

# Categories as type classes in the Scala Algebra System

Raphaël Jolly  
Databeans

CASC 2013  
Berlin

- \* The Scala Algebra System
- \* Categorical view of computer algebra
- \* Bounded polymorphism
- \* Type classes

- \* polynomial arithmetic over various base rings (integer, rational, complex, modular)
- \* rational and algebraic functions
- \* ring modules/algebras
- \* polynomial GCD
- \* Gröbner bases

## Trade-off

4 / 31

- \* type-safe
- \* generic
- \* mathematical syntax
- \* efficient ?

- \* 2003 : jscl-meditor
- \* 2004 : jscl was chosen as symbolic engine of GeoGebra
- \* 2007 : encounter with JAS and parametric polymorphism (Java 2.5)
- \* 2008 : port to Scala -> ScAS
- \* 2012 : ScAS 2.0 (type classes)
- \* 2013 : scripting support in Scala 2.11

1. arithmetic operations between elements are restricted based on their **domain**
2. domains must support abstraction by means of **categories**
3. categories must support multiple inheritance
4. (optional) categories may support default definition of operations

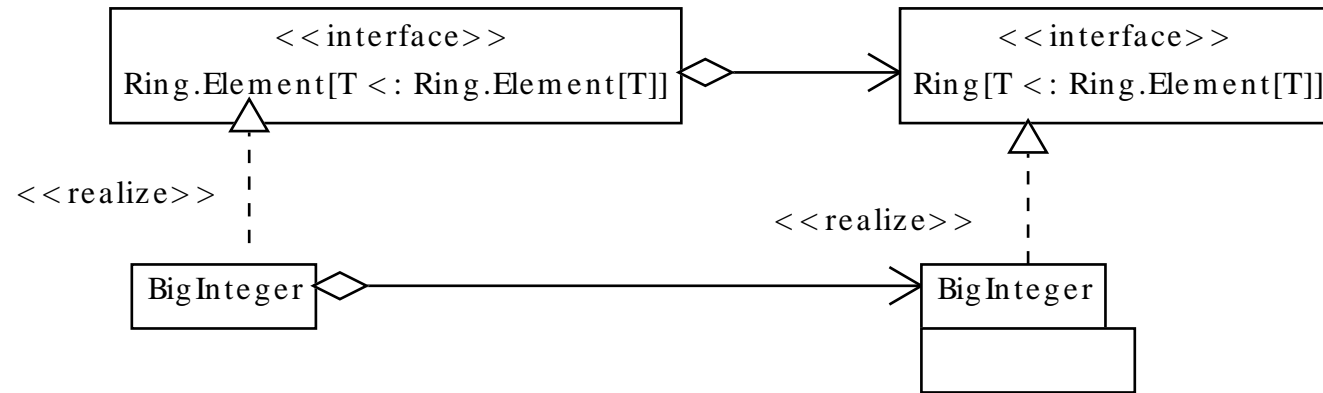
- \* Axiom [Davenport:1992]
- \* Gauss [Gruntz:1994]
- \* JAS [Kredel:2006]
- \* DoCon [Mechveliani:2001]
- \* Mathemagix [VanDerHoeven:2002]

