Progress report on the Scala Algebra System

Raphaël Jolly
Databeans

CASC 2020
Linz

* Idea : use type classes in the Scala language to model algebraic structures[1] as an alternative to F-bounded polymorphism, used in the Java Algebra System[2]
* Benefits: allows post-facto extensions[3] and makes it possible to reuse existing classes without wrappers
* Also allows generic numeric-symbolic implementations, with unboxed primitive types for improved efficiency
* There was however a problem with coercions and their interaction with type classes

[1] Jolly, R. Categories as type classes in the Scala Algebra System. CASC 2013
[2] Kredel, H. Parametric solvable polynomial rings and applications. CASC 2015
[3] Watt, S. Post facto type extensions for mathematical programming. DSAL 2006

In consequence, I had to devise a hybrid scheme: as type classes can operate with values of any type, why not exercise them on f-bounded classes, which are coercion-friendly.

The downside of this approach is that I could not use it to implement a Scala DSL to existing libraries (JAS) like is currently possible with Jython or JRuby. For this, I had to wait for improvements in the Scala language itself, which are now begining to emerge in Scala 3 ("Dotty").

In Dotty, type classes are now enhanced to support extension methods, which allow to define infix operators, with their parameters on each side.

But let us first look at a Scala 2 type class declaration.

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

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

https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html

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

* example definition

```
type BigInteger = java.math.BigInteger

given BigInteger as Ring[BigInteger]:
  def (x: BigInteger) + (y: BigInteger) = x.add(y)
  def zero = java.math.BigInteger.valueOf(0)
```

```
scala> 1l + 1
scala> 1 + 1l
// res1: Long = 2

scala> BigInt(1) + 1
scala> 1 + BigInt(1)
// res3: scala.math.BigInt = 2
```

* Polynomials : ZZ[x]
  x + 1
  1 + x
* Nested polynomials : ZZ[x][y]
  x + y
  y + 1

* This was not working in Scala 2 because type classes and coercions use the same underlying mechanism (implicits)

```
class Ring[T <: RingElem[T] : RingFactory]
    extends scas.structure.ordered.Ring[T] {
  def (x: T) + (y: T) = x.sum(y)
  def (x: T) - (y: T) = x.subtract(y)
  def (x: T) * (y: T) = x.multiply(y)
  def compare(x: T, y: T) = x.compareTo(y)
  def (x: T).isUnit = x.isUnit
  def characteristic = RingFactory[T].characteristic
  def zero = RingFactory[T].getZERO()
  def one = RingFactory[T].getONE()
  def (x: T).toCode(level: Level) = x.toString
  def (x: T).toMathML: String = ???
  def toMathML = ???

object RingFactory:
  def apply[T <: RingElem[T] : RingFactory] =
      summon[RingFactory[T]]
```

```
import jas.{ZZ, BigInteger, poly2scas, coef2poly,
    int2bigInt, bigInt2scas}

given r as GenPolynomialRing[BigInteger](ZZ,
    Array("x", "y", "z"), TermOrderByName.INVLEX)
val Array(one, x, y, z) = r.gens
val s = poly2scas(r)
import s.{+, *}

val p = 1 + x + y + z
val q = p \ 20
val q1 = q + 1
val q2 = q * q1
q2.length
// 12341
```

```
from jas import PolyRing, ZZ
# sparse polynomial powers

r = PolyRing( ZZ(), "(x,y,z)", PolyRing.lex );

# [one,x,y,z] = r.gens()




p = 1 + x + y + z;
q = p ** 20;
q1 = q + 1;
q2 = q * q1;
len(q2)
// 12341
```

The Rings project[4] has opted for a similar, typeclass-based design with its Scala DSL interface. To address the coercion problem, as far as I can tell the retained solution looks as follows in the new typeclass syntax.

```
trait Ring[E]:
  def (x: E) + (y: Int): E
  def (x: E) + (y: E): E
  def (x: Int) + (y: E): E

trait IPolynomialRing[Poly <: IPolynomial[Poly], E]
    extends Ring[Poly]:
  def (x: Poly) + (y: E): Poly
  def (x: E) + (poly: Poly): Poly
```

[4] Poslavsky, S. Rings: An efficient JVM library for commutative algebra (Invited Talk). CASC 2019

```
import cc.redberry.rings

import rings.poly.PolynomialMethods._
import rings.scaladsl._
import syntax._

implicit val ring = UnivariateRing(UnivariateRing(Z, "x"),
"y")
val x = ring("x")
val y = ring("y")

ring.show(x+y)
// x+y
```

```
implicit val r = UnivariateRing(Z, "x")
implicit val s = UnivariateRing(r, "y")
val x = r("x")
val y = s("y")

r.show(x+asBigInteger(1))
// 1+x

s.show(y+asBigInteger(1))
// javax.script.ScriptException: value + is not a member of
UnivariatePolynomial[UnivariatePolynomial[BigInteger]] in
s.show(y+asBigInteger(1))
```

```
abstract class Ring[T] extends
scas.structure.ordered.Ring[T]:
  def ring: cc.redberry.rings.Ring[T]
  def coder = Coder.mkCoder(ring)
  def (x: T) + (y: T) = ring.add(x, y)
  def (x: T) - (y: T) = ring.subtract(x, y)
  def (x: T) * (y: T) = ring.multiply(x, y)
  def compare(x: T, y: T) = ring.compare(x, y)
  def (x: T).isUnit = ring.isUnit(x)
  def characteristic = ring.characteristic
  def zero = ring.getZero()
  def one = ring.getOne()
  def (x: T).toCode(level: Level) = coder.stringify(x)
  def (x: T).toMathML = ???
  def toMathML = ???
```

