Updated: 2021-08-31

6.3
Numeric Programming

Northeastern
University

Numbers

- What's the first property of numbers that we should care about?
  - Order. That's why *Numeric[T]* extends *Ordering[T]*
  - *Numeric[T]* supports most of the simple operators, except for *div*
  - *Fractional[T]* supports *div*
  - Our *Rational* type extends *Fractional[Rational]*.
- But, to be honest, the Scala numbers aren't a huge improvement on Java. Most things are still done with *Double*. But *Double* is imprecise. Stuff like 2.9999999999 are just silly, right?
- An alternative is Spire. https://github.com/non/spire
- Both exact and inexact types. Example of exact type is *Real*. This will always be as precise as it can possibly be.

Mathologer video: this is just as valid a representation of 3

Using *Numeric*

- *Numeric[T]* is a trait / type-class and extends *Ordering[T]*
  - Methods:
    - *compare(x: T, y: T): Int*
    - *fromInt(x: Int): T*
    - *plus(x: T, y: T): T*
    - *toDouble(x: T): Double*
    - *zero: T = fromInt(0)*
    - etc.
  - The *sum* method in *IterableOnce[A]* is defined:

```
def sum[B >: A](implicit num: Numeric[B]): B = foldLeft(num.zero)(num.plus)
```

- *Numeric* now has a method:
  - *parseString(s: String): Option[T]*

Using *Fractional*

- *Fractional[T]* is a trait / type-class and extends *Numeric[T]*

  - Methods:
    - *div(x: T, y: T): T*
    - etc.

  - Standard imports:

```
trait DoubleIsFractional extends DoubleIsConflicted with Fractional[Double] {
    def div(x: Double, y: Double): Double = x / y
}
```

Numeric Parsing

- Our previous parser worked fine but it had the following issues:
  - If output doesn’t parse correctly, we will get an appropriate message. But, if anything goes wrong in the logic behind the parsers (e.g. division by zero) an exception will be thrown
  - Output type is fixed as *Double* which is not a very good type
  - Let’s try to improve it…

Improving our arithmetic expression parser

- How about we return *Try[T]* from each parser (generalizing the output type and wrapping it in *Try*)?
  - originally we had:

```
class Arith extends JavaTokenParsers {
  trait Expression {
    def eval: Double
  }
  abstract class Factor extends Expression
  case class Expr(t: Term, ts: List[String~Term]) extends Expression {
    def term(t: String~Term): Double = t match {case "+"~x => x.eval; case "-"~x => -x.eval }
    def eval = ts.foldLeft(t.eval)(_ + term(_))
  }
```

  - so now, let's do this (we also change its name to *ExpressionParser*):

```
abstract class ExpressionParser[T] extends JavaTokenParsers with (String => Try[T]) { self =>
  def apply(s: String): Try[T]
  def negate: (T)=>T
  def plus: (T,T)=>T
  trait Expression {
    def value: Try[T]
  }
  …
}
```

Improving (2)

- so we need:

```
def negate: (T)=>T = ???
def plus: (T,T)=>T = ???

case class Expr(t: Term, ts: List[String~Term]) extends Expression {
  def termVal(t: String~Term): Try[T] = t match {case "+"~x => x.value; case "-"~x =>
negate(x.value) }
  def value = ts.foldLeft(t.value)((a,x) => plus(a,termVal(x)))
}
```

- and therefore we need:

```
def lift(t: Try[T])(f: (T) => T): Try[T] = ???
def map2(t1: Try[T], t2: Try[T])(f: (T,T) => T): Try[T] = ???

case class Expr(t: Term, ts: List[String~Term]) extends Expression {
  def termVal(t: String~Term): Try[T] = t match {case "+"~x => x.value; case "-"~x =>
lift(x.value)(negate) }
  def value = ts.foldLeft(t.value)((a,x) => map2(a,termVal(x))(plus))
}
```

- how to implement *lift* and *map2*? The usual way!

```
def lift(t: Try[T])(f: (T) => T): Try[T] = t map f
def map2(t1: Try[T], t2: Try[T])(f: (T,T) => T): Try[T] = for { tt1 <- t1 ; tt2 <- t2 }
yield f(tt1,tt2)
```

