Computer Algebra in Physics Research

Stanislav Poslavsky

NRC "Kurchatov Institute" — IHEP, Protvino, Russia

ACAT19

Saas Fee, 2019

OUTLINE

COMPUTER ALGEBRA

↓

KEY MATHEMATICAL CONCEPTS

↓

KEY PROGRAMMING ASPECTS

↓

BENCHMARKS

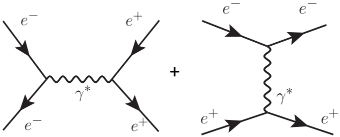
Computer algebra & hep-th

cross section $$\sigma = \int d\Phi \left| \sum_i \mathcal{A}mplitude_i \right|^2$$

COMPUTER ALGEBRA & HEP-TH

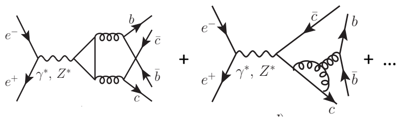
cross section $\sigma = \int d\Phi \left| \sum_i \mathcal{A}mplitude_i \right|^2$

Simple process
(textbook example):



$$\underbrace{\frac{\text{Tr}\{\gamma_\mu(\hat{p}+m)\gamma_\mu(\hat{k}+m)\dots\}}{(q^2-m^2)} + \dots}_{\text{with paper \& pencil}}$$

Real world:

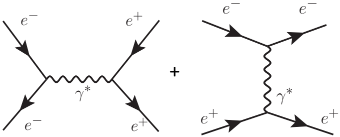


thousands of rational expressions, producing millions of terms in the intermediate results

Computer algebra & hep-th

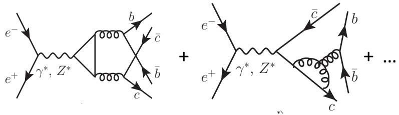
cross section $\sigma = \int d\Phi \left| \sum_i \mathcal{A}mplitude_i \right|^2$

Simple process *(textbook example)*:



$$\underbrace{\frac{\mathrm{Tr}\{\gamma_\mu(\hat{p}+m)\gamma_\mu(\hat{k}+m)\dots\}}{(q^2-m^2)} + \dots}_{\text{with paper \& pencil}}$$

Real world:



thousands of rational expressions, producing millions of terms in the intermediate results

Key performance bottlenecks:

Arithmetic	$\Longrightarrow$	1. *Multiplication, evaluation ...*
Putting terms together	$\Longrightarrow$	2. *Greatest Common Divisors*
Simplification	$\Longrightarrow$	3. *Polynomial Factorization*
Advanced	$\Longrightarrow$	4. *Gröbner bases, elimination ...*

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,	Singular, Macaulay2, CoCoA,
Fermat, ...	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*

RING HOMOMORPHISM

RING HOMOMORPHISM: *modular methods*

Euclidean algorithm (GCD):

```
function gcd(a, b)
    if b = 0
       return a;
    else
       return gcd(b, a mod b);
```

RING HOMOMORPHISM: *modular methods*

Euclidean algorithm (GCD):

```
function gcd(a, b)
    if b = 0
       return a;
    else
       return gcd(b, a mod b);
```

Applying it to

$$\texttt{gcd}(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1$$

will produce the following 3166 digit number at some intermediate step:

211782656677150921740253822599595701172055778537749109160433930907991796863546398308149265416417897047
716799242768103353160906637850318917854160052986686548498598432559533166777746185195074259067328652710
553540538380427535 …(3166 digits) …

RING HOMOMORPHISM: *modular methods*

Euclidean algorithm (GCD):

```
function gcd(a, b)
    if b = 0
       return a;
    else
       return gcd(b, a mod b);
```

Applying it to

$$\mathtt{gcd}(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1$$

will produce the following 3166 digit number at some intermediate step:

211782656677150921740253822599595701172055778537749109160433930907991796863546398308149265416417897047
716799242768103353160906637850318917854160052986686548498598432559533166777746185195074259067328652710
553540538380427535 ...(3166 digits) ...

- This is ***intermediate expression swell***. It occurs always in fact.
- Computations become ∞ slow due to exponential growth of coefficients

Ring Homomorphism: *modular methods*

Euclidean algorithm (GCD):

```
function gcd(a, b)
    if b = 0
       return a;
    else
       return gcd(b, a mod b);
```

Applying it to

$$\mathtt{gcd}(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1$$

will produce the following 3166 digit number at some intermediate step:

21178265667715092174025382259959570117205577853774910916043393090799179686354639830814926541641789 7047
71679924276810335316090663785031891785416005298668654849859843255953316677774618519507425906732865 2710
553540538380427535 ...(3166 digits) ...

- This is ***intermediate expression swell***. It occurs always in fact.
- Computations become ∞ slow due to exponential growth of coefficients

- **Observation**:
  If we compute modulo 17, we obtain the same result, but all intermediate numbers are bounded by 17

Ring Homomorphism: *modular methods*

- **Idea**:
  - compute GCD modulo several different 32-bit primes, than ”reconstruct” result

$$\begin{aligned} & gcd(a,b) \quad \text{mod} \quad 17 &&= 2+4\,x+3\,x^2 \\ & gcd(a,b) \quad \text{mod} \quad 19 &&= 3+6\,x+2\,x^2 \\ \Longrightarrow\ & gcd(a,b) \quad \text{mod} \quad 17\times 19 &&= 155+310\,x+173\,x^2 \end{aligned}$$

  - in practice this is ∞ times faster than direct computation

RING HOMOMORPHISM: *modular methods*

- **Idea**:
  - compute GCD modulo several different 32-bit primes, than ”reconstruct” result

$$\begin{aligned} & gcd(a,b) \quad \text{mod} \quad 17 && = \quad 2+4\,x+3\,x^2 \\ & gcd(a,b) \quad \text{mod} \quad 19 && = \quad 3+6\,x+2\,x^2 \\ \Longrightarrow\ & gcd(a,b) \quad \text{mod} \quad 17\times 19 && = \quad 155+310\,x+173\,x^2 \end{aligned}$$

