Numeric Programming with Spire

Lars Hupel
Krakow Scala User Group
2019-02-21

INNOQ

Numeric Programming with Cats, Algebra & Spire

Lars Hupel
Krakow Scala User Group
2019-02-21

INNOQ

What is Spire?

> *“Spire is a numeric library for Scala which is intended to be generic, fast, and precise.”*

What is Spire?

> *“Spire is a numeric library for Scala which is intended to be generic, fast, and precise.”*

- one of the “oldest” Typelevel libraries
- initial work by Eiríkr Åsheim & Tom Switzer
- 60 contributors
- started out in 2011 as a SIP for improving numerics

Generic Numeric Programming Through Specialized Type Classes

Scala Workshop 2012

Erik Osheim

Azavea

eosheim@azavea.com

Abstract

We describe an ongoing effort to build a system of type classes that support fast, accurate, flexible and generic numeric programming in Scala. This work combines Scala's support for user-directed type specialization with previous work on numeric type classes. In principle, these allow one to create generic numeric algorithms without sacrificing the speed of a direct implementation. In practice, these performance gains make very specific demands of both the language and the user.

This paper is a case study: we will explain the problems faced, discuss our strategies, and provide benchmarking results. We will also discuss ways in which Scala could be improved to more easily accommodate this kind of work. Finally, we will present a simple compiler plug-in that can be used to increase performance in many cases.

[14], is a general-purpose numerics library by Erik Osheim and Tom Switzer. Started as a set of proposed improvements to Scala's built-in numerics, Spire has evolved into a stand-alone library supporting new number types, a full type class hierarchy, and other functions.

Much of the underlying specialization work from the R&D project has been ported over to Spire, but some parts are only available in the original project (e.g. the compiler plug-in). Spire's number types and design philosophy have informed the design of its type classes, whereas the earlier project stayed closer to the design found in `scala.math`.

2. BACKGROUND

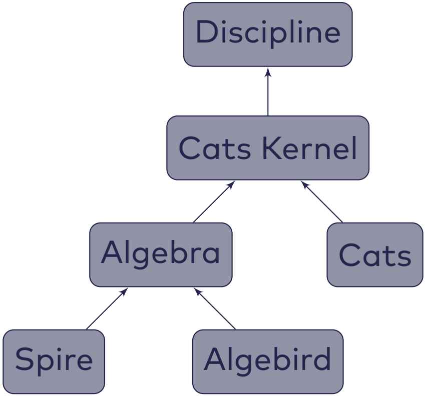
2.1 Motivating Examples

Programming is often an exercise in abstraction. Developers

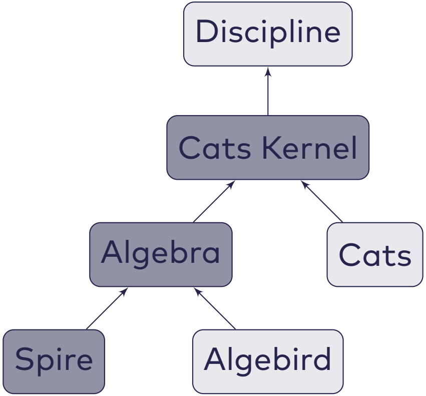
What's in Spire?

- algebraic tower
- number types
- numeric algorithms
- pretty syntax
- optimization macros
- laws

Project relationship



Project relationship



Algebra

> “*Algebra is the study of mathematical symbols and the rules for manipulating these symbols.*”

Algebra

> “*Algebra is the study of mathematical symbols and the rules for manipulating these symbols.*”

Mathematicians study algebra to discover **common properties** of various concrete structures.

