Rings Documentation

Release SNAPSHOT

Stanislav Poslavsky

Oct 08, 2017

Contents

1 Installation **3**

2 Quick tour **5**
- 2.1 Univariate polynomial factorization . . . 5
- 2.2 Multivariate polynomial factorization . . . 7
- 2.3 Polynomial GCD . . . 9
- 2.4 Constructing arbitrary rings . . . 10

3 User guide **13**
- 3.1 Integers . . . 13
  - 3.1.1 Prime numbers . . . 13
- 3.2 Modular arithmetic with machine integers . . . 14
- 3.3 Rings . . . 15
  - 3.3.1 Examples of rings . . . 16
    - 3.3.1.1 Galois fields . . . 17
    - 3.3.1.2 Fields of fractions . . . 18
  - 3.3.2 Scala DSL . . . 19
- 3.4 Polynomials and polynomial rings . . . 19
  - 3.4.1 Univariate polynomials . . . 22
    - 3.4.1.1 Division with remainder . . . 22
    - 3.4.1.2 Univariate GCD . . . 23
    - 3.4.1.3 Univariate factorization . . . 25
    - 3.4.1.4 Testing irreducibility . . . 26
    - 3.4.1.5 Univariate interpolation . . . 27
  - 3.4.2 Multivariate polynomials . . . 29
    - 3.4.2.1 Division with remainders . . . 29
    - 3.4.2.2 Multivariate GCD . . . 29
    - 3.4.2.3 Multivariate factorization . . . 29
    - 3.4.2.4 Multivariate Interpolation . . . 29
  - 3.4.3 Multivariate polynomials . . . 29

4 Index of algorithms implemented in Rings **31**
- 4.1 Univariate polynomials . . . 31
- 4.2 Multivariate polynomials . . . 33
- 4.3 References . . . 34

Bibliography **35**

Rings is a very efficient implementation of both univariate and multivariate polynomial algebra over arbitrary coefficient rings. **Rings** make use of asymptotically fast algorithms for basic algebraic operations as well as for advanced methods like GCDs and polynomial factorization and give performance comparable (or even better in some cases) to well known solutions like Singular/NTL/FLINT/Maple/Mathematica.

Features:

- Univariate and multivariate polynomials over arbitrary coefficient rings
- Polynomial factorization and GCDs over arbitrary coefficient rings
- Galois fields and abstract algebra
- Efficient implementation and overall high performance

CHAPTER 1

Installation

Rings are written in Java with DSL written in Scala. The Java library is available under `cc.redberry.rings` artifact from Maven Central:

```
<dependency>
    <groupId>cc.redberry</groupId>
    <artifactId>rings</artifactId>
    <version>1.0</version>
</dependency>
```

Scala interface with DSL is available with both Maven:

```
<dependency>
    <groupId>cc.redberry</groupId>
    <artifactId>rings.scaladsl</artifactId>
    <version>1.0</version>
</dependency>
```

and SBT:

```
libraryDependencies += "cc.redberry" % "rings.scaladsl" % "1.0"
```

Sources are avalable at GitHub: https://github.com/PoslavskySV/rings

CHAPTER 2

Quick tour

The following imports will be in general enough for most examples in this documentation:

Scala

Java

```
import cc.redberry.rings.poly.PolynomialMethods._
import cc.redberry.rings.scaladsl.Rings._
import cc.redberry.rings.scaladsl.implicits._
```

```
import cc.redberry.rings.*
import cc.redberry.rings.poly.*
import cc.redberry.rings.poly.univar.*
import cc.redberry.rings.poly.multivar.*

import static cc.redberry.rings.poly.PolynomialMethods.*
import static cc.redberry.rings.Rings.*
```

Univariate polynomial factorization

Factor polynomial in $Z_{17}[x]$:

Scala

Java

```
// Define ring Z/17[x]
implicit val ring = UnivariateRingZp64(17, "x")

// parse univariate poly from string
val poly = ring("4 + 8*x + 12*x^2 + 5*x^5 - x^6 + 10*x^7 + x^8")
```

```
// factorize poly
val factors = Factor(poly)
println(s"factorization : ${ring show factors}")
```

```
// the modulus
long modulus = 17;
// parse univariate poly over Z/17 from string
UnivariatePolynomialZp64 poly = UnivariatePolynomialZp64
    .parse("4 + 8*x + 12*x^2 + 5*x^5 - x^6 + 10*x^7 + x^8", modulus);

// factorize poly
FactorDecomposition<UnivariatePolynomialZp64> factors = PolynomialMethods.
↪Factor(poly);
System.out.println(factors);
```

This will give the following factorization:

$$\begin{aligned} &4 + 8x + 12x^2 + 5x^5 - x^6 + 10x^7 + x^8 = \\ &\quad = (6 + x)(8 + x)(14 + x)^2(15 + x)(7 + x + 4x^2 + x^3) \quad \mod 17 \end{aligned}$$

Coefficient rings with arbitrary large characteristic are available:

Scala

Java

```
// coefficient ring Z/1237940039285380274899124357 (the next prime to 2^100)
val cfRing = Zp(new BigInteger("1267650600228229401496703205653"))

// The ring Z/1237940039285380274899124357[x]
implicit val ring = UnivariateRing(cfRing, "x")

val poly = ring("4 + 8*x + 12*x^2 + 5*x^5 + 16*x^6 + 27*x^7 + 18*x^8")
println(s"factorization : ${ring show Factor(poly)}")
```

```
//// coefficient ring Z/1237940039285380274899124357 (the next prime to 2^100)
IntegersZp cfRing = Rings.Zp(new BigInteger("1267650600228229401496703205653"));

// parse univariate poly over Z/1267650600228229401496703205653 from string
UnivariatePolynomial<BigInteger> poly = UnivariatePolynomial
    .parse("4 + 8*x + 12*x^2 + 5*x^5 - x^6 + 10*x^7 + x^8", cfRing);

// factorize poly
FactorDecomposition<UnivariatePolynomial<BigInteger>> factors
                = PolynomialMethods.Factor(poly);
System.out.println(factors);
```

(large primes can be generated with `BigPrimes.nextPrime(BigInteger)` method).

This will give the following factorization:

$$\begin{aligned}
&4+8x+12x^2+5x^5-x^6+10x^7+x^8 = \\
&\quad = (448975734644581867134339749139+x)\times \\
&\qquad \times(924109545982468663492425885021+ \\
&\qquad +396701390689518624208584222054x+ \\
&\qquad +671565661754860153453068172251x^2+x^3)\times \\
&\qquad \times(493224222589341667858050863719+ \\
&\qquad +336789330550038195919829925685x+ \\
&\qquad +636344447485090019332177467857x^2+ \\
&\qquad +147109203828787380909295284273x^3+ \\
&\qquad\qquad +x^4) \quad \text{mod } 1267650600228229401496703205653
\end{aligned}$$

Multivariate polynomial factorization

Factor polynomial in $Z_2[x, y, z]$:

Scala

Java

```
// The ring Z/2[x, y, z]
implicit val ring = MultivariateRingZp64(2, Array("x", "y", "z"))

val (x, y, z) = ring("x", "y", "z")
// factorize polynomial
val factors = Factor(1 + (1 + x + y + z)**2 + (x + y + z)**4)
println(s"factorization : ${ring show factors}")
```

```
// coefficient ring Z/2
IntegersZp64 cfRing = new IntegersZp64(2);
MultivariatePolynomialZp64
        // create unit multivariate polynomial over
        // 3 variables over Z/2 using LEX ordering
        one = MultivariatePolynomialZp64.one(3, cfRing, MonomialOrder.LEX),
        // create "x" polynomial
        x = one.createMonomial(0, 1),
        // create "y" polynomial
        y = one.createMonomial(1, 1),
        // create "z" polynomial
        z = one.createMonomial(2, 1);

// (1 + x + y + z)^2
MultivariatePolynomialZp64 poly1 = one.copy().add(x, y, z);
poly1 = polyPow(poly1, 2);

// (x + y + z)^4
MultivariatePolynomialZp64 poly2 = x.copy().add(y, z);
poly2 = polyPow(poly2, 4);

// 1 + (1 + x + y + z)^2 + (x + y + z)^4
```

```
MultivariatePolynomialZp64 poly = one.copy().add(poly1, poly2);
FactorDecomposition<MultivariatePolynomialZp64> factors = PolynomialMethods.
↪Factor(poly);
System.out.println(factors);
```

This will give the following factorization:

$$1 + (1 + x + y + z)^2 + (x + y + z)^4 = (x + y + z)^2\,(1 + x + y + z)^2 \mod 2$$

Factor polynomial in $Z[a, b, c]$:

