Multivariate
Greatest Common Divisors
in the
Java Computer Algebra System

Heinz Kredel

ADG 2008, Shanghai

Automated Deduction in Geometry

- often leads to algebraic subproblems
- in polynomial rings over some coefficient ring
- resultants, greatest common divisors
- Gröbner Bases
  - ideal sum – intersection of geometric varieties
  - ideal intersection – union of geometric varieties
  - Comprehensive Gröbner Bases for parametric problems
- implementations by object-oriented software

Overview

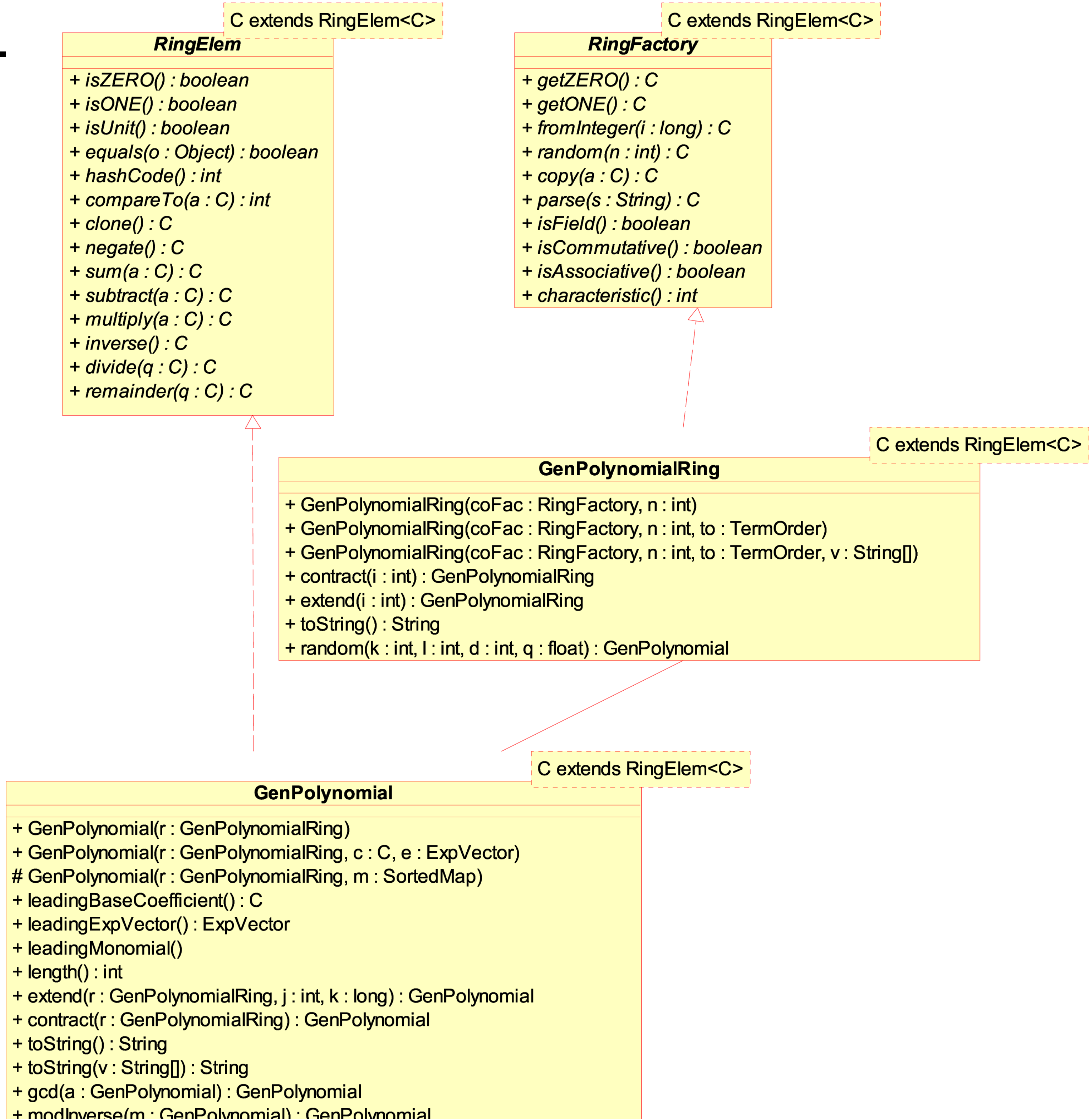
- Introduction to JAS
  - polynomial rings and polynomials
  - example with regular ring coefficients
- Greatest Common Divisors (GCD)
  - class layout
  - implementations
  - performance
- Evaluation
- Conclusions