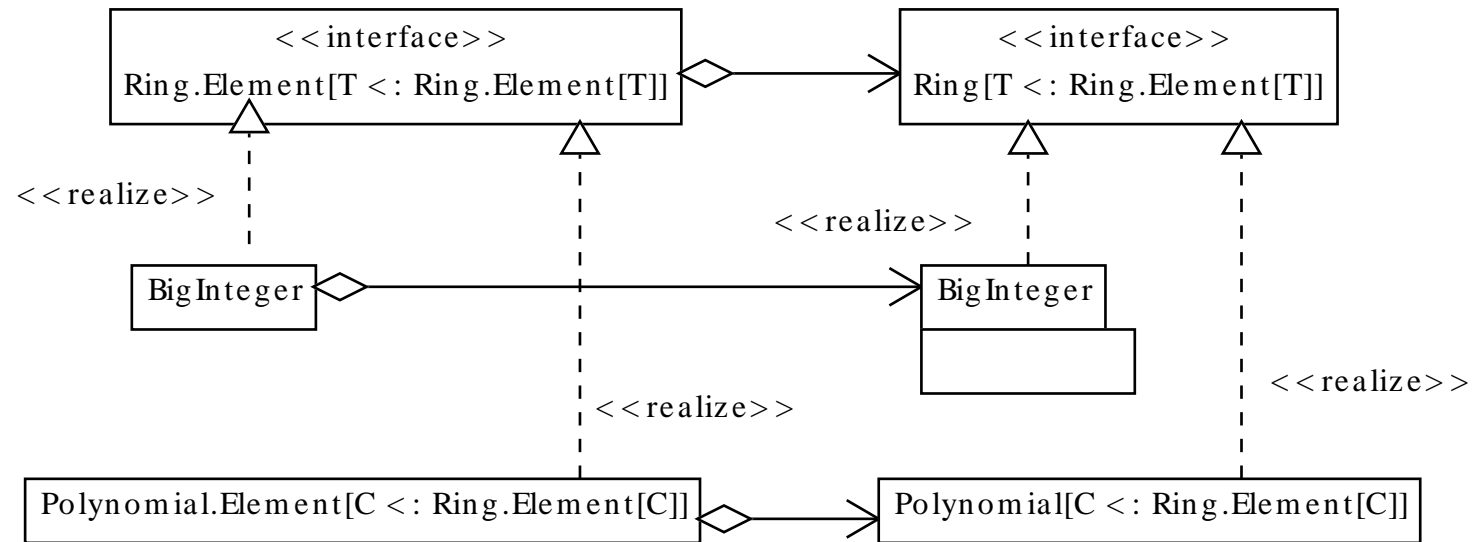
```
trait Ring[T <: Ring.Element[T]] {  
  def zero: T  
  ...  
}  
object Ring {  
  trait Element[T <: Element[T]] {  
    val factory: Ring[T]  
    def +(that: T): T  
    def unary_-: T  
    ...  
  }  
}
```



```
object BigInteger extends Ring[BigInteger] {
  def apply(i: Int): BigInteger =
    apply(java.math.BigInteger.valueOf(i))
  def apply(value: java.math.BigInteger) =
    new BigInteger(value)
  def zero = apply(0)
}
class BigInteger(val value: java.math.BigInteger)
  extends Ring.Element[BigInteger] {
  val factory = BigInteger
  def +(that: BigInteger) = factory(
    this.value.add(that.value))
  def unary_- = factory(value.negate())
  override def toString = value.toString
}
BigInteger(1) + BigInteger(1)
// 2
```

```
class Polynomial[C <: Ring.Element[C]](val ring: Ring[C],
    val variable: String) extends Ring[
    Polynomial.Element[C]] {
  def generator = new Polynomial.Element(...)(this)
}
object Polynomial {
  def apply[C <: Ring.Element[C]](ring: Ring[C],
    variable: String) = new Polynomial(ring, variable)
  class Element[C <: Ring.Element[C]](val value: Repr[C])(
    val factory: Polynomial[C]) extends Ring.Element[
    Element[C]]
}
val r = Polynomial(BigInteger, "x")
val x = r.generator
x + x
// 2*x
