nescala 30 Jan 2015

Intervals:

unifying uncertainty, ranges, and loops

Erik Osheim (**@d6**)

who am i?

- **typelevel** member λ
- maintain **spire** and several other scala libraries
- interested in expressiveness **and** performance ◕
- hack scala code at **meetup**

code at **http://github.com/non**

These slides assume the following imports.

```
import spire.algebra._
import spire.implicits._
import spire.math._
```

Together they import all of Spire's types, type classes, and syntax. It's ideal for exploratory work in the REPL.

You can find the code for Spire at:

https://github.com/non/spire

what will this talk cover?

- Basic description of the **Interval[_]** type
- Arithmetic, uncertainty, and error propagation
- Set operations and iteration
- Demo (**?**)
- Conclusion

intervals

what is an interval?

Intervals represent a range of real numbers.

For example: *all numbers between 3 and 6*.

Sound familiar?

what is an interval?

Reminds me of `scala.Range`:

```
scala> (3 to 6)
res0: Range.Inclusive = Range(3, 4, 5, 6)
```

Or maybe:

```
scala> (3 until 6)
res1: Range = Range(3, 4, 5)
```

what's the difference?

`Range` has two modes:

- *inclusive*: `(3 to 6)` includes 6
- *exclusive*: `(3 until 6)` excludes 6

(Both cases include 3.)

Our example is *all numbers between 3 and 6*.
Should that include 3 and 6?

being precise

The first argument of `Range` is always *inclusive*, so we wouldn't be able to exclude both end points.

Maybe `Range` is not the right way to express this idea?

notation

Let's use notation to be a bit clearer about what we mean.

We'll use:

```
* parenthesis as exclusive:     (3, 6)
* square brackets as inclusive: [3, 6]
* we can combine them too:      [3, 6)
```

(Note: this is not valid Scala code.)

special cases

There are some special intervals worth mentioning:

```
(∅)       empty interval (contains nothing)
[1]       the single value 1 (degenerate)
(3, ∞)    contains numbers > 3
(-∞, 0]   contains numbers <= 0
(-∞, ∞)   contains all numbers
```

(The bottom three are called *unbounded* intervals.)

numeric precision

Going back to the range example, we see:

```
scala> (3 to 6)
res0: Range.Inclusive = Range(3, 4, 5, 6)

scala> res0.contains(4.5)
res1: Boolean = false
```

(Ignore the fact that this type-checks.)

It seems clear that the interval `[3, 6]` should contain `4.5`.

time for polymorphism!

Range was written in terms of `Int`, to support things like:

```
(3 to 6).toList
(3 to 6).map(...)
for (i <- 3 to 6) { ... }
```

(Let's not talk about `NumericRange[_]` right now.)

In our case, we can support `Int`, `Double`, or any other type we want using Spire's type classes.

def length = uncountable

Remember: *intervals represent ranges of real numbers*.

Even `(0, 1)` has uncountably-many members.

This means we can't effectively model intervals with:

- `Range`
- `Set[_]`
- `Stream[_]`

...or any other Scala collection.

design goals

1. Defined in terms of endpoints -- not a container
2. Support unbounded intervals (above, below, or both)
3. Generic: parameterize the number type

(We want an intensional definition.)

generic constraints

Let's say we're representing `(x, y)` as an `Interval[A]`.

We know that `x < y` (i.e. the lower bound is below the upper).

Therefore, we definitely need to compare A values.

In Spire, we use `Order[A]` to do this.

interval arithmetic

why bother creating intervals?

- Uncertainty about a value (e.g. `3 ± 0.1`)
- Work with a set of values efficiently
- Provide approximate answers to harder problems
- Efficiently support distributed calculation

how do we use them?

Mathematically, interval arithmetic is equivalent to:

1. doing the operations over each element

2. building the smallest interval containing the results

For example:

```
[1, 2] + 1
[1 + 1, ... 2 + 1]
[2, 3]
```

a "graphical" view

We can see the same operation in ASCII art:

```
            0   1   2   3   4
  [1, 2]        [---]
+     1         x
= [2, 3]            [---]
```

By adding 1 to each end point we get the correct result.

more addition

We can add two intervals together just as easily:

```
            0   1   2   3   4   5
  [1, 2]        [---]
+ [1, 3]        [-------]
= [2, 5]            [-----------]
```

We are adding every possible value of `[1, 2]` to every possible value of `[1, 3]` to get `[2, 5]`.

it generalizes

This strategy works for *well-behaved* operations, i.e.

