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
An *incomplete* list of similar software:

- **Closed source (proprietary)**
  - Magma, Maple, Mathematica, Fermat, ...

- **Open source (free)**
  - 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](https://ringsalgebra.io)
Basic algebraic definitions

- **Ring**: a set of elements with ”+” and ”×” operations defined.  
  *Examples*:  
  - \( \mathbb{Z} \) — ring of integers  
  - \( \mathbb{Z}[i] \) — Gaussian integers  
  - \( R[X] \) — 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  
  - \( \text{Frac}(R[X]) \) — field of rational functions

- **Ideal**: a subset of ring elements closed under multiplication with ring.  
  *Examples*:  
  - Given a set of generators \( \{f_i(x, y, ...)\} \in R[x, y, ...] \) ideal is formed by all elements of the form  
    \[
    c_1(x, y, ...) \times f_1(x, y, ...) + ... + c_n(x, y, ...) \times f_n(x, y, ...)
    \]
Rings: *design by examples*

Simple example:

```scala
1  implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
2  val x = ring("x") // parse polynomial from string
3  val poly = x.pow(100) - 1 // construct polynomial programmatically
4  val factors = Factor(poly) // factorize polynomial
5  println(factors)
```
Rings: *design by examples*

Simple example:

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

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

```scala
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:

1. `implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]`
2. `val x = ring("x") // parse polynomial from string`
3. `val poly = x.pow(100) - 1 // construct polynomial programmatically`
4. `val factors = Factor(poly) // factorize polynomial`
5. `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:

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

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

```scala
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`

```scala
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:**

1. // ring of elements of type E
2. implicit val ring : Ring[E] = ...
3. val a : E = ...
4. val b : E = ...
5. val sum = a + b // compiles to ring.add(a, b)
6. val mul = a * b // compiles to ring.multiply(a, b)
7. val div = a / b // compiles to ring.divideExact(a, b)

**Example:**

1. val a : IntZ = Z(12)
2. val b : IntZ = Z(13)
3. assert (a * b == Z(156)) // no any implicit Ring[IntZ]
4. {
5.     implicit val ring = Zp(17) // implicit Ring[IntZ]
6.     assert (a * b == Z(3))     // multiplication modulo 17
7. }

Example: $a = 12, b = 13$

Assertion: $a 	imes b = 156$ (no implicit ring)

Example: $Z_p(17)$

Assertion: $a 	imes b = 3$ (multiplication modulo 17)
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

\[
\begin{align*}
(m_1 - m_2 - m_3)(m_1 + m_2 - m_3)(m_1 - m_2 + m_3)(m_1 + m_2 + m_3) \\
+ 2(-m_1^2 - m_2^2 + m_3^2) Y_1 + 2(m_1^2 - m_3^2 - m_2^2) Y_2 + 2(m_1^2 - m_2^2 - m_3^2) Y_3 \\
+ Y_1^2 + Y_2^2 + Y_3^2 - 2Y_1 Y_2 - 2Y_1 Y_3 - 2Y_2 Y_3 \equiv 0
\end{align*}
\]

\[Y_1 = ((s - t)^2 - m_3^2) \quad Y_2 = (s^2 - m_1^2) \quad Y_3 = (t^2 - m_2^2)\]
Rings: *design by examples*

**Multivariate polynomials & rational functions & simplifications**

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

8  val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9   / (s.pow(2) - m1.pow(2))
10  / (t.pow(2) - m2.pow(2)))
11  // or just parse from string
12  // val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")
13```

```
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")

// field of coefficients Frac(Z[m1, m2, m3])
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**

1. // field of coefficients \( \text{Frac}(\mathbb{Z}[m_1, m_2, m_3]) \)
2. `val cfs = Frac(MultivariateRing(\mathbb{Z}, \text{Array}("m1","m2","m3")))`
3. // field of rational functions \( \text{Frac}(\text{Frac}(\mathbb{Z}[m_1, m_2, m_3])[s, t]) \)
4. `implicit val field = Frac(MultivariateRing(cfs, \text{Array}("s", "t")))`
5. // parse variables from strings
6. `val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")`

8. `val frac = (1 / ((s - t).pow(2) - m3.pow(2))`
9. `\quad / (s.pow(2) - m1.pow(2))`
10. `\quad / (t.pow(2) - m2.pow(2)))`
11. `...`

**Result:**

\[
\begin{align*}
&\frac{1}{((s-t)^2-m_3^2)(s^2-m_1^2)(t^2-m_2^2)} = \\
&\quad - \frac{1}{8m_1m_2m_3(m_1 + m_2 + m_3)} \quad \frac{1}{(-m_3 - t + s)(t - m_2)} \\
&\quad - \frac{1}{8m_1m_2m_3(m_1 + m_2 + m_3)} \quad \frac{1}{(-m_3 - t + s)(s + m_1)} \\
&\quad + \ldots (+22 \text{ other terms})
\end{align*}
\]
Rings: *design by examples*