```





```
trait Ring[T] { outer =>
  def zero: T
  def plus(x: T, y: T): T
  implicit def mkOps(value: T): Ring.Ops[T] = new Ring.Ops[T] {
    val lhs = value
    val factory = outer
  }
}
object Ring {
  trait ExtraImplicits {
    implicit def infixRingOps[T: Ring](lhs: T) =
      implicitly[Ring[T]].mkOps(lhs)
  }
  trait Ops[T] {
    val lhs: T
    val factory: Ring[T]
    def +(rhs: T) = factory.plus(lhs, rhs)
  }
}
```

```
type BigInteger = java.math.BigInteger
object BigInteger extends Ring[BigInteger] {
  def apply(i: Int) = java.math.BigInteger.valueOf(i)
  def zero = apply(0)
  def plus(x: BigInteger, y: BigInteger) = x.add(y)
}
trait ExtraImplicits {
  implicit val ZZ = BigInteger
}
object Implicits extends ExtraImplicits
  with Ring.ExtraImplicits

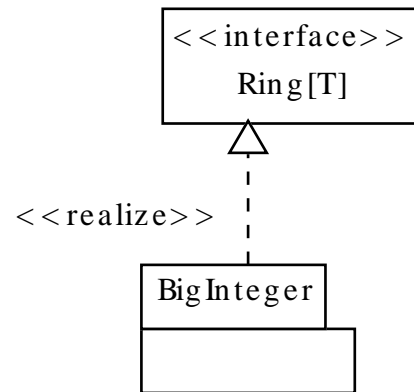
import Implicits.{ZZ, infixRingOps}
BigInteger(1) + BigInteger(1)
// 2
```

```
class Polynomial[C: Ring](val variable: String)
  extends Ring[Repr[C]] {
  val ring = implicitly[Ring[C]]
  def generator: Repr[C] = ...
}
object Polynomial {
  def apply[C: Ring](variable: String) =
    new Polynomial(variable)
}

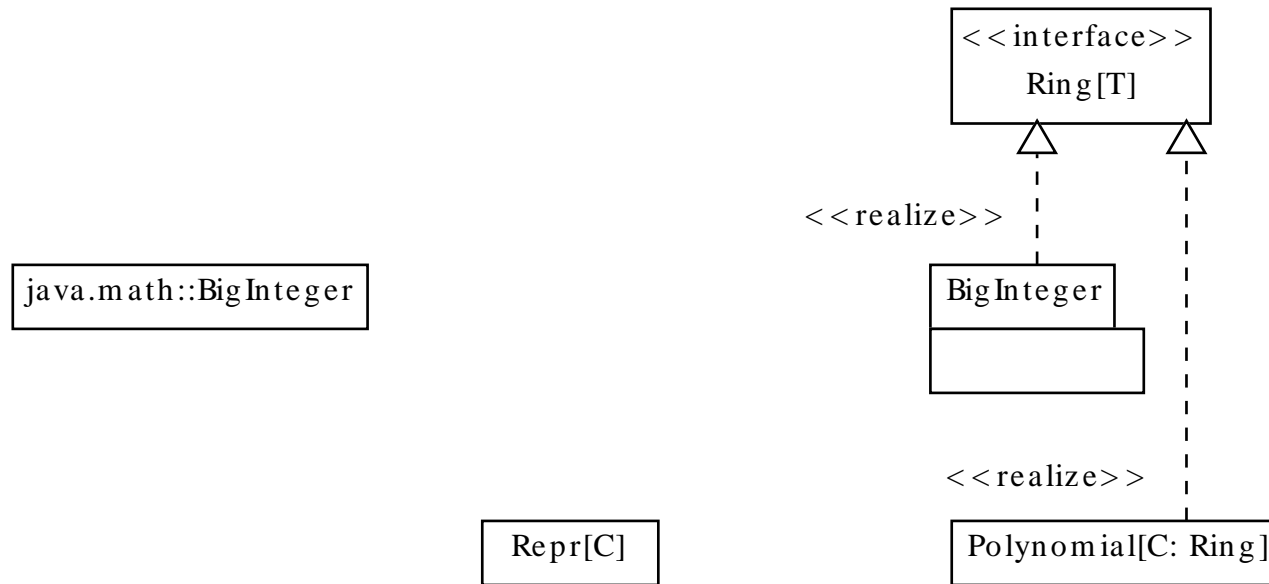
import Implicits.{ZZ, infixRingOps}
implicit val r = Polynomial[BigInteger]("x")
val x = r.generator
x + x
// 2*x
```

