Implicit conversions in the Scala Algebra System

Raphaël Jolly
Databeans

CASC 2024
Rennes

* ScAS (Scala Algebra System) : computer algebra system written in Scala
* Scala:
  - allows to define algebraic categories with type classes and extension methods
  - provides implicit conversions -> difficult to operate in conjunction with the above
  - discussions are on-going about restricting implicit conversions [1]
    -> complicates the subject even further
    -> might bring new opportunities

They might hide type errors
They make type inference less precise and less efficient

[1] https://contributors.scala-lang.org/t/proposed-changes-and-restrictions-for-implicit-conversions/4923

Static types

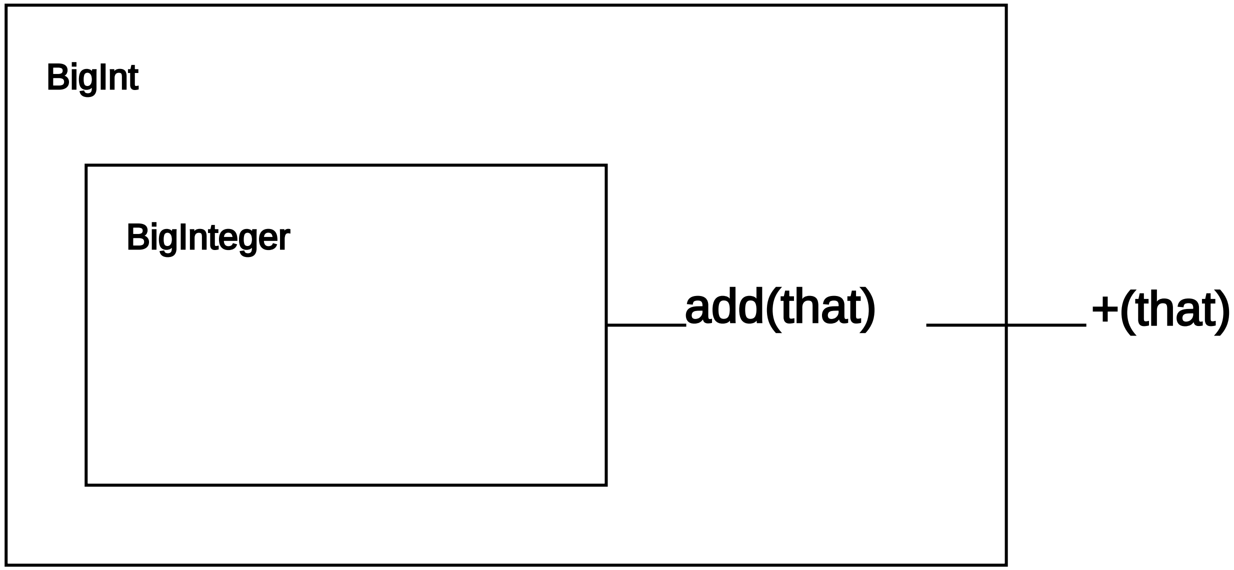
```
val a: Apple
val b: Orange

a+b // compile error
```