  - in practice this is ∞ times faster than direct computation
- **The same for linear systems**:
  - solving $Ax = B$:

$$\begin{aligned} & Ax = B \quad \text{mod} \quad p_1 \\ & Ax = B \quad \text{mod} \quad p_2 \\ \Longrightarrow\ & Ax = B \quad \text{mod} \quad p_1 \times p_2 \times \dots \end{aligned}$$

Ring Homomorphism: *modular methods*

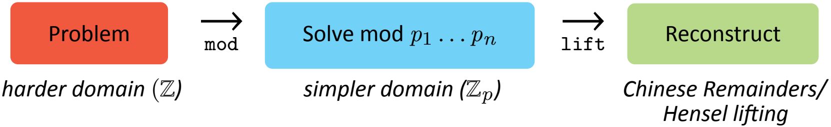
- **Idea**:
  - compute GCD modulo several different 32-bit primes, than ”reconstruct” result

$$
\begin{array}{llll}
 & gcd(a, b) \quad \text{mod} \quad 17 & = & 2 + 4\,x + 3\,x^2 \\
 & gcd(a, b) \quad \text{mod} \quad 19 & = & 3 + 6\,x + 2\,x^2 \\
\Longrightarrow & gcd(a, b) \quad \text{mod} \quad 17 \times 19 & = & 155 + 310\,x + 173\,x^2
\end{array}
$$

  - in practice this is ∞ times faster than direct computation
- **The same for linear systems**:
  - solving $Ax = B$:

$$
\begin{array}{ll}
 & Ax = B \quad \text{mod} \quad p_1 \\
 & Ax = B \quad \text{mod} \quad p_2 \\
\Longrightarrow & Ax = B \quad \text{mod} \quad p_1 \times p_2 \times \ldots
\end{array}
$$

- **The same everywhere**: factorization, resultant theory, Gröbner bases etc.



Problem $\xrightarrow{\texttt{mod}}$ Solve mod $p_1 \ldots p_n$ $\xrightarrow{\texttt{lift}}$ Reconstruct

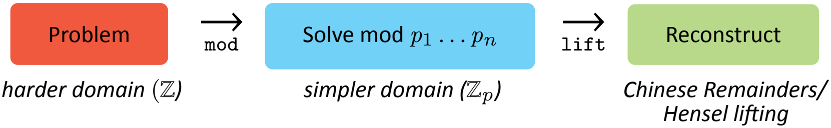
harder domain ($\mathbb{Z}$)

simpler domain ($\mathbb{Z}_p$)

Chinese Remainders/ Hensel lifting

RING HOMOMORPHISM: *ideal-adic methods*

- **Problems with integer (rational) coefficients:**



Problem $\xrightarrow[\texttt{mod}]{}$ Solve mod $p_1 \ldots p_n$ $\xrightarrow[\texttt{lift}]{}$ Reconstruct

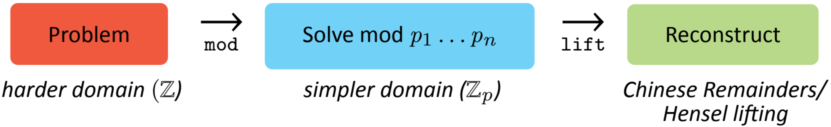
harder domain $(\mathbb{Z})$

simpler domain $(\mathbb{Z}_p)$

Chinese Remainders/
Hensel lifting

Ring Homomorphism: *ideal-adic methods*

- **Problems with integer (rational) coefficients:**



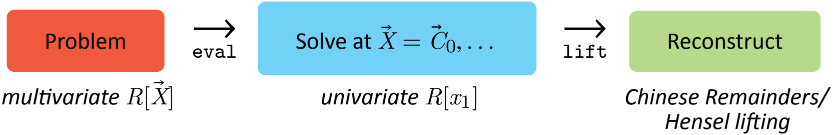
Problem $\xrightarrow{\texttt{mod}}$ Solve mod $p_1 \ldots p_n$ $\xrightarrow{\texttt{lift}}$ Reconstruct

harder domain $(\mathbb{Z})$

simpler domain $(\mathbb{Z}_p)$

Chinese Remainders/ Hensel lifting

- **Problems with multivariate polynomials:**



Problem $\xrightarrow{\texttt{eval}}$ Solve at $\vec{X} = \vec{C}_0, \ldots$ $\xrightarrow{\texttt{lift}}$ Reconstruct

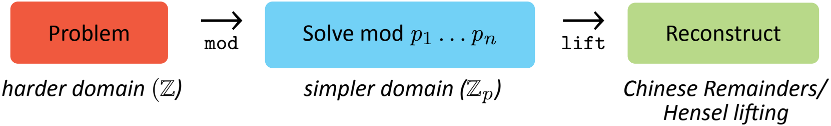
multivariate $R[\vec{X}]$

univariate $R[x_1]$

Chinese Remainders/ Hensel lifting

RING HOMOMORPHISM: *ideal-adic methods*

- **Problems with integer (rational) coefficients:**



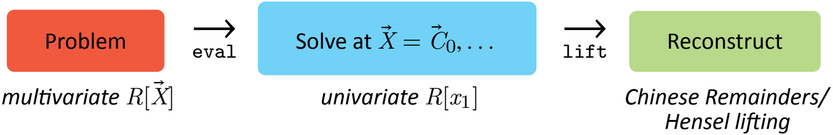
Problem $\xrightarrow[\texttt{mod}]{}$ Solve mod $p_1 \dots p_n$ $\xrightarrow[\texttt{lift}]{}$ Reconstruct

harder domain ($\mathbb{Z}$)

simpler domain ($\mathbb{Z}_p$)

Chinese Remainders/ Hensel lifting

- **Problems with multivariate polynomials:**



Problem $\xrightarrow[\texttt{eval}]{}$ Solve at $\vec{X} = \vec{C}_0, \dots$ $\xrightarrow[\texttt{lift}]{}$ Reconstruct

multivariate $R[\vec{X}]$

univariate $R[x_1]$

Chinese Remainders/ Hensel lifting

- **Example:** $\texttt{gcd}(x^3 - y^3, x^4 - y^4)$
  assume:

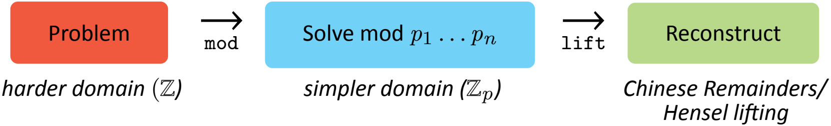
$$\texttt{gcd}(f(x,y), g(x,y)) = x^0\,(a_0 + \dots + a_3 y^3) + x^1\,(b_0 + \dots + b_3 y^3) + \dots$$

evaluate:

$$\left.\begin{array}{lll} y = 1: & \texttt{gcd}(f(x,1), g(x,1)) = & x - 1 \\ y = 2: & \texttt{gcd}(f(x,2), g(x,2)) = & x - 2 \\ & \dots & \end{array}\right\} \Longrightarrow a_0 = 0, a_1 = 1, \dots$$

Ring Homomorphism: *ideal-adic methods*

- **Problems with integer (rational) coefficients:**



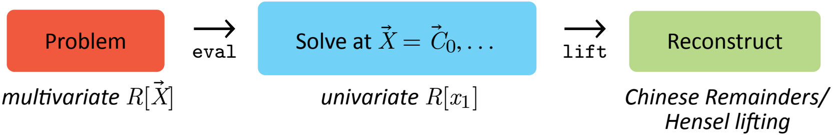
Problem $\xrightarrow{\text{mod}}$ Solve mod $p_1 \dots p_n$ $\xrightarrow{\text{lift}}$ Reconstruct

harder domain $(\mathbb{Z})$

simpler domain $(\mathbb{Z}_p)$

Chinese Remainders/ Hensel lifting

- **Problems with multivariate polynomials:**



Problem $\xrightarrow{\text{eval}}$ Solve at $\vec{X} = \vec{C}_0, \dots$ $\xrightarrow{\text{lift}}$ Reconstruct

multivariate $R[\vec{X}]$

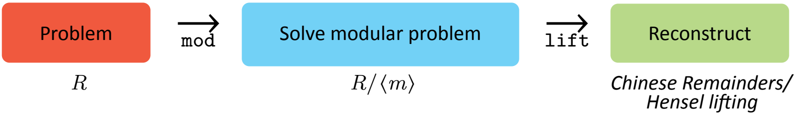
univariate $R[x_1]$

Chinese Remainders/ Hensel lifting

The math is the same: $R \to R/\langle m \rangle \to R$

Homomorphism	$\mathbb{Z} \to \mathbb{Z}_p$	$R[\vec{X}] \to R[x_1]$
Generator	prime number p	prime ideal $I = \langle x_2 - c_2, \dots \rangle$
Image function	$x \mod p$	$f(x) \mod I = f(x_1, c_2, \dots)$
Reconstruction	Chinese Remainders	Newton's formula

RING HOMOMORPHISM: *generic view*

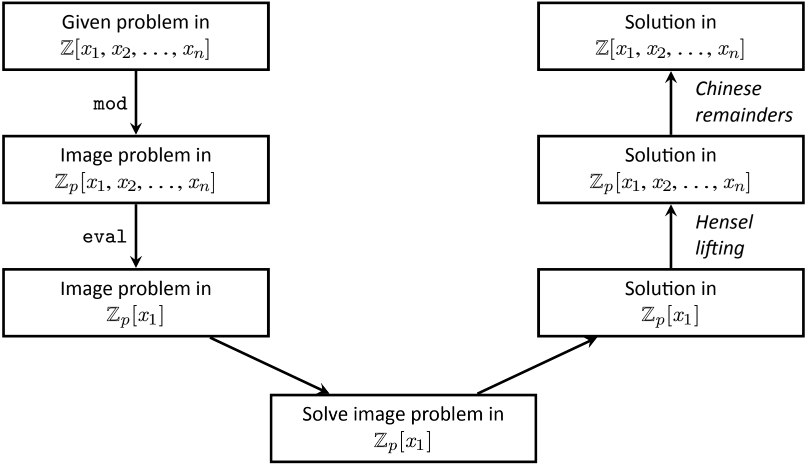


Problem $\xrightarrow{\texttt{mod}}$ Solve modular problem $\xrightarrow{\texttt{lift}}$ Reconstruct

R

$R/\langle m \rangle$

Chinese Remainders/ Hensel lifting



Given problem in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$

`mod`

Image problem in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$

`eval`

Image problem in $\mathbb{Z}_p[x_1]$

Solve image problem in $\mathbb{Z}_p[x_1]$

Solution in $\mathbb{Z}_p[x_1]$

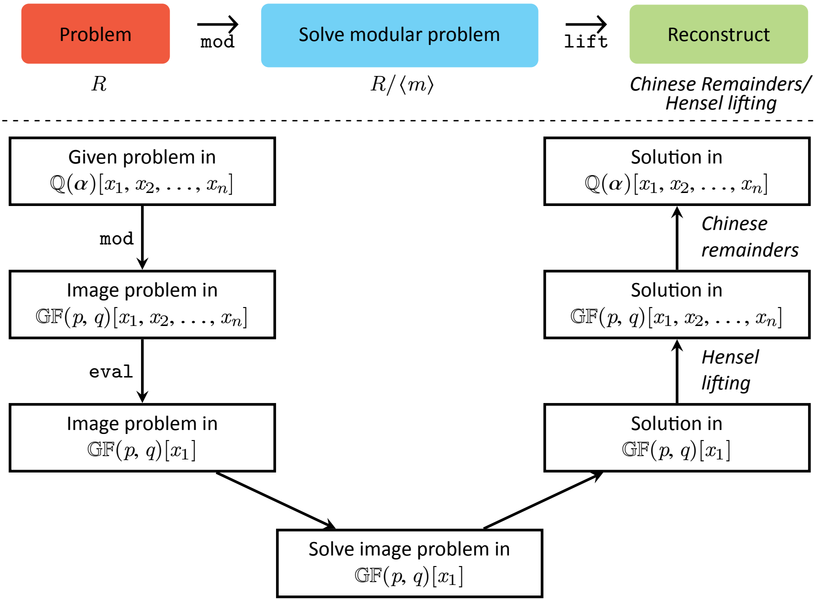
Hensel lifting

Solution in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$

Chinese remainders

Solution in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$

RING HOMOMORPHISM: *generic view*



Problem $\xrightarrow{\texttt{mod}}$ Solve modular problem $\xrightarrow{\texttt{lift}}$ Reconstruct

