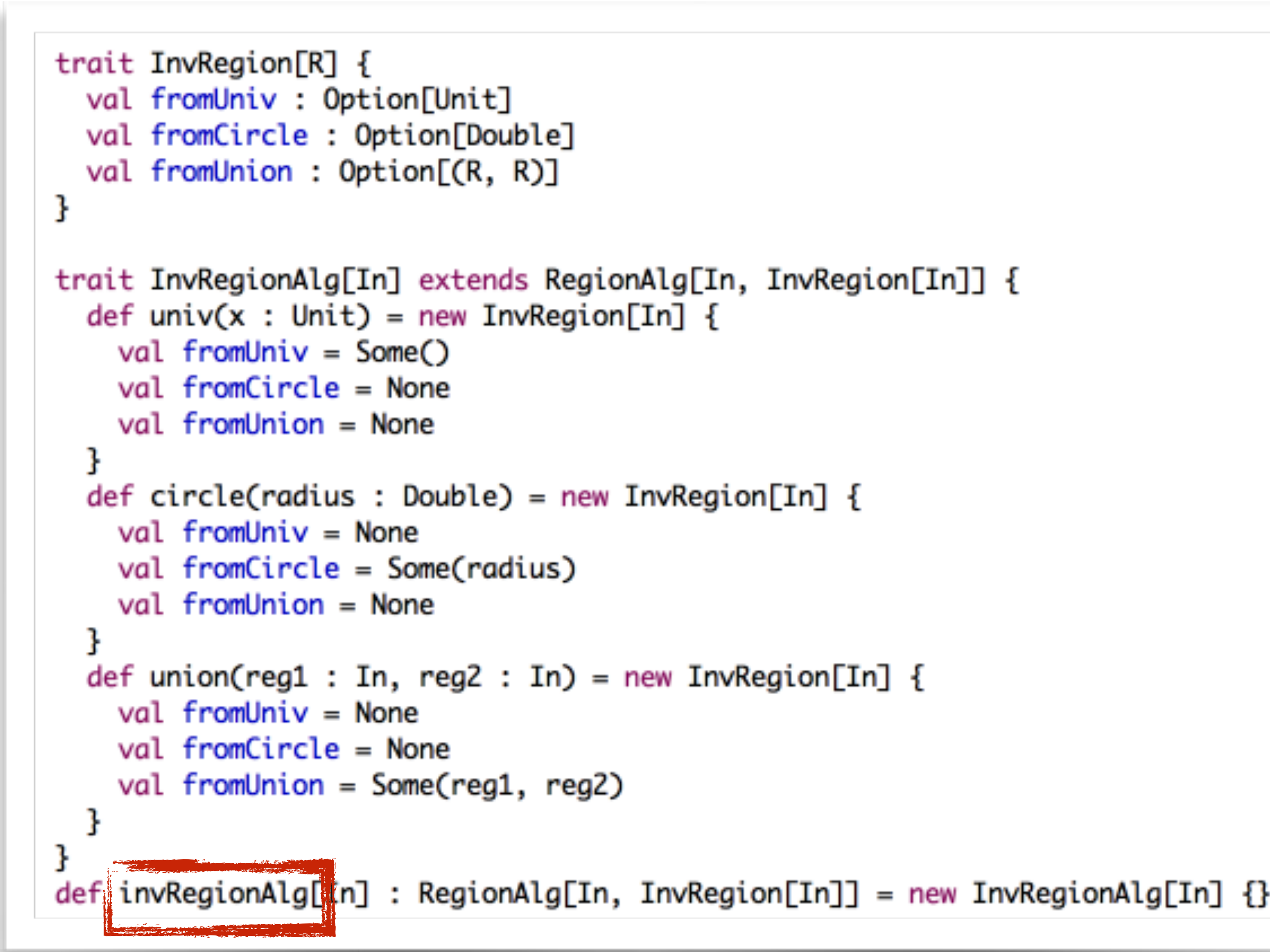
Allow for pattern matching!



