On the Design of a Java Computer Algebra System

Heinz Kredel

PPPJ 2006, Mannheim

Introduction

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

Overview

- Introduction
- Types
- Classes
- Functionality
- Implementation
- Conclusions

Polynomials

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

- multivariate polynomials
- polynomial ring
  - in n variables
  - over a coefficient ring

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

- polynomials as mappings
  - from terms to coefficients

$$p = T \to C$$

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

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

Polynomials (cont.)

$$one: \{ x_1^0 x_2^0 \ldots x_n^0 \to 1 \}$$

$$zero: \{ \}$$

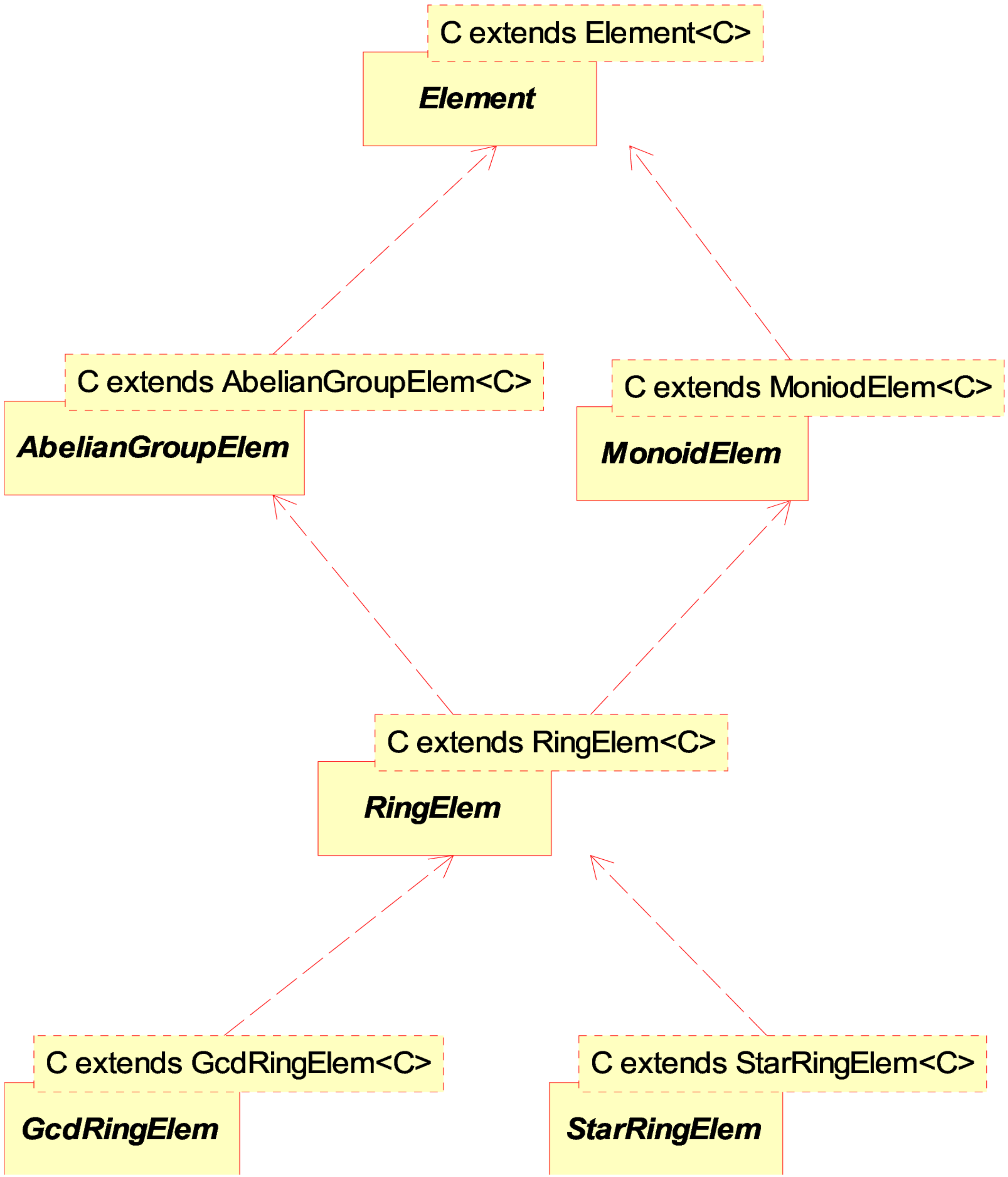
$$x_1^2 x_3^4 \;>_T\; x_2^5$$

- 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
- e.g. commutative $c_{ij}=1,\; p_{ij}=0$