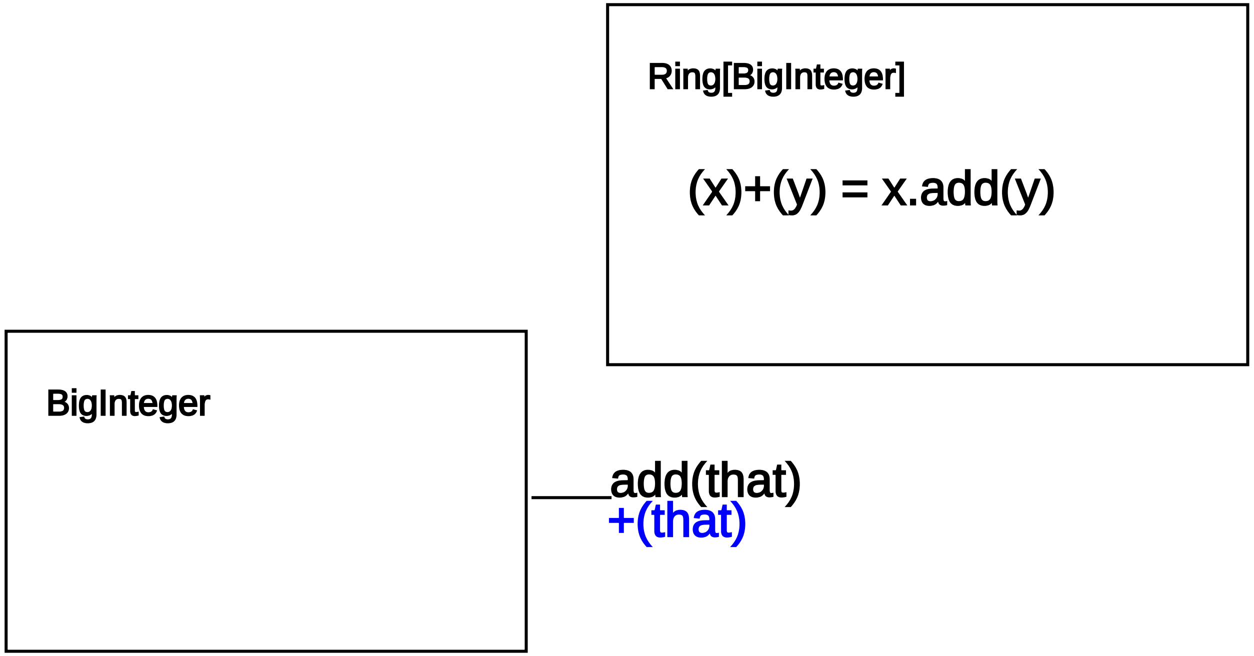
p in $\mathbb{Z}[x]$
1 also in $\mathbb{Z}[x]$

```
p+1 // fails
```

* too strict
  -> we need a mechanism to restore flexibility
  - numeric promotion
    -> already exists for build-in types (e.g. int/long)
    -> custom types : implicit conversion

* custom types
   - example : multiprecision arithmetic
   - Object oriented languages are well suited (Java)
     -> java.math.BigInteger
   - syntax is verbose, method names are literal "add"
   - two more features have to be found elsewhere
     - Enrichement : to endow a numeric type with arithmetic operators
     - numeric promotion : to lift an operand value from a subset type to the type of the other operand
   - two approaches to enrichment
     - objet-oriented (wrapper)
     - functional (type classes)


BigInt
BigInteger
add(that)
+(that)


Ring[BigInteger]
(x)+(y) = x.add(y)
BigInteger
add(that)
+(that)

```
val a = BigInt(1)

// actual type is Int
// expected type is BigInt
// conversion looked up in implicit scope of Int => BigInt
// includes BigInt's companion object
val b = a + 1

// implicit scope of Int => { def +(arg: BigInt): U }
// again BigInt's companion
val c = 1 + a

object BigInt:
  implicit def int2bigInt(i: Int): BigInt = apply(i)
```

https://docs.scala-lang.org/scala3/reference/changed-features/implicit-conversions-spec.html

```
import java.math.BigInteger
import scala.language.implicitConversions

trait Ring[T]:
  extension (x: T) def + (y: T): T

given r: Ring[BigInteger] with
  given Conversion[Int, BigInteger] = BigInteger.valueOf(_)
  extension (x: BigInteger) def + (y: BigInteger) =
x.add(y)

import r.given

val a = BigInteger("1")
val b = a + 1 // works
val c = 1 + a // fails
```

```
import java.math.BigInteger
import scala.language.implicitConversions

trait Ring[T]:
  extension (x: T) def + (y: T): T

given r: Ring[BigInteger] with
  given Conversion[Int, BigInteger] = BigInteger.valueOf(_)
  extension (x: BigInteger) def + (y: BigInteger) =
x.add(y)

import r.{given, *}

val a = BigInteger("1")
val b = a + 1 // works
val c = 1 + a // works
```

```
case class MyInt1(n: Int)
case class MyInt2(n: Int)

given r: Ring[MyInt1] with
  given Conversion[Int, MyInt1] = MyInt1(_)
  extension (x: MyInt1) def + (y: MyInt1) = MyInt1(x.n +
y.n)

given s: Ring[MyInt2] with
  given Conversion[Int, MyInt2] = MyInt2(_)
  extension (x: MyInt2) def + (y: MyInt2) = MyInt2(x.n +
y.n)

import r.{given, *}
import s.{given, *}
```

```
MyInt1(1) + 1 // ok
MyInt2(1) + 1 // ok

1 + MyInt1(1) // None of the overloaded alternatives
of method + in class Int with types ... match arguments
(MyInt1)
1 + MyInt2(1) // None of the overloaded alternatives
of method + in class Int with types ... match arguments
(MyInt2)
```