Scala

Java

```
// The ring Z[a, b, c]
implicit val ring = MultivariateRing(Z, Array("a", "b", "c"))

val (a, b, c) = ring("a", "b", "c")
// factorize polynomial
val factors = Factor(1 - (1 + a + b + c)**2 - (2 + a + b + c)**3)
println(s"factorization : ${ring show factors}")
```

```
MultivariatePolynomial<BigInteger>
        // create unit multivariate polynomial over
        // 3 variables over Z using LEX ordering
        one = MultivariatePolynomial.one(3, Rings.Z, MonomialOrder.LEX),
        // create "a" polynomial
        a = one.createMonomial(0, 1),
        // create "b" polynomial
        b = one.createMonomial(1, 1),
        // create "c" polynomial
        c = one.createMonomial(2, 1);

// (1 + a + b + c)^2
MultivariatePolynomial<BigInteger> poly1 = one.copy().add(a, b, c);
poly1 = polyPow(poly1, 2);

// (2 + a + b + c)**3
MultivariatePolynomial<BigInteger> poly2 = one.copy().multiply(2).add(a, b, c);
poly2 = polyPow(poly2, 3);

// 1 - (1 + a + b + c)^2 - (2 + a + b + c)**3
MultivariatePolynomial<BigInteger> poly = one.copy().subtract(poly1, poly2);
FactorDecomposition<MultivariatePolynomial<BigInteger>> factors
                = PolynomialMethods.Factor(poly);
System.out.println(factors);
```

Will give the following factorization:

$$1 - (1 + a + b + c)^2 - (2 + a + b + c)^3 = -(1 + a + b + c)(2 + a + b + c)(4 + a + b + c)$$

Factor polynomial in $Q[x, y, z]$:

Scala

Java

```
implicit val ring = MultivariateRing(Q, Array("x", "y", "z"))

val poly = ring(
  """
    |(1/6)*y*z + (1/6)*y^3*z^2 - (1/2)*y^6*z^5 - (1/2)*y^8*z^6
    |-(1/3)*x*z - (1/3)*x*y^2*z^2 + x*y^5*z^5 + x*y^7*z^6
    |+(1/9)*x^2*y^2*z - (1/3)*x^2*y^7*z^5 - (2/9)*x^3*y*z
    |+(2/3)*x^3*y^6*z^5 - (1/2)*x^6*y - (1/2)*x^6*y^3*z
    |+x^7 + x^7*y^2*z - (1/3)*x^8*y^2 + (2/3)*x^9*y
  """.stripMargin)

// factorize polynomial (in this example there will be 3 factors)
val factors = Factor(poly)
println(s"factorization : ${ring show factors}")
```

```
MultivariatePolynomial<Rational<BigInteger>>
                poly = MultivariatePolynomial.parse(
                        "(1/6)*y*z + (1/6)*y^3*z^2 - (1/2)*y^6*z^5 - (1/2)*y^8*z^6" +
                "-(1/3)*x*z - (1/3)*x*y^2*z^2 + x*y^5*z^5 + x*y^7*z^6" +
                "+(1/9)*x^2*y^2*z - (1/3)*x^2*y^7*z^5 - (2/9)*x^3*y*z" +
                "+(2/3)*x^3*y^6*z^5 - (1/2)*x^6*y - (1/2)*x^6*y^3*z" +
                "+x^7 + x^7*y^2*z - (1/3)*x^8*y^2 + (2/3)*x^9*y"
                , Rings.Q);

System.out.println(PolynomialMethods.Factor(poly));
```

Polynomial GCD

Univariate extended GCD in $Z_{17}[x]$:

Scala

Java

```
// The ring Z/17[x]
implicit var ring = UnivariateRingZp64(17, "x")

val x = ring("x")

val xgcd = PolynomialExtendedGCD(1 + x + x.pow(2) + x.pow(3), 1 + 2*x + 9*x.pow(2))
println(s"XGCD : ${ring show xgcd}")
```

```
UnivariatePolynomialZp64
        a = UnivariatePolynomialZ64.create(1, 1, 1, 1).modulus(17),
        b = UnivariatePolynomialZ64.create(1, 2, 9).modulus(17);

System.out.println(Arrays.toString(UnivariateGCD.PolynomialExtendedGCD(a, b)));
```

Multivariate GCD in $Z[a, b, c]$:

Scala

Java

```
// The ring Z[a, b, c]
implicit val ring = MultivariateRing(Z, Array("a", "b", "c"))

println(PolynomialGCD(
        ring("-b-b*c-b^2+a+a*c+a^2"),
        ring("b^2+b^2*c+b^3+a*b^2+a^2+a^2*c+a^2*b+a^3")))
```

```
MultivariatePolynomial<BigInteger>
        a = MultivariatePolynomial.parse("-b-b*c-b^2+a+a*c+a^2", Rings.Z),
        b = MultivariatePolynomial.parse("b^2+b^2*c+b^3+a*b^2+a^2+a^2*c+a^2*b+a^3",␣
↪Rings.Z);

System.out.println(PolynomialMethods.PolynomialGCD(a, b));
```

Constructing arbitrary rings

Polynomial rings over Z and Q:

Scala

Java

```
// Ring Z[x]
UnivariateRing(Z, "x")
// Ring Z[x, y, z]
MultivariateRing(Z, Array("x", "y", "z"))
// Ring Q[a, b, c]
MultivariateRing(Q, Array("a", "b", "c"))
```

```
// Ring Z[a]
Rings.UnivariateRing(Rings.Z);
// Ring Z[a, b, c]
Rings.MultivariateRing(3, Rings.Z);
// Ring Q[a, b, c]
Rings.MultivariateRing(3, Rings.Q);
```

Polynomial rings over Z_p:

Scala

Java

```
// Ring Z/3[x] (64 indicates that machine numbers are used in the basis)
UnivariateRingZp64(3, "x")
// Ring Z/3[x, y, z]
MultivariateRingZp64(3, Array("x", "y", "z"))
// Ring Z/p[x, y, z] with p = 2^107 - 1 (Mersenne prime)
MultivariateRing(Zp(BigInt(2).pow(107) - 1), Array("x", "y", "z"))
```

```
// Ring Z/3[a] (64 indicates that machine numbers are used in the basis)
Rings.UnivariateRingZp64(3);
// Ring Z/3[a, b, c]
Rings.MultivariateRingZp64(3, 3);
// Ring Z/p[a, b, c] with p = 2^107 - 1 (Mersenne prime)
Rings.MultivariateRing(3, Rings.Zp(BigInteger.ONE.shiftLeft(107).decrement()));
```

Galois fields:

Scala

Java

```
// Galois field with cardinality 7^10
// (irreducible polynomial will be generated automatically)
GF(7, 10, "x")
// GF(7^3) generated by irreducible polynomial "1 + 3*z + z^2 + z^3"
GF(UnivariateRingZp64(7, "z")("1 + 3*z + z^2 + z^3"), "z")
```

```
// Galois field with cardinality 7^10
// (irreducible polynomial will be generated automatically)
Rings.GF(7, 10);
// GF(7^3) generated by irreducible polynomial "1 + 3*z + z^2 + z^3"
Rings.GF(UnivariatePolynomialZ64.create(1, 3, 1, 2).modulus(7));
```

Fractional fields:

Scala

Java

```
// Field of fractions of univariate polynomials Z[x]
Rationals(UnivariateRing(Z, "x"))
// Field of fractions of multivariate polynomials Z/19[x, y, z]
Rationals(MultivariateRingZp64(19, Array("x", "y", "z")))
```

```
// Field of fractions of univariate polynomials Z[a]
Rings.Rationals(Rings.UnivariateRing(Rings.Z));
// Field of fractions of multivariate polynomials Z/19[a, b, c]
Rings.Rationals(Rings.MultivariateRingZp64(3, 19));
```

Ring of univariate polynomials over elements of Galois field $GF(7^3)[x]$:

Scala

Java

```
// Elements of GF(7^3) are represented as polynomials
// over "z" modulo irreducible polynomial "1 + 3*z + z^2 + z^3"
val cfRing = GF(UnivariateRingZp64(7, "z")("1 + 3*z + z^2 + z^3"), "z")

assert(cfRing.characteristic().intValue() == 7)
assert(cfRing.cardinality().intValue() == 343)

// Ring GF(7^3)[x]
implicit val ring = UnivariateRing(cfRing, "x")
```

```
// Coefficients of polynomials in GF(7^3)[x] are elements of GF(7^3)
val poly = ring("1 - (1 - z^3) * x^6 + (1 - 2*z) * x^33 + x^66")

// factorize poly (in this examples there will be 9 factors)
val factors = Factor(poly)
println(s"${ring show factors}")
```

```
// Elements of GF(7^3) are represented as polynomials
// modulo irreducible polynomial "1 + 3*z + z^2 + z^3"
FiniteField<UnivariatePolynomialZp64> cfRing
        = Rings.GF(UnivariatePolynomialZ64.create(1, 3, 1, 2).modulus(7));
assert cfRing.characteristic().intValue() == 7;
assert cfRing.cardinality().intValue() == 343;

// Ring GF(7^3)[a]
UnivariateRing<UnivariatePolynomial<UnivariatePolynomialZp64>>
        ring = Rings.UnivariateRing(cfRing);

// Coefficients of polynomials in GF(7^3)[a] are elements of GF(7^3)
UnivariatePolynomial<UnivariatePolynomialZp64>
        poly = ring.parse("1 - (1 - z^3) * x^6 + (1 - 2*z) * x^33 + x^66");

// factorize poly (in this examples there will be 9 factors)
FactorDecomposition<UnivariatePolynomial<UnivariatePolynomialZp64>> factors
        = PolynomialMethods.Factor(poly);
System.out.println(factors);
```

Ring of multivariate polynomials over elements of Galois field $GF_{7^3}[x, y, z]$:

Scala

Java

```
// Elements of GF(7^3) are represented as polynomials
// over "z" modulo irreducible polynomial "1 + 3*z + z^2 + z^3"
val cfRing = GF(UnivariateRingZp64(7, "z")("1 + 3*z + z^2 + z^3"), "z")
// Ring GF(7^3)[x]
implicit val ring = MultivariateRing(cfRing, Array("a", "b", "c"))

// Coefficients of polynomials in GF(7^3)[x] are elements of GF343
val poly = ring("1 - (1 - z^3) * a^6*b + (1 - 2*z) * c^33 + a^66")
```

```
// Elements of GF(7^3) are represented as polynomials
// modulo irreducible polynomial "1 + 3*z + z^2 + z^3"
FiniteField<UnivariatePolynomialZp64> cfRing
        = Rings.GF(UnivariatePolynomialZ64.create(1, 3, 1, 2).modulus(7));
assert cfRing.characteristic().intValue() == 7;
assert cfRing.cardinality().intValue() == 343;

// Ring GF(7^3)[a, b, c]
MultivariateRing<MultivariatePolynomial<UnivariatePolynomialZp64>>
        ring = Rings.MultivariateRing(3, cfRing);

// Coefficients of polynomials in GF(7^3)[a, b, c] are elements of GF(7^3)
MultivariatePolynomial<UnivariatePolynomialZp64>
        poly = ring.parse("1 - (1 - z^3) * a^6*b + (1 - 2*z) * c^33 + a^66");
```

CHAPTER 3

User guide

Integers

There are two basic types of integer numbers that we have to deal with when doing algebra in computer: machine integers and arbitrary-precision integers. For the machine integers the Java's primitive 64-bit `long` type is used (since most modern CPUs are 64-bit). Internally Rings use machine numbers for representing integers modulo prime numbers less than 2^{64} which is done for performance reasons (see *Modular arithmetic with machine integers*). For the arbitrary-precision integers Rings use improved `BigInteger` class github.com/tbuktu/bigint (cc.redberry.rings.bigint.BigInteger) instead of built-in `java.math.BigInteger`. The improved BigInteger has Schönhage-Strassen multiplication and Barrett division algorithms for large integers which is a significant performance improvement in comparison to native Java's implementation.

Prime numbers

In many applications it is necessary to test primality of integer number (`isPrime(number)`) or to generate some prime numbers at random (`nextPrime(number)`). This is realized in the following two classes:

- SmallPrimes for numbers less than 2^{32}. It uses *Miller-Rabin* probabilistic primality test for int type in such a way that a result is always guaranteed (code is adapted from Apache Commons Math).
- BigPrimes for arbitrary large numbers. It switches between *Pollard-Rho*, *Pollard-P1* and *Quadratic Sieve* algorithms for prime factorization and also uses probabilistic *Miller-Rabin test* and strong *Lucas test* for primality testing.

The following examples give some illustrations:

Java

```java
int intNumber = 1234567;
// false
boolean primeQ = SmallPrimes.isPrime(intNumber);
// 1234577
int intPrime = SmallPrimes.nextPrime(intNumber);
// [127, 9721]
```

```
int[] intFactors = SmallPrimes.primeFactors(intNumber);


long longNumber = 12345671234567123L;
// false
primeQ = BigPrimes.isPrime(longNumber);
// 12345671234567149
long longPrime = BigPrimes.nextPrime(longNumber);
// [1323599, 9327350077]
long[] longFactors = BigPrimes.primeFactors(longNumber);


BigInteger bigNumber = new BigInteger(
↪"321536584276145124691487234561832756183746531874567");
// false
primeQ = BigPrimes.isPrime(bigNumber);
// 321536584276145124691487234561832756183746531874827
BigInteger bigPrime = BigPrimes.nextPrime(bigNumber);
// [3, 29, 191, 797359, 1579057, 14916359, 1030298906727233717673336103]
List<BigInteger> bigFactors = BigPrimes.primeFactors(bigNumber);
```

Modular arithmetic with machine integers

There is one special ring — ring Z_p of integers modulo prime number $p < 2^{64}$ — which is used in the basis of many fundamental algorithms. In contrast to Z_p with arbitrary large characteristic, for characteristic that fits into 64-bit word one can use machine integers to significantly speed up basic math operations. Operations in Z_p require applying `mod` operation which in turn implies integer division. Integer division is a very slow CPU instruction; and what is more important is that it breaks CPU pipelining. Hopefully, operations in Z_p imply taking `mod` with a fixed modulus p, which allows to make some precomputation beforehand and then reduce integer divisions to multiplications that are over a magnitude times faster. The details of this trick can be found in Hacker's Delight. Rings use libdivide4j library for fast integer division with precomputation which is ported from the well known C/C++ libdivide library. With this precomputation the `mod` operation becomes several times faster than the native CPU instruction, which boosts the overall performance of many of Rings algorithms in more than 3 times.

The ring Z_p with $p < 2^{64}$ is implemented in IntegersZp64 class (while IntegersZp implements Z_p with arbitrary large characteristic). IntegersZp64 defines all arithmetic operations in Z_p:

Java

```
// Z/p with p = 2^7 - 1 (Mersenne prime)
IntegersZp64 field = new IntegersZp64(127);
//     1000 = 111 mod 127
assert field.modulus(1000) == 111;
// 100 + 100 = 73 mod 127
assert field.add(100, 100) == 73;
//  12 - 100 = 39 mod 127
assert field.subtract(12, 100) == 39;
//  55 * 78  = 73 mod 127
assert field.multiply(55, 78) == 99;
//   1 / 43  = 65 mod 127
assert field.reciprocal(43) == 65;
```

It is worst to mention, that multiplication defined in IntegersZp64 is especially fast when characteristic is less than 2^{32}: in this case multiplication of two numbers fits the machine 64-bit word, while in the opposite case Montgomery reduction will be used.

Java

```
// Z/p with p = 2^31 - 1 (Mersenne prime) - fits 32-bit word
IntegersZp64 field32 = new IntegersZp64((1L << 31) - 1L);
// does not cause long overflow - fast
assert field32.multiply(0xabcdef12, 0x12345678) == 0x7e86a4d6;


// Z/p with p = 2^61 - 1 (Mersenne prime) - doesn't fit 32-bit word
IntegersZp64 field64 = new IntegersZp64((1L << 61) - 1L);
// cause long overflow - Montgomery reduction will be used - no so fast
assert field64.multiply(0x0bcdef1234567890L, 0x0234567890abcdefL) ==␣
↪0xf667077306fd7a8L;
```

Implementation note: unfortunately, the price that we pay for fast arithmetic with machine integers is that IntegersZp64 stands separately from the elegant type hierarchy of generic rings implemented in Rings (see section *Rings*); that is because Java doesn't support generics with primitive types. This leads to that some of the fundamental algorithms have two implementations – one for rings over generic elements and one for IntegersZp64 over `longs`.

