pnwscala 15 Nov 2014

Unruly Creatures:

Strategies for dealing with *Real Numbers*

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**

(The following slides assume the following imports.)

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

(These are good for doing exploratory work in the REPL.)

You can find the code for Spire at:

https://github.com/non/spire

What will this talk cover?

- Primitive math blooper-reel
- Difficulties of numeric representation
- Quick summary of some types and trade-offs
- Detailed treatment of *computable real numbers*

⚠ Math warning ⚠

This talk should not require a ton of prior math knowledge. My goal is to cover this material in an approachable way.

(Almost) none of us are experts. I am not an expert.

Please ask questions if you are confused, or if I present something in a confusing way (quite likely).

Fixed-width integers

Most of us use `Int` and `Long` every day.

These types are fast, and usually work well.

But when they fail, they fail catastrophically.

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`?

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`? 2

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`? 2

How many `Int` values are there?

- Positive?
- Negative?

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`? 2

How many `Int` values are there? `4294967296` (`0x100000000L`)

- Positive? 2147483647
- Negative? 2147483648

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`? 2

How many `Int` values are there? `4294967296` (`0x100000000L`)

- Positive? `2147483647`
- Negative? `2147483648`

If `(x > 0)` and `(y > 0)`, is `(x + y > 0)`?

Pop Quiz

For how many `Int` values will `(x == -x)` return `true`? 2

How many `Int` values are there? `4294967296` (`0x100000000L`)

- Positive? `2147483647`
- Negative? `2147483648`

If `(x > 0)` and `(y > 0)`, is `(x + y > 0)`? not always

MaxValues

- `Int` fails if you need to:
  - measure US spending in dollars
  - count humans on Earth
- Long fails if you need to:
  - measure US energy consumption in Joules
  - weigh the Earth in pounds.

So integer types are pretty easy to understand.

But when they fail, they fail *catastrophically*.

How about floating point?

Improvements:

- Can represent fractional values ("real" division)
- Much wider range of values (max is ~1.8 x 10^308)
- Overflows to `PositiveInfinity`

So, it just works?

The other shoe

Adding numbers is what computers are good at.

```
0.15 + 0.15 // 0.3
0.1  + 0.2  // 0.30000000000000004

(0.15 + 0.15) == (0.1 + 0.2) // false
```

Wait, what?

Something about epsilons...

Four (unsuccessful) attempts, in order of complexity:

```
val x: Double = ...
val y: Double = ...
val eps = 0.000001

(x == y) // nope, as per previous slide
(x - y).abs < eps // what if x and y are very small?
((x - y).abs / y) < eps // what if y is negative?
((x - y).abs / (x.abs + y.abs)) < eps // zeros?
```

Maybe this is it?

```
def okFineUgh(x: Double, y: Double, eps: Double) = {
  import java.lang.Double.MIN_NORMAL
  val xa = x.abs
  val ya = y.abs
  val d = xa - ya
  if (x == y) true
  else if (x == 0 || y == 0 || d < MIN_NORMAL) {
    d < (eps * MIN_NORMAL)
  } else d / (xa + ya) < eps
}
```

Error propogation

Often you will see benign errors in floating point calculations.

```
val ns = (0 until 1000).map(_ ⇒ 0.01).toList
ns.sum // 9.999999999999831
```

The answer should be `10`, but this is close right?

<benign/malignant joke here>

```
val ns = (0 until 1000).map(_ ⇒ 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
```

<benign/malignant joke here>

```
val ns = (0 until 1000).map(_ ⇒ 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
ten(1e6)  // 10.000000009313226 weird but ok
```

<benign/malignant joke here>

```scala
val ns = (0 until 1000).map(_ => 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
ten(1e6)  // 10.000000009313226 weird but ok
ten(1e9)  //  9.999990463256836 back below 10
```

<benign/malignant joke here>

```scala
val ns = (0 until 1000).map(_ => 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
ten(1e6)  // 10.000000009313226 weird but ok
ten(1e9)  //  9.999990463256836 back below 10
ten(1e12) // 10.009765625 back above 10...?
```