Java Algebra System (JAS)

- object oriented design of a computer algebra system
  = software collection for symbolic (non-numeric) computations
- type safe through Java generic types
- thread safe, ready for multi-core CPUs
- use dynamic memory system with GC
- 64-bit ready
- jython (Java Python) interactive scripting front end

Implementation overview

- 170+ classes and interfaces
- plus 80+ JUnit test cases
- uses JDK 1.6 with generic types
    - Javadoc API documentation
    - logging with Apache Log4j
    - build tool is Apache Ant
    - revision control with Subversion
- jython (Java Python) scripts
    - support for Sage like polynomial expressions
- open source, license is GPL or LGPL


C extends RingElem<C>
RingElem
+ isZERO() : boolean
+ isONE() : boolean
+ isUnit() : boolean
+ equals(o : Object) : boolean
+ hashCode() : int
+ compareTo(a : C) : int
+ clone() : C
+ negate() : C
+ sum(a : C) : C
+ subtract(a : C) : C
+ multiply(a : C) : C
+ inverse() : C
+ divide(q : C) : C
+ remainder(q : C) : C
C extends RingElem<C>
RingFactory
+ getZERO() : C
+ getONE() : C
+ fromInteger(i : long) : C
+ random(n : int) : C
+ copy(a : C) : C
+ parse(s : String) : C
+ isField() : boolean
+ isCommutative() : boolean
+ isAssociative() : boolean
+ characteristic() : int
C extends RingElem<C>
GenPolynomialRing
+ GenPolynomialRing(coFac : RingFactory, n : int)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder, v : String[])
+ contract(i : int) : GenPolynomialRing
+ extend(i : int) : GenPolynomialRing
+ toString() : String
+ random(k : int, l : int, d : int, q : float) : GenPolynomial
C extends RingElem<C>
GenPolynomial
+ GenPolynomial(r : GenPolynomialRing)
+ GenPolynomial(r : GenPolynomialRing, c : C, e : ExpVector)
# GenPolynomial(r : GenPolynomialRing, m : SortedMap)
+ leadingBaseCoefficient() : C
+ leadingExpVector() : ExpVector
+ leadingMonomial()
+ length() : int
+ extend(r : GenPolynomialRing, j : int, k : long) : GenPolynomial
+ contract(r : GenPolynomialRing) : GenPolynomial
+ toString() : String
+ toString(v : String[]) : String
+ gcd(a : GenPolynomial) : GenPolynomial
+ modInverse(m : GenPolynomial) : GenPolynomial

Polynomials over regular rings

example:
`List<GenPolynomial<Product<Residue<BigRational>>>>`

$$R = \mathbb{Q}[x_1, \ldots, x_n]$$

$$S' = \left(\prod_{\wp \in spec(R)} R/\wp\right)[y_1, \ldots, y_r]$$ a von Neuman regular ring

$$L \subset S = (\underbrace{\mathbb{Q}[x_0, x_1, x_2]/ideal(F)})^4[a, b]$$

```
rr = ResidueRing[ BigRational( x0, x1, x2 ) IGRLEX
     ( ( x0^2 + 295/336  ),
     ( x2 - 350/1593 x1 - 1100/2301 ) ) ]
L = [
     {0=x1 - 280/93 , 2=x0 * x1 - 33/23 } a^2 * b^3
   + {0=122500/2537649 x1^3 + 770000/3665493 x1^2
       + 14460385/47651409 x1 + 14630/89739 ,
      3=350/1593 x1 + 23/6 x0 + 1100/2301 } ,
    ... ]
```

Regular ring construction

```
 1 List<GenPolynomial<Product<Residue<BigRational>>>> L
    = new ArrayList<GenPolynomial<Product<Residue<BigRational>>>>();

 2 BigRational bf = new BigRational(1);
 3 GenPolynomialRing<BigRational> pfac
    = new GenPolynomialRing<BigRational>(bf,3);
 4 List<GenPolynomial<BigRational>> F
    = new ArrayList<GenPolynomial<BigRational>>();
 5 GenPolynomial<BigRational> pp = null;
 6 for ( int i = 0; i < 2; i++) {
 7     pp = pfac.random(5,4,3,0.4f);
 8     F.add(pp);
 9 }
10 Ideal<BigRational> id = new Ideal<BigRational>(pfac,F);
11 id.doGB();
12 ResidueRing<BigRational> rr = new ResidueRing<BigRational>(id);
13 System.out.println("rr = " + rr);
14 ProductRing<Residue<BigRational>> pr
    = new ProductRing<Residue<BigRational>>(rr,4);
```