R

$R/\langle m \rangle$

Chinese Remainders/
Hensel lifting

Given problem in $\mathbb{Q}(\alpha)[x_1, x_2, \ldots, x_n]$

`mod`

Image problem in $\mathbb{GF}(p, q)[x_1, x_2, \ldots, x_n]$

`eval`

Image problem in $\mathbb{GF}(p, q)[x_1]$

Solve image problem in $\mathbb{GF}(p, q)[x_1]$

Solution in $\mathbb{GF}(p, q)[x_1]$

Hensel lifting

Solution in $\mathbb{GF}(p, q)[x_1, x_2, \ldots, x_n]$

Chinese remainders

Solution in $\mathbb{Q}(\alpha)[x_1, x_2, \ldots, x_n]$

SOME PROGRAMMING ASPECTS

PROGRAMMING ASPECTS: *general design*

- Algebraic concepts are perfect for translating into computer with object oriented programming
- But that’s not easy, only few libraries have e.g. strong typing
- Thanks to Java’s (and Scala’s) perfect OOP model, it became possible in `Rings`

Generic Euclidean algorithm:

```
def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
    if (b == 0) a else gcd(b, a % b)
```

Programming aspects: *general design*

- Algebraic concepts are perfect for translating into computer with object oriented programming
- But that’s not easy, only few libraries have e.g. strong typing
- Thanks to Java’s (and Scala’s) perfect OOP model, it became possible in `Rings`

Generic Euclidean algorithm:

```
def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
    if (b == 0) a else gcd(b, a % b)
```

Apply it to polynomials from $\mathbb{Q}[x]$:

```
implicit val ring1 = UnivariateRing(Q, "x")
val p1 = gcd(ring1("x^20 - 1"), ring1("x^30 - 1"))
// val p1 : UnivariatePolynomial[Rational[IntZ]] = ...
```

PROGRAMMING ASPECTS: *general design*

- Algebraic concepts are perfect for translating into computer with object oriented programming
- But that's not easy, only few libraries have e.g. strong typing
- Thanks to Java's (and Scala's) perfect OOP model, it became possible in Rings

Generic Euclidean algorithm:

```
def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
    if (b == 0) a else gcd(b, a % b)
```

Apply it to polynomials from $\mathbb{Q}[x]$:

```
implicit val ring1 = UnivariateRing(Q, "x")
val p1 = gcd(ring1("x^20 - 1"), ring1("x^30 - 1"))
// val p1 : UnivariatePolynomial[Rational[IntZ]] = ...
```

Apply it to polynomials from $\mathbb{Q}(\pm\sqrt{2})[x]$:

```
implicit zRing = Z
val num = gcd(zRing("213794398743"), zRing("34345"))
// val num : Int~
```

PROGRAMMING ASPECTS: *modular arithmetic & CPU*

— `mod` *is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more*

Real CPU: $N \mod p \equiv N - \lfloor N/p \rfloor \times p$ — one `DIV`, one `MUL` and one `SR`

- `DIV` has 20-80 times worth throughput than `MUL` (Intel Skylake)

PROGRAMMING ASPECTS: *modular arithmetic & CPU*

— `mod` *is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more*

Real CPU: $N \mod p \equiv N - \lfloor N/p \rfloor \times p$ — one `DIV`, one `MUL` and one `SR`

- `DIV` has 20-80 times worth throughput than `MUL` (Intel Skylake)

- **Old hack: use floats!**
  - Compute *once* the 64-bit *float* $magic = 1.0/p$
  - Then $\lfloor N/p \rfloor = [N \times magic]$ which is one float `MUL`
  - In practice **1.5-2 times speed up** (Skylake)
  - It was used in many CASs (`NTL`, `Mathematica`, `Maple` etc.)

Programming aspects: *modular arithmetic & CPU*

— `mod` *is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more*

Real CPU: $N \mod p \equiv N - \lfloor N/p \rfloor \times p$ — one `DIV`, one `MUL` and one `SR`

- `DIV` has 20-80 times worth throughput than `MUL` (Intel Skylake)

- **Old hack: use floats!**
  - Compute *once* the 64-bit *float* $magic = 1.0/p$
  - Then $\lfloor N/p \rfloor = [N \times magic]$ which is one float `MUL`
  - In practice **1.5-2 times speed up** (Skylake)
  - It was used in many CASs (`NTL`, `Mathematica`, `Maple` etc.)

- **Current hack:**
  - 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`
  - Used in many compilers when divisor is known at compile time:
    - Granlund & Montgomery (1994) — GCC, Go, ...
    - Warren's Hacker's delight (2002) — JVM, LLVM, ...