Rings

The concept of mathematical ring is implemented in the generic interface Ring<E> which defines all basic algebraic operations over the elements of type `E`. The simplest example is the ring of integers Z (Rings.Z), which operates with BigInteger instances and simply delegates all operations like `+` or `*` to methods of class BigInteger. A little bit more complicated ring is a ring of integers modulo some number Z_p:

Java

```
// The ring Z/17
Ring<BigInteger> ring = Rings.Zp(BigInteger.valueOf(17));

//     103 = 1 mod 17
BigInteger el  = ring.valueOf(BigInteger.valueOf(103));
assert  el.intValue() == 1;

// 99 + 88 = 0 mod 17
BigInteger add = ring.add(BigInteger.valueOf(99),
                          BigInteger.valueOf(88));
assert add.intValue() == 0;

// 99 * 77 = 7 mod 17
BigInteger mul = ring.multiply(BigInteger.valueOf(99),
                               BigInteger.valueOf(77));
assert mul.intValue() == 7;

// 1  / 99 = 11 mod 17
BigInteger inv = ring.reciprocal(BigInteger.valueOf(99));
assert inv.intValue() == 11;
```

In fact the interface Ring<E> defines algebraic operations inherent both for *GCD domains*, *Euclidean rings* and *Fields*. These operations can be summarized in the following methods from Ring<E>:

Java

```
// Methods from Ring<E> interface:
```

```
// GCD domain operation:
E gcd(E a, E b);

// Euclidean ring operation:
E[] divideAndRemainder(E dividend, E divider);

// Field operation:
E reciprocal(E element);
```

In the case when a particular ring is (e.g. Z) is not a field, the invocation of field method (`reciprocal`) will produce `ArithmeticException`. Each Ring<E> implementation provides the information about its mathematical nature (ring/Euclidean ring/field) and all properties like cardinality, characteristic etc. Additionally it defines `parse(String)` method to convert strings into ring elements:

Java

```
// Z is not a field
assert Rings.Z.isEuclideanRing();
assert !Rings.Z.isField();
assert !Rings.Z.isFinite();

// Q is an infinite field
assert Rings.Q.isField();
assert !Rings.Q.isFinite();
assert Rings.Q.parse("2/3").equals(
        new Rational<>(Rings.Z, BigInteger.valueOf(2), BigInteger.valueOf(3)));

// GF(2^10) is a finite field
FiniteField<UnivariatePolynomialZp64> gf = Rings.GF(2, 10);
assert gf.isField();
assert gf.isFinite();
assert gf.characteristic().intValue() == 2;
assert gf.cardinality().intValue() == 1 << 10;
System.out.println(gf.parse("1 + z + z^10"));

// Z/3[x] is Euclidean ring but not a field
UnivariateRing<UnivariatePolynomialZp64> zp3x = Rings.UnivariateRingZp64(3);
assert zp3x.isEuclideanRing();
assert !zp3x.isField();
assert !zp3x.isFinite();
assert zp3x.characteristic().intValue() == 3;
assert zp3x.parse("1 + 14*x + 15*x^10").equals(
        UnivariatePolynomialZ64.create(1, 2).modulus(3));
```

Examples of rings

The shortcut methods for different rings are placed in cc.redberry.rings.Rings class (Scala shortcuts are directly in `scaladsl` package object). Below is the list of basic rings defined in Rings:

Ring	Description	Code in Rings
Z	Ring of integers	`Z`
Q	Field of rationals	`Q`
Z_p	Integers modulo p	`Zp(p)`
Z_p with $p < 2^{64}$	Integers modulo $p < 2^{64}$	`Zp64(p)` *[0]
$GF(p^q)$	Galois field with cardinality p^q	`GF(p, q)` or `GF(irred)`
$Frac(R)$	Field of fractions of an integral domain R	`Frac(R)`
$R[x]$	Univariate polynomial ring over coefficient ring R	`UnivariatePolynomials(R)`
$Z_p[x]$ with $p < 2^{64}$	Univariate polynomial ring over coefficient ring Z_p with $p < 2^{64}$	`UnivariatePolynomialsZp64(p)`
$R[x_1, \ldots, x_N]$	Multivariate polynomial ring with exactly N variables over coefficient ring R	`MultivariatePolynomials(N, R)`
$Z_p[x_1, \ldots, x_N]$ with $p < 2^{64}$	Multivariate polynomial ring with exactly N variables over coefficient ring Z_p with $p < 2^{64}$	`MultivariatePolynomialsZp64(N, p)`

*

Galois fields

Galois field $GF(p^q)$ with prime characteristic p and cardinality p^q can be created by specifying p and q in which case the irreducible polynomial will be generated automatically or by explicitly specifying the irreducible:

Scala

Java

```
// Galois field GF(7^10) represented by univariate polynomials
// in variable "z" over Z/7 modulo some irreducible polynomial
// (irreducible polynomial will be generated automatically)
GF(7, 10, "z")
// GF(7^3) generated by irreducible polynomial "1 + 3*z + z^2 + z^3"
GF(UnivariateRingZp64(7, "z")("1 + 3*z + z^2 + z^3"), "z")
```

```
// Galois field GF(7^10)
// (irreducible polynomial will be generated automatically)
GF(7, 10);
// GF(7^3) generated by irreducible polynomial "1 + 3*z + z^2 + z^3"
GF(UnivariatePolynomialZ64.create(1, 3, 1, 2).modulus(7));
```

Galois fields with arbitrary large characteristic are available:

Scala

Java

```
// Mersenne prime 2^107 - 1
val characteristic : BigInteger = BigInt(2).pow(107) - 1
// Galois field GF((2^107 - 1) ^ 16)
implicit val field = GF(characteristic, 16, "z")

assert(field.cardinality() == characteristic.pow(16))
```

[0] Class IntegersZp64 which represents Z_p with $p < 2^{64}$ does not inherit Ring<E> interface (see *Modular arithmetic with machine integers*)

```
// Mersenne prime 2^107 - 1
BigInteger characteristic = BigInteger.ONE.shiftLeft(107).decrement();
// Galois field GF((2^107 - 1) ^ 16)
FiniteField<UnivariatePolynomial<BigInteger>> field = GF(characteristic, 16);

assert(field.cardinality().equals(characteristic.pow(16)));
```

Implementation of Galois fields uses precomputed inverses for fast division with Newton iterations (see `fastDivisionPreConditioning` in UnivariateDivision) which allows to achieve assymptotically fast performance.

Fields of fractions

Field of fractions can be defined over any integral domain R. The simplest example is the field Q of fractions over Z:

Scala

Java

```
implicit val field = Frac(Z) // the same as Q

assert( field("13/6") == field("2/3") + field("3/2") )
```

```
Rationals<BigInteger> field = Frac(Z); // the same as Q

Rational<BigInteger> a = field.parse("13/6");
Rational<BigInteger> b = field.parse("2/3");
Rational<BigInteger> c = field.parse("3/2");

assert a.equals(field.add(b, c));
```

The common GCD is automatically canceled in the numerator and denominator. Fractions may be defined over any GCD ring. For example, $Frac(Z[x, y, z])$ – rational functions over x, y and z:

Scala

Java

```
val ring = MultivariateRing(Z, Array("x", "y", "z"))
implicit val field = Frac(ring)

val a = field("(x + y + z)/(1 - x - y)")
val b = field("(x^2 - y^2 + z^2)/(1 - x^2 - 2*x*y - y^2)")

println(a + b)
```

```
Ring<MultivariatePolynomial<BigInteger>> ring = Rings.MultivariateRing(3, Z);
Ring<Rational<MultivariatePolynomial<BigInteger>>> field = Frac(ring);

Rational<MultivariatePolynomial<BigInteger>>
                a = field.parse("(x + y + z)/(1 - x - y)"),
                b = field.parse("(x^2 - y^2 + z^2)/(1 - x^2 - 2*x*y - y^2)");

System.out.println(field.add(a, b));
```

Scala DSL

Scala DSL allows to use standard mathematical operators for elements of rings:

Scala

```scala
import syntax._

implicit val ring = UnivariateRing(Zp(3), "x")
val (a, b) = ring("1 + 2*x^2", "1 - x")

// compiles to ring.add(a, b)
val add = a + b
// compiles to ring.subtract(a, b)
val sub = a - b
// compiles to ring.multiply(a, b)
val mul = a * b
// compiles to ring.divideExact(a, b)
val div = a / b
// compiles to ring.divideAndRemainder(a, b)
val divRem = a /% b
// compiles to ring.increment(a, b)
val inc = a ++
// compiles to ring.decrement(a, b)
val dec = a --
// compiles to ring.negate(a, b)
val neg = -a
```

Note that in the above example the ring is defined as `implicit val`, in which case the math operations are delegated to the implicit ring instance. Consider the difference:

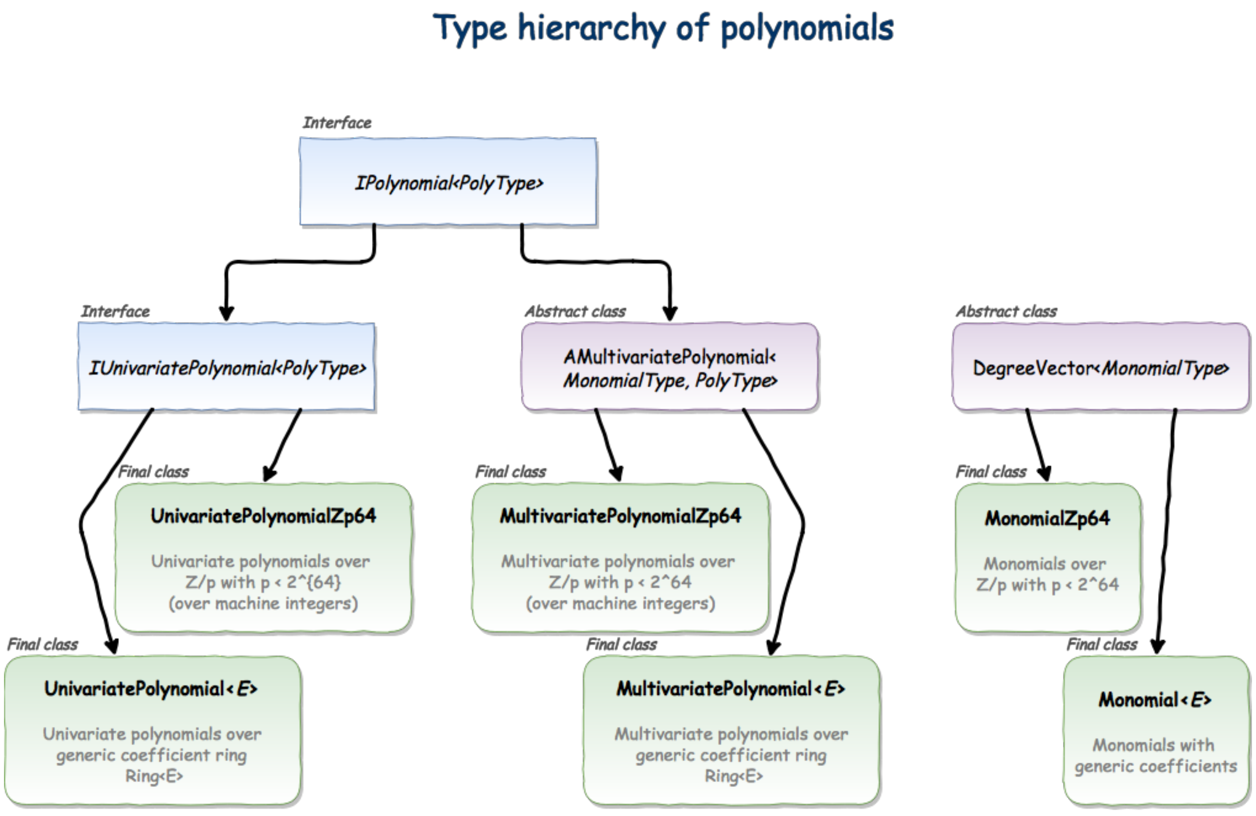
Scala

```scala
import syntax._

val a: Integer = 10
val b: Integer = 11

// no any implicit Ring[Integer] instance in the scope
// compiles to a.add(b) (integer addition)
assert(a + b === 21)

implicit val ring = Zp(13)
// compiles to ring.add(a, b) (addition mod 13)
assert(a + b === 8)
```

Polynomials and polynomial rings

Rings have separate implementation of univariate (dense) and multivariate (sparse) polynomials. Polynomials over Z_p with $p < 2^{64}$ are also implemented separately and specifically optimized (coefficients are represented as primitive machine integers instead of generic templatized objects and fast modular arithmetic is used, see *Modular arithmetic with machine integers*). Below the type hierarchy of polynomial classes is shown:

The first thing about the internal representation of polynomials is that polynomial instances do not store the information about particular string names of variables. Variables are treated just as “the first variable”, “the second variable” and so


Type hierarchy of polynomials
Interface
IPolynomial<PolyType>
Interface
IUnivariatePolynomial<PolyType>
Abstract class
AMultivariatePolynomial< MonomialType, PolyType>
Abstract class
DegreeVector<MonomialType>
Final class
UnivariatePolynomialZp64
Univariate polynomials over Z/p with p < 2^{64} (over machine integers)
Final class
MultivariatePolynomialZp64
Multivariate polynomials over Z/p with p < 2^64 (over machine integers)
Final class
MonomialZp64
Monomials over Z/p with p < 2^64
Final class
UnivariatePolynomial<E>
Univariate polynomials over generic coefficient ring Ring<E>
Final class
MultivariatePolynomial<E>
Multivariate polynomials over generic coefficient ring Ring<E>
Final class
Monomial<E>
Monomials with generic coefficients

on without specifying particular names ("x" or "y"). As result string names of variables should be specifically stored somewhere. Some illusrtations:

Scala

Java

```
import syntax._
// when parsing "x" will be considered as the "first variable"
// and "y" as "the second", then in the result the particular
// names "x" and "y" are erased
val poly1 = MultivariatePolynomial.parse("x^2 + x*y", "x", "y")
// parse the same polynomial but using "a" and "b" instead of "x" and "y"
val poly2 = MultivariatePolynomial.parse("a^2 + a*b", "a", "b")
// polynomials are equal (no matter which variable names were used when parsing)
assert(poly1 == poly2)
// degree in the first variable
assert(poly1.degree(0) == 2)
// degree in the second variable
assert(poly1.degree(1) == 1)

// this poly differs from poly2 since now "a" is "the second"
// variable and "b" is "the first"
val poly3 = MultivariatePolynomial.parse("a^2 + a*b", "b", "a")
assert(poly3 != poly2)
// swap the first and the second variables and the result is equal to poly2
assert(poly3.swapVariables(0, 1) == poly2)


// the default toString() will use the default
// variables "a", "b", "c"  and so on (alphabetical)
// the result will be "a*b + a^2"
println(poly1)
// specify which variable names use for printing
// the result will be "x*y + x^2"
println(poly1.toString(Array("x", "y")))
// the result will be "y*x + y^2"
println(poly1.toString(Array("y", "x")))
```

```
// when parsing "x" will be considered as the "first variable"
// and "y" as "the second" => in the result the particular
// names "x" and "y" are erased
MultivariatePolynomial<BigInteger> poly1 = MultivariatePolynomial.parse("x^2 + x*y",
↪"x", "y");
// parse the same polynomial but using "a" and "b" instead of "x" and "y"
MultivariatePolynomial<BigInteger> poly2 = MultivariatePolynomial.parse("a^2 + a*b",
↪"a", "b");
// polynomials are equal (no matter which variable names were used when parsing)
assert poly1.equals(poly2);
// degree in the first variable
assert poly1.degree(0) == 2;
// degree in the second variable
assert poly1.degree(1) == 1;

// this poly differs from poly2 since now "a" is "the second"
// variable and "b" is "the first"
MultivariatePolynomial<BigInteger> poly3 = MultivariatePolynomial.parse("a^2 + a*b",
↪"b", "a");
assert !poly3.equals(poly2);
// swap the first and the second variables and the result is equal to poly2
assert AMultivariatePolynomial.swapVariables(poly3, 0, 1).equals(poly2);


// the default toString() will use the default
// variables "a", "b", "c"  and so on (alphabetical)
// the result will be "a*b + a^2"
System.out.println(poly1);
// specify which variable names use for printing
// the result will be "x*y + x^2"
System.out.println(poly1.toString(new String[]{"x", "y"}));
// the result will be "y*x + y^2"
System.out.println(poly1.toString(new String[]{"y", "x"}));
```

With Scala DSL the information about string names of variables may be stored in the ring instance:

Scala

```
// "x" is the first variable "y" is the second
val ring = MultivariateRing(Z, Array("x", "y"))
// parse polynomial
val poly = ring("x^2 + x*y")
// the result will be "x*y + x^2"
println(ring show poly)
```

The second general note about implementation of polynomials is that polynomial instances are in general mutable. Methods which may modify the instance are available in Java API, while all math operations applied using Scala DSL (with operators +, − etc.) are not modifier:

Scala

Java

```
val ring = UnivariateRing(Z, "x")
val (p1, p2, p3) = ring("x", "x^2", "x^3")

// this WILL modify p1
p1.add(p2)
// this will NOT modify p2
p2.copy().add(p3)
// this will NOT modify p2
ring.add(p2, p3)
// this will NOT modify p2
p2 + p3
```

```
UnivariatePolynomial
        p1 = UnivariatePolynomial.parse("x", Z),
        p2 = UnivariatePolynomial.parse("x^2", Z),
        p3 = UnivariatePolynomial.parse("x^3", Z);

// this WILL modify p1
p1.add(p2);
// this will NOT modify p2
p2.copy().add(p3);
```

TODO polynomial API

Univariate polynomials

Rings have two separate implementations of univariate polynomials:

- UnivariatePolynomialZp64 — univariate polynomials over Z_p with $p < 2^{64}$. Implementation of UnivariatePolynomialZp64 uses specifically optimized data structure and efficient algorithms for arithmetics in Z_p (see *Modular arithmetic with machine integers*)
- UnivariatePolynomial<E> — univariate polynomials over generic coefficient ring Ring<E>

Internally both implementations use dense data structure (array of coefficients) and Karatsuba's algrotithm (Sec. 8.1 in *[vzGG03]*) for multiplication.

