Computer Algebra in Scala

Raphaël Jolly
Databeans

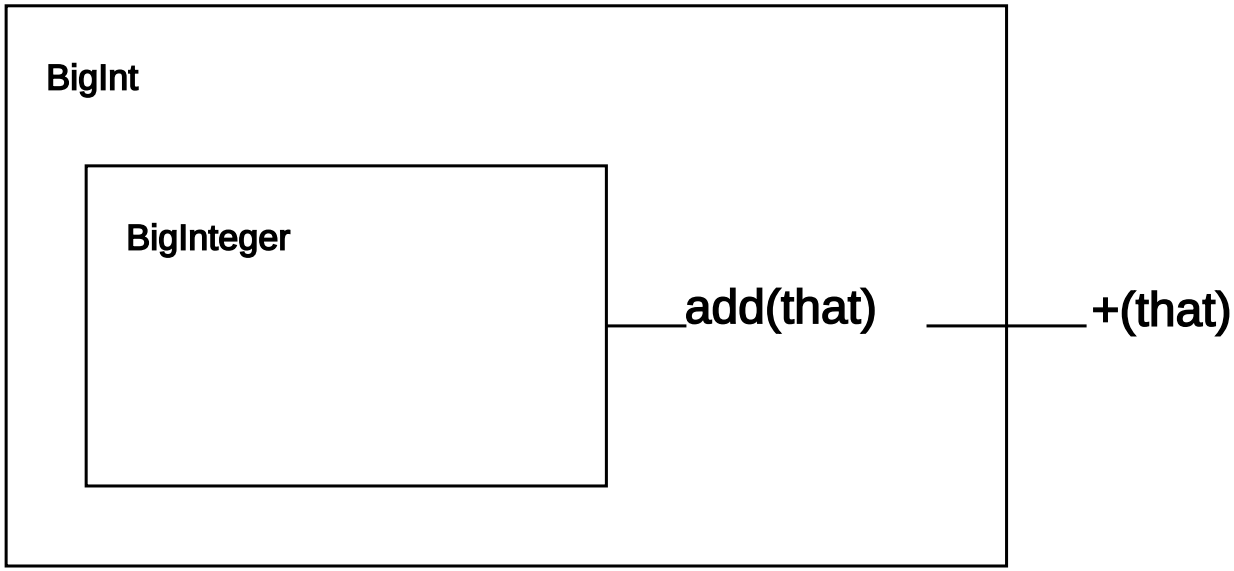
SCALAIO 2024
Paris @ Epita

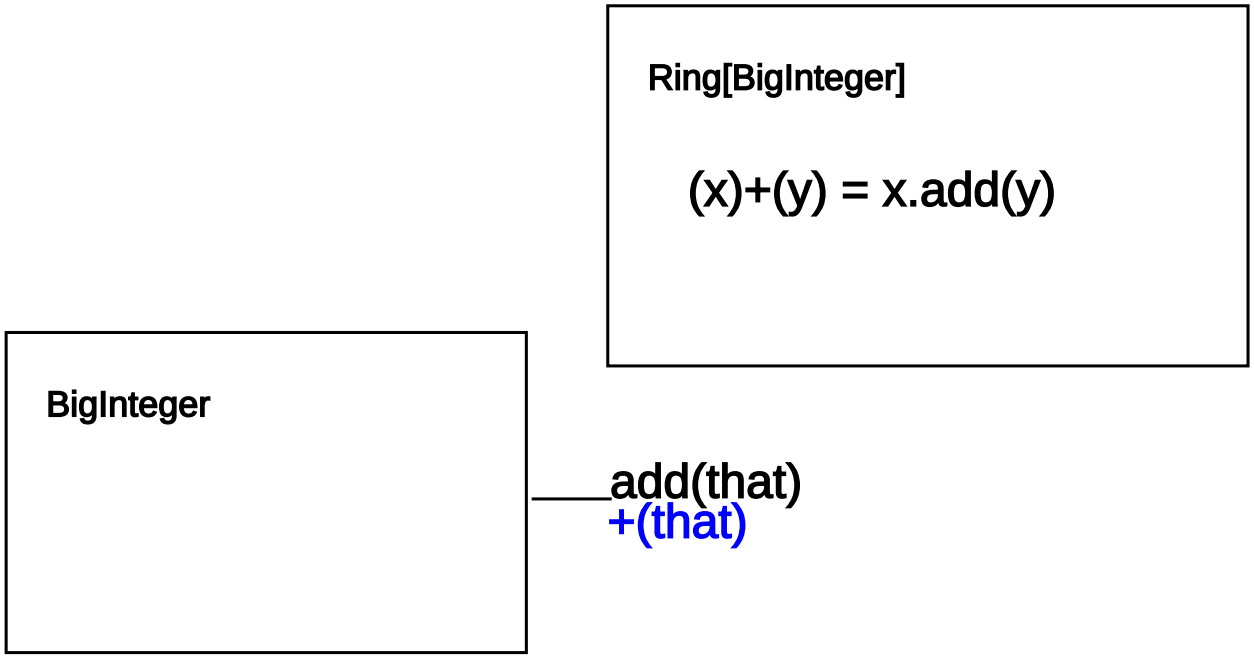
```
Polynomial gcd(Polynomial p, Polynomial q) {
        while (q.signum() != 0) {
                Polynomial r = p.remainder(q);
                p = q;
                q = r;
        }
        return p;
}
```

```
  @tailrec def gcd(x: T, y: T) = if (y.isZero) x else
gcd(y, x % y)
```

```
val a: Apple
val b: Orange

a+b // compile error

p in ℤ[x]
1 also in 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)
```

```
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.++
```

