`Rings`: *efficient Java/Scala library for polynomial rings*

Stanislav Poslavsky

Institute for High Energy Physics NRC "Kurchatov Institute", Protvino, Russia

ACAT19

Saas Fee, 2019

Rings: *an overview*

- **Computational Number Theory**
  - *primes: sieving, testing, factorization*
  - *univariate polynomials over arbitrary coefficient rings: fast arithmetic, gcd, factorization etc.*
  - *Galois fields & Algebraic number fields*
- **Computational Commutative Algebra**
  - *multivariate polynomials over arbitrary coefficient rings: fast arithmetic, gcd, factorization etc.*
  - *fast rational function arithmetic*
- **Computational Algebraic Geometry**
  - *Systems of polynomial equations*
  - *Gröbner bases*
  - *Ideals in multivariate polynomial rings*

RINGS LIBRARY

— Yet another program for math ?
Really ? What for ???

An *incomplete* list of similar software:

Closed source (proprietary)	Open source (free)
`Magma`, `Maple`, `Mathematica`, `Fermat`, ...	`Singular`, `Macaulay2`, `CoCoA`, `Reduce`, `Maxima`, `Pari/GP`, ... `FLINT`, `NTL`, `FORM`, ...

`Rings` is aimed to be:

- **Ultrafast:** *make it faster than existing tools*
- **Lightweight:** *portable, extensible and embeddable library (not a CAS)*
- **Modern:** *API which meets modern best programming practices*

`Rings`:

- *is the first such library written in JAVA (90%) & SCALA (10%)*
- *contains more than 100,000 lines of code*
- *well, see https://ringsalgebra.io*

Basic algebraic definitions

- `Ring`: a set of elements with "$+$" and "$\times$" operations defined.
  Examples:
  - $\mathbb{Z}$ — ring of integers
  - $\mathbb{Z}[i]$ — Gaussian integers
  - $R[\vec{X}]$ — polynomials with coefficients from ring R
- `Field`: a ring with "/" (division) operation.
  Examples:
  - $\mathbb{Q}$ — field of rational numbers
  - $\mathbb{Z}_p$ — field of integers modulo a prime number
  - $Frac(R[X])$ — field of rational functions
- `Ideal`: a subset of ring elements closed under multiplication with ring.
  Examples:
  - Given a set of generators $\{f_i(x, y, \ldots)\} \in R[x, y, \ldots]$ ideal is formed by all elements of the form

$$c_1(x, y, \ldots) \times f_1(x, y, \ldots) + \ldots + c_n(x, y, \ldots) \times f_n(x, y, \ldots)$$

Rings: *design by examples*

Simple example:

```
implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
val x = ring("x")              // parse polynomial from string
val poly = x.pow(100) - 1   // construct polynomial programmatically
val factors = Factor(poly)  // factorize polynomial
println(factors)
```

Rings: *design by examples*

Simple example:

```
implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
val x = ring("x")             // parse polynomial from string
val poly = x.pow(100) - 1    // construct polynomial programmatically
val factors = Factor(poly)   // factorize polynomial
println(factors)
```

- Explicit types are omitted for shortness, though Scala is fully statically typed

```
val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
val poly : UnivariatePolynomial[Rational[IntZ]] = ...
```

(types are inferred automatically at compile time if not specified explicitly)

Rings: *design by examples*

Simple example:

```
implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
val x = ring("x")            // parse polynomial from string
val poly = x.pow(100) - 1    // construct polynomial programmatically
val factors = Factor(poly)   // factorize polynomial
println(factors)
```

- Explicit types are omitted for shortness, though Scala is fully statically typed

  ```
  val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
  val poly : UnivariatePolynomial[Rational[IntZ]] = ...
  ```

  (types are inferred automatically at compile time if not specified explicitly)

- Trait `Ring[E]` implements the concept of mathematical ring and defines all basic algebraic operations over the elements of type `E`

  ```
  println( ring.isField )        // access ring properties
  println( ring.characteristic ) // access ring characteristic
  println( ring.cardinality )    // access ring cardinality
  ```

Rings: *design by examples*

Simple example:

```
implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
val x = ring("x")            // parse polynomial from string
val poly = x.pow(100) - 1    // construct polynomial programmatically
val factors = Factor(poly)   // factorize polynomial
println(factors)
```

- Explicit types are omitted for shortness, though `Scala` is fully statically typed

```
val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
val poly : UnivariatePolynomial[Rational[IntZ]] = ...
```

(types are inferred automatically at compile time if not specified explicitly)

- Trait `Ring[E]` implements the concept of mathematical ring and defines all basic algebraic operations over the elements of type `E`

```
println( ring.isField )          // access ring properties
println( ring.characteristic ) // access ring characteristic
println( ring.cardinality )     // access ring cardinality
```