<benign/malignant joke here>

```scala
val ns = (0 until 1000).map(_ => 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
ten(1e6)  // 10.000000009313226 weird but ok
ten(1e9)  //  9.999990463256836 back below 10
ten(1e12) // 10.009765625 back above 10...?
ten(1e15) //  0.0 wait what!?!?!?
```

<benign/malignant joke here>

```scala
val ns = (0 until 1000).map(_ => 0.01).toList

def ten(big: Double) = (big :: ns).sum - big

ten(0.0)  //  9.999999999999831 same, great
ten(1e6)  // 10.000000009313226 weird but ok
ten(1e9)  //  9.999990463256836 back below 10
ten(1e12) // 10.009765625 back above 10...?
ten(1e15) //  0.0 wait what!?!?!?
ten(1e14) // 15.625 what is this i don't even
```

Greatest Hits

```
10.0 / 0.0 // Infinity
 0.0 / 0.0 // NaN

Double.NaN == Double.NaN      // false
Double.NaN compare Double.NaN // 0

 2.0 * 0.0 //  0.0
-2.0 * 0.0 // -0.0

0.0 == -0.0      // true
0.0 compare -0.0 // 1
```

Is floating point dumb?

1. No.
2. Seriously, no.
3. IEEE-754 is incredibly good work.
4. William Kahan is way smarter than I am.
5. Given a *very hard* problem they did a great job.

See *http://floating-point-gui.de* for links and more info.

Floating point is amazing

- `Long` and `Double` have the same number of bits (64).
- `Long` can't represent fractional values (e.g. 0.5).
- `Long` fails catastrophically at the limits of its range.
- `Long` doesn't do any kind of rounding besides truncation.
- `Double` does all of these with the same number of bits.
- Therefore, `Double` has to "move" the problems somewhere.


MODE CONTROL
ATT HOLD
OFF
RADAR
RCS
UPLINK ACTY
TEMP
NO ATT
GIMBAL LOCK
STBY
PROG
KEY REL
RESTART
OPR ERR
TRACKER
PRIO DISP
ALT
NO DAP
VEL
COMP ACTY
PROG
VERB
NOUN
HARD OVER
DISABLE
VERB
NOUN
+
7
8
9
CLR
ENTR
−
4
5
6
PRO
RSET
0
1
2
3
KEY REL
4

Astronaut training

IEEE-754 was standardized in 1985.

It was intended to be useful for scientists and analysts who:

- Compute error bounds.
- Check for possible floating point traps and errors.
- Understand the structure and limitations of floating point.

(i.e. not true of ~99.9% of us using floating point today)

Problems we've seen

1. Violating mathematical intuitions
2. Using a fixed number of bits to describe numbers.
3. Rounding/precision errors
4. *kth*-roots and transcendental functions

Integers ($\mathbb{Z}$)

Fixing integer overflow is easy.

Use `SafeLong` (or `BigInt` if you hate speed).

```
SafeLong(1000) ** 5
// 1000000000000000

SafeLong(1000) ** 10
// 1000000000000000000000000000000

SafeLong(1000) ** 20
// 1000000000000000000000000000000000000000000000000000000000000000000000
```

Behaves exactly as a mathematician would expect.

What SafeLong has

- Whole numbers (e.g. 5)
- Addition (+), Subtraction (–), Multiplication (∗)
- Quotients (/) and Remainders (%)
- Non-negative integer exponentiation (∗∗)
- Fast implementation
- Guarantee that working with small numbers is fast.

What SafeLong lacks

- Fractional values (e.g. `0.5`)
- Real division
- Roots (e.g. `sqrt`)
- `log`, `exp`, `sin`, and so on...

Rational numbers ($\mathbb{Q}$)

- Fractions like `1/3`, `99/100`, and so on...
- `(p/q)` where `p` and `q` are integers and `(q != 0)`.
- Many ways to write the same number (e.g. 1/2, 2/4, ...)
- Formally, each "number" is the set of all equivalent fractions.