PROGRAMMING ASPECTS: *modular arithmetic & CPU*

— `mod` *is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more*

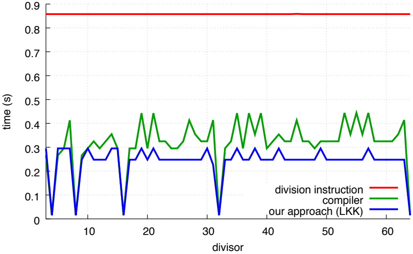
- **Fast modulo operation in `Rings` is approx 2 times faster than built-in %**
  - solving linear systems $O(n^3)$ — *8 times faster*
  - factoring polynomials $O(n^{1+\log_2 3})$ — *6 times faster*

Programming aspects: *modular arithmetic & CPU*

— `mod` *is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more*

- **Fast modulo operation in `Rings` is approx 2 times faster than built-in %**
  - solving linear systems $O(n^3)$ — *8 times faster*
  - factoring polynomials $O(n^{1+\log_2 3})$ — *6 times faster*

- **Can be even faster!**
  - *new algorithm to compute `MOD` with no `DIV` (Lemire, Kaser, Kurz, arXiv:1902.01961 [cs.MS] **Feb 2019**)*
  - *up to 25% speed up, really major achievement*



PROGRAMMING ASPECTS: *polynomial data structures*

- **Univariate polynomial:** $c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$

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

  `Array:`

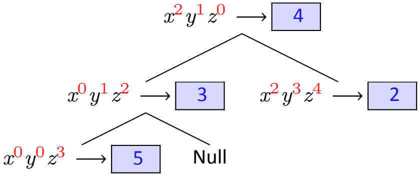
c_0	c_1	c_2	$\ldots$	c_n

  - Fast methods: *Karatsuba, FFT, Newton's iterations, etc.*

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

  $\downarrow\downarrow\downarrow$

  `Tree/Hash map:`



$x^2y^1z^0 \longrightarrow$ 4
$x^0y^1z^2 \longrightarrow$ 3
$x^2y^3z^4 \longrightarrow$ 2
$x^0y^0z^3 \longrightarrow$ 5
Null

`Sparse recursive:` $((5z^3) + (3z^2)\ y^1)\ x^0 + (4y^1 + (2z^4)\ y^3)\ x^2$

`Dense recursive:` $((0z^0 + 0z^1 + 0z^2 + 5z^3) + (0z^0 + 0z^1 + 3z^2)\ y^1)\ x^0 + \ldots$

Programming aspects: *polynomial data structures*

- *How the data structure affects the performance?*

Fateman's benchmark: multiply $f(f+1)$ with

$$f = (x+y+z+t+1)^{30}$$

(there will be 635,376 terms in the result...)

System/Library	Time, seconds	Comments
Rings (`hash map`)	15	- not used
Rings (`dense recursive`)	153	- used in Hensel lifting
Rings (`sparse recursive`)	365	- used for evaluation
Rings (`tree map`)	490*	- default
Maple 2018	27	- uses efficient tree map
Mathematica 11	171	-
Singular 4.1.1	198	- recursive
Magma V2.23	203	-
Sage 8.2	1075	- it's Python...

* 10s for multiply and 480s to rebalance the tree

https://ulthiel.com/math/other/benchmarks-of-computer-algebra-systems/

PROGRAMMING ASPECTS: *general notes*

Things which programmers pay attention, but scientists often do not:

- **Unit & integration tests:**
  - `Rings` covered with thousands of tests
  - Integration tests run external tools (e.g. SINGULAR CAS) to cross check the correctness
- **Randomized testing:**
  - It helped to fix *hundreds* of bugs
  - Several bugs in core routines were reported to MMA, SINGULAR, MAPLE etc.
- **Continuous integration (CI):**
  - `Rings` CI takes several hours to run all tests
  - Each new build may reveal new bugs (thanks to randomized tests!)

Benchmarks

Benchmarks

- **Polynomial GCD:**

  take random polynomials a, b, c and compute $\texttt{gcd}(ac, bc)$

- **Polynomial factorization:**

  take random polynomials a, b, c and compute $\texttt{factor}(abc)$ and $\texttt{factor}(abc + 1)$ (irreducibility test)

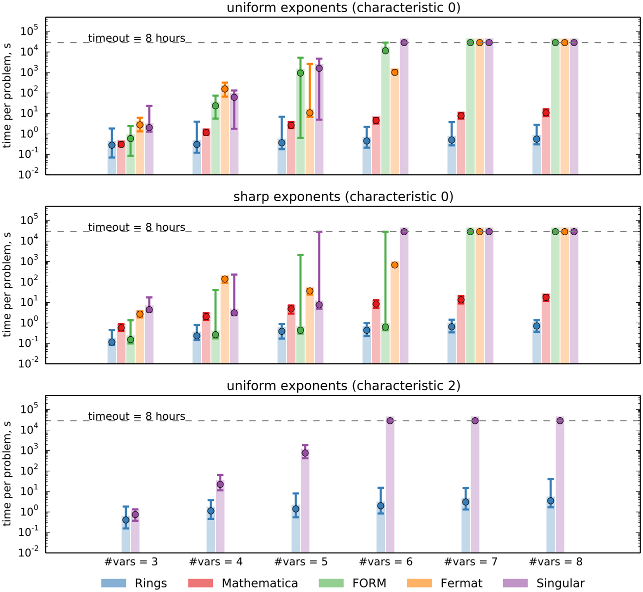
BENCHMARKS: *polynomial GCD*

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
$\text{exp}_{\text{min}} = 0$
$\text{exp}_{\text{max}} = 30$



BENCHMARKS: *polynomial factorization*

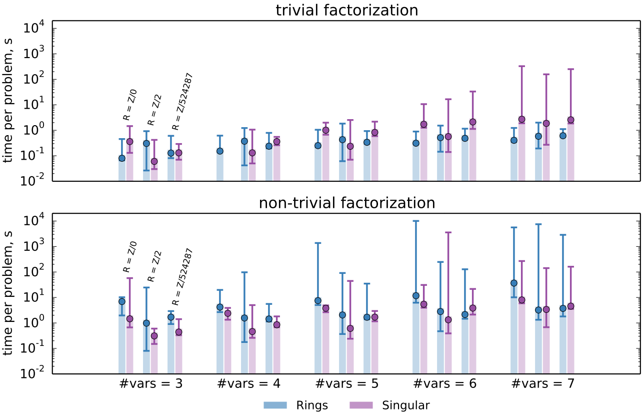
Params:

#factors = 3

#terms = 20

exp_{min} = 0

exp_{max} = 30



CONCLUSIONS

- MODERN HEP-TH REQUIRES HIGH PERFORMANCE COMPUTER ALGEBRA TOOLS
- FASTER ALGORITHMS AND MORE EFFICIENT IMPLEMENTATIONS APPEAR FROM TIME TO TIME
  - MORE DETAILS ON RINGS CAN BE FOUND AT
    - HTTPS://RINGSALGEBRA.IO
    - HTTPS://GITHUB.COM/POSLAVSKYSV/RINGS
- THE FURTHER SCALING MAY BE ACHIEVED BY USING DISTRIBUTED COMPUTING

THANKS FOR ATTENTION!