- the operation is deterministic
- the range of outputs is determined by end points and/or zero
- we know what to do in unbound cases

`Interval[_]` supports:

```
abs vmin vmax + - * / reciprocal pow sqrt nroot
```

in code

Let's start simple:

```
val w = Interval(3.0, 5.0)   // [3.0, 5.0]
val x = Interval.point(2.0)  // [2.0]
val y = 10.0 ± 0.1           // [9.9, 10.1]
val z = Interval.above(0.0)  // (0.0, ∞)

w.isEmpty          // false
x.isPoint          // true
y.contains(10.0)   // true
z.contains(0.0)    // false
```

arithmetic

```
val w = Interval(3.0, 5.0)   // [3.0, 5.0]
val x = Interval.point(2.0)  // [2.0]
val y = 10.0 ± 0.1           // [9.9, 10.1]
val z = Interval.above(0.0)  // (0.0, ∞)

w * x                  // [6.0, 10.0]
w * x + y              // [15.9, 20.1]
(x * y)                // [19.8, 20.2]
(x * y) vmin z         // (0.0, 20.2]
((y ** 2) * 2).sqrt // [14.000714267493642, 14.28355697996826]
```

interval capabilities

We can create intervals for anything with an order:

```
val a = Interval("bats", "cats")
val b = Interval.below(Option(6))
val c = Interval.empty[List[Long]]

a.contains("batter") // true
b + b                // (-∞, Some(12))
c + List(3L)         // (∅)
```

wait...

I thought you said intervals approximated real numbers?

What gives?

those aren't real numbers!

Spire generalizes `Interval[_]` to support all ordered types.

We then restrict specific interval operations.

```
val a = Interval("bats", "cats")
val b = Interval.below(Option(6))
val c = Interval.empty[List[Long]]

a * a   // does not compile
b.sqrt  // does not compile
c / 2L  // does not compile
```

type classes are capabilities

The compiler informs us that our types are not sufficiently *number-like* for the given operations.

(i.e. they don't behave sufficiently like real numbers.)

Here are the missing capabilities:

```
a * a            needs Semiring[_]
b.sqrt           needs Ring[_], NRoot[_]
c / List(2L)     needs Field[_]
```

intervals as specialized sets

basic set operations

We've shown how to combine intervals using arithmetic.

But we can also think of intervals as a kind of set:

```
val p = Interval.above(0) // (0, ∞)
val q = Interval.below(3) // (-∞, 3)

p.contains(0) // false
q.contains(0) // true
```

more set operations

```
val p = Interval.above(0) // (0, ∞)
val q = Interval.below(3) // (-∞, 3)

p & q                     // (0, 3)
p | q                     // (-∞, ∞)
val t = Interval(1, 2)    // [1, 2]
t intersects p            // true
t isSubsetOf q            // true
~p                        // List((-∞, 0])
~t                        // List((-∞, 1), (2, ∞))
p -- t                    // List((0, 1), (2, ∞))
```

set visualization

```
                    -2  -1   0   1   2   3   4
p = (0, ∞)                  (------------------
q = (-∞, 3)         ----------------------)
t = [1, 2]                      [---]

p & q                       (-----------)
p | q               ----------------------------
~p                  ---------]
~t                  -------------)   (----------
```

going even farther

Some operations (e.g. ~x) return a list of intervals.

Returning a single object would fit the *set* metaphor better.

If you want this, see Rüdiger Klaehn's *intervalset* project:

https://github.com/rklaehn/intervalset

(It's possible this structure will be added to a Spire module.)

if it's a set, can we map?