To avoid differing representations of the same number, Spire:

- Always uses 0/1 for zero.
- Always uses simplified fractions (e.g. 64/100 → 16/25).
- Always uses a positive denominator.

(Depending on the magnitude and precision number, Spire will use either `Long` or `BigInt` values internally.)

When doing arithmetic, rational numbers never need to round:

```
val ns = (1 to 5).map(n ⇒ Rational(n, n + 1))
// Vector(1/2, 2/3, 3/4, 4/5, 5/6)

ns.qsum // 71/20

def approx(n: Int) = (1 to n).map(n ⇒ Rational(n, n + 1)).qsum

approx(10) //            221209/27720
approx(20) //          89778475/5173168
approx(30) // 19474808125534 43/72201776446800
```

```
val q = approx(100)

// 269847828294911352159474207232539025749555 89
// divided by
// 281670315928038407744716588098661706369472

q.toInt // 95
```

Again, no surprises for mathematicians.

However, there are possibly some surprises for us.

When adding rationals with very different[*] bases:

- A common denominator has to be found
- This might be much larger than the inputs

```
r"3/17" + r"4/19" + r"5/23" + r"6/29"
// 174784/215441
```

This is not the case for multiplication/division.

What Rational has

- Fractional numbers (e.g. 5, 1/5, 200/7, etc.)
- Addition (+), Subtraction (–), Multiplication (∗)
- Division (/), Quotient (/~), and Remainders (%)
- Integer exponentiation (∗∗)

What Rational lacks

- Roots (e.g. `sqrt`)
- `log`, `exp`, `sin`, and so on...
- Guarantee that working with small numbers is fast.
- Guarantee that adding numbers is fast.

Real numbers ($\mathbb{R}$)

"Math Stuff"

- The set is described as *uncountably infinite*.
- Contains the rational numbers.
- Also contains irrational numbers, which can be:
  - Algebraic (e.g. $\sqrt{2}$)
  - Transcendental (e.g. π)
- *Almost all* of the numbers in $\mathbb{R}$ are transcendental.

"More Math Stuff"

Each Real number (`x` $\in$ $\mathbb{R}$) can be thought of as any of:

- Infinite sequence of approximations "getting closer" to `x`.
- Partition of the rationals into (`_ < x`) and (`x <= _`).
- *<insert explanation of Dedekind-completeness here>*

(Disclaimer: I was not great at Real Analysis in college.)

What does all of that mean?

- Many values in $\mathbb{R}$ are difficult/impossible to represent.
- We may need to use approximations:
  - Think of what we saw with `Double` earlier.
  - We will need to consider limiting error bounds.
- Also... *Real numbers may not actually exist!*

The Finitist Opinion

"God made the integers, all else is the work of man."

Leopold Kronecker

The Classical Opinion

"No one shall expel us from the Paradise that Cantor has created."

David Hilbert

Computers make us all finitists and constructivists.

So what can we do?

We cheat.

The *computable reals* are a countable subset of $\mathbb{R}$.

- In principle, "almost all" of numbers in $\mathbb{R}$ are uncomputable.
- In practice, it is fairly hard to "find" uncomputable numbers.

http://math.stackexchange.com/questions/462790/are-there-any-examples-of-non-computable-real-numbers

How are computable numbers defined?

A computable number `x` is defined as a function.

Given `bits`, return `num` so `(num / 2^bits)` is "closest" to `x`.

Basic idea in Scala

```scala
case class CR(f: Int ⇒ SafeLong) {
  // Approximate this real value as
  // a rational number with k bits,
  // i.e. as (n / 2^k).
  def apply(bits: Int): Rational = {
    val denom = SafeLong(2) ** bits
    Rational(f(bits), denom)
  }
}
```

Parenthetical aside

This is implemented in Spire as `Real`.

There are some complications we do for speed/accuracy.

But the idea is the same.