Defining several instances in the same scope (cont2) 11/20

```
1 ++ MyInt1(1)
^^^^
value ++ is not a member of Int.
An extension method was tried, but could not be fully
constructed:

    s.++(s.given_Conversion_Int_MyInt2.apply(1))

    failed with:

        Ambiguous extension methods:
        both s.++(s.given_Conversion_Int_MyInt2.apply(1))
        and  r.++(r.given_Conversion_Int_MyInt1.apply(1))
        are possible expansions of 1.++
```

```
given r: Ring[BigInteger] with
  given Conversion[Int, BigInteger] = BigInteger.valueOf(_)
  extension(x: BigInteger) def + (y: BigInteger) = x.add(y)

case class Poly[C](n: C)

trait PolyRing[C](using ring: Ring[C]) extends
Ring[Poly[C]]:
  given [D](using Conversion[D, C]): Conversion[D, Poly[C]]
= x => Poly(x)
  extension (x: Poly[C]) def + (y: Poly[C]) =
    import ring.*
    Poly(x.n + y.n)

given s: PolyRing[BigInteger] with {}
```

```
import r.{given} // do not import this operator
import s.{given, *} // import this one

val a = BigInteger("1")
val x = Poly(BigInteger("1"))

// everything is lifted to the top ring

val b = a + 1
val c = 1 + a
val d = x + a
val e = a + x
val f = x + 1
val g = 1 + x
```

```
trait Ring[T]:

  extension (x: T)
    def + (y: T): T
```

```
trait Ring[T]:
  extension (x: T) def add(y: T): T
  extension (x: T)
    def + (y: T) = x.add(y)
```

```
trait Ring[T]:
  extension (x: T) def add(y: T): T
  extension (x: into T)
    def + (y: into T) = x.add(y)
```

```
type Conversion[T] = [X] =>> X => T
extension [U](x: U)
  def unary_~[T](using c: U => T) = c(x)

trait Ring[T]:
  extension (x: T) def add(y: T): T
  extension[U: Conversion[T]](x: U)
    def + [V: Conversion[T]](y: V) = (~x).add(~y)
```

```
given r: Ring[BigInteger] with

  given Conversion[Int, BigInteger] = BigInteger.valueOf(_)
  extension (x: BigInteger) def + (y: BigInteger) =
x.add(y)

import r.{given, *}
...
val c = 1 + a // works
```

```
given r: Ring[BigInteger] with

  given (Int => BigInteger) = BigInteger.valueOf(_)
  extension (x: BigInteger) def add(y: BigInteger) =
x.add(y)

import r.given
...
val c = 1 + a // works
```

```
object r extends Ring[BigInteger]:
  given instance: r.type = this
  given (Int => BigInteger) = BigInteger.valueOf(_)
  extension (x: BigInteger) def add(y: BigInteger) =
x.add(y)

import r.given
...
val c = 1 + a // works
```

```
extension (x: Poly[C]) def + (y: Poly[C]) =
  import ring.*
  Poly(x.n + y.n)
```

```
trait Ring[T]:
  extension[U: Conversion[T]](x: U)
    def + [V: Conversion[T]](y: V) = (~x).add(~y)
```