Polynomial construction and GB

```
List<GenPolynomial<Product<Residue<BigRational>>>> L = ...

String[] vars = new String[] { "a", "b" };
GenPolynomialRing<Product<Residue<BigRational>>> fac
  = new GenPolynomialRing<Product<Residue<BigRational>>>(pr,2,vars)
GenPolynomial<Product<Residue<BigRational>>> p;
for ( int i = 0; i < 3; i++) {
    p = fac.random(2,4,4,0.4f);
    L.add(p);
}
System.out.println("L = " + L);

GroebnerBase<Product<Residue<BigRational>>> bb
  = new RGroebnerBasePseudoSeq<Product<Residue<BigRational>>>(pr);

List<GenPolynomial<Product<Residue<BigRational>>>> G = bb.GB(L);
System.out.println("G = " + G);
```

take primitive parts --> gcd

Overview

- Introduction to JAS
  - polynomial rings and polynomials
  - example with regular ring coefficients
- Greatest Common Divisors (GCD)
  - class layout
  - implementations
  - performance
- Evaluation
- Conclusions

Greatest common divisors

```
UFD euclidsGCD( UFD a, UFD b ) {
    while ( b != 0 ) {
          // let a = q b + r;                // remainder
          // let ldcf(b)^e a = q b + r; // pseudo remainder
          a = b;
          b = r; // simplify remainder
    }
    return a;
}
```

```
mPol gcd( mPol a, mPol b ) {
     a1 = content(a);     // gcd of coefficients
     b1 = content(b);     // or recursion
     c1 = gcd( a1, b1 ); // recursion
     a2 = a / a1;         // primitive part
     b2 = b / b1;
     c2 = euclidsGCD( a2, b2 );
     return c1 * c2;
}
```

GCD class layout

1. where to place the algorithms in the library ?
2. which interfaces to implement ?
3. which recursive polynomial methods to use ?

- place gcd in `GenPolynomial`
  - like Axiom
- ✓ place gcd in separate package `edu.jas.ufd`
  - like other libraries
  - gcd 3200 loc, polynomial 1200 loc

Interface `GcdRingElem`

- extend `RingElem` by defining `gcd()` and `egcd()`
- let `GenGcdPolynomial` extend `GenPolynomial`
  - not possible by type system
- let `GenPolynomial` implement `GcdRingElem`
  - must change nearly all classes (100+ restrictions)
- ✓ final solution
  - `RingElem` defines `gcd()` and `egcd()`
  - `GcdRingElem` (empty) marker interface
  - only 10 classes to change