Using Rational

- Here are examples of using *Rational* and its parser:

```
scala> import edu.neu.coe.scala.parse.RationalExpressionParser
import edu.neu.coe.scala.parse.RationalExpressionParser

scala> val parser = RationalExpressionParser
parser: edu.neu.coe.scala.parse.RationalExpressionParser.type = <function1>

scala> parser.parseAll(parser.expr,"2/3")
res2: parser.ParseResult[parser.Expr] = [1.4] parsed:
Expr(Term(FloatingPoint(2),List((/~FloatingPoint(3)))),List())

scala> res2.get.value
res3: scala.util.Try[edu.neu.coe.scala.numerics.Rational] = Success(Rational(2,3))
scala> import edu.neu.coe.scala.numerics.Rational
import edu.neu.coe.scala.numerics.Rational

scala> import edu.neu.coe.scala.numerics.Rational._
import edu.neu.coe.scala.numerics.Rational._

scala> implicit def convert(x: Int): Rational = Rational.apply(x)
warning: there was one feature warning; re-run with -feature for details
convert: (x: Int)edu.neu.coe.scala.numerics.Rational

scala> implicit def convert(s: String): Rational = Rational.apply(s)
warning: there was one feature warning; re-run with -feature for details
convert: (s: String)edu.neu.coe.scala.numerics.Rational

scala> val l = List[Rational]("1", "2/3", "5")
l: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(1,1), Rational(2,3), Rational(5,1))

scala> val (x,y) = (Rational(2,3),Rational(4,5))
x: edu.neu.coe.scala.numerics.Rational = Rational(2,3)
y: edu.neu.coe.scala.numerics.Rational = Rational(4,5)
```

Sorting on Rational

- And now, let's try to create a list of Rationals and sort it.

```
scala> val l = List[Rational]("1", "2/3", "5")
l: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(1,1), Rational(2,3),
Rational(5,1))
scala> l.sorted
res3: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(2,3), Rational(1,1),
Rational(5,1))
```

Some observations on precision

- Scientific/Engineering Observations
  - You observe some natural phenomenon, such as the period of oscillation of a pendulum.
  - Let's say your observations are (in seconds):
    - 6.4, 6.3, 6.5, 6.2, 6.4, 6.3, 6.5, 6.4, 6.2, 6.3
  - These are different! Does that mean that nine of them are wrong and one is right? Of course not!
  - We say that our best estimate of the period is 6.35s with a standard deviation of 0.1. That means we believe of the true period *T* that:
    - $6.25 < T < 6.45$ with 68% confidence; and
    - $6.15 < T < 6.55$ with 95% confidence.
  - We write this scientifically as $T = 6.35(10)$ seconds.

Precision, continued

- Let's say that what we really want to know is the length of the pendulum:
  - $l = g\,T^2 / 4\,\pi^2$
  - So, we calculate its value as 10.02 m.
- But wait a moment! Our estimate of *T* wasn't known to great precision. Come to think of it, we don't know *g* very precisely either (we do know π pretty well!)
- So, what's the precision of *l*?
  - T^2 is known: 40.32(20)
  - g is known: 9.805(10)
  - In general, if f = f(x,y) then $\Delta f = \delta f/\delta x \Delta x + \delta f/\delta y \Delta y$
  - We can simply add the relative errors to get 0.51%
  - Our estimate of *l* is therefore: 10.020(51)

Note that we doubled the standard deviation (0.5% relative error bound)

(0.1% relative error bound)

Fuzzy

```scala
trait Fuzzy[+T] {
  /**
    * Get method to return an exact value
    * @return the exact value of this Fuzzy, otherwise an exception is thrown
    * @throws FuzzyException
    */
  def get: T
  /**
    * Return true if this value is exact
    * @return true if exact, else false
    */
  def isExact: Boolean
  /**
    * Return a measure of the probability that this fuzzy instance could be equal to u.
    * If U is a continuous type, we return the probability density function at u.
    * If U is a discrete type, we return the actual probability that this equals u.
    * @param u
    * @tparam U
    * @return the probability that this is equal to u.
    */
  def pdf[U >: T : Discrete](u: U): Double
  /**
    * Map this fuzzy instance into an instance of Fuzzy[U].
    * @param f
    * @tparam U
    * @return
    */
  def map[U >: T : Discrete](f: T=>U)(implicit ev1: (U=>Double)): Fuzzy[U] = flatMap(t => unit(f(t)))
  /**
    * FlatMap this fuzzy instance into an instance of Fuzzy[U].
    * @param f
    * @tparam U
    * @return
    */
  def flatMap[U >: T : Discrete](f: T=>Fuzzy[U]): Fuzzy[U]
  def unit[U >: T : Discrete](u: U)(implicit ev1: (U=>Double)): Fuzzy[U]
  def foreach(f: T=>Unit): Unit
}
```

Libraries

- Spire
- Apache Commons math3 (Java library)
- Number: https://github.com/rchillyard/Number