Design credit

The original design is based on the ERA library for Haskell:

1. "The World's Shortest Correct Exact Real Arithmetic Program?" David Lester *http://www.sciencedirect.com/science/article/pii/S0890540112000727*

2. *https://hackage.haskell.org/package/numbers-2009.8.9/docs/src/Data-Number-CReal.html*

(Spire extends this basic design to speed up "common cases".)

How does it all work?

Error occurs when approximations are "too far" from the truth.

Imaging we are approximating values with one decimal place.

```
x is 1.1 (approximating 1.14)
y is 2.8 (approximating 2.84)

z = x + y
z is 3.9 (approximating 3.98)
```

A better approximation would have been `4.0`.

How does it all work?

To approximation an arithmetic operation, we usually require more precision on the inputs than we are expected to produce.

(Numerical analysts spend a lot of time figuring out exactly how to do this.)

How does it all work?

Here's how to implement addition on computable reals:

```
def roundDiv(n: SafeLong, d: SafeLong) =
  (n + (d / 2)) / d

def addition(x: CR, y: CR): CR =
  CR({ bits ⇒
    val nx = x(bits + 2)
    val ny = y(bits + 2)
    roundDiv(nx + ny, 4)
  })
```

How does it all work?

Negation is really easy!

```
def negation(x: CR): CR =
  CR(bits ⇒ -x(bits))
```

So are max and min!

```
def min(x: CR, y: CR): CR =
  CR(bits ⇒ x(bits) min y(bits))


def max(x: CR, y: CR): CR =
  CR(bits ⇒ x(bits) max y(bits))
```

Things start getting trickier

```
def sizeBaseTwo(n: SafeLong) = log2(n) + 1

def multiplication(x: CR, y: CR): CR =
  CR({ bits ⇒
    val sx = sizeBaseTwo(x(0).abs + 2) + 3
    val sy = sizeBaseTwo(y(0).abs + 2) + 3
    val nx = x(bits + sy)
    val ny = y(bits + sx)
    roundDiv(nx * ny, 2 ** (bits + sx + sy))
  })
```

The Calculus Affair

Let's try something harder: `exp(x)`.

This is `(e^x)` which is can be written as an infinite series:

```
exp(x) = 1 + x + (x^2 / 2!) + (x^3 / 3!) + ...

// ! is for factorial, n! = 1 * 2 * 3 * ... * n
```

The Calculus Affair

Let's simulate that in Scala to see what is going on:

```
def terms(x: SafeLong, n: Int): Stream[Rational] =
  Rational(x ** n, n.!) #:: terms(x, n + 1)

val series = 1 #:: terms(SafeLong.two, 1)

series.take(10).toList
// List(1, 2, 2, 4/3, 2/3, 4/15, 4/45, 8/315, 2/315, 4/2835)
```

The Calculus Affair

How well does this do?

```
math.exp(2) // 7.389056098930 65
```

```
series.take(2).qsum.toDouble  // 3.0
series.take(5).qsum.toDouble  // 7.0
series.take(10).qsum.toDouble // 7.388712522045855
series.take(20).qsum.toDouble // 7.389056098930174
series.take(30).qsum.toDouble // 7.389056098930 65
```

It works!

The Calculus Affair

We can get more precise estimates than `math.exp`:

```
series.take(40).qsum.toBigDecimal
// 7.389056098930650227230427460575007 81

series.take(80).qsum.toBigDecimal
// 7.389056098930650227230427460575 0
//   0781318031557055184732408712782
//   2522573796079057763843124850 79
```

This is how we handle irrational numbers:

- Find an infinite (but computable) series for `x`.
- Determine how many terms we need for a given precision.
- Compute the terms, add them, return the approximation.

(In practice we want to find "quickly converging" series.)