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

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

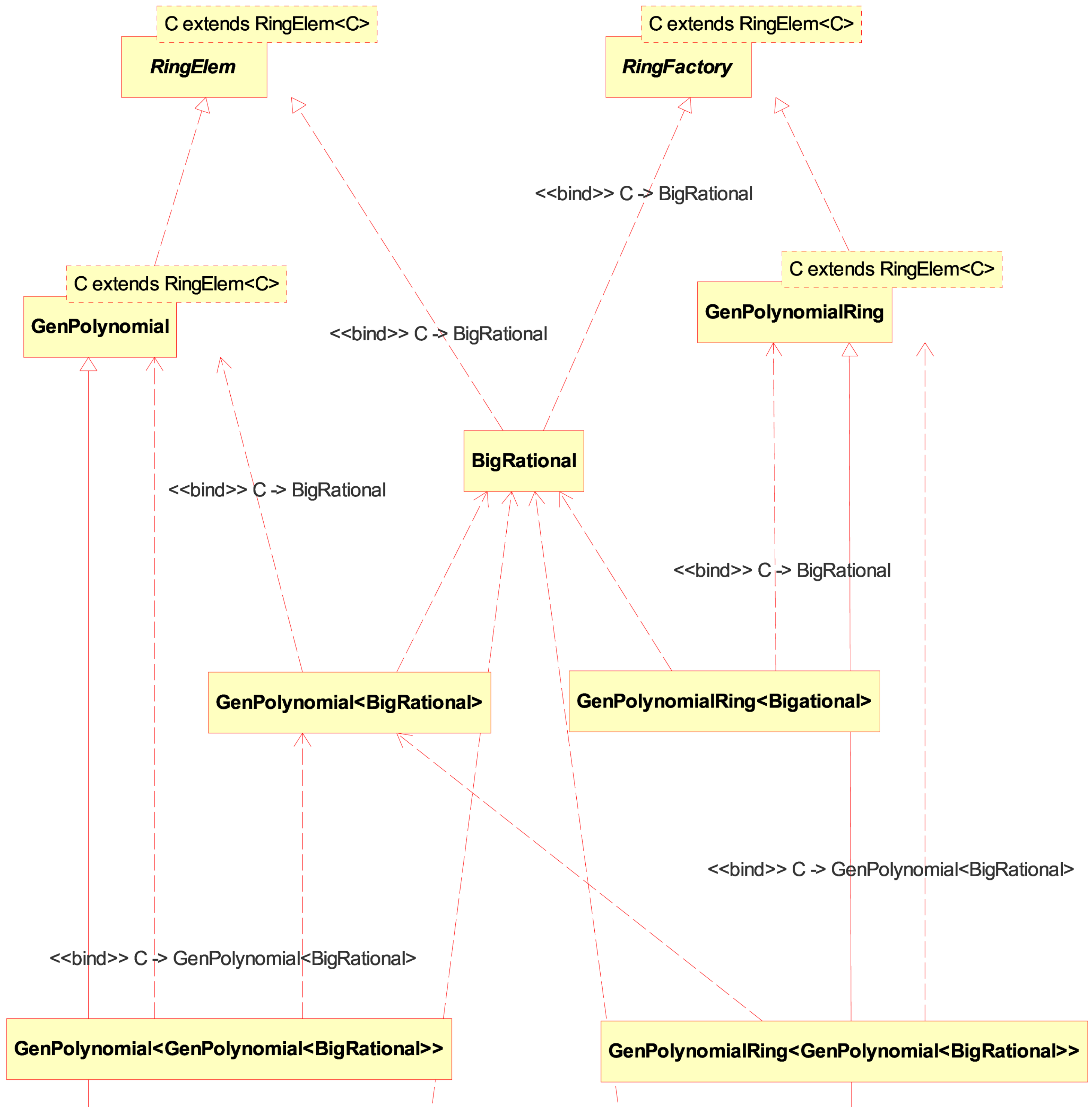
- `GenPolynomial<C extends RingElem<C>>`
- `C` is coefficient type in the following
- implements `RingElem<GenPolynomial<C>>`
- factory is `GenPolynomialRing<...>`
- implements `RingFactory<GenPolynomial<C>>`
- factory constructors require coefficient factory parameter

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)

Solvable polynomials

- extend generic polynomials
- called `GenSolvablePolynomial`
- inherit additive methods
- override `clone()` and `multiply()`
- uses factory for solvable polynomials also in inherited methods, hide super class factory
- factory stores table of relations $x_j * x_i = c_{ij} x_i x_j + p_{ij}$
- constructors permit `RelationTable` parameters (assumed commutative if omitted)



