Hilfsmittel zur Integration rationaler Funktionen in Java

Heinz Kredel

Computer-Algebra Seminar HWS 2009
Universität Mannheim

Overview

- Coefficients and Polynomials
  - Types and Classes
  - Functionality and Implementation
  - Examples and Performance
- Greatest common divisors, squarefree decomposition and factorization
- Integration of rational functions
  - partial fraction decomposition
  - Hermite, Rothstein-Trager

Chebychev polynomials

defined by recursion:

```
T[0] = 1
T[1] = x
T[n] = 2 x T[n-1] - T[n-2]
```

first 10 polynomials:

```
T[0] = 1
T[1] = x
T[2] = 2 x^2 - 1
T[3] = 4 x^3 - 3 x
T[4] = 8 x^4 - 8 x^2 + 1
T[5] = 16 x^5 - 20 x^3 + 5 x
T[6] = 32 x^6 - 48 x^4 + 18 x^2 - 1
T[7] = 64 x^7 - 112 x^5 + 56 x^3 - 7 x
T[8] = 128 x^8 - 256 x^6 + 160 x^4 - 32 x^2 + 1
T[9] = 256 x^9 - 576 x^7 + 432 x^5 - 120 x^3 + 9 x
```

Chebychev polynomial computation

```
int m = 10;
BigInteger fac = new BigInteger();
String[] var   = new String[]{ "x" };
GenPolynomialRing<BigInteger> ring
               = new GenPolynomialRing<BigInteger>(fac,1,var);
List<GenPolynomial<BigInteger>> T
               = new ArrayList<GenPolynomial<BigInteger>>(m);
GenPolynomial<BigInteger> t, one, x, x2;
one = ring.getONE();
x   = ring.univariate(0); // polynomial in variable 0
x2  = ring.parse("2 x");
T.add( one );     // T[0]
T.add( x );       // T[1]
for ( int n = 2; n < m; n++ ) {
    t = x2.multiply( T.get(n-1) ).subtract( T.get(n-2) );
    T.add( t );   // T[n]
}
for ( int n = 0; n < m; n++ ) {
    System.out.println("T["+n+"] = " + T.get(n) );
}
```

Introduction to software

- 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
- dynamic memory system with GC
- 64-bit ready
- jython (Java Python) front end

Implementation

- 230+ classes and interfaces
- plus 100+ JUnit test cases
- JDK 1.5 with generic types
- logging with Apache Log4j
- some jython scripts
- javadoc API documentation
- revision control with subversion
- build tool is Apache Ant
- open source, license is GPL

Polynomials

$$p \in R = C[x_1, \ldots, x_n]$$

$$p = \boxed{3x_1^2 x_3^4 + 7x_2^5 - 61} \in \mathbb{Z}[x_1, x_2, x_3]$$

- multivariate polynomials
- polynomial ring
  - in n variables
  - over a coefficient ring
- 3 variables $x_{1,} x_2 \text{and} \, x_3$
- with integer coefficients

Monoid rings

$T \quad Monoid\,, \quad C \quad Ring$

$$p: \; T \; \rightarrow \; C$$
$$t \rightarrow p(t) = c_t$$

$p(t) \neq 0$ *for only finitely many* $t \in T$

$$(p+q)(t) = p(t) + q(t)$$

$$(p \cdot q)(t) = \sum_{uv=t} p(u) \cdot q(v)$$

$$T \; = \; \{ \; x_1^{e_1} \ldots x_n^{e_n} \; | \; (e_1, \ldots, e_n) \; \in \; \mathbb{N}^n \; \}$$

$$C[x_{1,\ldots}, x_n] \; = \; \{ \; p \; | \; p : T \; \rightarrow \; C \; \}$$

- polynomials as mappings
  - from terms to coefficients
  - terms are power products of variables
  - with finite support
  - definition of sum and product

$$\boxed{x_1^2 x_3^4 \rightarrow 3, \;\; x_2^5 \rightarrow 7, \;\; x_1^0 x_2^0 x_3^0 \rightarrow -61}$$

$$else \quad x_1^{e_1} x_2^{e_2} x_3^{e_3} \rightarrow 0$$

Polynomials (cont.)

$$one: \left\{ x_1^0 x_2^0 \ldots x_n^0 \rightarrow 1 \right\}$$

$$zero: \left\{ \right\}$$

$$x_1^2 x_3^4 \; >_T \; x_2^5$$

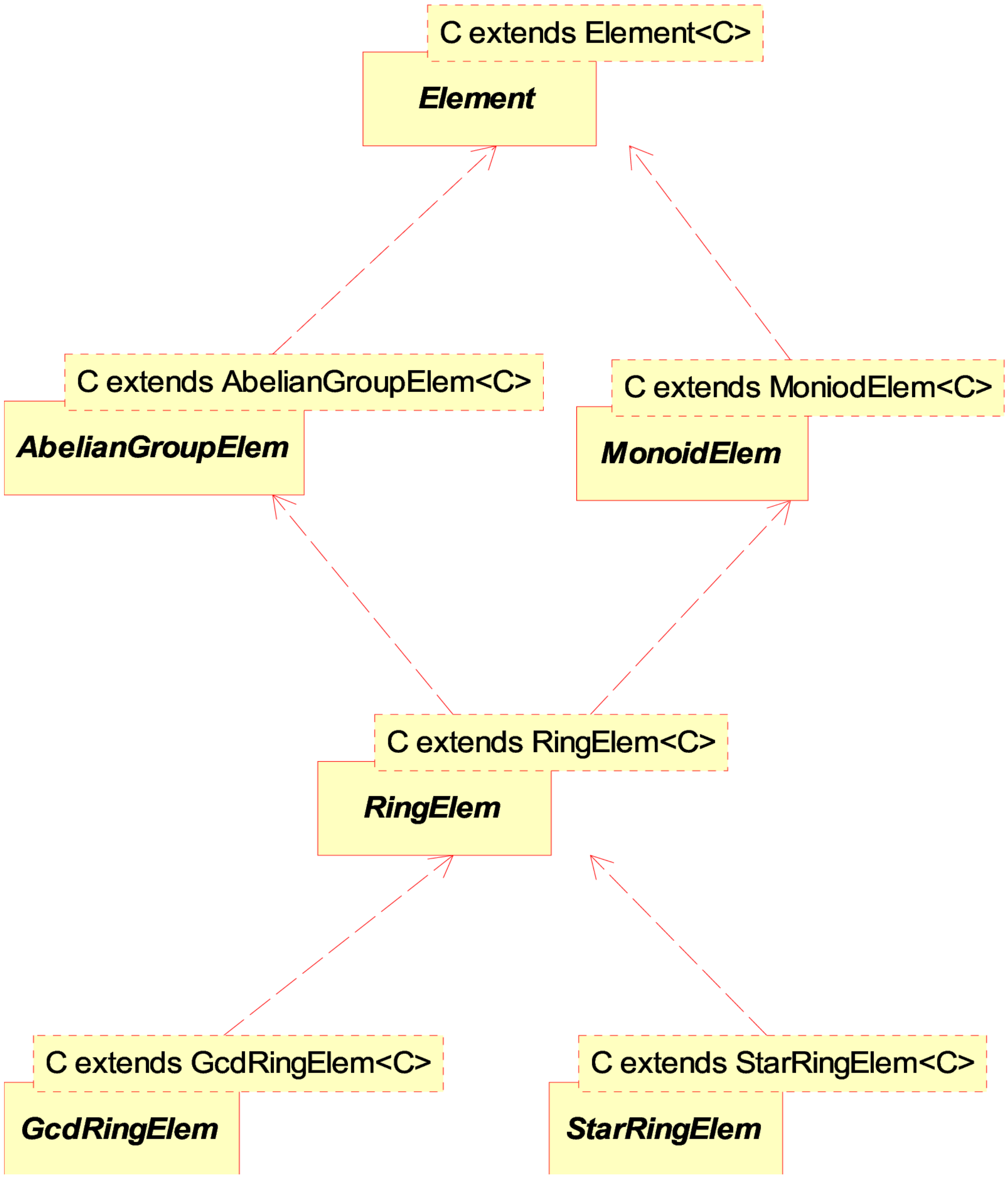
- mappings to zero are not stored
- terms are ordered / sorted

$$\boxed{x_j * x_i = c_{ij} x_i x_j + p_{ij}}$$

$$1 \leqslant i < j \leqslant n, \;\; 0 \neq c_{ij} \in C,$$

$$x_i x_j >_T p_{ij} \in R$$

- polynomials with non-commutative multiplication
- commutative is special case $c_{ij}=1,\ p_{ij}=0$