# Basic Types : BigInteger

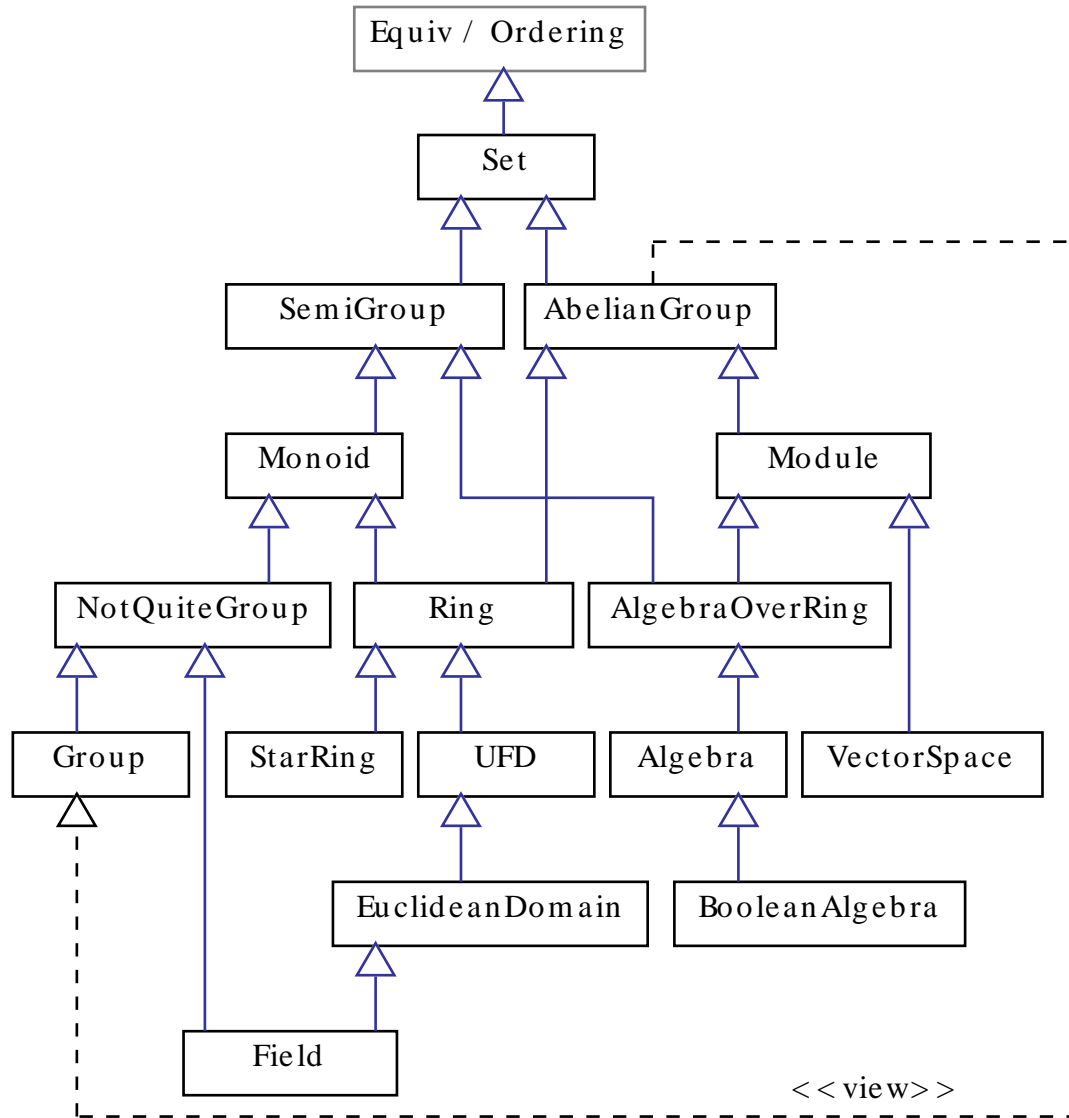
java.math::BigInteger







# Category hierarchy of ScAS



- \* avoid the dependent type problem : type system makes no distinction between types of elements from e.g. `ModInteger(2)` and `ModInteger(3)` but only one implicit value is allowed for the associated domain
- \* reuse of existing types : allows unboxed primitive types => generic numeric-symbolic implementations
- \* makes domain subclassing easier :

```
SolvablePolynomial <: Polynomial
```

```
RealAlgebraicNumber <: AlgebraicNumber
```

```
PolynomialWithSubresGCD <: Polynomial
```

- \* does not play well with coercions

$$\begin{array}{l} \text{BigInteger}(1) + 1 \Rightarrow \text{BigInteger}(1) + \text{BigInteger}(1) \\ x + 1 \qquad \qquad \qquad \Rightarrow x + r(1) \end{array}$$