Scala offers some mechanisms for automatic code specialization. In the former versions, there was a @specialized annotation, but it is abandonned in Dotty. Fortunately, there is a replacement mechanism.

```
abstract class MathLib[N : Numeric]:
  def dotProduct(xs: Array[N], ys: Array[N]): N
object MathLib:
  inline def apply[N : Numeric] = new MathLib[N]:
    def dotProduct(xs: Array[N], ys: Array[N]) =
      require(xs.length == ys.length)
      var i = 0
      var s: N = Numeric[N].zero
      while (i < xs.length)
        s = s + xs(i) * ys(i)
        i += 1
      s
```

```
val mlib = MathLib[Double]

val xs = Array(1.0, 1.0)
val ys = Array(2.0, -3.0)
mlib.dotProduct(xs, ys)
// -1.0
```

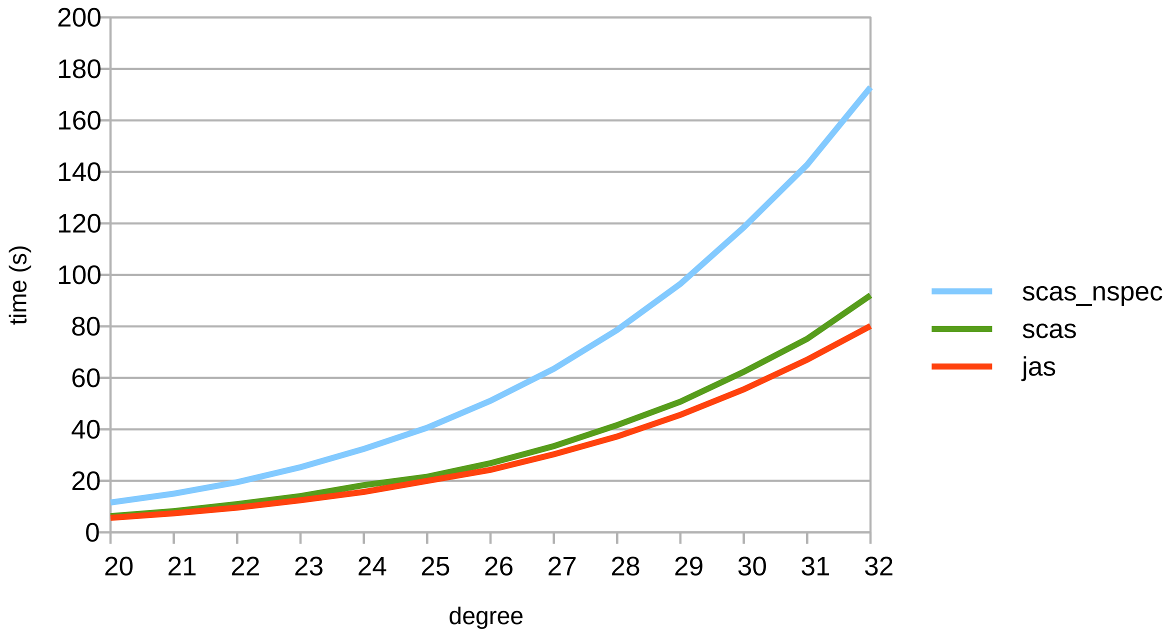
Inline : call site

```
val mlib = MathLib[Double]

val xs = Array(1.0, 1.0)
val ys = Array(2.0, -3.0)
mlib.dotProduct(xs, ys)
// -1.0
```

I have used this principle to specialize a generic implementation of power product exponents in the Scala Algebra System. To assess the improved efficiency of inline specialized code, I have used the Polypower benchmark that we have seen previously.

Inline : polypower benchmark



This graphic shows the execution times versus degree of polynomial in multiplication with non-specialized and specialized generic exponent vectors, respectively, together with originally specialized JAS code for comparison. As we can see, there is considerable cost for boxing, which is removed by specialization.

* The redesign of type classes and implicits in Scala 3 offers a solution to the coercion problem

* It makes it possible to use Scala as DSL for existing libraries

* There are proof of concept implementations for JAS, Rings

* There is a small improvement over how coefficients are lifted to the target ring in Rings' own Scala DSL

* The construct is reasonably efficient and might be used to implement the whole library as well as the interface

Thank you !

http://github.com/rjolly/scas