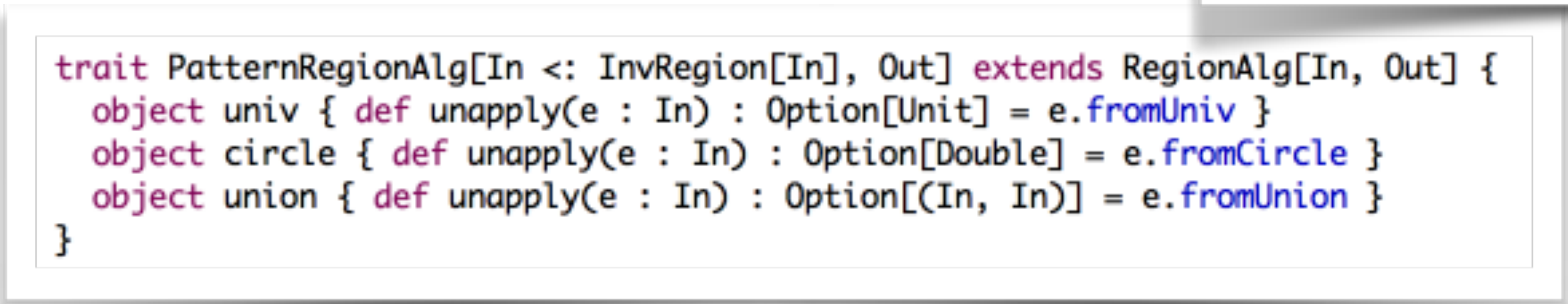
```
object Twice {
  def apply(x: Int): Int = x * 2
  def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
}

object TwiceTest extends Application {
  val x = Twice(21)
  x match { case Twice(n) => Console.println(n) } // prints 21
}
```