C extends Element<C>
Element
C extends AbelianGroupElem<C>
AbelianGroupElem
C extends MoniodElem<C>
MonoidElem
C extends RingElem<C>
RingElem
C extends GcdRingElem<C>
GcdRingElem
C extends StarRingElem<C>
StarRingElem

Ring element creation

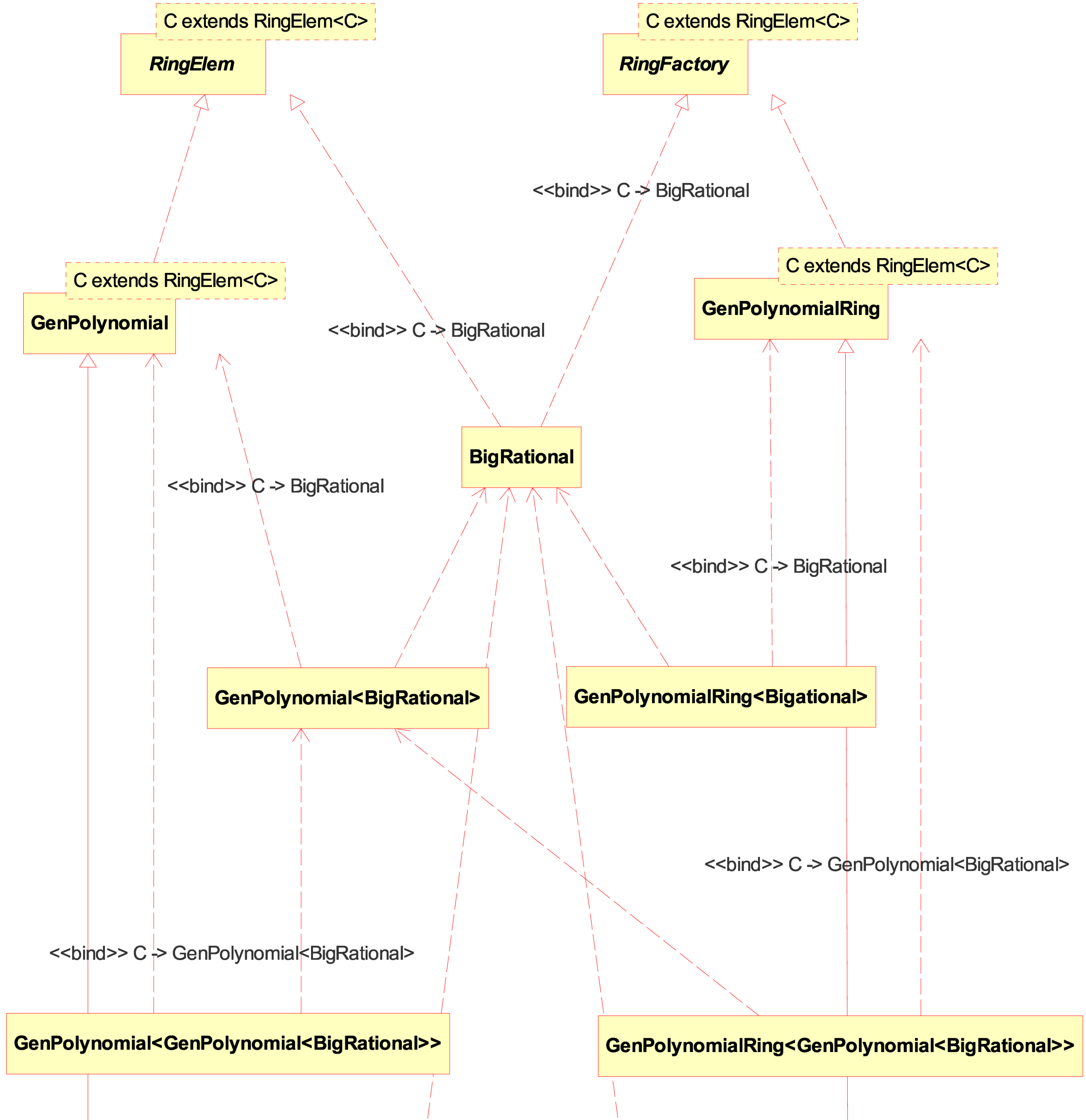
- recursive type for coefficients and polynomials
- creation of ZERO and ONE needs information about the ring
- `new C()` not allowed in Java, `C` type parameter
- solution with factory pattern: `RingFactory`
- factory has sufficient information for creation of ring elements
- eventually has references to other factories, e.g. for coefficients

Ring element functionality

- `C` is type parameter
- `C sum(C S), C subtract(C S), C negate(), C abs()`
- `C multiply(C s), C divide(C s), C remainder(C s), C inverse()`
- `boolean isZERO(), isONE(), isUnit(), int signum()`
- `equals(Object b), int hashCode(), int compareTo(C b)`
- `C clone()` versus `C copy(C a)`
- `Serializable` interface is implemented

Ring factory functionality

- create 0 and 1
  - `C getZERO(), C getONE()`
- `C copy(C a)`
- embed integers `C fromInteger(long a)`
  - `C fromInteger(java.math.BigInteger a)`
- random elements `C random(int n)`
- parse string representations
  - `C parse(String s), C parse(Reader r)`
- `isCommutative(), isAssociative()`



C extends RingElem<C>
RingElem
C extends RingElem<C>
RingFactory
<<bind>> C -> BigRational
C extends RingElem<C>
GenPolynomial
<<bind>> C -> BigRational
C extends RingElem<C>
GenPolynomialRing
BigRational
<<bind>> C -> BigRational
<<bind>> C -> BigRational
GenPolynomial<BigRational>
GenPolynomialRing<Bigational>
<<bind>> C -> GenPolynomial<BigRational>
<<bind>> C -> GenPolynomial<BigRational>
GenPolynomial<GenPolynomial<BigRational>>
GenPolynomialRing<GenPolynomial<BigRational>>