Interlude

Interlude : Categorial view of computer algebra

1965
1990 Axiom/Aldor
1990 MAS (Modula-2 Algebra System)
2000 JAS (Java)
2000 DoCon (algebraic Domain Constructor) Haskell
2010 ScAS (Scala Algebra System)
2015 DoCon-A (Agda)

Serge D. Mechveliani, Ph.D dissertation "Functional programming and categorial approach in computer algebra". In Russian. Pereslavl-Zalessky, Russia, 2002
Heinz Kredel, On the Design of a Java Computer Algebra System, Proceedings Principles and Practices of Programming in Java 2006
Raphael Jolly, Heinz Kredel, Computer algebra in Java: libraries and scripting, arXiv:0811.1061v1, 2008

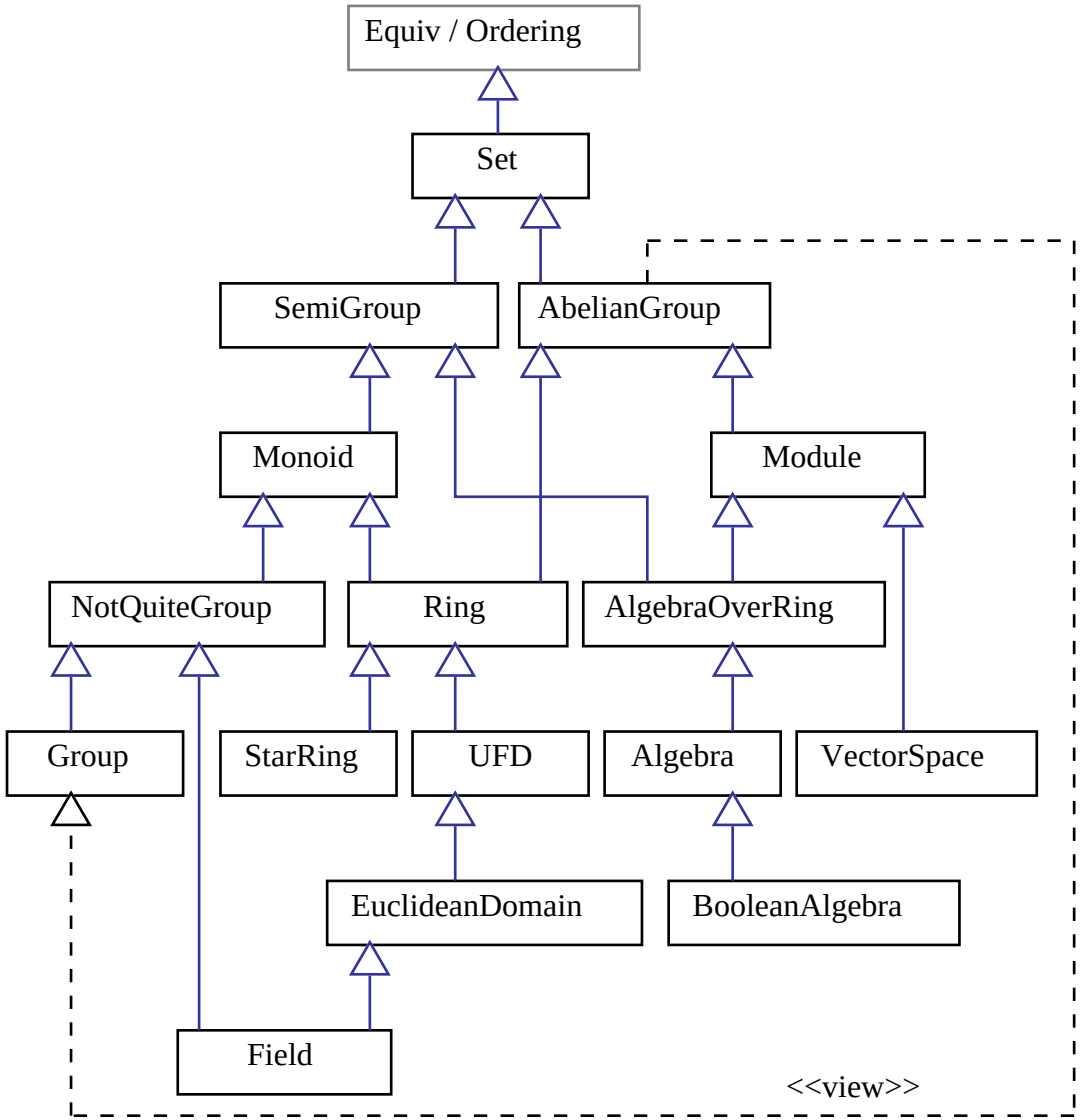
Semigroups and monoids

Here's the `Monoid` type class definition:

```
1  trait SemiGroup[T]:
2    extension (x: T) def combine (y: T): T
3
4  trait Monoid[T] extends SemiGroup[T]:
5    def unit: T
```

An implementation of this `Monoid` type class for the type `String` can be the following:

```
1  given Monoid[String] with
2    extension (x: String) def combine (y: String): String = x.concat(
3    def unit: String = ""
```


Equiv / Ordering
Set
SemiGroup
AbelianGroup
Monoid
Module
NotQuiteGroup
Ring
AlgebraOverRing
Group
StarRing
UFD
Algebra
VectorSpace
EuclideanDomain
BooleanAlgebra
Field
<<view>>

Applications

- symbolic computation is niche
- numeric applications can take advantage of some algebraic structure
- linear algebra : vectors/matrices, scientific computing in general
- complex numbers (standard library?)
- boolean algebra : logic

alpha=>(beta=>alpha)
(alpha=>(beta=>gamma))=>((alpha=>beta)=>(alpha=>gamma))
((p=>q)=>p)=>p

```
  val take = pp.take(1)
  val drop = pp.drop(1)
  def newInstance: [C] => (UniqueFactorizationDomain[C],
PowerProduct[M]) => MultivariatePolynomial[T, C, M]
  def gcd1(x: T[C, M], y: T[C, M]): T[C, M]
  def gcd(x: T[C, M], y: T[C, M]) = if (length > 1) then
    val s = newInstance(newInstance(ring, drop), take)
    s.gcd(x.convertTo(using s), y.convertTo(using
s)).convertFrom(s)