Division with remainder

There are several algorithms for division with remainder of univariate polynomials implemented in Rings:

- `UnivariateDivision.divideAndRemainderClassic` Plain division
- `UnivariateDivision.pseudoDivideAndRemainder` Plain pseudo division of polynomials over non-fields
- `UnivariateDivision.divideAndRemainderFast` Fast division via Newton iterations (Sec. 11 in *[vzGG03]*)

The upper-level methods `UnivariateDivision.divideAndRemainder` switches between plain and fast division depending on the input. The algorithm with Newton iterations allows to precompute Newton inverses for the divider and then use it for divisions by that divider. This allows to achieve considerable performance boost when need to do several divisions with a fixed divider (e.g. for implementation of Galois fields). Examples:

Scala

Java

```
implicit val ring = UnivariateRingZp64(17, "x")
// some random divider
val divider = ring.randomElement()
// some random dividend
val dividend = 1 + 2 * divider + 3 * divider.pow(2)

// quotient and remainder using built-in methods
val (divPlain, remPlain) = dividend /% divider

// precomputed Newton inverses, need to calculate it only once
implicit val invMod = divider.precomputedInverses
// quotient and remainder computed using fast
// algorithm with precomputed Newton inverses
val (divFast, remFast) = dividend /%% divider

// results are the same
assert((divPlain, remPlain) == (divFast, remFast))
```

```
UnivariateRing<UnivariatePolynomialZp64> ring = UnivariateRingZp64(17);
// some random divider
UnivariatePolynomialZp64 divider = ring.randomElement();
// some random dividend
UnivariatePolynomialZp64 dividend = ring.add(
        ring.valueOf(1),
        ring.multiply(ring.valueOf(2), divider),
        ring.multiply(ring.valueOf(3), ring.pow(divider, 2)));

// quotient and remainder using built-in methods
UnivariatePolynomialZp64[] divRemPlain
        = UnivariateDivision.divideAndRemainder(dividend, divider, true);

// precomputed Newton inverses, need to calculate it only once
UnivariateDivision.InverseModMonomial<UnivariatePolynomialZp64> invMod
        = UnivariateDivision.fastDivisionPreConditioning(divider);
// quotient and remainder computed using fast
// algorithm with precomputed Newton inverses
UnivariatePolynomialZp64[] divRemFast
        = UnivariateDivision.divideAndRemainderFast(dividend, divider, invMod, true);

// results are the same
assert Arrays.equals(divRemPlain, divRemFast);
```

Details of implementation can be found in UnivariateDivision.

Univariate GCD

Rings have several algorithms for univariate GCD:

- `UnivariateGCD.EuclidGCD` and `UnivariateGCD.Extednded EuclidGCD` Euclidean algorithm (and its extended version)
- `UnivariateGCD.HalfGCD` and `UnivariateGCD.ExtedndedHalfGCD` Half-GCD (and its extended version) (Sec. 11 *[vzGG03]*)
- `UnivariateGCD.SubresultantRemainders` Subresultant sequences (Sec. 7.3 in *[GCL92]*)

- `UnivariateGCD.ModularGCD` Modular GCD (Sec. 6.7 in *[vzGG03]*, small primes version)

The upper-level method `UnivariateGCD.PolynomialGCD` switches between Euclidean algorithm and Half-GCD for polynomials in $F[x]$ where F is a finite field. For polynomials in $Z[x]$ and $Q[x]$ the modular algorithm is used (small primes version). In other cases algorithm with subresultant sequences is used. Examples:

Scala

Java

```
import poly.univar.UnivariateGCD._

// Polynomials over field
val ringZp = UnivariateRingZp64(17, "x")
val a = ringZp("1 + 3*x + 2*x^2")
val b = ringZp("1 - x^2")
// Euclid and Half-GCD algorithms for polynomials over field
assert(EuclidGCD(a, b) == HalfGCD(a, b))
// Extended Euclidean algorithm
val (gcd, s, t) = ExtendedEuclidGCD(a, b) match {case Array(gcd, s, t) => (gcd, s, t)}
assert(a * s + b * t == gcd)
// Extended Half-GCD algorithm
val (gcd1, s1, t1) = ExtendedHalfGCD(a, b) match {case Array(gcd, s, t) => (gcd, s,␣
↪t)}
assert((gcd1, s1, t1) == (gcd, s, t))


// Polynomials over Z
val ringZ = UnivariateRing(Z, "x")
val aZ = ringZ("1 + 3*x + 2*x^2")
val bZ = ringZ("1 - x^2")
// GCD for polynomials over Z
assert(ModularGCD(aZ, bZ) == ringZ("1 + x"))


// Bivariate polynomials represented as Z[y][x]
val ringXY = UnivariateRing(UnivariateRing(Z, "y"), "x")
val aXY = ringXY("(1 + y) + (1 + y^2)*x + (y - y^2)*x^2")
val bXY = ringXY("(3 + y) + (3 + 2*y + y^2)*x + (3*y - y^2)*x^2")
// Subresultant sequence
val subResultants = SubresultantRemainders(aXY, bXY)
// The GCD
val gcdXY = subResultants.gcd.primitivePart
assert(aXY % gcdXY === 0 && bXY % gcdXY === 0)
```

```
// Polynomials over field
UnivariatePolynomialZp64 a = UnivariatePolynomialZ64.create(1, 3, 2).modulus(17);
UnivariatePolynomialZp64 b = UnivariatePolynomialZ64.create(1, 0, -1).modulus(17);
// Euclid and Half-GCD algorithms for polynomials over field
assert EuclidGCD(a, b).equals(HalfGCD(a, b));
// Extended Euclidean algorithm
UnivariatePolynomialZp64[] xgcd = ExtendedEuclidGCD(a, b);
assert a.copy().multiply(xgcd[1]).add(b.copy().multiply(xgcd[2])).equals(xgcd[0]);
// Extended Half-GCD algorithm
UnivariatePolynomialZp64[] xgcd1 = ExtendedHalfGCD(a, b);
assert Arrays.equals(xgcd, xgcd1);


// Polynomials over Z
```

```
UnivariatePolynomial<BigInteger> aZ = UnivariatePolynomial.create(1, 3, 2);
UnivariatePolynomial<BigInteger> bZ = UnivariatePolynomial.create(1, 0, -1);
// GCD for polynomials over Z
assert ModularGCD(aZ, bZ).equals(UnivariatePolynomial.create(1, 1));


// Bivariate polynomials represented as Z[y][x]
UnivariateRing<UnivariatePolynomial<UnivariatePolynomial<BigInteger>>>
        ringXY = UnivariateRing(UnivariateRing(Z));
UnivariatePolynomial<UnivariatePolynomial<BigInteger>>
        aXY = ringXY.parse("(1 + y) + (1 + y^2)*x + (y - y^2)*x^2"),
        bXY = ringXY.parse("(3 + y) + (3 + 2*y + y^2)*x + (3*y - y^2)*x^2");
//// Subresultant sequence
PolynomialRemainders<UnivariatePolynomial<UnivariatePolynomial<BigInteger>>>
        subResultants = SubresultantRemainders(aXY, bXY);
// The GCD
UnivariatePolynomial<UnivariatePolynomial<BigInteger>> gcdXY = subResultants.gcd().
↪primitivePart();
assert UnivariateDivision.remainder(aXY, gcdXY, true).isZero();
assert UnivariateDivision.remainder(bXY, gcdXY, true).isZero();
```

Details of implementation can be found in UnivariateGCD.