**Multivariate polynomials & rational functions & simplifications**

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

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

14 // bring in the form with algebraically independent denominators
15 val dec = GroebnerMethods.LeinartDecomposition(frac)
16 // simplify fractions (factorize)
17 val decSimplified = dec.map(f => field.factor(f))
18 // pretty print
19 decSimplified.map(f => field.stringify(f)).foreach(println)
```
Rings: *design by examples*

**Multivariate polynomials & rational functions & simplifications**

```scala
// 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**

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

8 val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9     / (s.pow(2) - m1.pow(2))
10     / (t.pow(2) - m2.pow(2)))
11 ...

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

<table>
<thead>
<tr>
<th>Built-in ring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{Z} )</td>
<td>ring of integers</td>
</tr>
<tr>
<td>( \mathbb{Q} )</td>
<td>field of rationals</td>
</tr>
<tr>
<td>GaussianRationals</td>
<td>field of complex rational numbers ( \mathbb{Q}(i) )</td>
</tr>
<tr>
<td>( \mathbb{Z}_p(p) )</td>
<td>integers modulo ( p )</td>
</tr>
<tr>
<td>( \text{GF}(p,q) )</td>
<td>finite field with cardinality ( p^q )</td>
</tr>
<tr>
<td>AlgebraicNumberField(alpha)</td>
<td>algebraic number field ( F(\alpha_1, \ldots, \alpha_s) )</td>
</tr>
<tr>
<td>( \text{Frac}(R) )</td>
<td>field of fractions over Euclidean ring ( R )</td>
</tr>
<tr>
<td>( \text{UnivariateRing}(R, x) )</td>
<td>univariate ring ( R[x] )</td>
</tr>
<tr>
<td>( \text{MultivariateRing}(R, vars) )</td>
<td>multivariate ring ( R[x_1, x_2, \ldots] )</td>
</tr>
<tr>
<td>( \text{QuotientRing}(R, ideal) )</td>
<td>multivariate quotient ring ( R[x_1, x_2, \ldots]/I )</td>
</tr>
</tbody>
</table>
Rings: implementation aspects

- Efficient Z/p rings
- Arbitrary precision integer arithmetic
- Prime numbers: sieving / testing
- Fast univariate arithmetic
  - Subresultant sequences
  - Univariate (e)GCD (Half-GCD, Modular, Subresultant)
  - Square-free factorization
  - Univariate factorization in finite fields
  - Univariate factorization in \( \mathbb{Z}[x] \)
  - Univariate Hensel (p-adic) lifting
- Multivariate (e)GCD (Brown / Zippel / EEZ / generic)
  - Multivariate Hensel (ideal-adic) lifting (dense / sparse)
- Multivariate factorization
- Univariate interpolation
- Multivariate interpolation
- Galois fields
- CRT / Rational reconstruction

Dependency graph
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 $\exp_{\text{min}} \leq \exp_i < \exp_{\text{max}}$; the total degree will be $N_{\text{vars}}\exp_{\text{min}} \leq \exp_{\text{tot}} < N_{\text{vars}}\exp_{\text{max}}$
    **Example** ($\exp_{\text{min}} = 0, \exp_{\text{max}} = 10$):
    \[
    \cdots + x^5 y^2 z^8 + x^3 y^8 z^6 + \ldots
    \]
  - *Sharp exponents* (multinomial distribution):
    choose the total degree $\exp_{\text{tot}}$, then for the first variable $0 \leq \exp_1 \leq \exp_{\text{tot}}$, for the second variable $0 \leq \exp_2 \leq (\exp_{\text{tot}} - \exp_1)$ and so on
    **Example** ($\exp_{\text{tot}} = 10$):
    \[
    \cdots + x^7 y^2 z^1 + x^0 y^8 z^2 + \ldots
    \]
**Benchmarks**: polynomial GCD

**Params** \((a, b, g)\):

- \#terms = 40
- \#bits = 32
- \(\text{exp}_{\text{min}} = 0\)
- \(\text{exp}_{\text{max}} = 30\)

---

- \#terms = 40
- \#bits = 32
- \(\text{exp}_{\text{tot}} = 50\)

---

- \#terms = 40
- \#bits = 1
- \(\text{exp}_{\text{min}} = 0\)
- \(\text{exp}_{\text{max}} = 30\)

▷ reFORM (see Ben’s talk) seems has comparable performance
Benchmarks: polynomial GCD

“Huge” problems:

<table>
<thead>
<tr>
<th>size of input polynomials</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Params (a,b,g):
exp\text{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 \(\text{factor}(abc)\) (non-trivial) and \(\text{factor}(abc + 1)\) (trivial)