Recursive methods

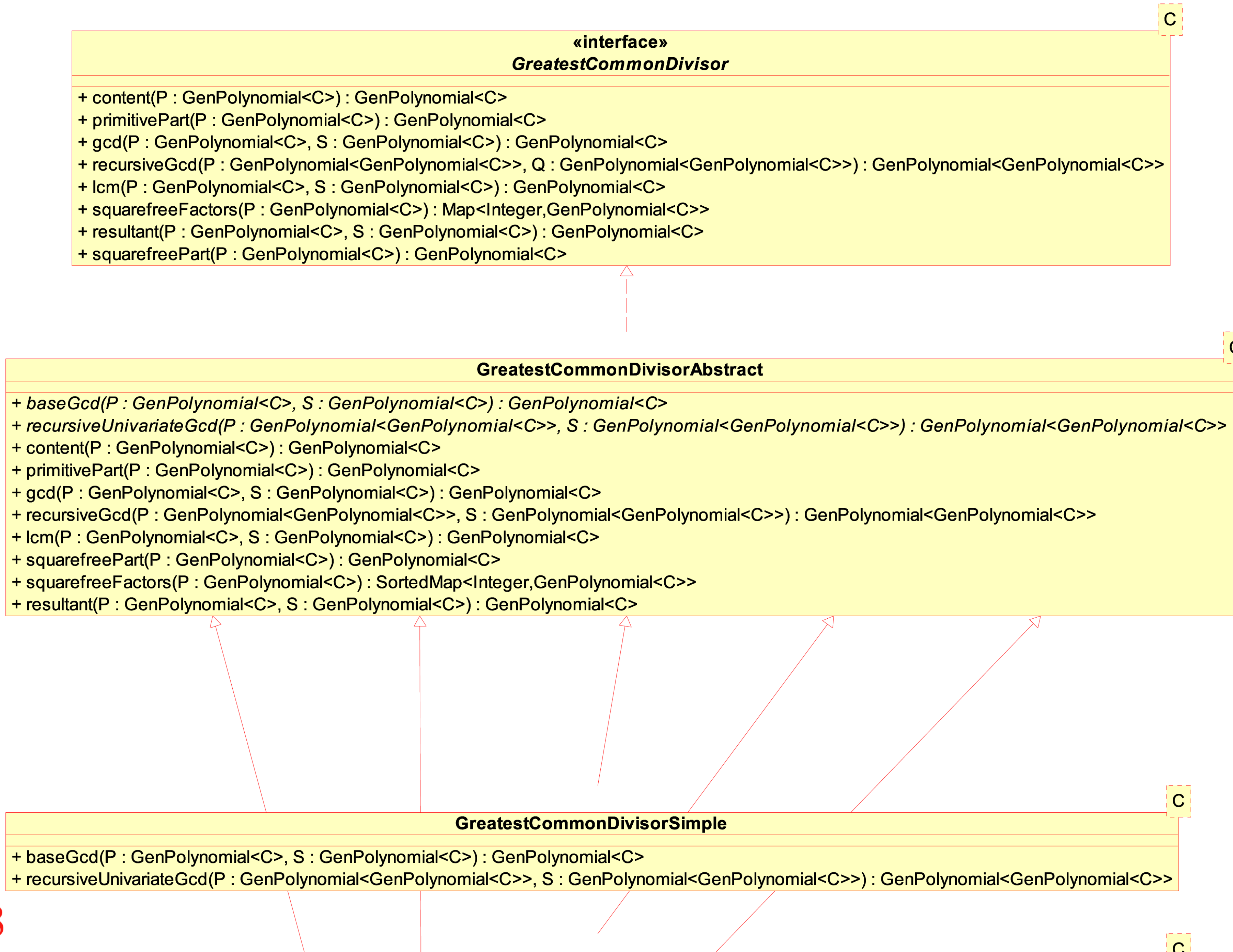
- recursive type `RingElem<C extends RingElem<C>>`
- so polynomials can have polynomials as coefficients
  - `GenPolynomial<GenPolynomial<BigRational>>`
- leads to code duplication due to type erasure
  - `GenPolynomial<C> gcd(GenPolynomial<C> P, S)`
  - `GenPolynomial<C> baseGcd(GenPolynomial<C> P,S)`
  - `GenPolynomial<GenPolynomial<C>> recursiveUnivariateGcd( GenPolynomial<GenPolynomial<C>> P, S )`
  - and also required `recursiveGcd(.,.)`

Conversion of representation

- static conversion methods in class `PolyUtil`
- convert to recursive representation
  - `GenPolynomial<GenPolynomial<C>> recursive( GenPolynomialRing<GenPolynomial<C>> rf, GenPolynomial<C> A )`
- convert to distributive representation
  - `GenPolynomial<C> distribute( GenPolynomialRing<C> dfac, GenPolynomial<GenPolynomial<C>> B)`
- must provide (and construct) result polynomial ring
- performance of many conversions ?

GCD implementations

- Polynomial remainder sequences (PRS)
  - primitive PRS
  - simple / monic PRS
  - sub-resultant PRS
- modular methods
  - modular coefficients, Chinese remaindering (CR)
  - recursion by modular evaluation and CR
  - modular coefficients, Hensel lifting wrt. p^e
  - recursion by modular evaluation and Hensel lifting


C
«interface»
GreatestCommonDivisor
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveGcd(P : GenPolynomial<GenPolynomial<C>>, Q : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : Map<Integer,GenPolynomial<C>>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
GreatestCommonDivisorAbstract
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<Integer,GenPolynomial<C>>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
C
GreatestCommonDivisorSimple
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
C


+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<Integer,GenPolynomial<C>>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
C
GreatestCommonDivisorSimple
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
C
GreatestCommonDivisorPrimitive
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
C
GreatestCommonDivisorSubres
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ baseResultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveResultant(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
GreatestCommonDivisorModular
+ gcd(P : GenPolynomial<BigInteger>, S : GenPolynomial<BigInteger>) : GenPolynomial<BigInteger>
GreatestCommonDivisorModEval
+ gcd(P : GenPolynomial<ModInteger>, S : GenPolynomial<ModInteger>) : GenPolynomial<ModInteger>