Univariate factorization

Implementation of univariate factorization in Rings is distributed over several classes:

- `UnivariateSquareFreeFactorization` Square-free factorization of univariate polynomials. In the case of zero characteristic Yun's algorithm is used (Sec. 14.6 in *[vzGG03]*), otherwise Musser's algorithm is used (Sec. 8.3 in *[GCL92]*, *[Mus71]*).
- `DistinctDegreeFactorization` Distinct-degree factorization. Internally there are several algorithms: plain (Sec. 14.2 in *[vzGG03]*), adapted version with precomputed x-powers, and Victor Shoup's baby-step giant-step algorithm *[Sho95]*. The upper-level method swithces between these algorithms depending on the input.
- `EqualDegreeFactorization` Equal-degree factorization using Cantor-Zassenhaus algorithm in both odd and even characteristic (Sec. 14.3 in *[vzGG03]*).
- `UnivariateFactorization` Defines upper-level methods and implements factorization over Z. In the latter case Hensel lifting (combined linear/quadratic) is used to lift factoization modulo some prime number to actual factorization over Z and naive recombination to reconstruct correct factors. Examples:

Univariate factorization is supported for polynomials in $F[x]$ where F is either finite field or Z or Q. Examples:

Scala

Java

```
// ring GF(13^5)[x] (coefficient domain is finite field)
val ringF = UnivariateRing(GF(13, 5, "z"), "x")
// some random polynomial composed from some factors
val polyF = ringF.randomElement() * ringF.randomElement() * ringF.randomElement().
↪pow(10)
// perform square-free factorization
println(ringF show FactorSquareFree(polyF))
// perform complete factorization
println(ringF show Factor(polyF))
```

```
// ring Q[x]
val ringQ = UnivariateRing(Q, "x")
// some random polynomial composed from some factors
val polyQ = ringQ.randomElement() * ringQ.randomElement() * ringQ.randomElement().
↪pow(10)
// perform square-free factorization
println(ringQ show FactorSquareFree(polyQ))
// perform complete factorization
println(ringQ show Factor(polyQ))
```

```
// ring GF(13^5)[x] (coefficient domain is finite field)
UnivariateRing<UnivariatePolynomial<UnivariatePolynomialZp64>> ringF =␣
↪UnivariateRing(GF(13, 5));
// some random polynomial composed from some factors
UnivariatePolynomial<UnivariatePolynomialZp64> polyF = ringF.randomElement().
↪multiply(ringF.randomElement().multiply(polyPow(ringF.randomElement(), 10)));

// perform square-free factorization
System.out.println(FactorSquareFree(polyF));
// perform complete factorization
System.out.println(Factor(polyF));


// ring Q[x]
UnivariateRing<UnivariatePolynomial<Rational<BigInteger>>> ringQ = UnivariateRing(Q);
// some random polynomial composed from some factors
UnivariatePolynomial<Rational<BigInteger>> polyQ = ringQ.randomElement().
↪multiply(ringQ.randomElement().multiply(polyPow(ringQ.randomElement(), 10)));
// perform square-free factorization
System.out.println(FactorSquareFree(polyQ));
// perform complete factorization
System.out.println(Factor(polyQ));
```

The details about implementation can be found in UnivariateSquareFreeFactorization, DistinctDegreeFactorization, EqualDegreeFactorization and UnivariateFactorization.

Testing irreducibility

Irreducibility test and generation of random irreducible polynomials are availble from `IrreduciblePolynomials`. For irreducibility testing of polynomials over finite fields the algorithm described in Sec. 14.9 in *[vzGG03]* is used. Methods implemented in `IrreduciblePolynomials` are used for construction of arbitrary Galois fields. Examples:

Scala

Java

```
import rings.poly.univar.IrreduciblePolynomials._
val random = new Random()

// random irreducible polynomial in Z/2[x] of degree 10 (UnivariatePolynomialZp64)
val poly1 = randomIrreduciblePolynomial(2, 10, random)
assert(poly1.degree() == 10)
assert(irreducibleQ(poly1))

// random irreducible polynomial in Z/2[x] of degree 10␣
↪(UnivariatePolynomial[Integer])
```

```
val poly2 = randomIrreduciblePolynomial(Zp(2).theRing, 10, random)
assert(poly2.degree() == 10)
assert(irreducibleQ(poly2))

// random irreducible polynomial in GF(11^15)[x] of degree 10 (this may take few␣
↪seconds)
val poly3 = randomIrreduciblePolynomial(GF(11, 15).theRing, 10, random)
assert(poly3.degree() == 10)
assert(irreducibleQ(poly3))

// random irreducible polynomial in Z[x] of degree 10
val poly4 = randomIrreduciblePolynomialOverZ(10, random)
assert(poly4.degree() == 10)
assert(irreducibleQ(poly4))
```

```
Well44497b random = new Well44497b();

// random irreducible polynomial in Z/2[x] of degree 10
UnivariatePolynomialZp64 poly1 = randomIrreduciblePolynomial(2, 10, random);
assert poly1.degree() == 10;
assert irreducibleQ(poly1);

// random irreducible polynomial in Z/2[x] of degree 10
UnivariatePolynomial<BigInteger> poly2 = randomIrreduciblePolynomial(Zp(2), 10,␣
↪random);
assert poly2.degree() == 10;
assert irreducibleQ(poly2);

// random irreducible polynomial in GF(11^15)[x] of degree 10 (this may take few␣
↪seconds)
UnivariatePolynomial<UnivariatePolynomialZp64> poly3 =␣
↪randomIrreduciblePolynomial(GF(11, 15), 10, random);
assert poly3.degree() == 10;
assert irreducibleQ(poly3);

// random irreducible polynomial in Z[x] of degree 10
UnivariatePolynomial<BigInteger> poly4 = randomIrreduciblePolynomialOverZ(10, random);
assert poly4.degree() == 10;
assert irreducibleQ(poly4);
```

The details about implementation can be found in IrreduciblePolynomials.

Univariate interpolation

Polynomial interpolation via Newton method can be done in the following way:

Scala

Java

```
import rings.poly.univar.UnivariateInterpolation._

// points
val points = Array(1L, 2L, 3L, 12L)
// values
val values = Array(3L, 2L, 1L, 6L)

```

```
// interpolate using Newton method
val result = new InterpolationZp64(Zp64(17))
  .update(points, values)
  .getInterpolatingPolynomial

// result.evaluate(points(i)) = values(i)
assert(points.zipWithIndex.forall { case (point, i) => result.evaluate(point) ==␣
↪values(i) })
```

```
// points
long[] points = {1L, 2L, 3L, 12L};
// values
long[] values = {3L, 2L, 1L, 6L};

// interpolate using Newton method
UnivariatePolynomialZp64 result = new InterpolationZp64(Zp64(17))
        .update(points, values)
        .getInterpolatingPolynomial();

// result.evaluate(points(i)) = values(i)
assert IntStream.range(0, points.length).allMatch(i -> result.evaluate(points[i]) ==␣
↪values[i]);
```

With Scala DSL it is quite easy to implement Lagrange interpolation formula:

Scala

```
/*  Lagrange interpolation formula */
def lagrange[Poly <: IUnivariatePolynomial[Poly], E](points: Seq[E], values:␣
↪Seq[E])(implicit ring: IUnivariateRing[Poly, E]) = {
  import syntax._
  points.indices
    .foldLeft(ring getZero) { case (sum, i) =>
      sum + points.indices
        .filter(_ != i)
        .foldLeft(ring getConstant values(i)) { case (product, j) =>
          implicit val cfRing = ring.cfRing
          val E: E = points(i) - points(j)
          product * (ring.`x` - points(j)) / E
        }
    }
}

import rings.poly.univar.UnivariateInterpolation._
import syntax._

// coefficient ring GF(13, 5)
implicit val cfRing = GF(13, 5, "z")
val z = cfRing("z")
// some points
val points = Array(1 + z, 2 + z, 3 + z, 12 + z)
// some values
val values = Array(3 + z, 2 + z, 1 + z, 6 + z)

// interpolate with Newton iterations
val withNewton = new Interpolation(cfRing)
  .update(points, values)
  .getInterpolatingPolynomial
```

```
// interpolate using Lagrange formula
val withLagrange = lagrange(points, values)(UnivariateRing(cfRing, "x"))
// results are the same
assert(withNewton == withLagrange)
```

Multivariate polynomials

Division with remainders

Multivariate GCD

Multivariate factorization

Multivariate Interpolation

Multivariate polynomials

CHAPTER 4

Index of algorithms implemented in Rings

Univariate polynomials

1. *Karatsuba multiplication* [vzGG03] (Sec. 8.1) used (in some adapted form) for multiplication of univariate polynomials:

- UnivariatePolynomial.multiply
- UnivariatePolynomialZp64.multiply

2. *Half-GCD and Extended Half-GCD* [vzGG03] (Sec. 11) used (in some adapted form inspired by [NTL]) for univariate GCD:

- UnivariateGCD.HalfGCD
- UnivariateGCD.ExtendedHalfGCD

3. *Subresultant polynomial remainder sequences* [GCL92] (Sec. 7.3):