Coefficients

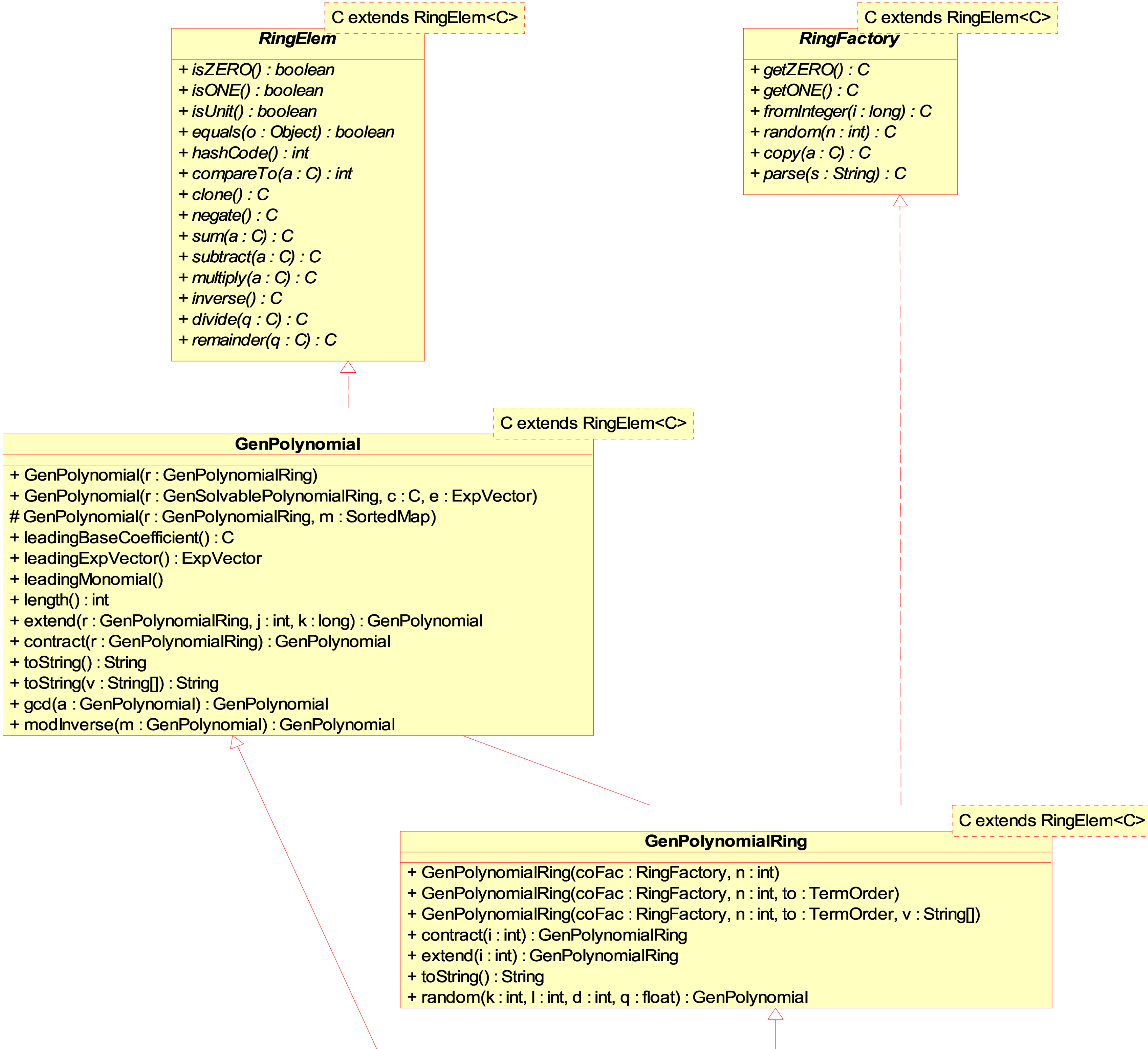
- e.g. `BigRational, BigInteger`
- implement both interfaces
- creation of rational number 2 from long 2:
  - `new BigRational(2)`
  - `cfac.fromInteger(2)`
- creation of rational number 1/2 from two longs:
  - `new BigRational(1,2)`
  - `cfac.parse("1/2")`

Polynomials

- `GenPolynomial<C extends RingElem<C>>`
- `C` is coefficient type in the following
- implements `RingElem<GenPolynomial<C>>`
- factory is `GenPolynomialRing<...>`
- implements `RingFactory<GenPolynomial<C>>`
- factory constructors require coefficient factory parameter

Polynomial creation

- types are
  - GenPolynomial<BigRational>
  - GenPolynomialRing<BigRational>
- creation is
  - new GenPolynomialRing<BigRational>(cfac,5)
  - pfac.getONE()
  - pfac.parse(“1”)
- polynomials as coefficients
  - GenPolynomial<GenPolynomial<BigRational>>
  - GenPolynomialRing<GenPolynomial<...>>(pfac,3)


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
C extends RingElem<C>
GenPolynomial
+ GenPolynomial(r : GenPolynomialRing)
+ GenPolynomial(r : GenSolvablePolynomialRing, 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
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

Polynomial factory constructors

- coefficient factory of the corresponding type
- number of variables
- term order (optional) $x_1^2 x_3^4 >_T x_2^5$
- names of the variables (optional)
- GenPolynomialRing<C>(RingFactory<C> cf, int n, TermOrder t, String[] v)

Polynomial factory functionality

- ring factory methods plus more specific methods
- `GenPolynomial<C> random(int k, int l, int d, float q, Random rnd)`
- embed and restrict polynomial ring to ring with more or less variables
  - `GenPolynomialRing<C> extend(int i)`
  - `GenPolynomialRing<C> contract(int i)`
  - `GenPolynomialRing<C> reverse()`
- handle term order adjustments

Polynomial functionality

- ring element methods plus more specific methods
- constructors all require a polynomial factory
  - GenPolynomial(GenPolynomialRing<C> r, C c, ExpVector e)
  - GenPolynomial(GenPolynomialRing<C> r, SortedMap<ExpVector,C> v)
- access parts of polynomials
  - ExpVector leadingExpVector()
  - C leadingBaseCoefficient()
  - Map.Entry<ExpVector,C> leadingMonomial()
- extend and contract polynomials

Example

```
BigInteger z = new BigInteger();
TermOrder to = new TermOrder();
String[] vars = new String[] { "x1", "x2", "x3" };
GenPolynomialRing<BigInteger> ring

 = new GenPolynomialRing<BigInteger>(z,3,to,vars);

GenPolynomial<BigInteger> pol
 = ring.parse( "3 x1^2 x3^4 + 7 x2^5 - 61" );


toString output:
ring = BigInteger(x1, x2, x3) IGRLEX
pol = GenPolynomial[
      3 (4,0,2), 7 (0,5,0), -61 (0,0,0) ]
pol = 3 x1^2 * x3^4 + 7 x2^5 - 61
```

Example (cont.)

```
p1 = pol.subtract(pol);
p2 = pol.multiply(pol);
```

```
p1 = GenPolynomial[  ]
p1 = 0
p2 =  9 x1^4 * x3^8 + 42 x1^2 * x2^5 * x3^4
     + 49 x2^10
     - 366 x1^2 * x3^4 - 854 x2^5 + 3721
```

Coefficient implementation

- `BigInteger` based on `java.math.BigInteger`
- implemented in pure Java, no GMP C-library
- using adaptor pattern to implement `RingElem` (and `RingFactory`) interface
- about 10 to 15 times faster than the Modula-2 implementation `SACI` (in 2000)
- other classes: `BigRational, ModInteger, BigComplex, BigQuaternion` and `BigOctonion`
- `AlgebraicNumber` class can be used over `BigRational` or `ModInteger`

Polynomial implementation

- are (ordered) maps from terms to coefficients
- implemented with `SortedMap` interface and `TreeMap` class from Java collections framework
- alternative implementation with `Map` and `LinkedHashMap`, which preserves the insertion order
- but had inferior performance
- terms (the keys) are implemented by class `ExpVector`
- coefficients implement `RingElem` interface

Polynomial implementation (cont.)