Polynomial remainder sequences

- Euclids algorithm applied to polynomials lead to
  - intermediate expression swell / explosion
  - result can be small nevertheless, e.g. one
- avoid this by simplifying the successive remainders
  - take primitive part: primitive PRS
  - divide by computed factor: sub-resultant PRS
  - make monic if field: monic PRS
- implementations work for all rings with a gcd
  - for example `Product<Residue<BigRational>>`

Modular CR method overview

1. Map the coefficients of the polynomials modulo some prime number p. If the mapping is not ‘good’, choose a new prime and continue with step 1.
2. Compute the gcd over the modulo p coefficient ring. If the gcd is 1, also the ‘real’ gcd is one, so return 1.
3. From gcds modulo different primes reconstruct an approximation of the gcd using Chinese remaindering. If the approximation is ‘correct’, then return it, otherwise, choose a new prime and continue with step 1.

Modular methods

- algorithm variants
  - modular on base coefficients with Chinese remainder reconstruction
    - monic PRS on multivariate polynomials
    - modulo prime polynomials to remove variables until univariate, polynomial version of Chinese remainder reconstruction
  - modular on base coefficients with Hensel lifting with respect to p^e

    future work
    - monic PRS on multivariate polynomials
    - modulo prime polynomials to remove variables until univariate, polynomial version of Hensel lifting

Performance: PRS - modular

a,b,c random polynomials

d=gcd(ac,bc)
c|d ?

degrees, e	s	p	sr	ms	me
a=7, b=6, c=2	23	23	36	1306	2176
a=5, b=5, c=2	12	19	13	36	457
a=3, b=6, c=2	1456	117	1299	1380	691
a=5, b=5, c=0	508	6	6	799	2

BigInteger coefficients, s = simple, p = primitive, sr = sub-resultant, ms = modular simple monic, me = modular evaluation.
`random()` parameters: r = 4, k = 7, l = 6, q = 0.3,

degrees, e	sr	ms	me
a=5, b=5, c=0	3	29	27
a=6, b=7, c=2	181	695	2845
a=5, b=5, c=0	235	86	4
a=7, b=5, c=2	1763	874	628
a=4, b=5, c=0	26	1322	12

BigInteger coefficients, sr = sub-resultant, ms = modular simple monic, me = modular evaluation.
`random()` parameters: r = 4, k = 7, l = 6, q = 0.3,