- The `implicit` brings operator overloading via type enrichment (continue =>)

Rings: *design by examples*

Meaning of implicits:

```
// ring of elements of type E
implicit val ring : Ring[E] = ...
val a : E = ...
val b : E = ...

val sum = a + b // compiles to ring.add(a, b)
val mul = a * b // compiles to ring.multiply(a, b)
val div = a / b // compiles to ring.divideExact(a, b)
```

Example:

```
val a : IntZ = Z(12)
val b : IntZ = Z(13)
assert (a * b == Z(156))   // no any implicit Ring[IntZ]

{
    implicit val ring = Zp(17) // implicit Ring[IntZ]
    assert (a * b == Z(3))     // multiplication modulo 17
}
```

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

- **Example:**
  Given polynomial fraction

$$\frac{1}{((s-t)^2 - m_3^2)(s^2 - m_1^2)(t^2 - m_2^2)}$$

decompose it in a sum of fractions such that denominators in each fraction are algebraically independent in (s, t)

NOTE: denominators are dependent since

$$\begin{aligned}(m_1 - m_2 - m_3)(m_1 + m_2 - m_3)(m_1 - m_2 + m_3)(m_1 + m_2 + m_3)\\ + 2(-m_1^2 - m_2^2 + m_3^2)\, Y_1 + 2(m_1^2 - m_3^2 - m_2^2)\, Y_2 + 2(m_1^2 - m_2^2 - m_3^2)\, Y_3\\ + Y_1^2 + Y_2^2 + Y_3^2 - 2\, Y_1\, Y_2 - 2\, Y_1\, Y_3 - 2\, Y_2\, Y_3 \equiv 0\end{aligned}$$

$$Y_1 = ((s-t)^2 - m_3^2) \qquad Y_2 = (s^2 - m_1^2) \qquad Y_3 = (t^2 - m_2^2)$$

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```
// field of coefficients Frac(Z[m1, m2, m3])
val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(Z[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
              / (s.pow(2) - m1.pow(2))
              / (t.pow(2) - m2.pow(2)))
// or just parse from string
// val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

```

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```
// field of coefficients Frac(Z[m1, m2, m3])
val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(Z[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
              / (s.pow(2) - m1.pow(2))
              / (t.pow(2) - m2.pow(2)))
// or just parse from string
// val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

// bring in the form with algebraically independent denominators
val dec = GroebnerMethods.LeinartDecomposition(frac)
// simplify fractions (factorize)
val decSimplified = dec.map(f => field.factor(f))
// pretty print
decSimplified.map(f => field.stringify(f)).foreach(println)
```

`Rings`: *design by examples*

Multivariate polynomials & rational functions & simplifications

```
// field of coefficients Frac(Z[m1, m2, m3])
val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(Z[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
              / (s.pow(2) - m1.pow(2))
              / (t.pow(2) - m2.pow(2)))
...
```

- **Result:** $$\frac{1}{((s-t)^2 - m_3^2)(s^2 - m_1^2)(t^2 - m_2^2)} = $$ $$- \frac{1}{8 m_1 m_2 m_3 (m_1 + m_2 + m_3)} \frac{1}{(-m_3 - t + s)(t - m_2)}$$ $$- \frac{1}{8 m_1 m_2 m_3 (m_1 + m_2 + m_3)} \frac{1}{(-m_3 - t + s)(s + m_1)}$$ $+ \ldots$ **(+22 other terms)**

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```scala
// field of coefficients Frac(Z[m1, m2, m3])
val cfs = Frac(MultivariateRing( Z , Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(Z[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
             / (s.pow(2) - m1.pow(2))
             / (t.pow(2) - m2.pow(2)))
// or just parse from string
// val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

// bring in the form with algebraically independent denominators
val dec = GroebnerMethods.LeinartDecomposition(frac)
// simplify fractions (factorize)
val decSimplified = dec.map(f => field.factor(f))
// pretty print
decSimplified.map(f => field.stringify(f)).foreach(println)
```

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```
// field of coefficients Frac(GF(2,16)[m1, m2, m3])
val cfs = Frac(MultivariateRing(GF(2,16,"e"), Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(GF(2,16)[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
              / (s.pow(2) - m1.pow(2))
              / (t.pow(2) - m2.pow(2)))
// or just parse from string
// val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

// bring in the form with algebraically independent denominators
val dec = GroebnerMethods.LeinartDecomposition(frac)
// simplify fractions (factorize)
val decSimplified = dec.map(f => field.factor(f))
// pretty print
decSimplified.map(f => field.stringify(f)).foreach(println)
```