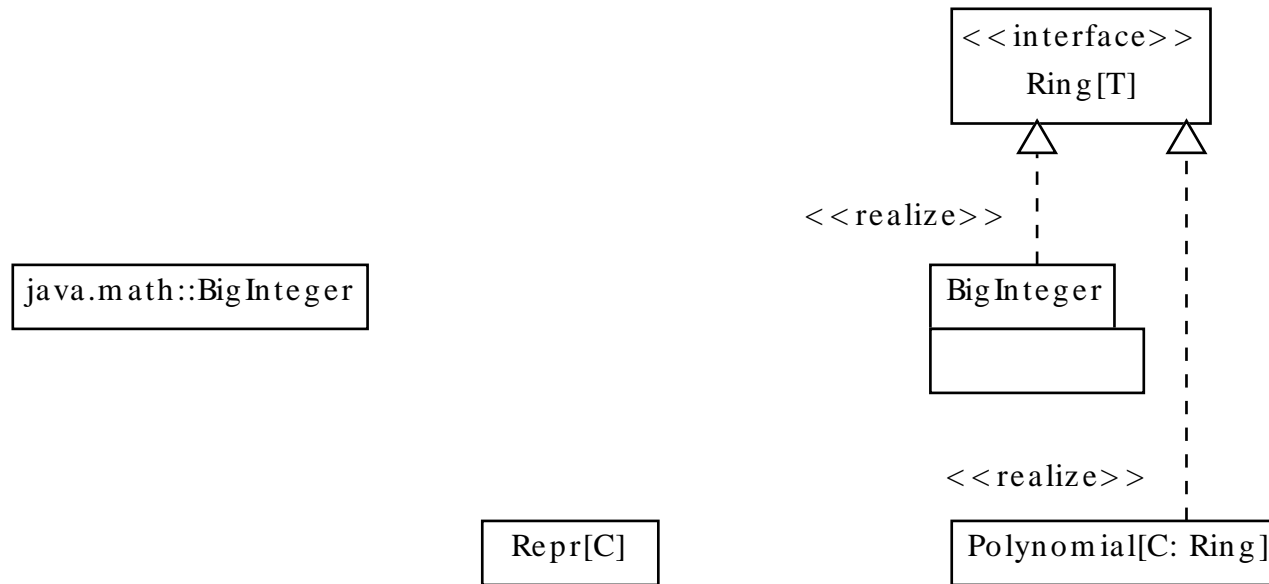
- \* Need an idea to solve this

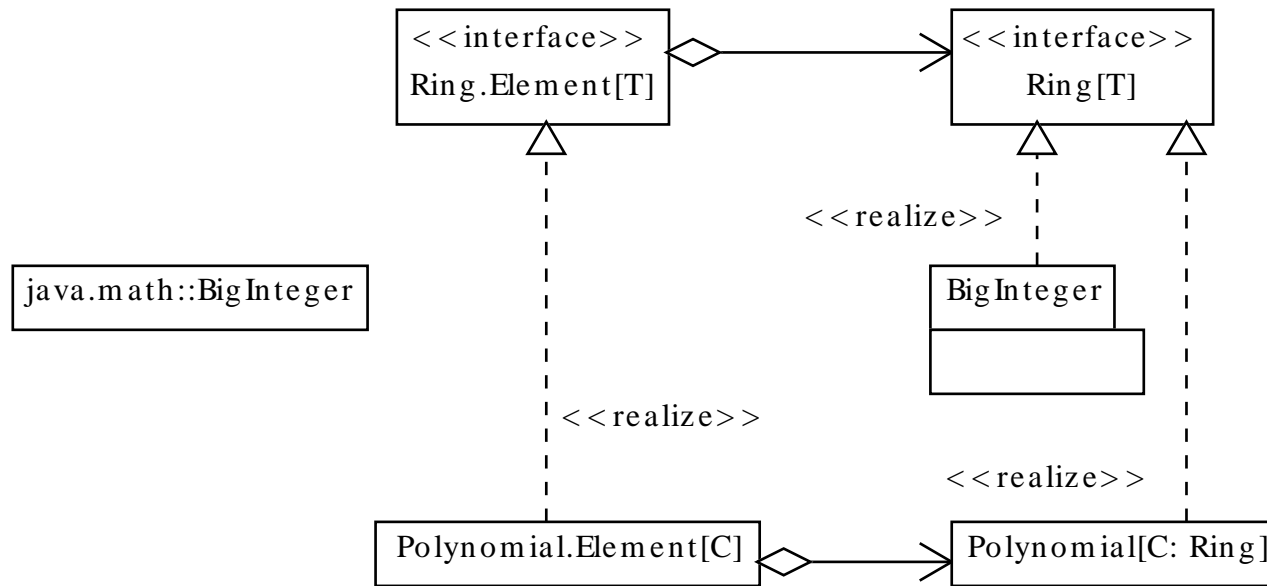
```
object Ring {  
  trait ExtraImplicits {  
    implicit def infixRingOps[T: Ring](lhs: T) =  
      implicitly[Ring[T]].mkOps(lhs)  
  }  
  trait Ops[T] {  
    val lhs: T  
    val factory: Ring[T]  
    def +(rhs: T) = factory.plus(lhs, rhs)  
  }  
  
}
```

```
object Ring {  
  trait ExtraImplicits {  
    implicit def infixRingOps[T: Ring](lhs: T) =  
      implicitly[Ring[T]].mkOps(lhs)  
  }  
  trait Ops[T] {  
    val lhs: T  
    val factory: Ring[T]  
    def +(rhs: T) = factory.plus(lhs, rhs)  
  }  
  trait Element[T <: Element[T]] extends Ops[T] {  
    this: T =>  
    val lhs = this  
  }  
}
```

# Hybrid abstraction scheme

22 / 31







- \* introduced in Haskell [Oliveira:2010]
- \* first mentioned as possible abstractions for computer algebra structures [Weber:1993] [Santas:1995]
- \* DoCon : actually uses them
- \* Scala : a code example in the language documentation explicitly involves abstract algebraic constructs
- \* Mathemagix uses a concept of categories that is completely equivalent to type classes

```
class CommutativeRing a => GCDRing a where
```

```
    gcd : a -> a -> a
```

```
    canAssoc : a -> a
```

```
instance GCDRing Integer where
```

```
    canAssoc n = if n < 0 then -n else n
```

```
    gcd n 0 = n
```

```
    gcd n m = gcd m (mod n m)
```

```
data Fraction a = a :/ a ...
```

```
instance GCDRing a =>
```

```
    AdditiveSemigroup (Fraction a) where
```

```
    (x :/ y) + (x' :/ y') =
```