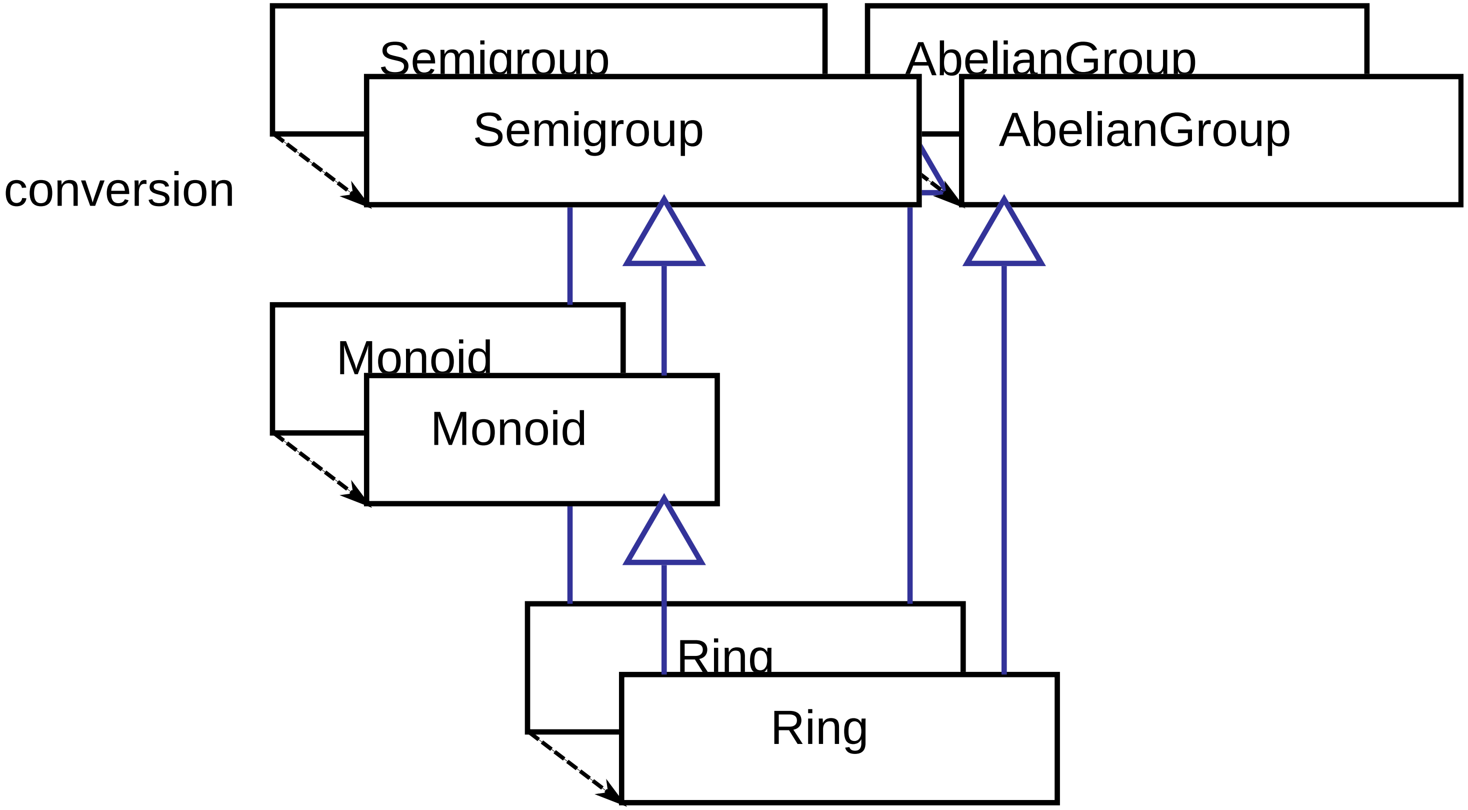
```
extension (x: Poly[C]) def + (y: Poly[C]) =

  Poly(x.n + y.n)
```

```
trait Ring[T]:
  extension(x: T)
    def + [V: Conversion[T]](y: V) = x.add(~y)
```

```
extension (x: Poly[C]) def + (y: Poly[C]) =

  Poly(x.n + y.n)
```

```
trait Ring[T]:
  extension(x: T)
    def + [V: Conversion[T]](y: V) = x.add(~y)
```

```
trait conversion.Ring[T] extends Ring[T]:
  extension[U: Conversion[T]](x: U)
    def + [V: Conversion[T]](y: V) = (~x).add(~y)
```


Semigroup
Semigroup
AbelianGroup
AbelianGroup
conversion
Monoid
Monoid
Ring
Ring

```
object Complex extends Complex.Impl with
conversion.Field[Complex]:
  given instance: Complex.type = this
  class Impl extends AlgebraicNumber(Rational)
(Variable.sqrt(BigInteger("-1"))) with StarUFD[Complex]:
    def real(x: Complex) = x.coefficient(one)
    def imag(x: Complex) = x.coefficient(generator(0))
    override def conjugate(x: Complex) = real(x) - sqrt(-1)
* imag(x)
    override def toString = "Complex"
    override def toMathML = "<complexes/>"
  update(1 + sqrt(-1)\2)
```

Summary

* we want static types (no run-time type cast)
* functional (typeclass based) rather than object oriented
  -> more flexible, efficient
  -> used to mandate operator import
* introduce context bounds approach to implicit conversion
  -> allows to omit the operator import
* given instance as member field of its own typeclass
  -> instances are superseding one another
  -> allows to omit operator imports even in case there are several
* in case of nested algebraic structures:
  -> split the typeclass hierarchy to isolate left-handed promotion
* a lot of contortions
  -> still the price to pay for a nice mathematical notation

Thank you !

https://github.com/rjolly/scas