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

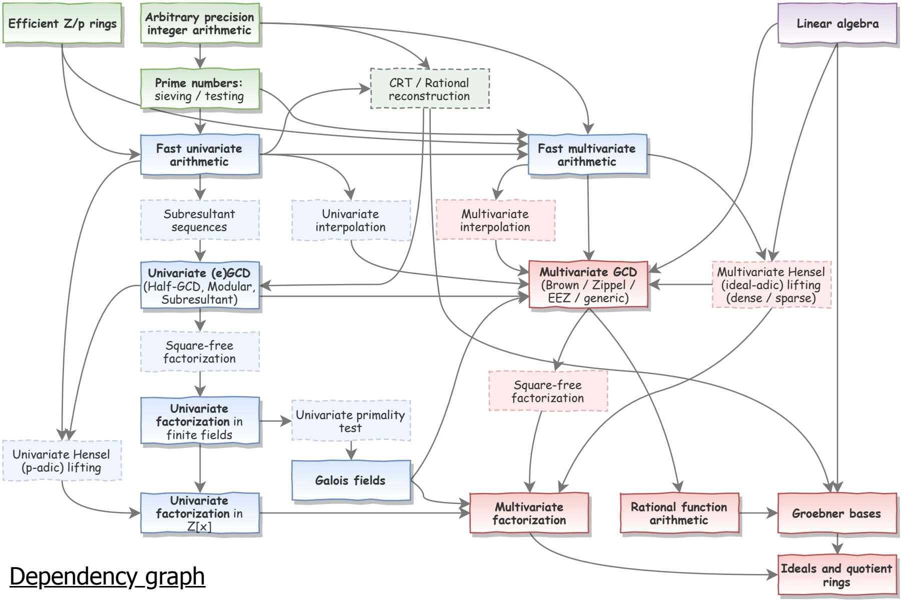
```
// field of coefficients Frac(GF(2,16)[m1, m2, m3])
val cfs = Frac(MultivariateRing(GF(2,16,"e"), Array("m1","m2","m3")))
// field of rational functions  Frac(Frac(GF(2,16)[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
              / (s.pow(2) - m1.pow(2))
              / (t.pow(2) - m2.pow(2)))
...
```

- **Result:** $$\frac{1}{((s-t)^2-m_3^2)(s^2-m_1^2)(t^2-m_2^2)} = \frac{1}{(m_1+m_2+m_3)^2}\frac{1}{(m_3+t+s)^2(s+m_1)^2} + \frac{1}{(m_1+m_2+m_3)^2}\frac{1}{(m_3+t+s)^2(t+m_2)^2} + \frac{1}{(m_1+m_2+m_3)^2}\frac{1}{(t+m_2)^2(s+m_1)^2}$$

Rings: *design by examples*

Built-in ring	**Description**
`Z`	ring of integers
`Q`	field of rationals
`GaussianRationals`	field of complex rational numbers $\mathbb{Q}(i)$
`Zp(p)`	integers modulo `p`
`GF(p,q)`	finite field with cardinality p^q
`AlgebraicNumberField(alpha)`	algebraic number field $F(\alpha_1, \ldots, \alpha_s)$
`Frac(R)`	field of fractions over Euclidean ring R
`UnivariateRing(R, x)`	univariate ring $R[x]$
`MultivariateRing(R, vars)`	multivariate ring $R[x_1, x_2, \ldots]$
`QuotientRing(R, ideal)`	multivariate quotient ring $R[x_1, x_2, \ldots]/I$

Rings: *implementation aspects*



<u>Dependency graph</u>

Benchmarks

• Polynomial GCD •

• Polynomial factorization •

• Gröbner bases •

Benchmarks: *polynomial GCD*

Benchmarks:

- Generate three polynomials a, b and g at random and compute $gcd(ag, bg)$ (non-trivial) and $gcd(ag + 1, bg)$ (trivial)
- Terms of polynomials are generated independently
- Two ways to generate exponents inside terms:
  - *Uniform exponents* (uniform distribution):
    choose each exponent independently in range $\text{exp}_{\text{min}} \leq \text{exp}_i < \text{exp}_{\text{max}}$; the total degree will be $N_{\text{vars}}\text{exp}_{\text{min}} \leq \text{exp}_{\text{tot}} < N_{\text{vars}}\text{exp}_{\text{max}}$
    Example ($\text{exp}_{\text{min}} = 0$, $\text{exp}_{\text{max}} = 10$):
    $$\cdots + x^5y^2z^8 + x^3y^8z^6 + \ldots$$
  - *Sharp exponents* (multinomial distribution):
    choose the total degree exp_{tot}, then for the first variable $0 \leq \text{exp}_1 \leq \text{exp}_{\text{tot}}$, for the second variable $0 \leq \text{exp}_2 \leq (\text{exp}_{\text{tot}} - \text{exp}_1)$ and so on
    Example ($\text{exp}_{\text{tot}} = 10$):
    $$\cdots + x^7y^2z^1 + x^0y^8z^2 + \ldots$$