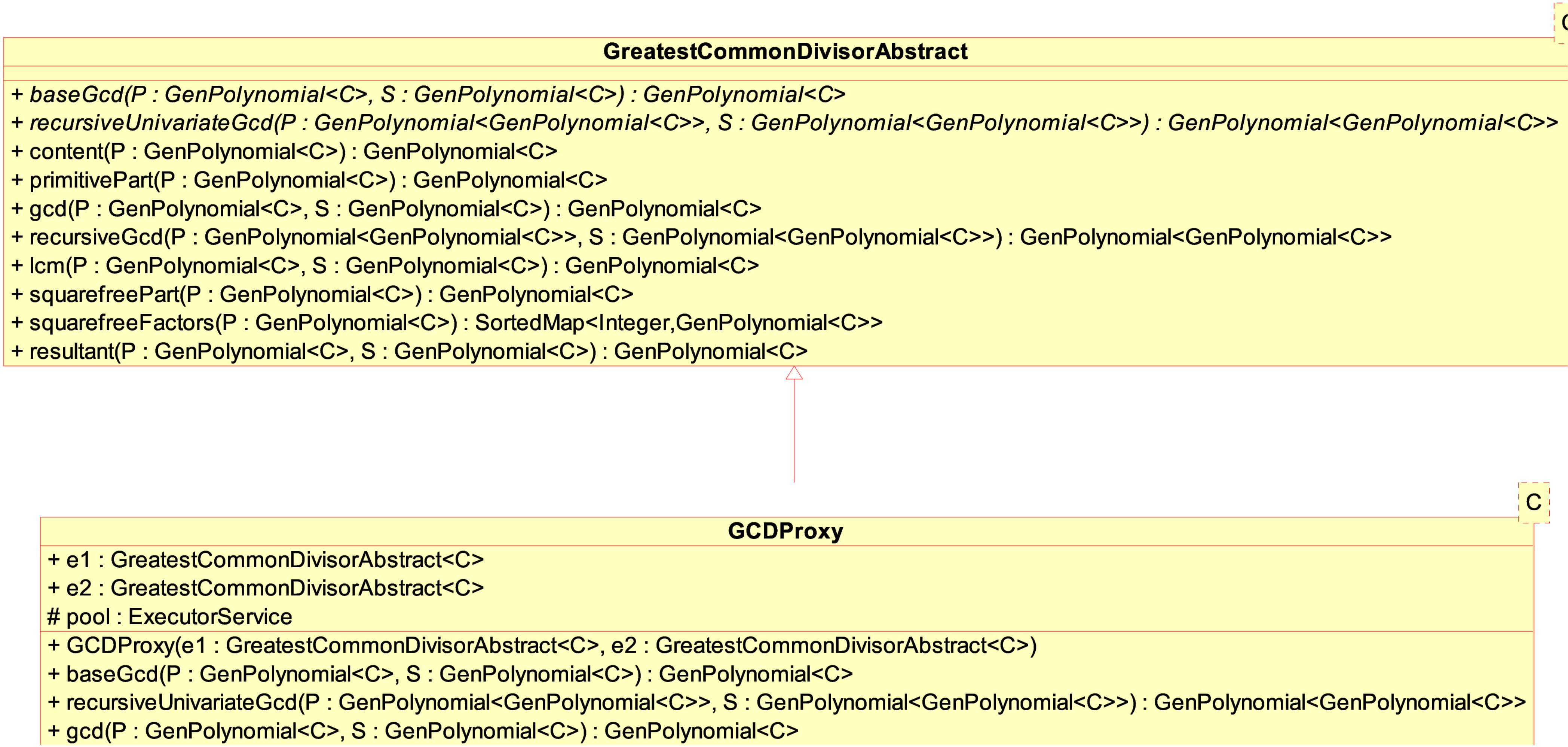
GCD factory

- all gcd variants have pros and cons
  - computing time differ in a wide range
  - coefficient rings require specific treatment
- solve by object-oriented factory design pattern: a factory class creates and provides a suitable implementation via different methods
  - `GreatestCommonDivisor<C> GCDFactory.<C>getImplementation( cfac );`
  - type `C` triggers selection at compile time
  - coefficient factory `cfac` triggers selection at runtime

GCD factory (cont.)

- four versions of `getImplementation()`
  - `BigInteger`, `ModInteger` and `BigRational`
  - and a version for undetermined type parameter
- last version tries to determine concrete coefficient at run-time
  - try to be as specific as possible for coefficients
- `ModInteger`:
  - if modulus is prime then optimize for field
  - otherwise use general version

GCD proxy (1)


GreatestCommonDivisorAbstract
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<Integer,GenPolynomial<C>>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
C
GCDProxy
+ e1 : GreatestCommonDivisorAbstract<C>
+ e2 : GreatestCommonDivisorAbstract<C>
# pool : ExecutorService
+ GCDProxy(e1 : GreatestCommonDivisorAbstract<C>, e2 : GreatestCommonDivisorAbstract<C>)
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ recursiveUnivariateGcd(P : GenPolynomial<GenPolynomial<C>>, S : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>

GCD proxy (2)

- different performance of algorithms
  - mostly modular methods are faster
  - but some times (sub-resultant) PRS faster
- hard to predict run-time of algorithm for given inputs
  - (worst case) complexity measured in:
    - the size of the coefficients,
    - the degrees of the polynomials, and
    - the number of variables,
    - the density or sparsity of polynomials,
    - and the density of the exponents

GCD proxy (3)

- improvement by speculative parallelism
- execute two (or more) algorithms in parallel
- most computers now have two or more CPUs
- use `java.uitl.concurrent.ExecutorService`
- provides method `invokeAny()`
  - executes several methods in parallel
  - when one finishes the others are interrupted
- interrupt checked in polynomial creation (only)
- `PreemptingException` exception aborts execution

GCD proxy (4)

```
final GreatestCommonDivisorAbstract<C> e1,e2;
protected ExecutorService pool;
          // set in constructor
List<Callable<GenPolynomial<C>>> cs = ...init..;
cs.add(
   new Callable<GenPolynomial<C>>() {
        public GenPolynomial<C> call() {
               return e1.gcd(P,S);
        }
   }
);
cs.add( ... e2.gcd(P,S); ... );
GenPolynomial<C> g = pool.invokeAny( cs );
```

in polynomial constructor:

```
if ( Thread.currentThread().isInterrupted() ) {
       throw new PreemptingException();
}
```