- UnivariateGCD.SubresultantRemainders

4. *Modular GCD in $Z[x]$ and $Q[x]$* [vzGG03] (Sec. 6.7), small primes version is used:

- UnivariateGCD.ModularGCD

5. *Fast univariate division with Newton iteration* [vzGG03] (Sec. 9.1) used everywhere where multiple divisions (remainders) by the same divider are performed:

- UnivariateDivision.fastDivisionPreConditioning
- UnivariateDivision.divideAndRemainderFast

6. *Univariate square-free factorization in zero characteristic (Yun's algorithm)* [vzGG03] (Sec. 14.6):

- UnivariateSquareFreeFactorization.SquareFreeFactorizationYunZeroCharacteristics

7. *Univariate square-free factorization in non-zero characteristic (Musser's algorithm)* [Mus71], [GCL92] (Sec. 8.3):

- UnivariateSquareFreeFactorization.SquareFreeFactorizationMusser

- UnivariateSquareFreeFactorization.SquareFreeFactorizationMusserZeroCharacteristics

8. *Distinct-degree factorization* *[vzGG03]* (Sec. 14.2) plain version and adapted version (which is actually used) with precomputed x-powers:

- DistinctDegreeFactorization.DistinctDegreeFactorizationPlain
- DistinctDegreeFactorization.DistinctDegreeFactorizationPrecomputedExponents

9. *Shoup's baby-step giant-step algorithm for distinct-degree factorization* *[Sho95]* used for factorization over fields with large cardinality:

- DistinctDegreeFactorization.DistinctDegreeFactorizationShoup

10. *Univariate modular composition* plain algorithm with Horner schema:

- ModularComposition.compositionHorner

11. *Brent-Kung algorithm* for univariate modular composition *[BK98]*, *[Sho95]*:

- ModularComposition.compositionBrentKung

12. *Cantor-Zassenhaus algorithm (equal-degree splitting)* *[vzGG03]* (Sec. 14.3) both for odd and even characteristic:

- EqualDegreeFactorization.CantorZassenhaus

13. *Univaraite linear p-adic Hensel lifting* *[GCL92]* (Sec. 6.5):

- univar.HenselLifting.createLinearLift
- univar.HenselLifting.liftFactorization

14. *Univaraite quadratic p-adic Hensel lifting* *[vzGG03]* (Sec. 15.4-15.5):

- univar.HenselLifting.createQuadraticLift
- univar.HenselLifting.liftFactorization

15. *Univariate polynomial factorization over finite fields* Musser's square free factorization -> distinct-degree factorization (either x-powers or Shoup's algorithm) -> Cantor-Zassenhaus equal-degree factorization:

- UnivariateFactorization.FactorInGF

16. *Univariate polynomial factorization over Z and Q* factorization modulo small prime -> Hensel lifting (adaptive linear/quadratic):

- UnivariateFactorization.FactorInZ
- UnivariateFactorization.FactorInQ

17. *Univariate irreducibility test* *[vzGG03]* (Sec. 14.9):

- IrreduciblePolynomials.irreducibleQ

18. *Ben-Or's generation of irreducible polynomials* *[vzGG03]* (Sec. 14.9):

- IrreduciblePolynomials.randomIrreduciblePolynomial

19. *Univariate polynomial interpolation* Lagrange and Newton methods:

- UnivariateInterpolation

Multivariate polynomials

20. *Brown GCD over finite fields [Bro71], [GCL92]* (Sec. 7.4), *[Yan09]*:

- MultivariateGCD.BrownGCD

21. *Zippel's sparse GCD over finite fields [Zip79], [Zip93], [dKMW05], [Yan09]* both for monic (with fast Vandermonde systems) and non-monic (LINZIP) cases:

- MultivariateGCD.ZippelGCD

22. *Extended Zassenhaus GCD (EZ-GCD) over finite fields [GCL92]* (Sec. 7.6), *[MY73]*:

- MultivariateGCD.EZGCD

23. *Enhanced Extended Zassenhaus GCD (EEZ-GCD) over finite fields [Wan80]*:

- MultivariateGCD.EEZGCD

24. *Modular GCD over Z with sparse interpolation [Zip79], [Zip93], [dKMW05]* (the same interpolation techniques as in *ZippelGCD* is used):

- MultivariateGCD.ModularGCD

25. *Kaltofen's & Monagan's generic modular GCD [KM99]* used for computing multivariate GCD over finite fields of very small cardinality

- MultivariateGCD.ModularGCDInGF

26. *Multivariate square-free factorization in zero characteristic (Yun's algorithm) [Lee13]*:

- MultivariateSquareFreeFactorization.SquareFreeFactorizationYunZeroCharacteristics

27. *Multivariate square-free factorization in non-zero characteristic (Musser's algorithm) [Mus71], [GCL92]* (Sec. 8.3):

- MultivariateSquareFreeFactorization.SquareFreeFactorizationMusser
- MultivariateSquareFreeFactorization.SquareFreeFactorizationMusserZeroCharacteristics

28. *Bernardin's fast dense multivariate Hensel lifting [Ber99], [Lee13]* both for bivariate case (original Bernardin's paper) and multivariate case (Lee thesis) and both with and without precomputed leading coefficients:

- multivar.HenselLifting

29. *Fast dense bivariate factorization with recombination [Ber99], [Lee13]*:

- MultivariateFactorization.bivariateDenseFactorSquareFreeInGF
- MultivariateFactorization.bivariateDenseFactorSquareFreeInZ

30. *Kaltofen's multivariate factorization in finite fields [Kal85], [Lee13]*; modified version of original Kaltofen's algorithm for leading coefficient precomputation with square-free decomposition (instead of distinct variables decomposition) due to Lee is used; further adaptations are made to work in finite fields of very small cardinality; the resulting algorithm is close to *[Lee13]*, but at the same time has many differences in details:

- MultivariateFactorization.factorInGF

31. *Kaltofen's multivariate factorization Z [Kal85], [Lee13]* (the same modifications as for finite field algorithm are made):

- MultivariateFactorization.factorInZ

32. *Multivariate polynomial interpolation with Newton method*:

- MultivariateInterpolation

References

Bibliography

[vzGG03] 10. von zur Gathen and J. Gerhard. Modern computer algebra (2. ed.). Cambridge University Press, 2003.

[NTL] 22. Shoup. NTL: A library for doing number theory. www.shoup.net/ntl

[GCL92] 11. (a) Geddes, S. R. Czapor, G. Labahn. Algorithms for Computer Algebra. 1992.

[Mus71] D.R. Musser, Algorithms for polynomial factorization, Ph.D. Thesis, University of Wisconsin, 1971.

[Sho95] 22. Shoup. A new polynomial factorization algorithm and its implementation. J. Symb. Comput., 20(4):363–397, 1995.

[BK98] R.P. Brent and H.T. Kung. Fast algorithms for manipulating formal power series. J. Assoc. Comput. Math. 25:581-595, 1978

[Bro71] 23. (a) Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors. J. ACM, 18(4):478–504, 1971.

[Zip79] 18. (a) Zippel. Probabilistic algorithms for sparse polynomials. In Proceedings of the International Symposiumon on Symbolic and Algebraic Computation, EUROSAM '79, pages 216–226, London, UK, UK, 1979. Springer-Verlag.

[Zip93] 18. (a) Zippel. Effective Polynomial Computation. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.

[dKMW05] 10. de Kleine, M. B. Monagan, A. D. Wittkopf. Algorithms for the Non-monic Case of the Sparse Modular GCD Algorithm. Proceeding of ISSAC '05, ACM Press, pp. 124-131 , 2005.

[Yan09] 19. Yang. Computing the greatest common divisor of multivariate polynomials over finite fields. Master's thesis, Simon Fraser University, 2009.

[MY73] 10. Moses and D.Y.Y.Yun, "The EZGCD Algorithm," pp. 159-166 in Proc. ACM Annual Conference, (1973).

[Wan80] P.S. Wang, "The EEZ-GCD Algorithm," ACM SIGSAMBull., 14 pp. 50-60 (1980).

[KM99] 5. Kaltofen, M. B. Monagan. On the Genericity of the Modular Polynomial GCD Algorithm. Proceeding of ISSAC '99, ACM Press, 59-66, 1999.

[Ber99] 12. Bernardin. Factorization of Multivariate Polynomials over Finite Fields. PhD thesis, ETH Zu rich, 1999.

[Lee13] 13. M.-D. Lee, Factorization of multivariate polynomials, Ph.D. thesis, University of Kaiserslautern, 2013

[Kal85] 5. Kaltofen. Sparse Hensel lifting. In EUROCAL 85 European Conf. Comput. Algebra Proc. Vol. 2, pages 4–17, 1985.