How does this look with Real?

```
Real(2)            // 2
Real(2).sqrt       // 1.4142135623730950488016887242096980785697
Real(2).nroot(3)   // 1.2599210498948731647672106072782283505703
Real(2).nroot(10)  // 1.0717734625362931642130063250233420229064

Real.pi            // 3.1415926535897932384626433832795028841972
Real.e             // 2.7182818284590452353602874713526624977572
Real.exp(2)        // 7.3890560989306502272304274605750078131803
```

Euler would be pleased

```
import Complex.i
import Real.{e, pi, one}

val i = Complex.i[Real]
val e = Complex(Real.e)

e ** (i * pi) + one // (0 + 0i)
```

What's the catch?

There is a catch, right?

You got me

Achilles Heel

Computable numbers have one major flaw:

There's no 100% reliable way to compare them.

Achilles Heel

Computable numbers have one major flaw:

There's no 100% reliable way to compare them.

What? That's crazy

Let me explain...

It depends what you mean

Remember that computable reals produce *approximations*

We can get as good an approximation as we want...

...but we have to ask for it.

(And we only get what we ask for.)

Caveats

1. When we have a computable real `r`
2. ...and we ask for an "n-bit" approximation
3. ...and get back `x`
4. ...we know `x` is the best n-bit approximation for `r`.

Crucially, we don't know if `(x < r)`, `(x = r)`, or `(r < x)`.

When is this a problem?

It's not a problem when being "near" is fine.

It *is* a problem testing for branches:

```
// for input near zero, output is also near zero.
def probablyOk(r: Real): Real =
  if (r < 0) r * 3 else r * 2

// it's possible small non-zero inputs values will
// have wildly-wrong output values.
def ughOh(r: Real): Real =
  if (r == 0) Real(0) else Real(100)
```

Couldn't you ask for more precision?

How would you know if it's ever enough?

You could get a correct sign for any non-zero number...

...provided you're willing to let the program hang on zero.

When is this a problem?

- Sign tests (e.g. `.signum`)
- `compare`, `<`, `==`, and so on
- Trig functions (e.g. `atan`) which require branching

However, methods like `abs`, `min`, and `max` are fine.

Future plans for Real

Currently `Real` provides these methods that can have error:

- It's possible we will make it restrictive-by-default soon.
- If so, an import will restore the current behavior.
- We should make precision used for sign tests pluggable.
- Your feedback and ideas are valuable.

Alternative solutions

There is a Spire branch which implements continued fractions:

- Alternative strategy from the one presented here.
- Different performance characteristics.
- Mostly the same drawbacks

(Continued fractions are fairly brain-bending.)

Alternative solutions

Spire also supports an `Algebraic` type:

1. Able to do exact sign tests.
2. Supports algebraic irrational numbers (e.g. roots)
3. Does not support transcendental numbers.

Often useful in conjunction with the `FpFilter` type.

(Tom Switzer should do a presentation on this!)

Alternative solutions

1. "On Guaranteed Accuracy Computation." C. K. Yap. *http://www.cs.nyu.edu/exact/doc/guaranteed.pdf*
2. "A Separation Bound for Real Algebraic Expressions." C. Burnikel, et al. *http://stubber.math-inf.uni-greifswald.de/informatik/PEOPLE/Papers/ESA01/sepbound01.pd*
3. "A New Constructive Root Bound for Algebraic Expressions." C. Li and C. Yap.

Alternative solutions

You could imagine implementing symbolic evaluation.

(Similar to systems like *Mathematica* or *Sympy*.)

That would be... a *lot* of work.

☞ There is no magic bullet. ☜

Summary

1. Int and Long are fast and dumb.
2. SafeLong is still pretty fast but safe.
3. Double is fast and as-smart-as-possible, but no smarter.
4. Using primitives "by default" is probably not smart.
5. If you use Double you already work with approximations.
6. Use Rational for exactness, and Real for the "best" estimates.
7. There is no silver bullet.

Summary (continued)

- Previous talks have discussed generic numeric code.
- Spire goes to great lengths to support generic algebra.
- Hopefully these trade-offs help motivate that work.

You can learn more about Spire's types and type classes in the *Spire User's Guide*:

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

the end