Sort of.

```
def mapBounds[B: Order](f: A => B) = {
  val lb = lowerBound.map(f)
  val ub = upperBound.map(f)
  Interval.fromBounds(lb, ub)
}
```

However, this isn't called `map` because it's not very well-behaved.

functor problems

Easy to pass `mapBounds` functions that violate our expectations.

```
val h = Interval(r"-2", r"2")  // [-2, 2]

h ** 2                         // [0, 4]
h.mapBounds(x => x ** 2)       // [4]
-h                             // [-2, 2]
h.mapBounds(x => -x)           // (∅)
```

Is there a better way?

polynomial power!

For our purposes, polynomials are:

- functions of a single variable (e.g. x)
- limited set of operations they can use

```
f(x) = x^2 + 9
f(x) = 3x^2
f(x) = 2x^2 + x
```

(Not valid scala code.)

polynomials in spire

Here's how to encode those polynomials in Spire:

```
// fancy string interpolation
poly"x^2 + 9" // Polynomial[Rational] = (x^2 + 9)

// sequence of terms
Polynomial(poly.Term(r"3", 2) :: Nil) // (3x^2)

// map of exponents -> coefficients
Polynomial(Map(2 -> r"2", 1 -> r"1")) // (2x^2 + x)
```

(Thanks to James Thompson for this excellent polynomial type.)

what does that do for us?

The clue is in the *limited set of operations polynomials can use*.

(These correspond to the *well-behaved* interval operations.)

```
val h = Interval(r"-2", r"2")  // [-2, 2]
val p = poly"x^2 + 9"          // (x^2 + 9)

h.translate(p)                 // [9, 13]
h.translate(poly"-x")          // [-2, 2]
h.translate(poly"x^3 + 3x^2")  // [-8, 20]
```

how does it work?

We turn `Polynomial[A]` into `Polynomial[Interval[A]]`.

```
def translate(p: Polynomial[A])(implicit A: Field[A]) = {
  val ts = p.terms.map { case Term(c, e) =>
    Term(Interval.point(c), e)
  }
  val p: Polynomial[Interval[A]] = Polynomial(ts)
  p(this)
}
```

Then we evaluate the new polynomial with `this`.

intervals as souped-up ranges

ranges

In addition to all this `Interval[_]` is an acceptable range.

Translation:

- `(x to y)` becomes `Interval(x, y)`
- `(x until y)` becomes `Interval.openUpper(x, y)`

Caveat: if you're looping over ints, stick with `Range` *(or* `cfor`*).*

advantages

You could just use `NumericRange[_]`; why `Interval[_]`?

```
1. No need to specify a step up-front.
2. Very fast contains() checks.
3. Math operations (e.g. +)
4. Set operations (e.g. intersection)
5. Overflow checking via NumberTag[_]
6. infinite loop (TM) support
```

Let's look at some examples...

basic folding/iteration

```
val ns = Interval(0, 100)
ns.foldOver(0, step = 1)(_ + _) // 5050

val x = ns.iterator(1).sum // 5050

ns.loop(25)(println)
// 0
// 25
// 50
// 75
// 100
```

getting fancier

We can use union/intersection to encode loop range calculations:

```
val x = Interval(x, y)
val y = Interval(w, z)
val z = (loop1 & loop2)

z.loop(step) { ... }
```

how often have you wanted this?

```
// divide the interval [0, N] into M distinct parts.
// return boundaries as a list.
def divide(n: Int, m: Int): List[Double] = {
  val h = Interval(r"0", Rational(n))
  val step = Rational(n, m)
  h.iterator(step).map(_.toDouble).toList
}

divide(7, 5)  // List(0.0, 1.4, 2.8, 4.2, 5.6, 7.0)
divide(13, 4) // List(0.0, 3.25, 6.5, 9.75, 13.0)
divide(3, 7) // List(0.0, 3.25, 6.5, 9.75, 13.0)
divide(10, 3)
// List(0.0, 3.3333333333333335, 6.666666666666667, 10.0)
```

how often have you wanted this?

Works on very hairy fractions too!

```
divide(3, 7)
// List(0.0,
//      0.42857142857142855,
//      0.8571428571428571,
//      1.2857142857142858,
//      1.7142857142857142,
//      2.142857142857143,
//      2.5714285714285716, 3.0)
```

infinite loops

```
Interval.above(r"0").loop(r"1") { n =>
  println(s"$n bottles of beer on the wall")
}
```

cool demo

caveats

There are a few things to keep in mind when working with `Interval[_]`:

1. It is unspecialized (re loops).
2. `Order[_]` needs to obey additional laws when used with rings.
3. Possible rounding issues with `Double`.
4. Interpretation of `contains` and `isEmpty`.
5. The "dependency problem"

why unspecialized?