- ExpVector is dense array of exponents (as long) of variables
- sparse array, array of int, Long not implemented
- would like to have ExpVector<long>
- polynomials are intended as immutable objects
- object variables are final and the map is not modified after creation
- eventually wrap with unmodifiableSortedMap()
- avoids synchronization in multi threaded code

Performance

- polynomial arithmetic performance:
  - performance of coefficient arithmetic
    - `java.math.BigInteger` in pure Java, faster than GMP style JNI C version
  - sorted map implementation
    - from Java collection classes with known efficient algorithms
  - exponent vector implementation
    - using `long[]`, have to consider also `int[]` or `short[]`
    - want `ExpVector<C>` but generic types may not be elementary types
  - JAS comparable to general purpose CA systems but slower than specialized systems

Performance

$$compute\ q = p \times (p+1)$$

$$p = (1+x+y+z)^{20}$$
$$p = (10000000001\,(1+x+y+z))^{20}$$
$$p = (1+x^{2147483647}+y^{2147483647}+z^{2147483647})^{20}$$

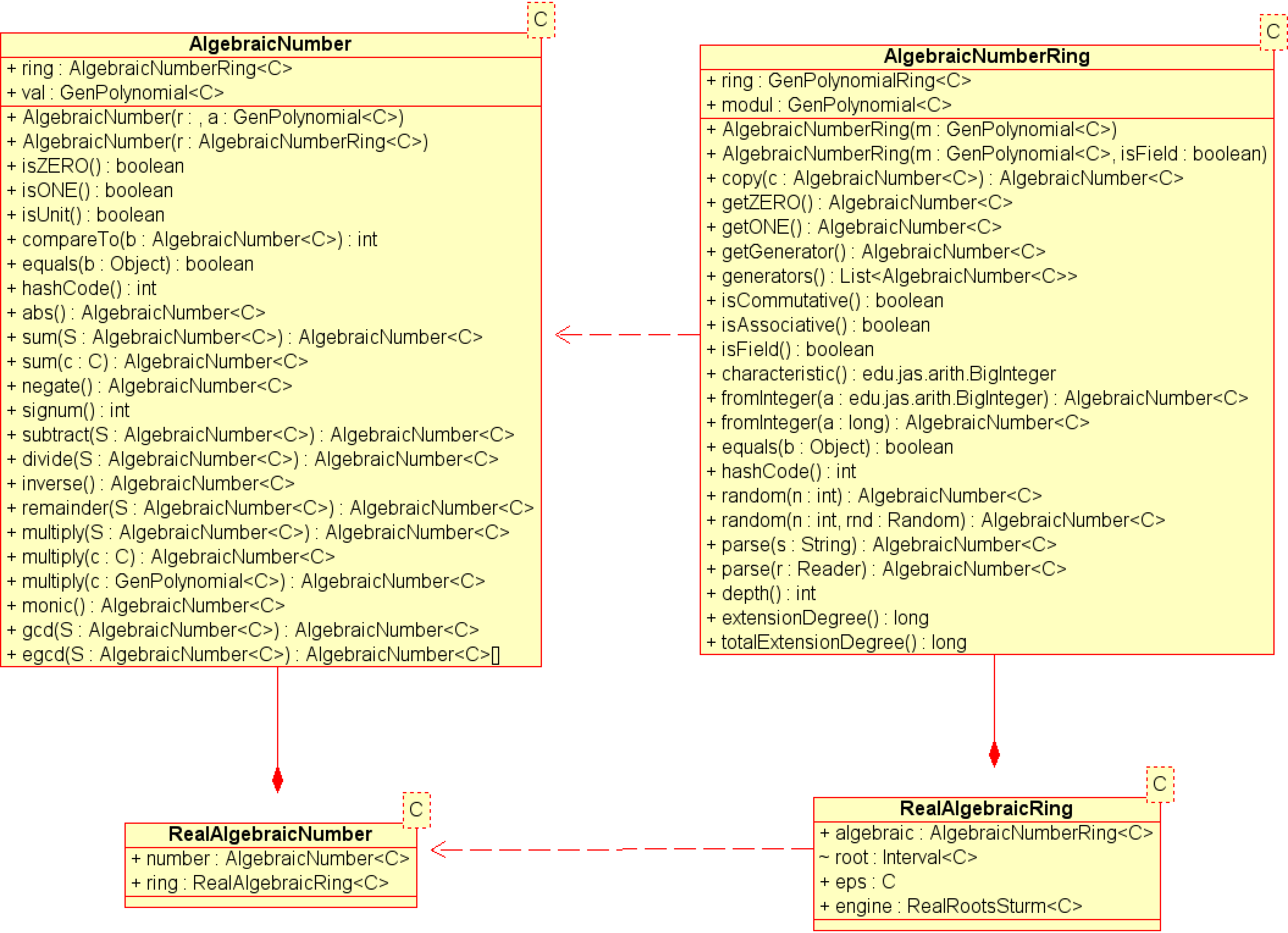
JAS: options, system	JDK 1.5	JDK 1.6
BigInteger, G	16.2	13.5
BigInteger, L	12.9	10.8
BigRational, L, s	9.9	9.0
BigInteger, L, s	9.2	**8.4**
BigInteger, L, big e, s	9.2	**8.4**
BigInteger, L, big c	66.0	59.8
BigInteger, L, big c, s	45.0	**43.2**

options, system	time	@2.7GHz
MAS 1.00a, L, GC = 3.9	**33.2**	
Singular 2-0-6, G	2.5	
Singular, L	**2.2**	
Singular, G, big c	12.9	
Singular, L, big exp	out of memory	
Maple 9.5	**15.2**	9.1
Maple 9.5, big e	19.8	11.8
Maple 9.5, big c	64.0	38.0
Mathematica 5.2	**22.8**	13.6
Mathematica 5.2, big e	30.9	18.4
Mathematica 5.2, big c	30.6	18.2
JAS, s	8.4	5.0
JAS, big e, s	8.6	5.1
JAS, big c, s	47.8	28.5

Computing times in seconds on AMD 1.6 GHz or 2.7 GHz Intel XEON CPU.
Options are: coefficient type, term order: G = graded, L = lexicographic,
big c = using the big coefficients, big e = using the big exponents, s = server JVM.

Algebraic numbers

- Residue classes modulo univariate polynomials
- K[x]/p = K(alpha), p polynomial, K field
- for example `BigRational, ModInteger`
- implement `RingElem` interface
- can be used as coefficients for other polynomials


C
AlgebraicNumber
+ ring : AlgebraicNumberRing<C>
+ val : GenPolynomial<C>
+ AlgebraicNumber(r : , a : GenPolynomial<C>)
+ AlgebraicNumber(r : AlgebraicNumberRing<C>)
+ isZERO() : boolean
+ isONE() : boolean
+ isUnit() : boolean
+ compareTo(b : AlgebraicNumber<C>) : int
+ equals(b : Object) : boolean
+ hashCode() : int
+ abs() : AlgebraicNumber<C>
+ sum(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ sum(c : C) : AlgebraicNumber<C>
+ negate() : AlgebraicNumber<C>
+ signum() : int
+ subtract(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ divide(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ inverse() : AlgebraicNumber<C>
+ remainder(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ multiply(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ multiply(c : C) : AlgebraicNumber<C>
+ multiply(c : GenPolynomial<C>) : AlgebraicNumber<C>
+ monic() : AlgebraicNumber<C>
+ gcd(S : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ egcd(S : AlgebraicNumber<C>) : AlgebraicNumber<C>[]
C
AlgebraicNumberRing
+ ring : GenPolynomialRing<C>
+ modul : GenPolynomial<C>
+ AlgebraicNumberRing(m : GenPolynomial<C>)
+ AlgebraicNumberRing(m : GenPolynomial<C>, isField : boolean)
+ copy(c : AlgebraicNumber<C>) : AlgebraicNumber<C>
+ getZERO() : AlgebraicNumber<C>
+ getONE() : AlgebraicNumber<C>
+ getGenerator() : AlgebraicNumber<C>
+ generators() : List<AlgebraicNumber<C>>
+ isCommutative() : boolean
+ isAssociative() : boolean
+ isField() : boolean
+ characteristic() : edu.jas.arith.BigInteger
+ fromInteger(a : edu.jas.arith.BigInteger) : AlgebraicNumber<C>
+ fromInteger(a : long) : AlgebraicNumber<C>
+ equals(b : Object) : boolean
+ hashCode() : int
+ random(n : int) : AlgebraicNumber<C>
+ random(n : int, rnd : Random) : AlgebraicNumber<C>
+ parse(s : String) : AlgebraicNumber<C>
+ parse(r : Reader) : AlgebraicNumber<C>
+ depth() : int
+ extensionDegree() : long
+ totalExtensionDegree() : long
C
RealAlgebraicNumber
+ number : AlgebraicNumber<C>
+ ring : RealAlgebraicRing<C>
C
RealAlgebraicRing
+ algebraic : AlgebraicNumberRing<C>
~ root : Interval<C>
+ eps : C
+ engine : RealRootsSturm<C>