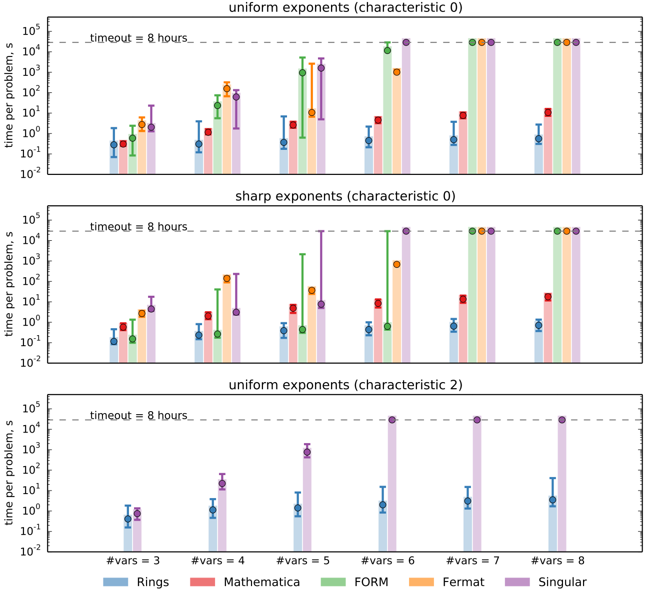
Benchmarks: *polynomial GCD*

Params (a,b,g):

$\#terms = 40$
$\#bits = 32$
$exp_{min} = 0$
$exp_{max} = 30$

$\#terms = 40$
$\#bits = 32$
$exp_{tot} = 50$

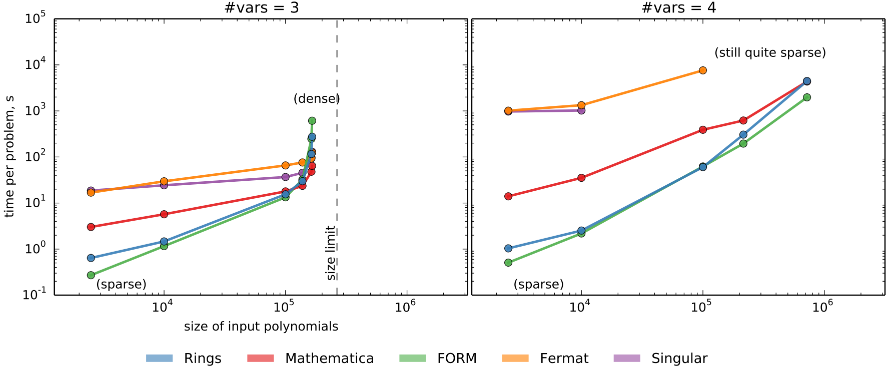
$\#terms = 40$
$\#bits = 1$
$exp_{min} = 0$
$exp_{max} = 30$



▷ reFORM (see Ben's talk) seems has comparable performance

Benchmarks: *polynomial GCD*

"Huge" problems:



Params (a,b,g):

$\mathrm{exp}_{\mathrm{tot}}$ = 50 / #bits = 128 / #terms = 50, 100, 500, 1000, 5000

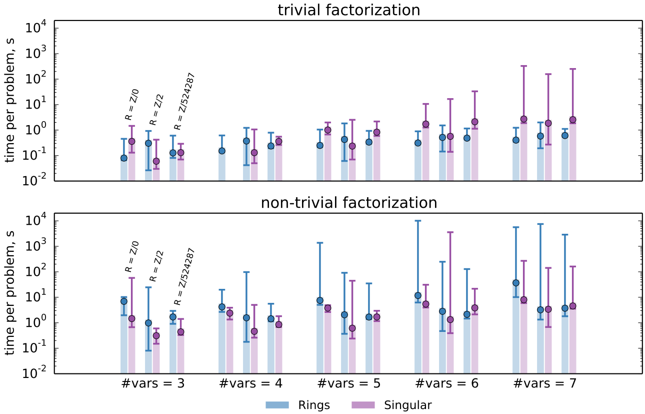
Benchmarks: *polynomial factorization*

- Polynomial factorization is much harder than GCD
- Few tools may do factorization on routine basis (`Singular`, `Magma` and `Maple`)

Benchmark: generate three polynomials a, b and c at random and compute $factor(abc)$ (non-trivial) and $factor(abc+1)$ (trivial)