  else
    val (a, p) = contentAndPrimitivePart(x)
    val (b, q) = contentAndPrimitivePart(y)
    primitivePart(gcd1(p, q))%* ring.gcd(a, b)
```

```
  def gcd(x: T, y: T) =
    val (a, p) = contentAndPrimitivePart(x)
    val (b, q) = contentAndPrimitivePart(y)
    given module: Module[T, C, N] = new Module(using this)
("c", 3)
    val list = module.gb(
      Array(p, 1, 0),
      Array(q, 0, 1)
    )
    val Array(_, u, v) = list.last
    (p / v)%* ring.gcd(a, b)
```

Algorithm GCD-by-syzygy

- Input: A pair of polynomials $\{p(x_1, \ldots, x_n),\ q(x_1, \ldots, x_n)\}$ in $\mathbb{K}[x_1, \ldots, x_n]$.
- Output: GCD $(p,\ q)$.
- Steps:

1. Set up the matrix $M = \begin{pmatrix} p & 1 & 0 \\ q & 0 & 1 \end{pmatrix}$. We regard the rows as generators of a submodule of $\mathbb{K}[x_1, \ldots, x_n]^3$, so the columns index the vector components.
2. Form a modification of POT ordering that has column 1 larger than columns 2 and 3, and also larger than all ring variables.
   a. For purposes of efficiency we will actually use term over position (TOP) ordering for our handling of ordering of polynomial variables with respect to columns 2 and 3.
   b. If we use tag variables $\{c_1,\ c_2,\ c_3\}$ for the columns, then our variable order becomes $c_1 > \{x_1, \ldots, x_n\} > \{c_2,\ c_3\}$.
   c. We use a degree reverse lexicographic term order on the subset $\{x_1, \ldots, x_n\}$. This helps for efficiency and also simplifies the proof of correctness.
3. Compute the Gröbner basis for this module with respect to the above term order. The result, again represented as a matrix, will have the form $G = \begin{pmatrix} g_1 & * & * \\ g_2 & * & * \\ \vdots & \vdots & \vdots \\ 0 & u & v \end{pmatrix}$.
4. Take the last row of G. As in the univariate case, the syzygy relation is $u(x_1, \ldots, x_n)\ p(x_1, \ldots, x_n) + v(x_1, \ldots, x_n)\ q(x_1, \ldots, x_n) = 0$.
5. Compute the quotient $g(x_1, \ldots, x_n) = \frac{p(x_1, \ldots, x_n)}{v(x_1, \ldots, x_n)}$. This is our GCD.