Algebra

> *“Algebra is the study of mathematical symbols and the rules for manipulating these symbols.”*

Examples

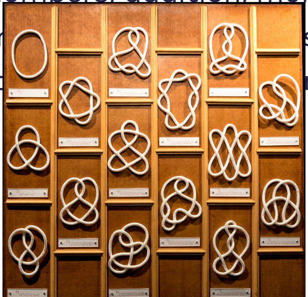
- numbers, addition, multiplication
- matrices, vector spaces, linear algebra
- lattices, boolean algebra

Algebra

> *“Algebra is the study of mathematical symbols and the rules for manipulating these symbols.”*

Examples

- numbers, addition, multiplication
- m s, linear algebra
- lo ra

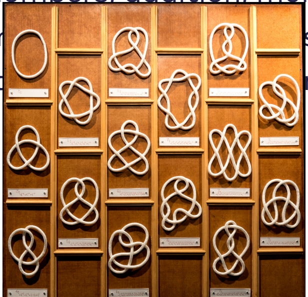


Algebra

> *“Algebra is the study of mathematical symbols and the rules for manipulating these symbols.”*

Examples

- numbers, addition, multiplication
- m s, linear algebra
- lc ra



Semigroup

```scala
trait Semigroup[A] {
  def append(x: A, y: A): A
}
```

Semigroup

```
trait Semigroup[A] {
  def append(x: A, y: A): A
}
```

Law: Associativity

```
append(x, append(y, z)) == append(append(x, y), z)
```

Monoids

```scala
trait Monoid[A] extends Semigroup[A] {
  def append(x: A, y: A): A // Semigroup
  def zero: A
}
```

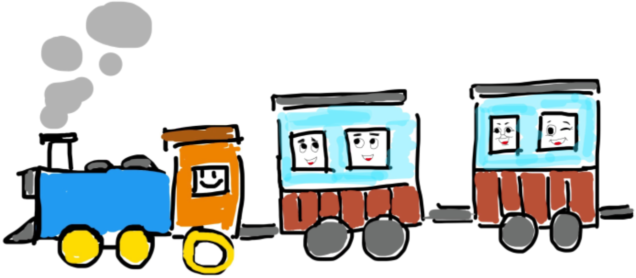
Law: Neutral element

```scala
append(x, zero) == x
```

Monoidal structures

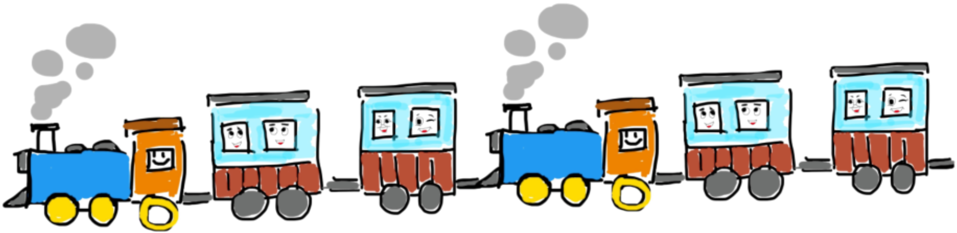
Lots of things are monoids.

Trains are monoids



Locomotive Carriages

Trains are monoids



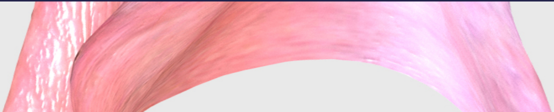
append(train, train)

Monoidal structures

Lots of things are monoids.

- (Train, no train, couple)
- (`Int`, 0, +)
- (`List[T]`, `Nil`, concat)
- (`Map[K, V]`, `Map.empty`, merge)

Demo



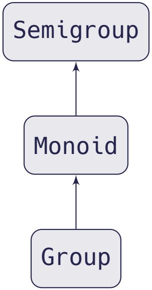
Monoidal structures

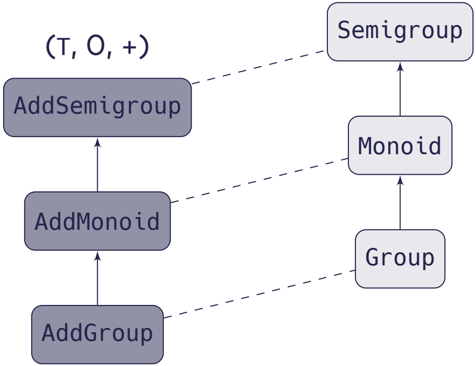
Lots of things are monoids.