BACKUP

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*
  - *Ideals in multivariate polynomial rings*
- **Programming in Scala**
  - *object-oriented and functional programming in one concise, high-level and statically typed language*

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

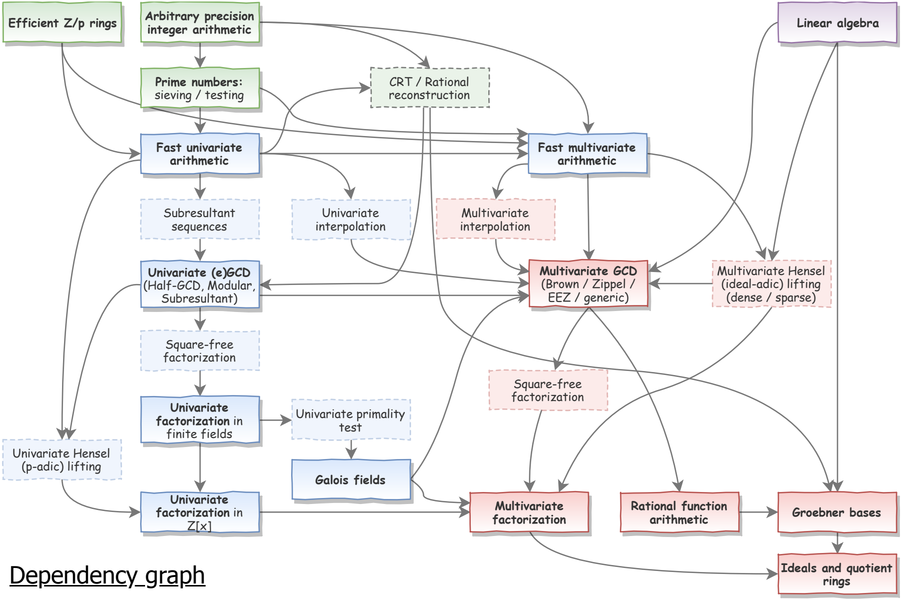
- `Ideal`: a subset of ring elements closed under multiplication with ring.

  Examples:

  - Given a set of generators $\{f_i(x, y, ...)\} \in R[x, y, ...]$ ideal is formed by all elements of the form

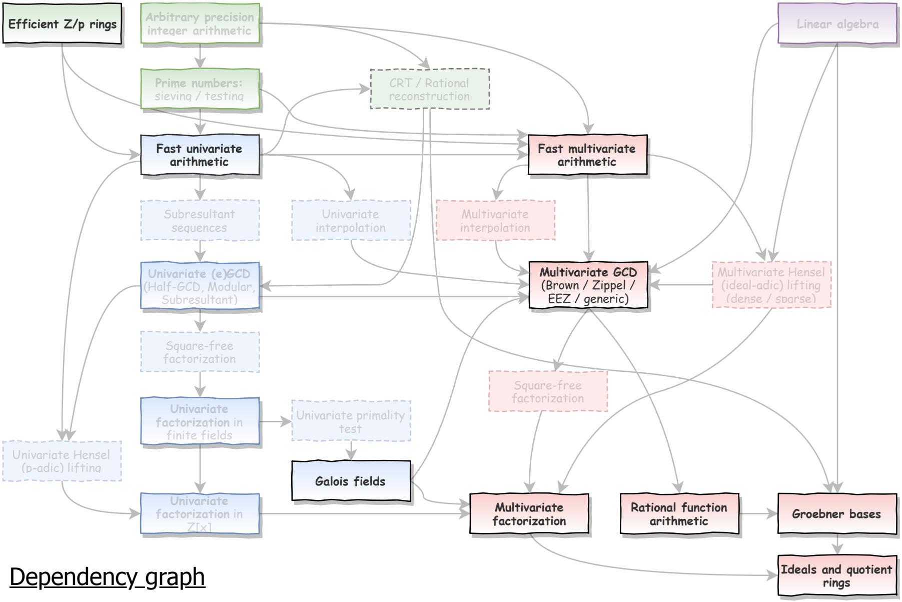
$$c_1(x, y, ...) \times f_1(x, y, ...) + ... + c_n(x, y, ...) \times f_n(x, y, ...)$$

Rings: *implementation aspects*


Efficient Z/p rings
Arbitrary precision integer arithmetic
Linear algebra
Prime numbers: sieving / testing
CRT / Rational reconstruction
Fast univariate arithmetic
Fast multivariate arithmetic
Subresultant sequences
Univariate interpolation
Multivariate interpolation
Univariate (e)GCD (Half-GCD, Modular, Subresultant)
Multivariate GCD (Brown / Zippel / EEZ / generic)
Multivariate Hensel (ideal-adic) lifting (dense / sparse)
Square-free factorization
Square-free factorization
Univariate factorization in finite fields
Univariate primality test
Univariate Hensel (p-adic) lifting
Galois fields
Univariate factorization in Z[x]
Multivariate factorization
Rational function arithmetic
Groebner bases
Ideals and quotient rings

Dependency graph

Rings: *implementation aspects*


Efficient Z/p rings
Arbitrary precision integer arithmetic
Linear algebra
Prime numbers: sieving / testing
CRT / Rational reconstruction
Fast univariate arithmetic
Fast multivariate arithmetic
Subresultant sequences
Univariate interpolation
Multivariate interpolation
Univariate (e)GCD (Half-GCD, Modular, Subresultant)
Multivariate GCD (Brown / Zippel / EEZ / generic)
Multivariate Hensel (ideal-adic) lifting (dense / sparse)
Square-free factorization
Square-free factorization
Univariate factorization in finite fields
Univariate primality test
Univariate Hensel (p-adic) lifting
Galois fields
Univariate factorization in Z[x]
Multivariate factorization
Rational function arithmetic
Groebner bases
Ideals and quotient rings