Overview

- Coefficients and Polynomials
  - Types and Classes
  - Functionality and Implementation
  - Examples and Performance
- Greatest common divisors, squarefree decomposition and factorization
- Integration of rational functions
  - partial fraction decomposition
  - Hermite, Rothstein-Trager

Unique factorization domains

- elements of a UFD can be written as

$$a = u \; p_1^{e_1} \; \ldots \; p_n^{e_n}$$

- polynomial rings over UFDs are UFDs

$$R = UFD[x_1, \; \ldots, \; x_n]$$

- Gauss Lemma

$$cont(a\,b) = cont(a) \; cont(b)$$

- primitive part

$$a = cont(a) \; pp(a)$$

- squarefree

$$\frac{a}{gcd(a, a')} \quad is\,squarefree$$

- squarefree factorization

$$a = a_1^1 \; \ldots \; a_d^d$$

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

Recursive methods

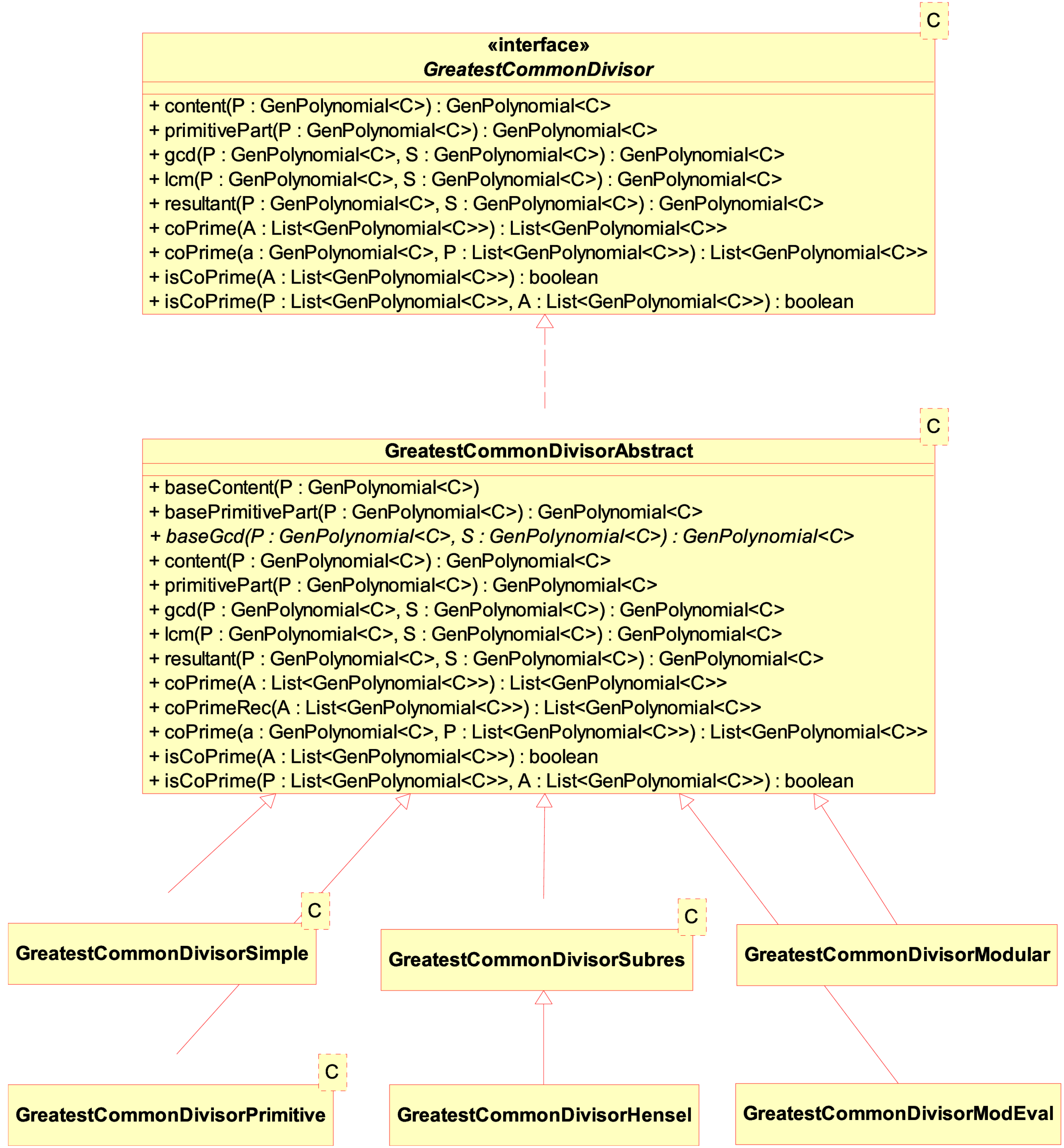
- recursive type RingElem<C extends RingElem<C>>
- so polynomials can have polynomials as coefficients
  - GenPolynomial<GenPolynomial<BigRational>>
- leads to code duplication due to type erasure
  - GenPolynomial<C> gcd(GenPolynomial<C> P, S)
  - GenPolynomial<C> baseGcd(GenPolynomial<C> P,S)
  - GenPolynomial<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 multivariate 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>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ coPrime(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ coPrime(a : GenPolynomial<C>, P : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ isCoPrime(A : List<GenPolynomial<C>>) : boolean
+ isCoPrime(P : List<GenPolynomial<C>>, A : List<GenPolynomial<C>>) : boolean
C
GreatestCommonDivisorAbstract
+ baseContent(P : GenPolynomial<C>)
+ basePrimitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ coPrime(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ coPrimeRec(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ coPrime(a : GenPolynomial<C>, P : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ isCoPrime(A : List<GenPolynomial<C>>) : boolean
+ isCoPrime(P : List<GenPolynomial<C>>, A : List<GenPolynomial<C>>) : boolean
C
GreatestCommonDivisorSimple
C
GreatestCommonDivisorSubres
GreatestCommonDivisorModular
C
GreatestCommonDivisorPrimitive
GreatestCommonDivisorHensel
GreatestCommonDivisorModEval

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

- algorithmic 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
    - monic PRS on multivariate polynomials
    - modulo prime polynomials to remove variables until univariate, multivariate 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,