**Params:**
- \#factors = 3
- \#terms = 20
- \(\exp_{\text{min}} = 0\)
- \(\exp_{\text{max}} = 30\)
Benchmarks: *polynomial factorization*

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

**Params:**
- #factors = 3
- #terms = 20
- $\exp_{\text{min}} = 0$
- $\exp_{\text{max}} = 30$
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

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

**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
- 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!
**Rings: design by examples**

**Rational function arithmetic:**

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

5  // construct expression
6  val expr1 = x / y + z.pow(2) / (x + y - 1)

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

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

▶ Fractions are always reduced to a common denominator and GCD is canceled automatically;
Rings: *design by examples*

**Diophantine equations:** solve $\sum f_is_i = \gcd(f_1, \ldots, f_N)$ for given $f_i$ and unknown $s_i$: 
**Rings: design by examples**

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

```scala
1  def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) =
2    fi.foldLeft((ring(0), Seq.empty[E])) { case ((gcd, seq), f) =>
3      val xgcd = ring.extendedGCD(gcd, f)
4      (xgcd(0), seq.map(_ * xgcd(1)) :+ xgcd(2))
5    }
```
Rings: *design by examples*

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

```scala
1  def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) =
2    fi.foldLeft((ring(0), Seq.empty[E])) { case ((gcd, seq), f) =>
3      val xgcd = ring.extendedGCD(gcd, f)
4      (xgcd(0), seq.map(_ * xgcd(1)) ++ xgcd(2))
5    }
```