Dependency graph

Rings: *design by examples*

Simple example:

```scala
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:

```scala
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

  ```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
  ```

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

```
 // base ring Q[x, y, z]
implicit val ring = MultivariateRing(Q, Array("x", "y", "z"))
val (x, y, z) = ring("x", "y", "z")   // parse polynomials from strings

val poly1 = (x + y + z).pow(10) - 1   // construct poly
val poly2 = ring("(x + y + z)^3 + 1") // or just parse from string

println( PolynomialGCD(poly1, poly2) ) // compute GCD
println( Factor(poly1) )               // factorize polynomial

// construct some non-trivial polynomial ideal
implicit val ideal = Ideal(Seq(poly1 - x, poly2 - y), LEX)
assert ( ideal.dimension == 1 )

// reduce poly modulo ideal
assert ( poly1 %% ideal == x )
assert ( poly2 %% ideal == y )
```

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 cancelled automatically;

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: *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:

```scala
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:

```scala
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))
  }
```

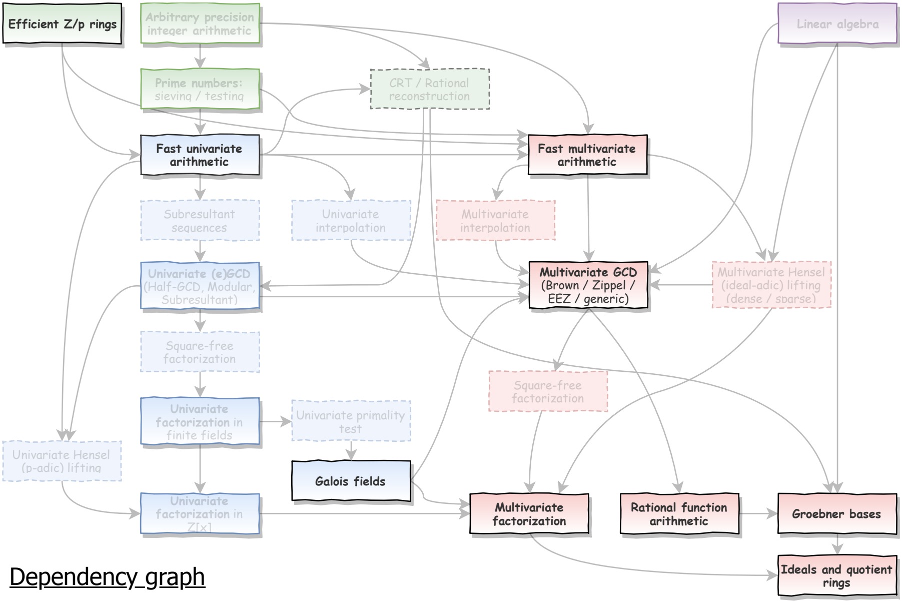
Diophantine equations in $Frac(GF(17^3)[x, y, z])[W]$:

```scala
// 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: *implementation aspects*


Efficient Z/p rings
Arbitrary precision integer arithmetic
Linear algebra
Prime numbers: sieving / testing
CRT / Rational reconstruction
Fast univariate arithmetic
Fast multivariate arithmetic
Subresultant sequences
Univariate interpolation
Multivariate interpolation
Univariate (e)GCD (Half-GCD, Modular, Subresultant)
Multivariate GCD (Brown / Zippel / EEZ / generic)
Multivariate Hensel (ideal-adic) lifting (dense / sparse)
Square-free factorization
Square-free factorization
Univariate factorization in finite fields
Univariate primality test
Univariate Hensel (p-adic) lifting
Galois fields
Univariate factorization in Z[x]
Multivariate factorization
Rational function arithmetic
Groebner bases
Ideals and quotient rings

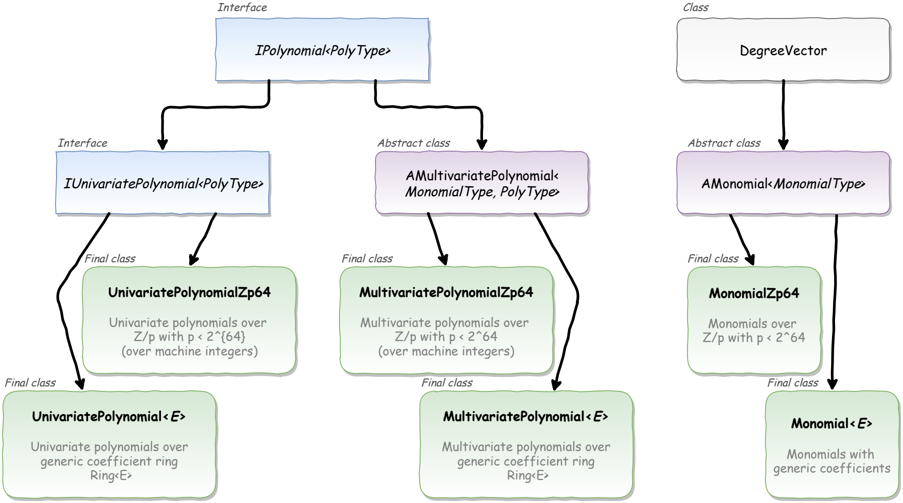
Dependency graph

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>

Class

DegreeVector

Interface

IUnivariatePolynomial<PolyType>

Abstract class

AMultivariatePolynomial<*MonomialType, PolyType*>

Abstract class

AMonomial<*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

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 6,000 l.o.c.)*

Rings: *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 ($\text{exp}_{\text{min}} = 0$, $\text{exp}_{\text{max}} = 10$):