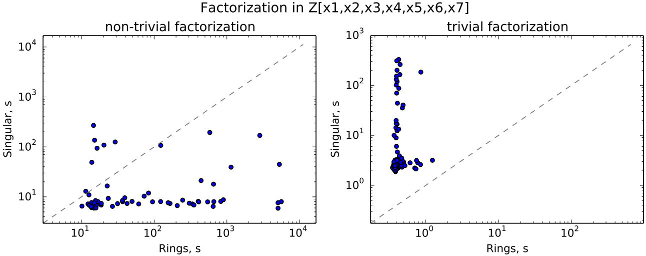
Params:

```
#factors = 3
#terms = 20
```
$\mathtt{exp}_{\mathtt{min}} = 0$
$\mathtt{exp}_{\mathtt{max}} = 30$



Benchmarks: *polynomial factorization*

Benchmark: generate three polynomials a, b and c at random and compute $factor(abc)$ (non-trivial) and $factor(abc + 1)$ (trivial)



Params:

```
#factors = 3
#terms = 20
```
$exp_{min} = 0$
$exp_{max} = 30$

Benchmarks: *Gröbner bases*

- **Computation of Gröbner bases is a key component of:**
  - *Solving polynomial equations*
  - *Computations with polynomial ideals*
  - *Different decomposition methods (like partial fractions)*
  - ...

- Efficient algorithms for Gröbner bases is one of the ”hottest” research areas
- There are very dedicated and efficient tools for Gröbner bases like `FGB` (proprietary, built-in in `Maple`) or `OpenF4` (free)
- Neither of considered here tools (except `Singular` to some extent) are specifically optimized for ”record” Gröbner bases computation

Benchmarks: *Gröbner bases*

Problem	Cf. ring	Rings	Mathematica	Singular
cyclic-7	$\mathbb{Z}_{1000003}$	3s	26s	N/A
cyclic-8	$\mathbb{Z}_{1000003}$	51s	897s	39s
cyclic-9	$\mathbb{Z}_{1000003}$	14603s	∞	8523s
katsura-7	$\mathbb{Z}_{1000003}$	0.5s	2.4s	0.1s
katsura-8	$\mathbb{Z}_{1000003}$	2s	24s	1s
katsura-9	$\mathbb{Z}_{1000003}$	2s	22s	1s
katsura-10	$\mathbb{Z}_{1000003}$	9s	216s	9s
katsura-11	$\mathbb{Z}_{1000003}$	54s	2295s	65s
katsura-12	$\mathbb{Z}_{1000003}$	363s	28234s	677s
katsura-7	$\mathbb{Z}$	5s	4s	1.2s
katsura-8	$\mathbb{Z}$	39s	27s	10s
katsura-9	$\mathbb{Z}$	40s	29s	10s
katsura-10	$\mathbb{Z}$	1045s	251s	124s

Rings *is fast enough for practice and speed improvement in case of* $\mathbb{Z}$ *is the subject of the upcoming release*

`Rings`: *some technical aspects*

- `Rings` is **93,137** `(.java)` **+ 8,386** `(.scala)` lines of code
- **GitHub**: **https://github.com/PoslavskySV/rings**
- **RT*D**: **https://rings.readthedocs.io**
- **Ref:** **CPC, Vol. 235, 2019, pp. 400-413, arXiv:1712.02329 [cs.SC]**
- Thousands of unit and integration:

  http://circleci.com/gh/PoslavskySV/rings (CI)
- Interactive **REPL**:

  ```
  sh> brew install PoslavskySV/rings/rings.repl
  ```

Rings: *an overview*

- **Computational Number Theory**
  - *primes: sieving, testing, factorization*
  - *univariate polynomials over arbitrary coefficient rings: fast arithmetic, gcd, factorization etc.*
  - *Galois fields & Algebraic number fields*
- **Computational Commutative Algebra**
  - *multivariate polynomials over arbitrary coefficient rings: fast arithmetic, gcd, factorization etc.*
  - *fast rational function arithmetic*
- **Computational Algebraic Geometry**
  - *Gröbner bases*
  - *Systems of polynomial equations*
  - *Ideals in multivariate polynomial rings*

THANKS!