Diophantine equations in $\text{Frac}(\mathbb{GF}(17^3)[x, y, z])[W]$:

```scala
1  // Galois field GF(17, 3)
2  implicit val gf = GF(17, 3, "t")
3  // Rational functions in x, y, z over GF(17, 3)
4  implicit val fracs = Frac(MultivariateRing(gf, Array("x", "y", "z")))
5  // univariate ring Frac(GF(17, 3)[x,y,z])[W]
6  implicit val ring = UnivariateRing(fracs, "W")
7
8  val f1 = ring("1 + t^2 + x/y - W^2") // parse elements from strings
9  val f2 = ring("1 + W + W^3/(t - x)") // parse elements from strings
10 val f3 = ring("t^2 - x - W^4") // parse elements from strings
11 // do the job
12 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 inefficient 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: \[ c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \]

  array:

  \[
  \begin{array}{c}
  c_0 \\
  c_1 \\
  c_2 \\
  \vdots \\
  c_n \\
  \end{array}
  \]

  ➤ *native arrays are used to store univariate polynomials*

- **Multivariate polynomials are sparse:**

  polynomial: \[ 2 x^2 y^3 z^4 + 3 y z^2 + 4 x^2 y + 5 z^3 \]

  ➤ *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

Interface

IPolynomial<PolyType>

IUnivariatePolynomial<PolyType>

UnivariatePolynomial<
E
>

Univariate polynomials over
generic coefficient ring
Ring<E>

UnivariatePolynomialZp64

Univariate polynomials over
Z/p with p < 2^{64}
(over machine integers)

MultivariatePolynomial<
E
>

Multivariate polynomials over
generic coefficient ring
Ring<E>

MultivariatePolynomialZp64

Multivariate polynomials over
Z/p with p < 2^{64}
(over machine integers)

AMonomial<
MonomialType
>

Monomial with
generic coefficients

AMonomial<MonomialType>

Monomials with
generic coefficients

AMonomial<
MonomialType
>

Monomials over
Z/p with p < 2^{64}

DegreeVector

Final class

Implementation: polynomials
### Rings: polynomials

<table>
<thead>
<tr>
<th></th>
<th>Univariate</th>
<th>Multivariate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addition/Subtraction</strong></td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
<td>$O\left(n^{\log_2 3}\right)$ via Karatsuba method (with lots of heuristic to reduce the constant)</td>
<td>$O\left(nm \log(n) \log(m)\right)$ via plain method (Kronecker trick is used to significantly reduce the constant)</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>$O\left(n^{\log_2 3}\right)$ via Newton’s iteration (with lots of heuristic to reduce the constant)</td>
<td>$O\left(nm \log(n) \log(m)\right)$ via plain method</td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td>$O(n)$ via Horner method</td>
<td>$O(n \log(d))$ via plain method with caching or via recursive Horner scheme</td>
</tr>
</tbody>
</table>
Rings: *polynomial GCD*

**Univariate (e)GCD:**
- Rings switches between Euclidean GCD, Half-GCD and Brown’s GCD (for coefficient rings with characteristic zero)

**Multivariate 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 algrothhm (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:

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gcd(a, bg))</td>
<td>(\mathbb{Z})</td>
<td>104s</td>
<td>115s</td>
<td>148s</td>
<td>1759s</td>
<td>141s</td>
</tr>
<tr>
<td>(gcd(a, bg + 1))</td>
<td>(\mathbb{Z})</td>
<td>0.4s</td>
<td>2s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>(gcd(a, bg))</td>
<td>(\mathbb{Z}_{524287})</td>
<td>25s</td>
<td>33s</td>
<td>N/A</td>
<td>147s</td>
<td>46s</td>
</tr>
<tr>
<td>(gcd(a, bg + 1))</td>
<td>(\mathbb{Z}_{524287})</td>
<td>0.5s</td>
<td>2s</td>
<td>N/A</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>
Rings: *polynomial GCD*

Dense input:

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd(a, bg)</td>
<td>(\mathbb{Z})</td>
<td>104s</td>
<td>115s</td>
<td>148s</td>
<td>1759s</td>
<td>141s</td>
</tr>
<tr>
<td>gcd(a, bg + 1)</td>
<td>(\mathbb{Z})</td>
<td>0.4s</td>
<td>2s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>gcd(a, bg)</td>
<td>(\mathbb{Z}_{524287})</td>
<td>25s</td>
<td>33s</td>
<td>N/A</td>
<td>147s</td>
<td>46s</td>
</tr>
<tr>
<td>gcd(a, bg + 1)</td>
<td>(\mathbb{Z}_{524287})</td>
<td>0.5s</td>
<td>2s</td>
<td>N/A</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>

- 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:

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gcd(a, , b) )</td>
<td>( \mathbb{Z} )</td>
<td>104s</td>
<td>115s</td>
<td>148s</td>
<td>1759s</td>
<td>141s</td>
</tr>
<tr>
<td>( \gcd(a, , b + 1) )</td>
<td>( \mathbb{Z} )</td>
<td>0.4s</td>
<td>2s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>( \gcd(a, , b) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>25s</td>
<td>33s</td>
<td>N/A</td>
<td>147s</td>
<td>46s</td>
</tr>
<tr>
<td>( \gcd(a, , b + 1) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>0.5s</td>
<td>2s</td>
<td>N/A</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>

- 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:

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

\[ p_2 = -1 + (1 + 3x_1 x_2 + 5x_2 x_3 + 7x_3 x_4 + 9x_4 x_5 + 11x_5 x_6 + 13x_6 x_7 + 15x_7 x_1)^3 \]
\[ \times (1 + 3x_1 x_3 + 5x_2 x_4 + 7x_3 x_5 + 9x_6 x_5 + 11x_7 x_6 + 13x_6 x_1 + 15x_7 x_2)^3 \]
\[ \times (1 + 3x_1 x_4 + 5x_2 x_5 + 7x_3 x_6 + 9x_6 x_7 + 11x_7 x_1 + 13x_6 x_2 + 15x_7 x_3)^3 \]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Singular</th>
<th>Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor(p₁)</td>
<td>( \mathbb{Z} )</td>
<td>55s</td>
<td>20s</td>
<td>271s</td>
</tr>
<tr>
<td>factor(p₁)</td>
<td>( \mathbb{Z}_2 )</td>
<td>0.25s</td>
<td>&gt; 1h</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p₁)</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>28s</td>
<td>109s</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p₂)</td>
<td>( \mathbb{Z} )</td>
<td>23s</td>
<td>12s</td>
<td>206s</td>
</tr>
<tr>
<td>factor(p₂)</td>
<td>( \mathbb{Z}_2 )</td>
<td>6s</td>
<td>3s</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p₂)</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>26s</td>
<td>9s</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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. $\mathbb{Q}$)

**Multivariate 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:

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

This benchmark covers almost all aspects of univariate arithmetic in finite fields.
Rings: *parametric number fields*

```scala
1 // Q[c, d]
2 val params = Frac(MultivariateRing(Q, Array("c", "d")))
3 // A minimal polynomial X^2 + c = 0
4 val generator = UnivariatePolynomial(params("c"), params(0), params(1))
5 // params

6 // Algebraic number field Q(sqrt(c)), here "s" denotes square root of c
7 implicit val cfRing = AlgebraicNumberField(generator, "s")
8 // ring of polynomials Q(sqrt(c))(x, y, z)
9 implicit val ring = MultivariateRing(cfRing, Array("x", "y", "z"))
10 // bring variables
11 val (x,y,z,s) = ring("x", "y", "z", "s")
12 // some polynomials
13 val poly1 = (x + y + s).pow(3) * (x - y - z).pow(2)
14 val poly2 = (x + y + s).pow(3) * (x + y + z).pow(4)
15 // compute gcd
16 val gcd = PolynomialGCD(poly1, poly2)
17 println(ring stringify gcd)
```