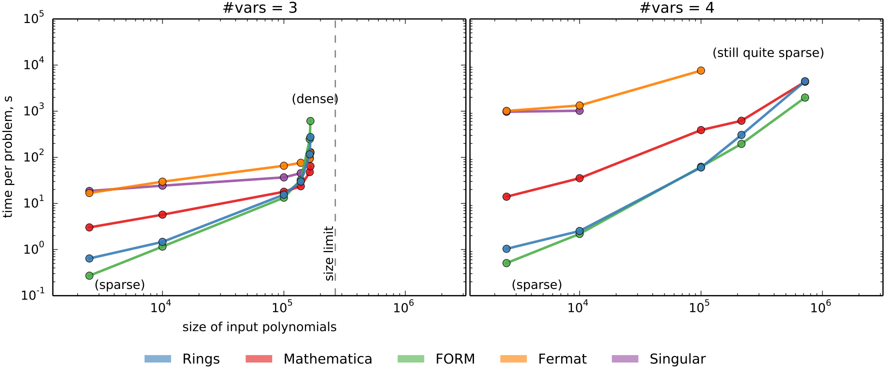
$$\cdots + x^5 y^2 z^8 + x^3 y^8 z^6 + \ldots$$

  - *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 ($\text{exp}_{\text{tot}} = 10$):

$$\cdots + x^7 y^2 z^1 + x^0 y^8 z^2 + \ldots$$

Rings: *polynomial GCD*

“Record” problems:



Params (a,b,g):

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

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:

$$
\begin{aligned}
a &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1 \\
b &= (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1 \\
g &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3
\end{aligned}
$$

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*

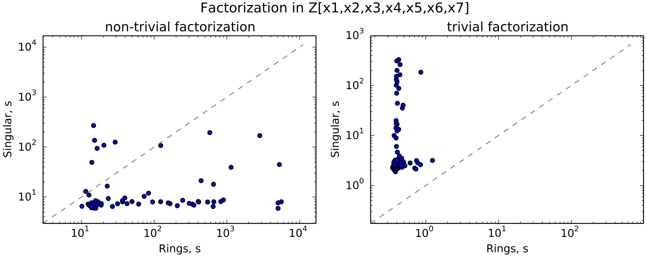
- **Univariate factorization:**
  - Rings switches between Cantor-Zassenhaus and Shoup's baby-step-giant-step algorithms for polynomials over finite fields
  - p-adic Hensel lifting is used to compute factorization over $\mathbb{Z}$ (resp. $\mathbb{Q}$)
- **Multvariate factorization:**
  - for bivariate 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*

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
```
$\mathtt{exp_{min}} = 0$
$\mathtt{exp_{max}} = 30$



Rings: *polynomial factorization*

Dense input:

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

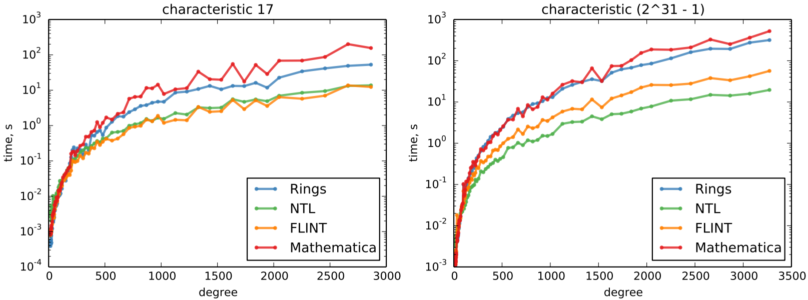
$$\begin{aligned} p_2 = -1 &+ (1 + 3x_1x_2 + 5x_2x_3 + 7x_3x_4 + 9x_4x_5 + 11x_5x_6 + 13x_6x_7 + 15x_7x_1)^3 \\ &\times(1 + 3x_1x_3 + 5x_2x_4 + 7x_3x_5 + 9x_6x_5 + 11x_7x_6 + 13x_6x_1 + 15x_7x_2)^3 \\ &\times(1 + 3x_1x_4 + 5x_2x_5 + 7x_3x_6 + 9x_6x_7 + 11x_7x_1 + 13x_6x_2 + 15x_7x_3)^3 \end{aligned}$$

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

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: *Gröbner bases*

- **Note**: Rings is not optimized for computing Gröbner bases for “challenging” problems yet (like those arise in post-quantum cryptography)
- Gröbner bases for graded orders for polynomials over finite fields computed with Faugere’s F4 algorithm (hardly based on fast sparse linear algebra)
- In other cases Rings may switch between Buchberger algorithm (with different selection strategies), Hilbert-driven methods or modular algorithms
- Again, many heuristics applied

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: *note on the programming languages*

- The choice of programming language is not so important as e.g. the choice of algorithms and careful design of the API
- **Rings is written in Java and also provides extensive Scala API**
  - Java: *just the most popular language*
    - extremely fast, very simple, cross-platform, has the largest community, comes with a dependency manager
    - with the same simplicity can be executed on PC, cluster or a wash machine
  - Scala: *object-oriented and functional programming in one concise, high-level and statically typed language*
    - has many recent developments from the theory of programming languages
    - very flexible and expressive: allows to write code very fast
    - also popular: e.g. Twitter and Spark are written in Scala

Rings: *note on the programming languages*

- The choice of programming language is not so important as e.g. the choice of algorithms and careful design of the API
- **Rings is written in Java and also provides extensive Scala API**
  - Java: *just the most popular language*
    - extremely fast, very simple, cross-platform, has the largest community, comes with a dependency manager
    - with the same simplicity can be executed on PC, cluster or a wash machine
  - Scala: *object-oriented and functional programming in one concise, high-level and statically typed language*
    - has many recent developments from the theory of programming languages
    - very flexible and expressive: allows to write code very fast
    - also popular: e.g. Twitter and Spark are written in Scala

---

- If you need to compute something quickly, you will find that it is easy
- If you need to program something, you will find that it is convenient

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:**

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

$$-\frac{1}{8m_1m_2m_3(m_1+m_2+m_3)}\frac{1}{(-m_3-t+s)(t-m_2)}$$

$$-\frac{1}{8m_1m_2m_3(m_1+m_2+m_3)}\frac{1}{(-m_3-t+s)(s+m_1)}$$

$+\ldots$ **(+22 other terms)**

Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```
// 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: *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)
```