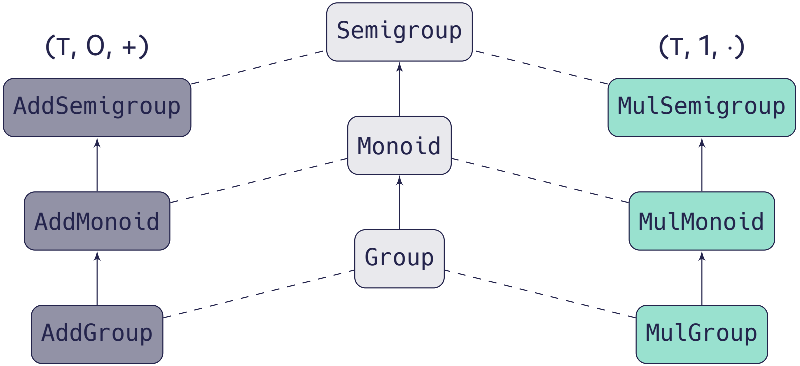
- (Train, no train, couple)
- (`Int`, 0, +)
- (`List[T]`, `Nil`, concat)
- (`Map[K, V]`, `Map.empty`, merge)

But some **are not!**

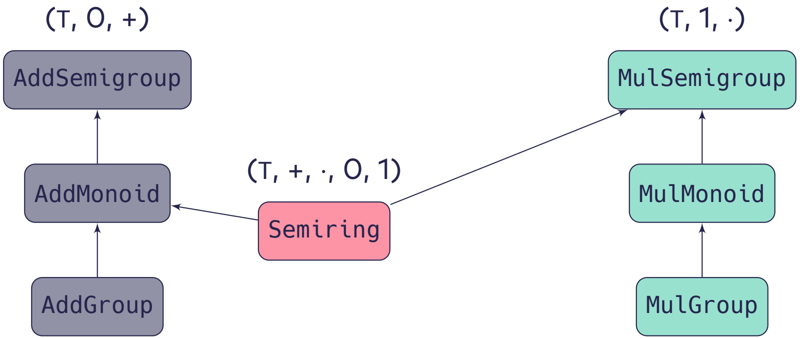
- (`Float`, 0, +)


Semigroup
Monoid
Group

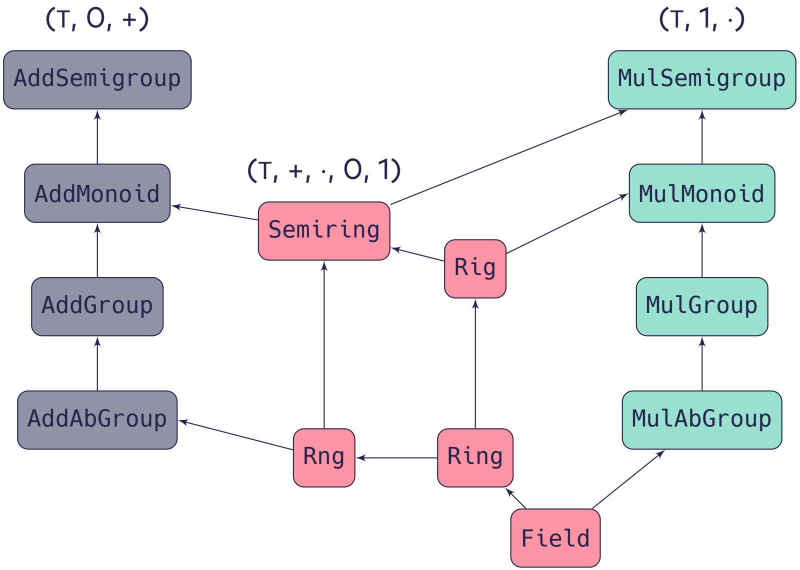

(T, 0, +)
Semigroup
AddSemigroup
Monoid
AddMonoid
Group
AddGroup



(T, 0, +)
Semigroup
(T, 1, ·)
AddSemigroup
MulSemigroup
Monoid
AddMonoid
MulMonoid
Group
AddGroup
MulGroup