BACKUP

Rings: *design by examples*

Rational function arithmetic:

```
 // rational functions Frac(Z[x, y, z])
implicit val ring = Frac(MultivariateRing(Z, Array("x", "y", "z")))
val (x, y, z) = ring("x", "y", "z") // parse elements from strings

// construct expression
val epxr1 = x / y + z.pow(2) / (x + y - 1)

// or import from file
import scala.io.Source
val expr2 = ring(Source.fromFile("myFile.txt").mkString)

val expr3 = expr1 * expr2
// unique factor decomposition of fraction
println ( ring.factor(expr3) )
```

- Fractions are always reduced to a common denominator and GCD is canceled automatically;

Rings: *design by examples*

Diophantine equations: solve $\sum f_i s_i = gcd(f_1, \ldots, f_N)$ for given f_i and unknown s_i:

Rings: *design by examples*

Diophantine equations: solve $\sum f_i s_i = gcd(f_1, \ldots, f_N)$ for given f_i and unknown s_i:

```
def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) =
  fi.foldLeft((ring(0), Seq.empty[E])) { case ((gcd, seq), f) =>
    val xgcd = ring.extendedGCD(gcd, f)
    (xgcd(0), seq.map(_ * xgcd(1)) :+ xgcd(2))
  }
```

Rings: *design by examples*

Diophantine equations: solve $\sum f_i s_i = gcd(f_1, \ldots, f_N)$ for given f_i and unknown s_i:

```
def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) =
  fi.foldLeft((ring(0), Seq.empty[E])) { case ((gcd, seq), f) =>
    val xgcd = ring.extendedGCD(gcd, f)
    (xgcd(0), seq.map(_ * xgcd(1)) :+ xgcd(2))
  }
```

Diophantine equations in $Frac(GF(17^3)[x, y, z])[W]$**:**

```
// Galois field GF(17, 3)
implicit val gf    = GF(17, 3, "t")
// Rational functions in x, y, z over GF(17, 3)
implicit val fracs = Frac(MultivariateRing(gf, Array("x", "y", "z")))
// univariate ring Frac(GF(17, 3)[x,y,z])[W]
implicit val ring  = UnivariateRing(fracs, "W")

val f1 = ring("1 + t^2 + x/y - W^2") // parse elements from strings
val f2 = ring("1 + W + W^3/(t - x)") // parse elements from strings
val f3 = ring("t^2 - x  - W^4")      // parse elements from strings
// do the job
val solve = solveDiophantine(Seq(f1, f2, f3))
```

- this is a piece of one-loop master integral reduction algorithm

`Rings`: *modular arithmetic with machine numbers*

- *Arithmetic in $\mathbb{Z}_p$ with word-sized p ($p < 2^{64}$) lies in the basis of the most part of fundamental algorithms and directly affects performance of all computations*

- $N \mod p \equiv N - \lfloor N/p \rfloor \times p$ — this is how remainder is computed by the CPU
- Integer division (`DIV`) is one of the most inneficient CPU instructions:
  - *it has 20-80 times worth throughput than e.g.* `MUL` *(for Intel Skylake)*
  - *it breaks CPU pipelining*
- **The hack** (*Barret reduction; see Hacker's delight*):
  - Compute *once* the $magic = \lfloor 2^m/p \rfloor$ for sufficiently large m
  - Then $\lfloor N/p \rfloor = (N \times magic)/2^m$ which is one `MUL` and one `SHIFT`
- **Another hack**:
  - $(a \times b)_{\mathbb{Z}_p} = (a \times b) \mod p$ if a and b are less than 2^{32} (fast)
  - else, the Montogomery multiplication is used

- Modular arithmetic in `Rings` is 3-5 times faster than with native CPU instructions and especially fast in $\mathbb{Z}_p$ rings with $p < 2^{32}$

Rings: *polynomials*

- **Univariate polynomials are always dense:**

polynomial: $c_0 \quad + \quad c_1 x \quad + \quad c_2 x^2 \quad + \quad \ldots \quad + \quad c_n x^n$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

array:

c_0	c_1	c_2	$\ldots$	c_n

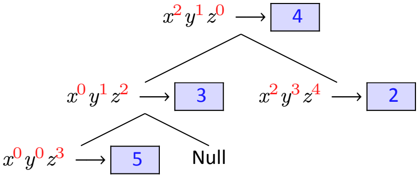
  - *native arrays are used to store univariate polynomials*

- **Multivariate polynomials are sparse:**

polynomial: $2\,x^2 y^3 z^4 + 3\,yz^2 + 4\,x^2 y + 5\,z^3$

$\downarrow\downarrow\downarrow$

tree map:



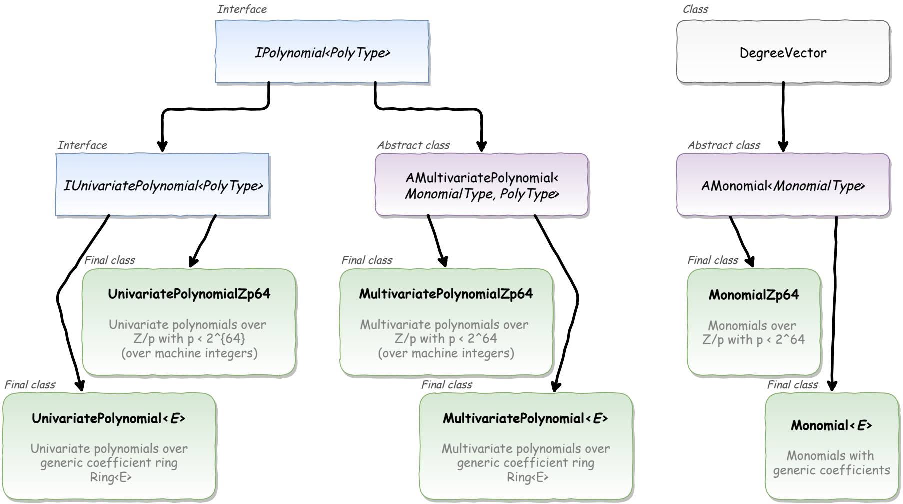
  - *red-black tree map is used to store multivariate terms*

`Rings`: *polynomials*

- Polynomials over $\mathbb{Z}_p$ with $p < 2^{64}$ (machine numbers) have separate implementations
  - `E[] data` — *generic array for univariate polynomials over generic rings (with elements of reference type* E*)*
  - `long[] data` — *native array for univariate polynomials over* $\mathbb{Z}_p$ *with* $p < 2^{64}$ *(machine words)*

- Motivation:
  - $\mathbb{Z}_p$ with $p < 2^{64}$ already has separate implementation
  - more specific and optimized algorithms
  - avoid inefficient generics with primitive types in `Java` (however, e.g. in `C/C++` one would have to do the same, like in `NTL`)

Rings: *polynomials*



Rings: *polynomials*

	Univariate (n is polynomial degree)	**Multivariate** (n is polynomial size)
Addition/Subtraction	$O(n)$	$O(n \log n)$
Multiplication	$O\left(n^{\log_2 3}\right)$ via Karatsuba method (with lots of heuristic to reduce the constant)	$O\left(nm \log(n) \log(m)\right)$ via plain method (Kronecker trick is used to significantly reduce the constant)
Division	$O\left(n^{\log_2 3}\right)$ via Newton's iteration (with lots of heuristic to reduce the constant)	$O\left(nm \log(n) \log(m)\right)$ via plain method
Evaluation	$O\left(n\right)$ via Horner method	$O\left(n \log(d)\right)$ via plain method with caching or via recursive Horner scheme

Rings: *polynomial GCD*

- **Univariate (e)GCD:**
  - Rings switches between Euclidean GCD, Half-GCD and Brown's GCD (for coefficient rings with characteristic zero)
- **Multvariate GCD:**
  - for sparse inputs Rings uses Zippel's algorithm based on linear algebra
  - for relatively dense polynomials Rings uses Enhanced Extended Zassenhaus (EEZ) approach based on multivariate (ideal-adic) Hensel lifting
  - when the coefficient ring has very small cardinality Rings uses a version of Kaltofen-Monagan generic GCD algrotithm
  - for coefficient rings of characteristic zero, modular algrotihm (Zippel-like for sparse or Brown-like with EEZ for dense inputs) is used
  - *all these contain tons of heuristic (code for algorithms spans more than 5,000 l.o.c.)*

Rings: *polynomial GCD*

Dense input:

$$a = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1$$
$$b = (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1$$
$$g = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3$$

Problem	Cf. ring	Rings	Mathematica	FORM	Fermat	Singular
$gcd(ag, bg)$	$\mathbb{Z}$	104s	115s	148s	1759s	141s
$gcd(ag, bg+1)$	$\mathbb{Z}$	0.4s	2s	0.3s	0.1s	0.4s
$gcd(ag, bg)$	$\mathbb{Z}_{524287}$	25s	33s	N/A	147s	46s
$gcd(ag, bg+1)$	$\mathbb{Z}_{524287}$	0.5s	2s	N/A	0.2s	0.2s

Rings: *polynomial GCD*

Dense input:

$$a = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1$$
$$b = (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1$$
$$g = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3$$

Problem	Cf. ring	Rings	Mathematica	FORM	Fermat	Singular
$gcd(ag, bg)$	$\mathbb{Z}$	104s	115s	148s	1759s	141s
$gcd(ag, bg+1)$	$\mathbb{Z}$	0.4s	2s	0.3s	0.1s	0.4s
$gcd(ag, bg)$	$\mathbb{Z}_{524287}$	25s	33s	N/A	147s	46s
$gcd(ag, bg+1)$	$\mathbb{Z}_{524287}$	0.5s	2s	N/A	0.2s	0.2s

- GCD performance on trivial input is very important (since e.g. most part of GCDs computed in rational function arithmetic are trivial)

Rings: *polynomial GCD*

Dense input:

$$a = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1$$
$$b = (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1$$
$$g = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3$$