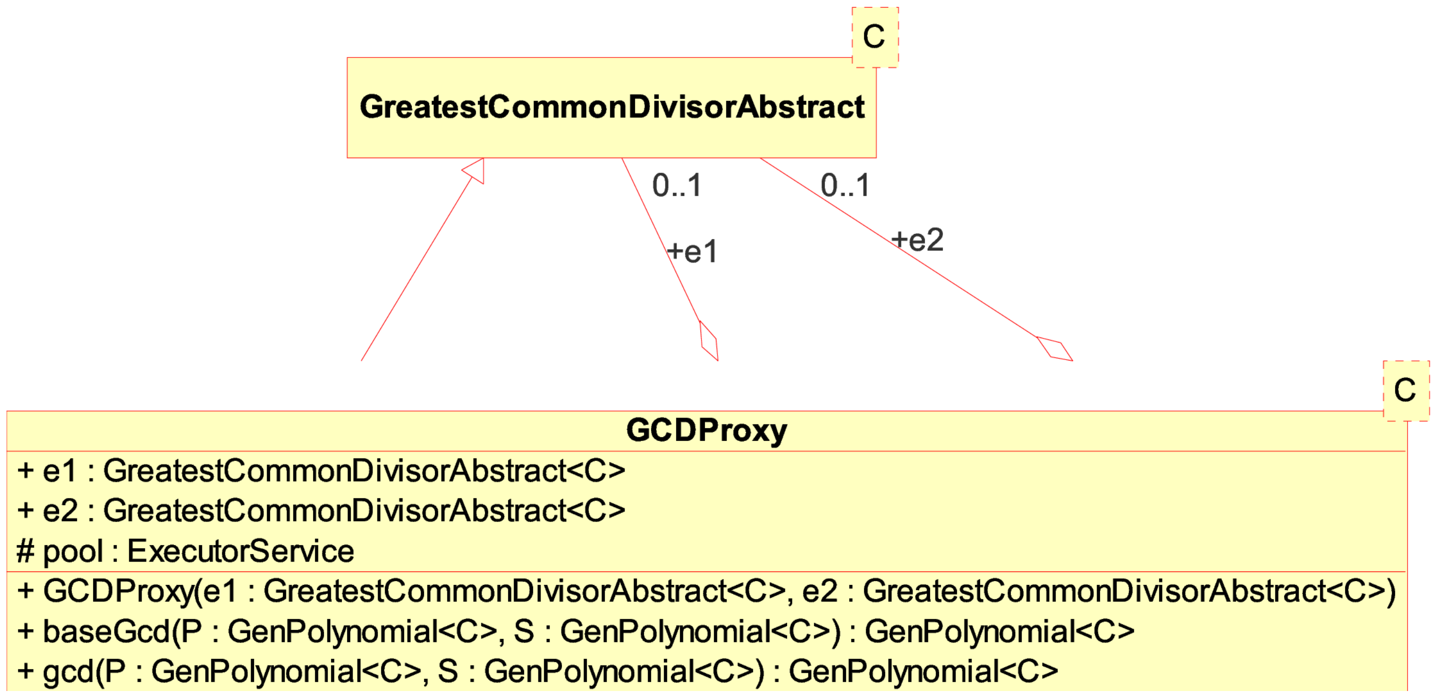
GCD factory

- 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)


C
GreatestCommonDivisorAbstract
0..1
0..1
+e1
+e2
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>
+ 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,

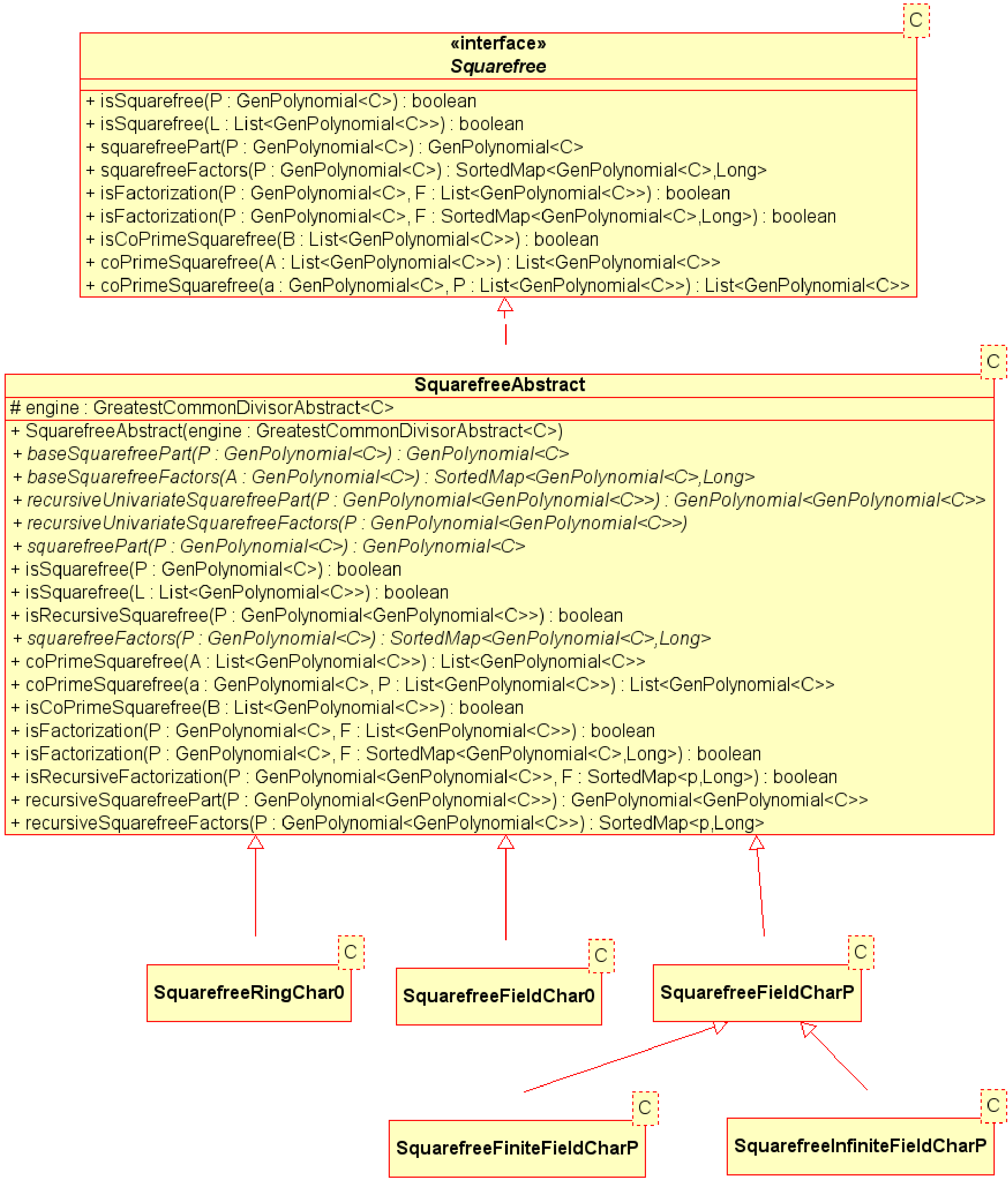
single CPU, 32-bit, 1.6 GHz

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,

Squarefree decomposition

- interface Squarefree
- abstract class SquarefreeAbstract
  - implements tests and co-prime squarefree set construction
- other classes for coefficients
  - ring or fields of characteristic zero
  - fields of characteristic p > 0
    - infinite rings
    - finite and infinite fields