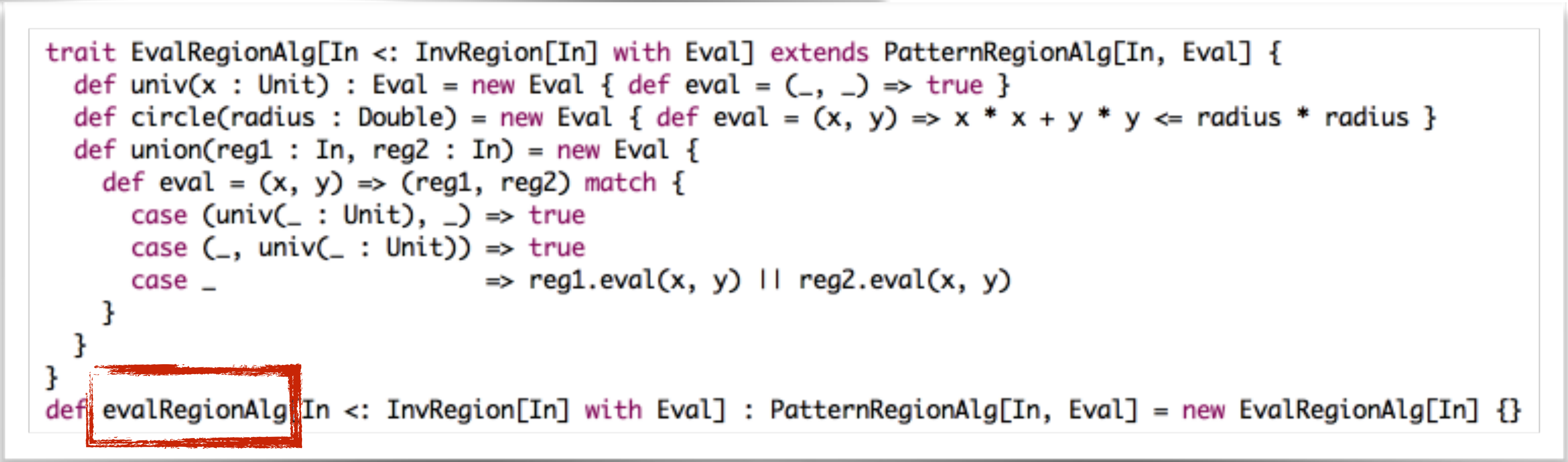
```scala
trait RegionAlg[In, Out] {
  def univ(x : Unit) : Out
  def circle(radius : Double) : Out
  def union(reg1 : In, reg2 : In) : Out
}

trait Eval { def eval : (Double, Double) => Boolean }
```



```scala
trait InvRegion[R] {
  val fromUniv : Option[Unit]
  val fromCircle : Option[Double]
  val fromUnion : Option[(R, R)]
}

trait InvRegionAlg[In] extends RegionAlg[In, InvRegion[In]] {
  def univ(x : Unit) = new InvRegion[In] {
    val fromUniv = Some()
    val fromCircle = None
    val fromUnion = None
  }
  def circle(radius : Double) = new InvRegion[In] {
    val fromUniv = None
    val fromCircle = Some(radius)
    val fromUnion = None
  }
  def union(reg1 : In, reg2 : In) = new InvRegion[In] {
    val fromUniv = None
    val fromCircle = None
    val fromUnion = Some(reg1, reg2)
  }
}
def invRegionAlg[In] : RegionAlg[In, InvRegion[In]] = new InvRegionAlg[In] {}
```



```scala
trait PatternRegionAlg[In <: InvRegion[In], Out] extends RegionAlg[In, Out] {
  object univ { def unapply(e : In) : Option[Unit] = e.fromUniv }
  object circle { def unapply(e : In) : Option[Double] = e.fromCircle }
  object union { def unapply(e : In) : Option[(In, In)] = e.fromUnion }
}
```



```scala
trait EvalRegionAlg[In <: InvRegion[In] with Eval] extends PatternRegionAlg[In, Eval] {
  def univ(x : Unit) : Eval = new Eval { def eval = (_, _) => true }
  def circle(radius : Double) = new Eval { def eval = (x, y) => x * x + y * y <= radius * radius }
  def union(reg1 : In, reg2 : In) = new Eval {
    def eval = (x, y) => (reg1, reg2) match {
      case (univ(_ : Unit), _) => true
      case (_, univ(_ : Unit)) => true
      case _                   => reg1.eval(x, y) || reg2.eval(x, y)
    }
  }
}
def evalRegionAlg[In <: InvRegion[In] with Eval] : PatternRegionAlg[In, Eval] = new EvalRegionAlg[In] {}
```

```
invRegionAlg[In] : RegionAlg[In, InvRegion[In]]
evalRegionAlg[In <: InvRegion[In] with Eval] : RegionAlg[In, Eval]
```

$$\Big\downarrow \textbf{combine}$$

```
evalInvAlg : RegionAlg[EvalInv, InvRegion[EvalInv] with Eval]
```

```
def o = makeRegion(closeS(evalInvAlg))

println("Is (0.5,0.5) inside it? " + o.eval(0.5, 0.5))
```

```
def combine[A, B, S <: A with B]
               (alg1 : F[S, A], alg2 : F[S, B])
               : F[S, A with B]

def closeS[A, B, S <: A with B]
               (alg : F[S, A with B])
               : F[S, S]
```

```
def makeRegion[R](alg : RegionAlg[R, R]) = { import alg._; union(circle(1.0), circle(1.0)) }

trait EvalInv extends Eval with InvRegion[EvalInv]
```

Contribution

- The technique to support pattern matching in OAs

Future Work

- More abstraction
- Fix the library

References

1. Hofer, C., Ostermann, K.: Modular domain-specific language components in scala.

2. Oliveira, B.C.d.S., van der Storm, T., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras.

3. Scala Option tutorial: http://www.scala-lang.org/old/node/112

Q&A