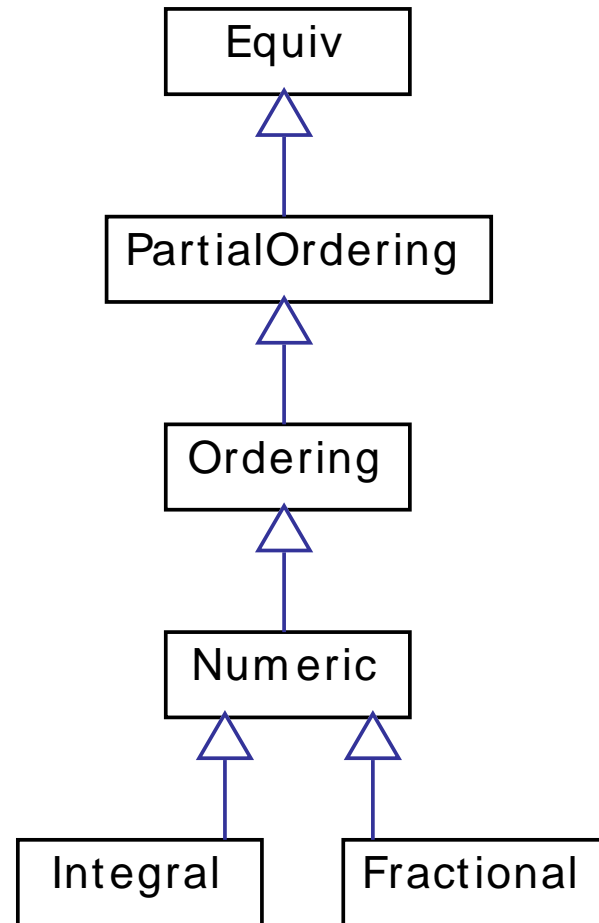
```
        ... usual way to sum fractions
```

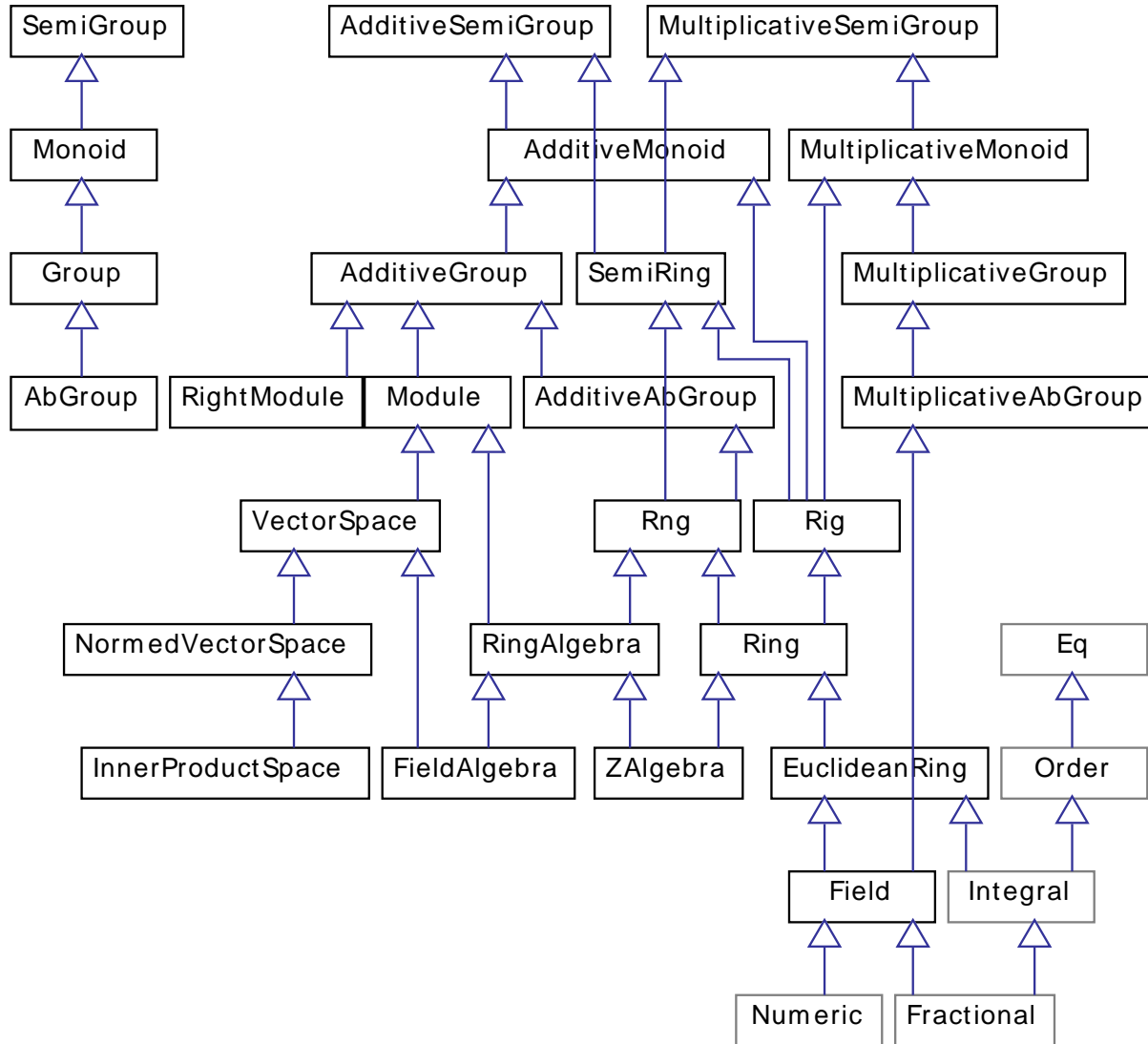
```
instance GCDRing a => Field (Fraction a) where ...
```