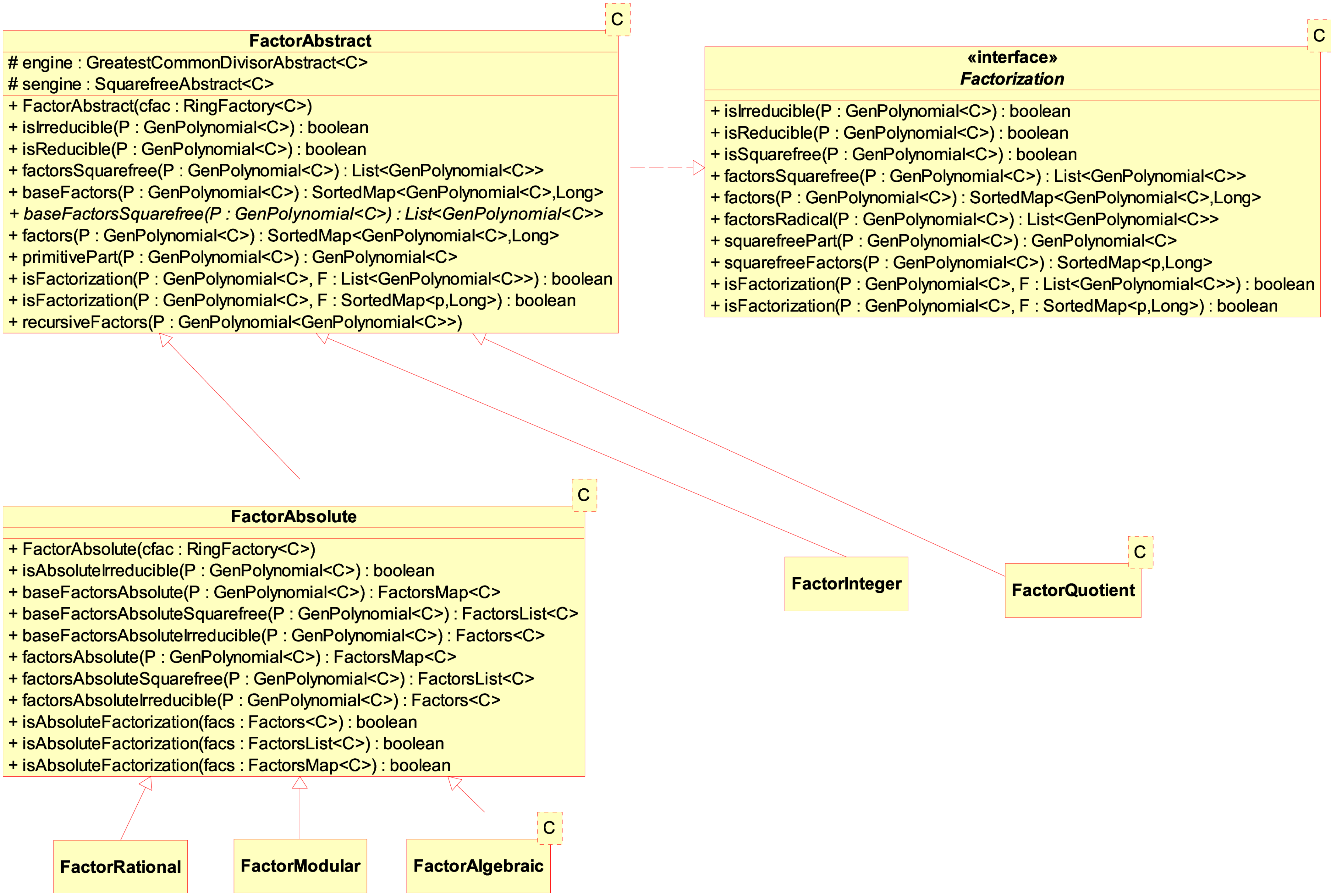
«interface»
Squarefree
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ isSquarefree(L : List<GenPolynomial<C>>) : boolean
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
+ isCoPrimeSquarefree(B : List<GenPolynomial<C>>) : boolean
+ coPrimeSquarefree(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ coPrimeSquarefree(a : GenPolynomial<C>, P : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
C
SquarefreeAbstract
engine : GreatestCommonDivisorAbstract<C>
+ SquarefreeAbstract(engine : GreatestCommonDivisorAbstract<C>)
+ baseSquarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ baseSquarefreeFactors(A : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ recursiveUnivariateSquarefreePart(P : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ recursiveUnivariateSquarefreeFactors(P : GenPolynomial<GenPolynomial<C>>)
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ isSquarefree(L : List<GenPolynomial<C>>) : boolean
+ isRecursiveSquarefree(P : GenPolynomial<GenPolynomial<C>>) : boolean
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ coPrimeSquarefree(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ coPrimeSquarefree(a : GenPolynomial<C>, P : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ isCoPrimeSquarefree(B : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
+ isRecursiveFactorization(P : GenPolynomial<GenPolynomial<C>>, F : SortedMap<p,Long>) : boolean
+ recursiveSquarefreePart(P : GenPolynomial<GenPolynomial<C>>) : GenPolynomial<GenPolynomial<C>>
+ recursiveSquarefreeFactors(P : GenPolynomial<GenPolynomial<C>>) : SortedMap<p,Long>
C
SquarefreeRingChar0
C
SquarefreeFieldChar0
C
SquarefreeFieldCharP
C
SquarefreeFiniteFieldCharP
C
SquarefreeInfiniteFieldCharP

Factorization

- interface Factorization
- abstract class FactorAbstract
  - implements nearly everything only baseFactorSquarefree must be implemented for each coefficient ring
  - uses Kronecker substitution for reduction to univariate case
- FactorModular
  - implements distinctDegreeFactor and equalDegreeFactor

Factorization (cont)

- FactorInteger
  - computes modulo primes, lifts with Hensel and does combinatorial factor search
- FactorRational
  - clears denominators and uses factorization over integers
- FactorAlgebraic
  - can eventually be used for modular and rational coefficients
  - computes and factors norm and then used gcds between factors of norm and polynomial


C
FactorAbstract
engine : GreatestCommonDivisorAbstract<C>
sengine : SquarefreeAbstract<C>
+ FactorAbstract(cfac : RingFactory<C>)
+ isIrreducible(P : GenPolynomial<C>) : boolean
+ isReducible(P : GenPolynomial<C>) : boolean
+ factorsSquarefree(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ baseFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ baseFactorsSquarefree(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ factors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<p,Long>) : boolean
+ recursiveFactors(P : GenPolynomial<GenPolynomial<C>>)
C
«interface»
Factorization
+ isIrreducible(P : GenPolynomial<C>) : boolean
+ isReducible(P : GenPolynomial<C>) : boolean
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ factorsSquarefree(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ factors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ factorsRadical(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<p,Long>
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<p,Long>) : boolean
C
FactorAbsolute
+ FactorAbsolute(cfac : RingFactory<C>)
+ isAbsoluteIrreducible(P : GenPolynomial<C>) : boolean
+ baseFactorsAbsolute(P : GenPolynomial<C>) : FactorsMap<C>
+ baseFactorsAbsoluteSquarefree(P : GenPolynomial<C>) : FactorsList<C>
+ baseFactorsAbsoluteIrreducible(P : GenPolynomial<C>) : Factors<C>
+ factorsAbsolute(P : GenPolynomial<C>) : FactorsMap<C>
+ factorsAbsoluteSquarefree(P : GenPolynomial<C>) : FactorsList<C>
+ factorsAbsoluteIrreducible(P : GenPolynomial<C>) : Factors<C>
+ isAbsoluteFactorization(facs : Factors<C>) : boolean
+ isAbsoluteFactorization(facs : FactorsList<C>) : boolean
+ isAbsoluteFactorization(facs : FactorsMap<C>) : boolean
FactorInteger
C
FactorQuotient
FactorRational
FactorModular
C
FactorAlgebraic

Overview

- Coefficients and Polynomials
  - Types and Classes
  - Functionality and Implementation
  - Examples and Performance
- Greatest common divisors, squarefree decomposition and factorization
- Integration of rational functions
  - partial fraction decomposition
  - Hermite, Rothstein-Trager

Integration of a rational function

```
integrateRational(A,D) {

  (p,r) = quotientRemainder(A,D);
  c = gcd(r,D);
  r = r/c; D = D/c;
  (g,h) = integrateHermite(r,D);
  (hp,hr) = quotientRemainder(num(h),denom(h));
  P = integrate(p+hp);
  if ( hr == 0 ) {
      return P + g;
  }
  L = integrateLogPart(hr,denom(h));

  return P + g + L;
}
```

Hermite Reduction

```
integrateHermite(A,D) { // gcd(A,D) == 1

 (D_1, ..., D_n) = squarefree(D);
 (P,A_1,...,A_n) = partialFraction(A,(D_1,D_2^2,...,D_n^n));
 g = 0;
 h = P + A_1/D_1;
 for ( k = 2, k <= n; k++ ) {
   if ( deg(D_k) == 0 ) { continue;
   }
   V = D_k;
   for ( j = k-1; k >= 1; j-- ) {
      (B,C) = exdendedEuclideanDiophant( dV/dx, V, -A_k/j );
      g = g + B/(V^j);
      A_k = -j C - dB/dx;
   }
   h = h + A_k/V;
 }
 return (g,h);
}
```