Problem	Cf. ring	Rings	Mathematica	FORM	Fermat	Singular
$gcd(ag, bg)$	$\mathbb{Z}$	104s	115s	148s	1759s	141s
$gcd(ag, bg+1)$	$\mathbb{Z}$	0.4s	2s	0.3s	0.1s	0.4s
$gcd(ag, bg)$	$\mathbb{Z}_{524287}$	25s	33s	N/A	147s	46s
$gcd(ag, bg+1)$	$\mathbb{Z}_{524287}$	0.5s	2s	N/A	0.2s	0.2s

- GCD performance on trivial input is very important (since e.g. most part of GCDs computed in rational function arithmetic are trivial)
- one have to make a trade-off between performance on non-trivial and trivial inputs

Rings: *polynomial factorization*

Dense input:

$$p_1 = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^{15} - 1$$

$$\begin{aligned} p_2 = -1 + &(1 + 3x_1x_2 + 5x_2x_3 + 7x_3x_4 + 9x_4x_5 + 11x_5x_6 + 13x_6x_7 + 15x_7x_1)^3 \\ &\times(1 + 3x_1x_3 + 5x_2x_4 + 7x_3x_5 + 9x_6x_5 + 11x_7x_6 + 13x_6x_1 + 15x_7x_2)^3 \\ &\times(1 + 3x_1x_4 + 5x_2x_5 + 7x_3x_6 + 9x_6x_7 + 11x_7x_1 + 13x_6x_2 + 15x_7x_3)^3 \end{aligned}$$

Problem	Cf. ring	Rings	Singular	Mathematica
$factor(p_1)$	$\mathbb{Z}$	55s	20s	271s
$factor(p_1)$	$\mathbb{Z}_2$	0.25s	$>$ 1h	N/A
$factor(p_1)$	$\mathbb{Z}_{524287}$	28s	109s	N/A
$factor(p_2)$	$\mathbb{Z}$	23s	12s	206s
$factor(p_2)$	$\mathbb{Z}_2$	6s	3s	N/A
$factor(p_2)$	$\mathbb{Z}_{524287}$	26s	9s	N/A

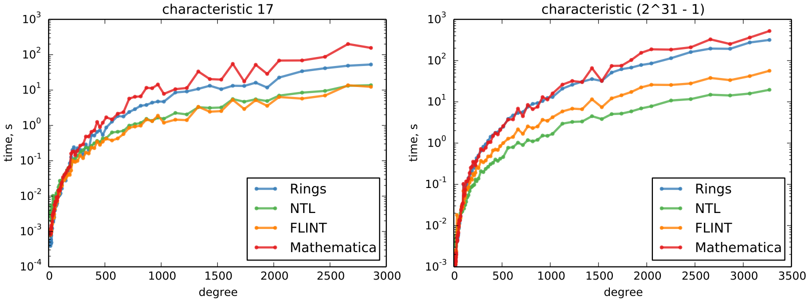
Benchmarks: *polynomial factorization*

- **Univariate factorization:**
  - `Rings` switches between Cantor-Zassenhaus and Shoup's baby-step-giant-step algorithms for polynomials over finite fields
  - p-adic Hensel lifting is used to compute factorization over $\mathbb{Z}$ (resp. $\mathbb{Q}$)
- **Multvariate factorization:**
  - for bivariate polynomials Bernardin's algorithm is used
  - Kaltofen's algorithm is used in all other cases
  - ideal-adic Hensel lifting switches between sparse (based on linear algebra) and dense (based on Bernardin's algorithm)
  - *all these contain tons of heuristic*

Rings: *polynomial factorization*

Univariate input:

$$p_{\text{deg}}[x] = 1 + \sum_{i=1}^{i \le \text{deg}} i \times x^i$$



- This benchmark covers almost all aspects of univariate arithmetic in finite fields

Rings: *parametric number fields*

```
// Q[c, d]
val params = Frac(MultivariateRing(Q, Array("c", "d")))
// A minimal polynomial X^2 + c = 0
val generator = UnivariatePolynomial(params("c"), params(0), params(1))(params)
// Algebraic number field Q(sqrt(c)), here "s" denotes square root of c
implicit val cfRing = AlgebraicNumberField(generator, "s")
// ring of polynomials  Q(sqrt(c))(x, y, z)
implicit val ring = MultivariateRing(cfRing, Array("x", "y", "z"))
// bring variables
val (x,y,z,s) = ring("x", "y", "z", "s")
// some polynomials
val poly1 = (x + y + s).pow(3) * (x - y - z).pow(2)
val poly2 = (x + y + s).pow(3) * (x + y + z).pow(4)

// compute gcd
val gcd = PolynomialGCD(poly1, poly2)
println(ring stringify gcd)
```