Workarounds

Nested algebraic structure

```
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 {}
```

Workarounds : avoiding

```
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
```

Introduction of the into keyword

```
trait Ring[T]:

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

Introduction of the into keyword

```
trait Ring[T]:

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

The context bounds alternative to implicit conversion

```
trait Ring[T]:

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

The context bounds alternative to implicit conversion

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

The context bounds alternative to implicit conversion

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

The context bounds alternative to implicit conversion

```
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)
```

```
Context bounds : before

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
```

```
Context bounds : after

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
```

Workarounds : overriding

```
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
```

Nested scope

```
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)
```

Nested scope

```
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)
```

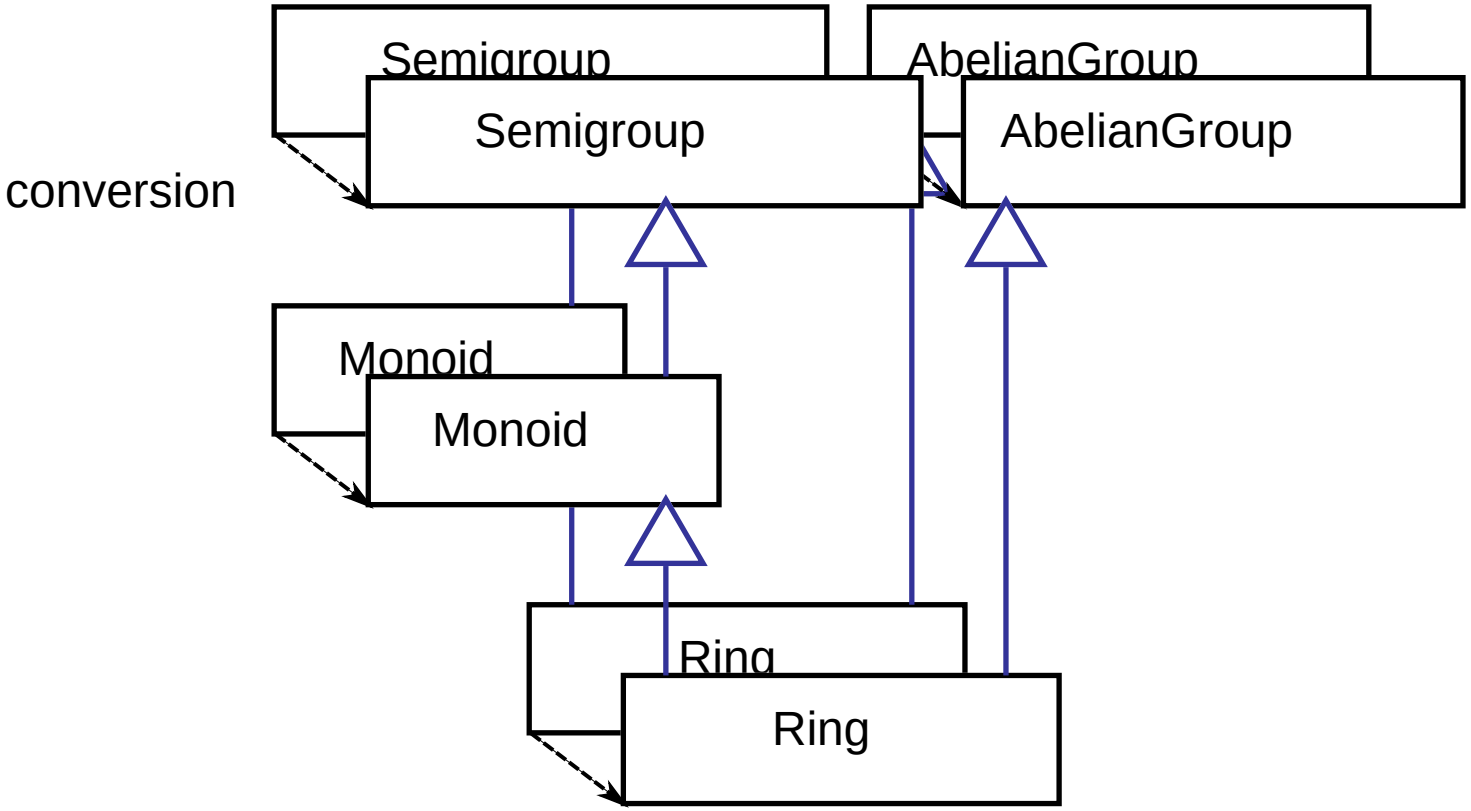
Nested scope

```
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)
```

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

Workaround : splitting



Example

```
object Complex extends Complex.Impl with
Field.Conv[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
  -> mandates 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