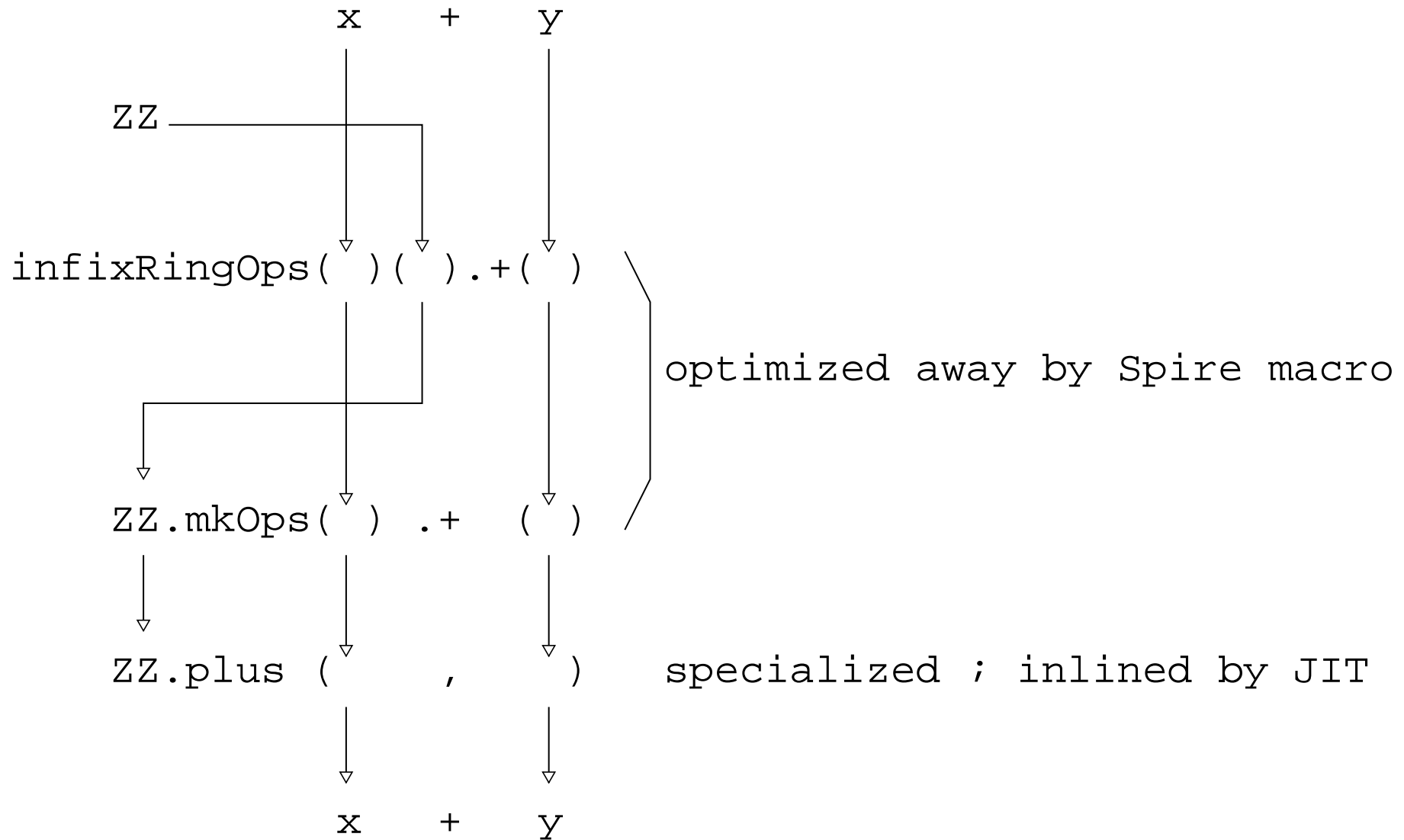
```
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
object ImplicitTest extends App {
  implicit object StringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }
  implicit object IntMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

  println(sum(List(1, 2, 3))) // 6
  println(sum(List("a", "b", "c"))) // abc
}
```