In most cases it is more efficient for `Interval[_]` to box.

This means that using `Interval[Int]` purely for looping will be slower than code that avoids boxing altogether.

Prefer `while`, Spire's `cfor`, Scalaxy/Loops, `Range` or similar.

[1] Fields in specialized classes are duplicated, and often fail to specialize access correctly.

additional order laws?

Our intervals require `Order[_]`, a total ordering.

However, when using `Order[_]` with `Field[_]` it must:

1. preserve the *algebraic structure*, i.e.
2. `if a ⩽ b then a + c ⩽ b + c` and
3. `if 0 ⩽ a and 0 ⩽ b then 0 ⩽ a * b`

e.g. `Interval[_]` cannot work correctly with `Complex[_]`.

but aren't complex intervals possible?

Yes, but `Interval[_]` doesn't support them.

If you need these, it's a *great opportunity* to contribute! ☺

rounding issues

```
1. The JVM doesn't support floating point rounding modes
2. Even if it did, using them would be a global side-effect.
3. Correct approach:
   * use java.lang.Math.{nextUp, nextAfter}
   * But quite ugly to bake this in for generic types
```

We probably need to extend the `NumberTag` to help support this.

Mostly I just use `Interval[Rational]` instead. ☃

contains and isEmpty

One potential puzzler is this:

```
val e = Interval.empty[Int]  // (∅)
val h = Interval.open(3, 4)  // (3, 4)

e.isEmpty                    // true
(e.contains(3), e.contains(4)  // (false, false)

h.isEmpty                    // false
(h.contains(3), h.contains(4)  // (false, false)
```

What gives?

contains and isEmpty

Remember: A is an approximation of the reals.

Even though Long has no values between 3 and 4, they exist.

Consider:

```
val ee = e * 2    // (Ø)
val hh = h * 2    // (6, 8)
hh.contains(7)    // true
```

So h was never empty at all!

dependency problem

Here's something strange:

```
val h = Interval(-3, 3)  // [-3, 3]
h ** 2                   // [0, 9]
h * h                    // [-9, 9]
```

What's going on?

dependency problem

If you take one value from `[-3, 3]` and square it, the smallest value you can get is `0` (e.g. `0 ** 2`).

However, if you take two values from `[-3, 3]` and multiply them, you could get `-9` (e.g. `-3 * 3`).

This is called the *dependency problem*.

These results are a correct interval, but could be more precise (i.e. tighter bounds).

conclusion

Hopefully I've helped explain what intervals are.

I believe there are many problems where intervals could be useful.

(Even if it wouldn't be worth it to stop and implement them then.)

Let us know what you think! ☏

there's even more

Learn more about Spire's various types in the *Spire User's Guide*:

https://github.com/non/spire/blob/master/GUIDE.md

If you want to contribute to Spire, we also have:

https://github.com/non/spire/blob/master/CONTRIBUTING.md

AUTHORS.md

Thanks to thank everyone who's helped build Spire:

Luc J. Bourhis
Olivier Chafik
Jean-Remi Desjardins
Christopher Hodapp
Kamushin
Christian Krause
Brian McKenna
Adam Pingel
Tom Switzer
Titorenko
Kenji Yoshida
Eugene Burmako
Adelbert Chang
Suminda Dharmasena
Lars Hupel
Rex Kerr
Dušan Kysel
Simon Ochsenreither
Denis Rosset
Flaviu Tamas
Vlad Ureche
Benito van der Zander
Franco Callari
Eric Christianse
Rob Emanuele
Ben Hutchison
Grzegorz Kossakowski
Josh Marcus
Erik Osheim
Lukas Rytz
James Thompson
Volth

special thanks

Denis Rosset rewrote large parts of `Interval[_]` and ensured that the many cases were handled properly

Without Denis Rosset's hard work this talk would not exist.

Also, thanks to Alan Eliason for his excellent work on Frink.

Thanks!

the end