(T, 0, +)
AddSemigroup
AddMonoid
AddGroup
(T, +, ·, 0, 1)
Semiring
(T, 1, ·)
MulSemigroup
MulMonoid
MulGroup

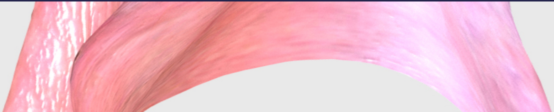


(T, 0, +)
(T, 1, ·)
AddSemigroup
MulSemigroup
(T, +, ·, 0, 1)
AddMonoid
MulMonoid
Semiring
Rig
AddGroup
MulGroup
AddAbGroup
MulAbGroup
Rng
Ring
Field

Law Checking

```
// Float and Double fail these tests
checkAll("Int",      RingLaws[Int].euclideanRing)
checkAll("Long",     RingLaws[Long].euclideanRing)
checkAll("BigInt",   RingLaws[BigInt].euclideanRing)
checkAll("Rational", RingLaws[Rational].field)
checkAll("Real",     RingLaws[Real].field)
```

Demo



Numbers

- machine floats are fast, but imprecise
- good tradeoff for many purposes, but not all!
- there is no “one size fits all” number type

Rational numbers

$$\frac{n}{d} \in \mathbb{Q} \quad \text{where } n, d \in \mathbb{Z}$$

Properties

- closed under addition, multiplication, ...
- decidable comparison

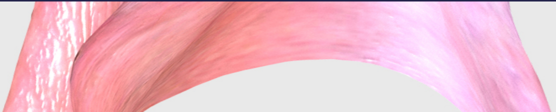
Rational numbers

$$\frac{n}{d} \in \mathbb{Q} \quad \text{where } n, d \in \mathbb{Z}$$

Properties

- closed under addition, multiplication, ...
- decidable comparison
- **may grow large**

Demo



Real numbers

We can't represent **all** real numbers on a computer ...

Real numbers

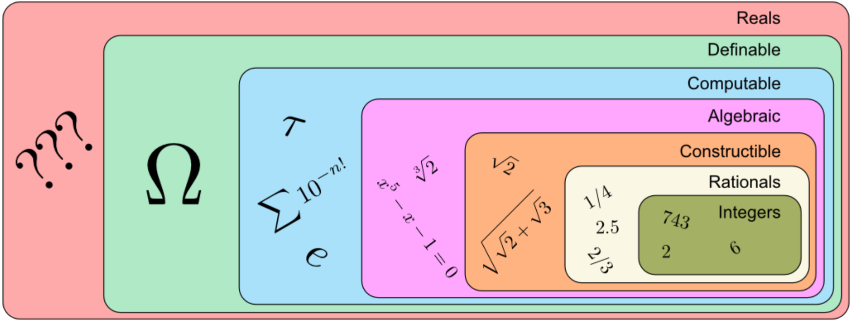
We can't represent **all** real numbers on a computer ...

... but we can get **arbitrarily close**

Real numbers

We can't represent **all** real numbers on a computer ...

... but we can get **arbitrarily close**



Real numbers, approximated

```
trait Real {
  def approximate(precision: Int): Rational
}
```

Real numbers, approximated

```
trait Real { self =>
  def approximate(precision: Int): Rational

  def +(that: Real): Real = new Real {
    def approximate(precision: Int) = {
      val r1 = self.approximate(precision + 2)
      val r2 = that.approximate(precision + 2)
      r1 + r2
    }
  }
}
```

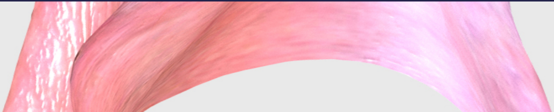
Real numbers, approximated

```
trait Real {
  def approximate(precision: Int): Rational
}

object Real {
  def apply(f: Int => Rational) = // ...

  def fromRational(rat: Rational) =
    apply(_ => rat)
}
```