Rothstein-Trager

```
integrateLogPart(A,D) {
 // gcd(A,D) == 1, D squarefree, deg(A) < deg(D)

 R = resultant_x(D, A - t dD/dx ); // in K[t,x]
 ( u, R_1^e_1, ..., R_m^e_m ) = factor(R); // in K[t]

 for ( i = 1, i <= m; i++ ) {
   a = rootOf( R_i );            // K(a) = K[t]/R_i
   G_i = gcd( D, A - a dD/dx );
 }
 return sum_i( sum_(a in rootOf(R_i)) a log(G_i);
}
```

partial fraction decomposition

```
/** Univariate GenPolynomial partial fraction decomposition.
 * @param A univariate GenPolynomial.
 * @param P univariate GenPolynomial.
 * @param S univariate GenPolynomial.
 * @return [ A0, Ap, As ] with A/(P*S) = A0 + Ap/P + As/S with deg(Ap) < deg(P) and
 */
public GenPolynomial<C>[] basePartialFraction(GenPolynomial<C> A, GenPolynomial<C>
    GenPolynomial<C>[] ret = new GenPolynomial[3];
    GenPolynomial<C> ps = P.multiply(S);
    GenPolynomial<C>[] qr = PolyUtil.<C> basePseudoQuotientRemainder(A, ps);
    ret[0] = qr[0];
    GenPolynomial<C> r = qr[1];
    GenPolynomial<C>[] diop = baseGcdDiophant(S,P,r); // switch arguments
    ret[1] = diop[0];  ret[2] = diop[1];
    if ( ret[1].degree(0) >= P.degree(0) ) {
        qr = PolyUtil.<C> basePseudoQuotientRemainder(ret[1], P);
        ret[0] = ret[0].sum( qr[0] );
        ret[1] = qr[1];
    }
    if ( ret[2].degree(0) >= S.degree(0) ) {
        qr = PolyUtil.<C> basePseudoQuotientRemainder(ret[2], S);
        ret[0] = ret[0].sum( qr[0] );
        ret[2] = qr[1];
    }
    return ret;
}
```

Algebraic field extension in R-T

```
// compute A - t P' and P in K[t][x]
// compute resultant in K[t][x]
GenPolynomial<GenPolynomial<C>> Rc
                              = engine.recursiveResultant(Pc, At);
GenPolynomial<C> res = Rc.leadingBaseCoefficient();
// factor resultant
SortedMap<GenPolynomial<C>,Long> resfac = irr.baseFactors(res);
for ( GenPolynomial<C> r : resfac.keySet() ) {
    // construct extension field
    AlgebraicNumberRing<C> afac
       = new AlgebraicNumberRing<C>(r, true); // since irreducible
    AlgebraicNumber<C> a = afac.getGenerator();
    // construct K(alpha)[x]
    GenPolynomialRing<AlgebraicNumber<C>> pafac
       = new GenPolynomialRing<AlgebraicNumber<C>>(afac, Pc.ring);
    // convert polynomials to K(alpha)[x]
    // compute gcd
    GenPolynomial<AlgebraicNumber<C>> Ga = aengine.baseGcd(Pa,Ap);
    // record factor and gcd
    afactors.add( a );
    adenom.add( Ga );
}
```

Examples (1)

```
integral (1 ) / (x^2 - 2 ) =
(z_929) log( x - { 4 z_929 } ) + (-1 z_929) log( x + { 4 z_929 } )


integral (1 ) / (x^3 + x) =
(1) log( x) + (-1/2 ) log( x^2 + { 1  } )


integral (1 ) / (x^6 - 5 x^4 + 5 x^2 + 4 ) =
sum_(z_347 in
 rootOf(z_347^6 + 35/1712 z_347^4 + 55/366368 z_347^2 + 1/2930944 ) )
(z_347) log( x + { 686940 z_347^5 + 28355/4 z_347^3 + 459/16
z_347 } )


integral (1 ) / (x^4 + 4 ) =
(z_93) log( x + { 16 z_93 } ) + (-1 z_93) log( x - { 16 z_93 - 2  } )
+ (z_167) log( x + { 16 z_167 } )
+ (-1 z_167) log( x - { 16 z_167 + 2  } )
```

Examples (2)

```
integral (7 x^6 + 1 ) / (x^7 + x + 1 ) =
(1 ) log( x^7 + x + { 1  } )


integral (1 ) / (x^3 - 6 x^2 + 11 x - 6 ) =
(1/2) log( x - 3 ) + (-1) log( x - 2 ) + (1/2) log( x - 1 )


integral (1 ) / (x^3 - 2 ) =
sum_(z_29 in rootOf(z_29^3 - 1/108 ) ) (z_29) log( x - { 6 z_29 } )

with absolute factorization:
integral (1 ) / (x^3 - 2 ) =
sum_(z_885 in rootOf(z_885^3 - 2 ) ) (1/6 z_885) log( x - { z_885 } )
+ ({ 1/6 } z_734) log( x - { z_734 } )
+ ({ -1/6 } z_734 - { 1/6 z_885 }) log( x + { z_734 + { z_885 }  } )
```

Example with Maple (1)

Maple Version 11

```
F := 1/(x**3 - 2);
                                                  1
                                           F := ------
                                                 3
                                                x  - 2

> int(F,x);
     (1/3)          (1/3)           (1/3)    2       (1/3)    (2/3)
1/6 2      ln(x - 2     ) - 1/12 2      ln(x  + x 2      + 2     )

                                  1/2   (2/3)
               (1/3)  1/2        3    (2      x + 1)
     - 1/6 2      3    arctan(-------------------)
                                          3
```

Examples with Mathematica (1)

Mathematica Version 6.0

```
In[10]:= F = 1 / ( x^3 -2 )

             1
Out[10]= -------
                3
         -2 + x

In[11]:= Integrate[F,x]

                                       2/3
                                  1 + 2    x                     2/3
Out[11]= -(2 Sqrt[3] ArcTan[----------] - 2 Log[-2 + 2    x] +
                                   Sqrt[3]

                 2/3      1/3  2        2/3
>       Log[2 + 2    x + 2    x ]) / (6 2   )
```

Examples (3)

```
Result: integral( (1 ) / (x^5 + x - 7 ) ) =
sum_(z_127 in rootOf(z_127^5 - 160/7503381 z_127^3
                     - 80/7503381 z_127^2 - 5/2501127 z_127
                     - 1/7503381 ) )
(z_127) log( x + { 480216384/214375 z_127^4
             - 120054096/214375 z_127^3 + 30003284/214375 z_127^2
             - 7505941/214375 z_127 - 256/214375  } )
```

Examples with Maple (2)

```
 F := 1 / (x**5 + x - 7);
                                                                    1
                                                          F := ----------
                                                                5
                                                               x  + x - 7

> int(F,x);
   -----
    \                 480216384   4   120054096   3   30003284   2   75059
     )    _R ln(x + --------- _R  - --------- _R  + -------- _R  - -----
    /                  214375           214375           214375         21437
   -----
  _R = %1

                                5         3        2
       %1 := RootOf(7503381 _Z  - 160 _Z  - 80 _Z  - 15 _Z - 1)
```

Examples with Mathematica (2)

```
In[1]:= F = 1/ ( x**5 + x - 7 )

                  1
Out[1]= ---------------
        -7 + x + x ** 5

In[2]:= Integrate[F,x]

                              1
Out[2]= Integrate[---------------, x]
                  -7 + x + x ** 5
```

Conclusions

- consistent object oriented design and implementation of a library for algebraic computations
- Java advantages
  - generic types for type safety
  - multi-threaded, networked, 64-bit ready
  - dynamic memory management
- benefits for computer algebra
  - non-trivial structures can be implemented
  - library can be used from any Java program

Thank you

- Questions?
- Comments?
- http://krum.rz.uni-mannheim.de/jas
- http://krum.rz.uni-mannheim.de/kredel/ca-sem-2009.pdf
- Thanks to
  - Thomas Becker
  - Wolfgang K. Seiler
  - Aki Yoshida
  - Raphael Jolly
  - many others