Parallelization

- thread safety from the beginning
  - explicit synchronization where required
  - immutable algebraic objects to avoid synchronization
- utility classes now from `java.util.concurrent`

Performance: proxy

degrees, e	time	algorithm
a=6, b=6, c=2	3566	subres
a=5, b=6, c=2	1794	modular
a=7, b=7, c=2	1205	subres
a=5, b=5, c=0	8	modular

BigInteger coefficients, winning algorithm: subres = sub-resultant, modular = modular simple monic.
`random()` parameters: r = 4, k = 24, l = 6, q = 0.3,

single CPU, 32-bit, 1.6 GHz

degrees, e	time	algorithm
a=6, b=6, c=2	3897	modeval
a=7, b=6, c=2	1739	modeval
a=5, b=4, c=0	905	subres
a=5, b=5, c=0	10	modeval

ModInteger coefficients, winning algorithm: subres = sub-resultant, modeval = modular evaluation.
`random()` parameters: r = 4, k = 6, l = 6, q = 0.3,

Application performance

- polynomial arithmetic performance
- gcd performance
- application performance: Gröbner bases
- computing time in milliseconds on
  - AMD 1.6 GHz 32-bit single CPU, Java 6 (JDK 1.6)
  - AMD Opteron 2.6 GHz 64-bit 16 CPUs, JDK 1.5
- differences for
  - client VM: fast to result
  - server VM: faster for long runs, just-in-time compiler
- different times after warm-up

Polynomial performance

- performance of coefficient arithmetic
    - `java.math.BigInteger` in pure Java
- sorted map implementation
    - from Java collection classes
- exponent vector implementation
    - using `long[]`, also `int[]`, `short[]` or `byte[]`
    - want `ExpVector<C>` but not with elementary types
    - can be selected at compile time
- JAS comparable to general purpose CA systems but slower than specialized systems

Performance: Gröbner base (1)

example	MAS	JAS, clientVM	JAS, serverVM
Raksanyi, G	50	311 (53)	479 (205)
Raksanyi, L	40	267 (52)	419 (198)
Hawes2, G	610	528 (237)	1106 (1351)
Hawes2, L	26030	9766 (8324)	11061 (5966)

time in milliseconds for Gröbner base examples, Term order: G = graded, L = lexicographical, timings in parenthesis are for second run.

example/algorithm	Subres	Modular
Hawes2, G	1215	105
Hawes2, L	4030	125

counts for winning algorithm

single CPU, 32-bit, 1.6 GHz

Performance: Gröbner base (2)

gcd algorithm	time	first	second
Subres and Modular $p < 2^{28}$	5799	3807	2054
Subres and Modular $p < 2^{59}$	10817	3682	2100
Modular $p < 2^{28}$ and Subres	5662	2423	3239
Subres	5973		
Modular $p < 2^{28}$ with ModEval	21932		
Modular $p < 2^{28}$ with Monic	27671		
Modular $p < 2^{59}$ with Monic	34732		
Modular $p < 2^{59}$ with ModEval	24495		

time in milliseconds for Hawes2 lex Gröbner base example, first, second = count for respecive algorithm, p shows the size of the used prime numbers.

16 CPUs, 64-bit, 2.6 GHz

Recursive polynomials

future work

- new type `RecPolynomial`
  - univariate polynomials with polynomial coefficients
  - must allow `RingElem` as (base) coefficients
  - must itself implement the `RingElem` interface
- mapping between terms and coefficients
  - no `ExpVector` class required
  - as Java array, dense representation
  - ✓ as `SortedMap, TreeMap` from `java.util`
  - can store polynomials and coefficients as `RingElem`

Recursive polynomials (cont.)

future work

- How to handle recursion base or recursion?
- case distinction on the number of variables `nvar`
- `nvar == 0`: obtain the polynomial as coefficient ?
  - better exclude this case
- `nvar == 1`: use collection of base coefficients
  - `Collection<C> A = val.getValues();`
- `nvar > 1`: use collection of polynomial coefficients
  - `Collection<C> A = val.getValues();`
  - `RecPolynomial<C> a = (RecPolynomial<C>) A.get(0);`

Overview

- Introduction to JAS
  - polynomial rings and polynomials
  - example with regular ring coefficients
- Greatest Common Divisors (GCD)
  - class layout
  - implementations
  - performance
- Evaluation
- Conclusions

Evaluation (1)

- Distributed representation with conversions to recursive representation on demand.
  - How expensive are the many conversions between distributed and recursive representation?
  - Manipulations of ring factories to setup the correct polynomial ring for recursions.
  - Compared to MAS (based on Aldes/SAC-2 with elaborated recursive polynomial representation) the conversions seem not to be too expensive.