Irrational numbers

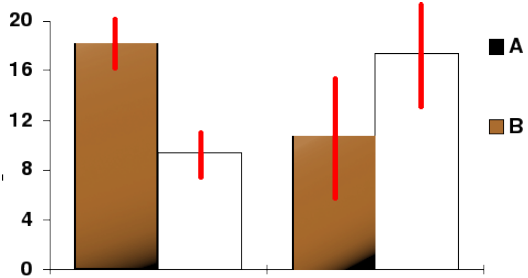
```
val pi: Real =
  Real(16) * atan(Real(Rational(1, 5))) -
    Real(4) * atan(Real(Rational(1, 239)))
```

Demo



Error bounds

- often, inputs are not accurate
- e.g. measurements (temperature, work, time, ...)
- What to do with error bounds?



Interval arithmetic

```scala
case class Interval[A](lower: A, upper: A)
```

Interval arithmetic

```
case class Interval[A](lower: A, upper: A) {
  def +(that: Interval[A]) =
    Interval(this.lower + that.lower,
             this.upper + that.upper)
}
```

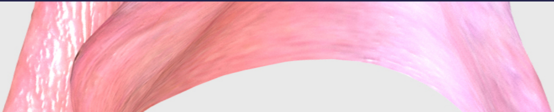
Interval arithmetic

```
case class Interval[A](lower: A, upper: A) {
  def +(that: Interval[A]) =
    Interval(this.lower + that.lower,
             this.upper + that.upper)
}
```

Spire generalizes this even further:

- open/closed intervals
- bounded/unbounded intervals

Demo



What else?

Spire is full of tools you didn't know you needed.

What else?

Spire is full of tools you didn't know you needed.

- `SafeLong`: like `BigInt`, but faster

What else?

Spire is full of tools you didn't know you needed.

- `SafeLong`: like `BigInt`, but faster
- `Trilean`: tri-state boolean value

What else?

Spire is full of tools you didn't know you needed.

- `SafeLong`: like `BigInt`, but faster
- `Trilean`: tri-state boolean value
- `UByte`, `UShort`, `UInt`, `ULong`: unsigned machine words

What else?

Spire is full of tools you didn't know you needed.

- `SafeLong`: like `BigInt`, but faster
- `Trilean`: tri-state boolean value
- `UByte`, `UShort`, `UInt`, `ULong`: unsigned machine words
- `Natural`: non-negative, arbitrary-sized integers

Q & A

INNOQ
www.innoq.com

Lars Hupel

lars.hupel@innoq.com

@larsr_h

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim a. Rh.
Germany
+49 2173 3366-0

Ohlauer Str. 43
10999 Berlin
Germany

Ludwigstr. 180 E
63067 Offenbach
Germany

Kreuzstr. 16
80331 München
Germany

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
+41 41 743 01 11

Albulastr. 55
8048 Zürich
Switzerland

LARS HUPEL

Consultant
innoQ Deutschland GmbH

Lars enjoys programming in a variety of languages, including Scala, Haskell, and Rust. He is known as a frequent conference speaker and one of the founders of the Typelevel initiative which is dedicated to providing principled, type-driven Scala libraries.

Image sources

- Rubik's Cube: `https://en.wikipedia.org/wiki/File:Rubik%27s_Cube_variants.jpg`, Hellbus
- Knots: `https://en.wikipedia.org/wiki/File:Tabela_de_n%C3%B3s_matem%C3%A1ticos_01,_crop.jpg`, Rodrigo.Argenton
- Intervals: `https://commons.wikimedia.org/wiki/File:Confidenceinterval.png`, Audrius Meskauskas
- Number venn diagram: `http://www.science4all.org/article/numbers-and-constructibility/`, Lê Nguyên Hoang
- Lavaux: `https://en.wikipedia.org/wiki/File:Lake_Geneva_with_Vineyards_in_Lavaux.jpg`, Severin.stalder
- Drawings: Yifan Xing