<http://docs.scala-lang.org/tutorials/tour/implicit-parameters.html>

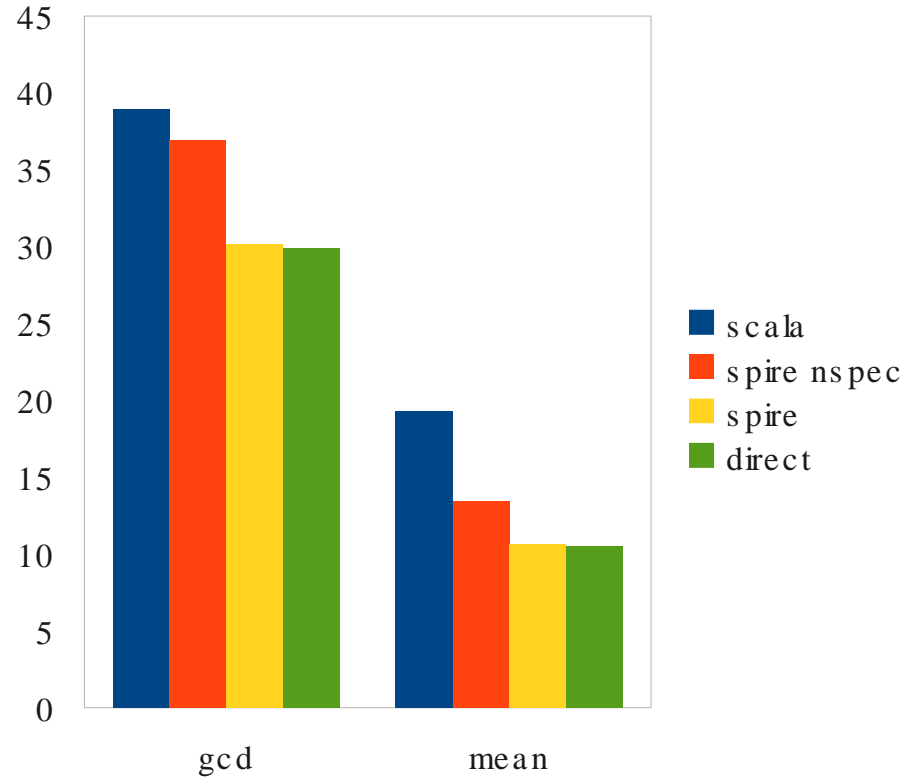






# Spire is Fast

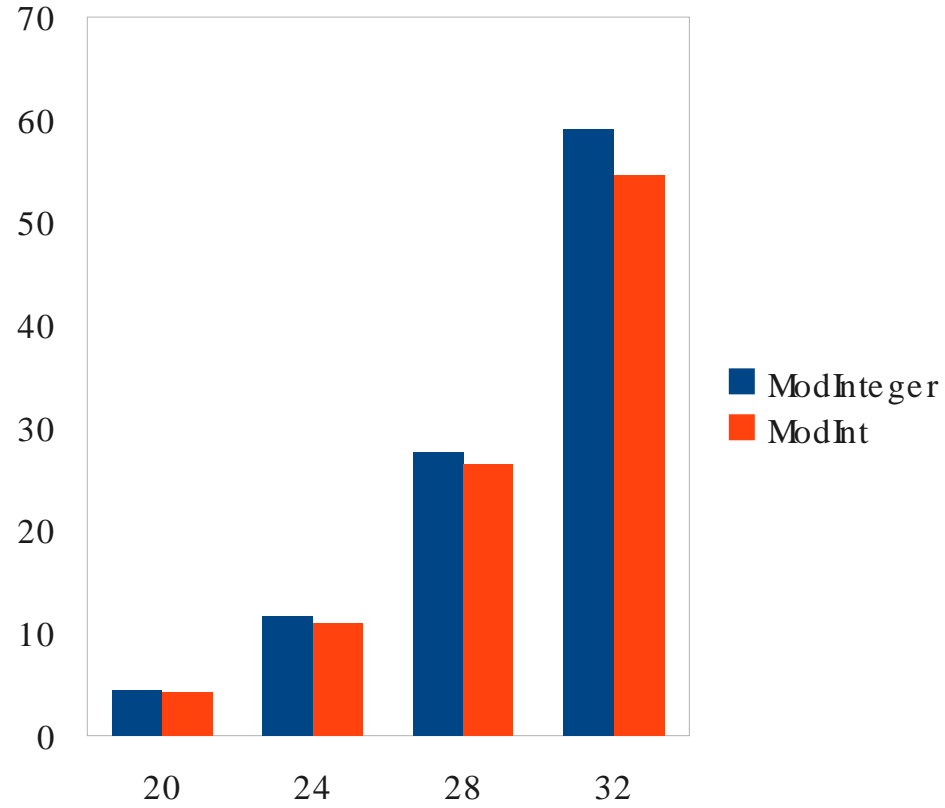
	scala	spire nspec	spire	direct
gcd	38.9	36.9	30.1	29.9
mean	19.3	13.4	10.6	10.5



<http://typelevel.org/blog/2013/07/07/generic-numeric-programming.html>

# ScAS : polypower benchmark

n	ModInteger	ModInt
20	4.4	4.2
24	11.6	10.9
28	27.6	26.4
32	59.1	54.6





Thank you !

<http://github.com/rjolly/scas>