Evaluation (2)

- `ModInteger` for polynomial coefficients is implemented using `BigInteger`.
  - Systems like Singular, MAS and Aldes/SAC-2, use `int`s for modular integer coefficients.
  - This can have great influence on the computing time.
  - However, JAS is with this choice able to perform computations with arbitrary long modular integers.
  - The right choice of prime size for modular integers is not yet determined.
  - We experimented with primes of size less than `Long.maxValue` and less than `Integer.maxValue`.

Evaluation (3)

- The bounds used to stop iteration over primes, are not yet state of the art.
  - We currently use the bounds found in [Aldes SAC-2]. The bounds derived in [Cohen] and [Geddes] are not yet incorporated.
  - However, we try to detect factors by exact division, early.
- The univariate polynomials and methods are not separate implementations tuned for this case.
  - We simply use the multivariate polynomials and methods with only one variable.

Evaluation (4)

- There are no methods for extended gcds and half extended gcds for multivariate polynomials yet.
  - Better algorithms for the gcd computation of lists of polynomials are not yet implemented.
- For generic parametric polynomials, such as `GenPolynomial<GenPolynomial<C>>` the `gcd()` method can not be used.
  - Since the coefficients are itself polynomials and do not implement a multivariate polynomial gcd.
  - In this case the method `recursiveGcd()` must be called explicitly.

Evaluation (5)

- Design of a interface `GreatestCommonDivisor` with only the most useful methods.
  - `gcd(), lcm(), primitivePart(), content(), squarefreePart(), squarefreeFactors(), resultant().`
- The generic algorithms work for all implemented coefficients from (commutative) fields.
- The implementations can be used in very general settings, as exemplified in the regular ring example.

Evaluation (6)

- The class `GreatestCommonDivisorAbstract` implements the full set of methods, as specified by the interface.
  - Only two methods `baseGcd` and `recursiveUnivariateGcd` must be implemented for the different PRS algorithms.
  - The modular algorithms overwrite the `gcd` method
- The abstract class should eventually be refactored to provide an abstract class for PRS algorithms and an abstract class for the modular algorithms.

Evaluation (7)

- The gcd factory allows non-experts of computer algebra to choose the right algorithm for their problem.
  - First selection by coefficient type at compile time and more precisely by the field property at run-time.
  - In case of `BigInteger` and `ModInteger` coefficients the modular algorithms are selected.
- Different approach taken in [Musser] and [Schupp] to provide programming language constructs to specify the requirements for the implementations.
  - Constructs direct the selection of algorithms, some at compile time and some at run-time.

Evaluation (8)

- Proxy class with gcd interface provides effective selection of the fastest algorithms at run-time.
  - Achieved at the cost of a parallel execution of two different gcd algorithms.
  - This could waste maximally the time for the computation of the fastest algorithm.
  - If two CPUs are working on the problem, the work of one of them is discarded.
  - In case there is only one CPU, the computing time is two times that of the fastest algorithm.

Conclusions (1)

- Design and implementation of a first part of ‘multiplicative ideal theory’: the computation of multivariate polynomial greatest common divisors.
    - Not yet covered is the complete factorization,
- GCDFactory for the selection of one of the several implementations for non experts.
    - Selection is based on the coefficient type and if the coefficient ring is a field.
- A parallel GCDProxy runs different implementations in parallel and takes the result from the first finishing method.

Conclusions (2)

- The new package is also type-safe designed with Java’s generic types.
- We exploited the gcd package in the `Quotient`, `Residue` and `Product` classes.
- Provides a new coefficient ring of rational functions for the polynomials and also new coefficient rings of residue class rings and product rings.
  - With an efficient gcd implementation we are now able to compute Gröbner bases over those coefficient rings.

Conclusions (3)

- For small Gröbner base computations the performance is equal to MAS and for bigger examples the computing time with JAS is better by a factor of two or three.
- Java improvements leverage the performance and capabilities of JAS.
- Future topics to explore, include the complete factorization of polynomials and the investigation of a new recursive polynomial representation.

Thank you

- Questions or Comments?
- http://krum.rz.uni-mannheim.de/jas
- Thanks to
  - Raphael Jolly
  - Thomas Becker, Samuel Kredel
  - Markus Aleksy, Hans-Günther Kruse
  - Adrian Schneider
  - the referees
  - and other colleagues