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>>

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()`

Polynomial factory constructors

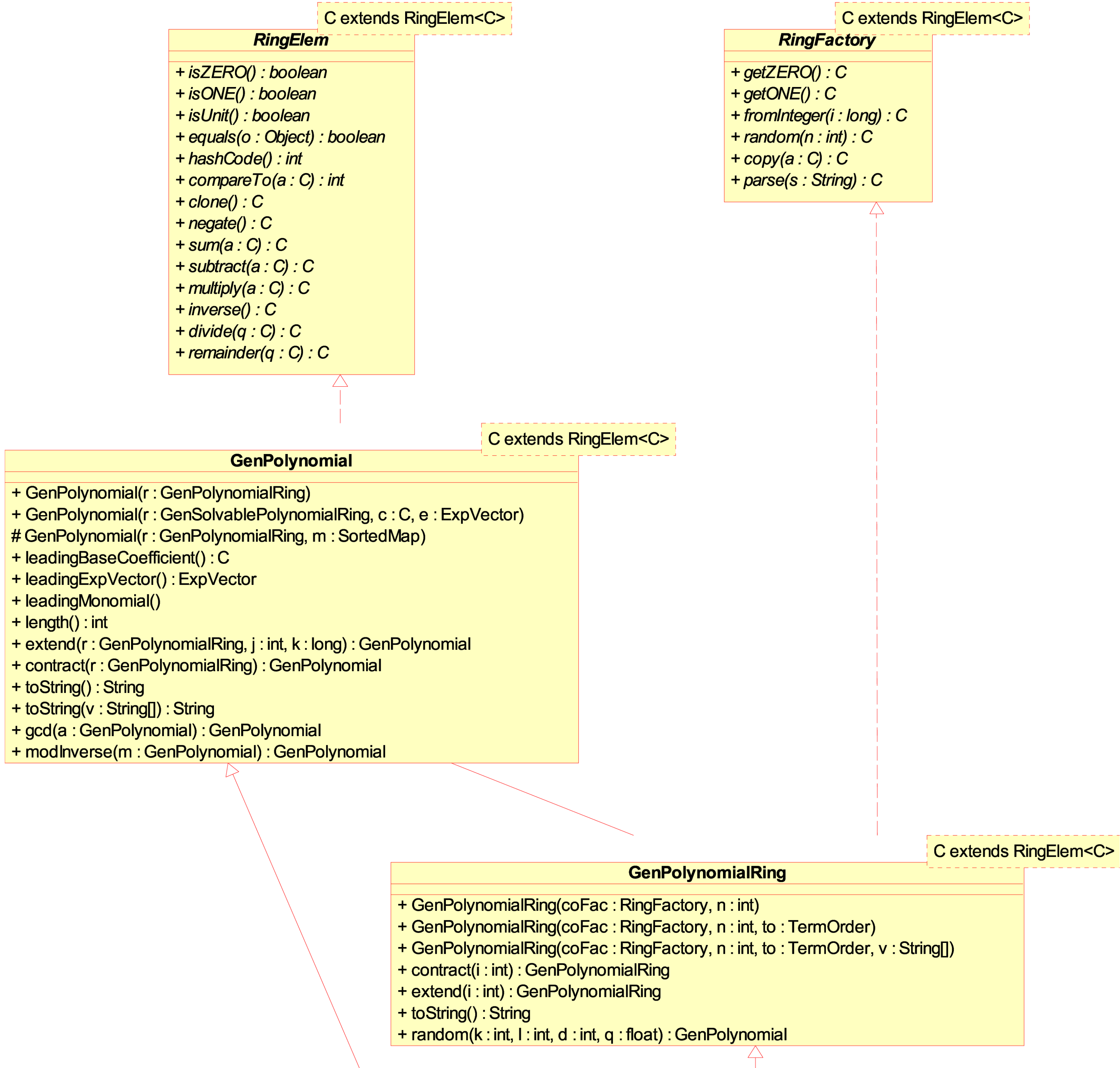
- coefficient factory of the corresponding type
- number of variables
- term order (optional) $x_1^2 x_3^4 >_T x_2^5$
- names of the variables (optional)
- `GenPolynomialRing<C>( RingFactory<C> cf, int n, TermOrder t, String[] v)`

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


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

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
```

Solvable polynomials

- extend generic polynomials, new multiplication
- want: implement ring element with solvable polynomial as type parameter
  - `RingElem<GenSolvablePolynomial<C>>`
- but: already implement ring element with polynomial as type parameter by inheritance
  - `RingElem<GenPolynomial<C>>`
- problem because type erasure makes them equal
- Java forbids implementation of same interface twice

Solvable polynomial functionality

- non commutative multiplication
  - `multiply(GenSolvablePolynomial<C> p)`
  - `multiply(C b, ExpVector e)`
  - `multiplyLeft(C b, ExpVector e)`
- return type is `GenSolvablePolynomial<C>`
- but `sum()` returns formal type `GenPolynomial<C>` but run time type `GenSolvablePolynomial<C>`
- so one must often use a cast `(GenSolvablePolynomial<C>) p.sum(q)`

Solvable polynomial factory

- same problem with interface implementation as for solvable polynomials
- factory constructor with relation table
  - `GenSolvablePolynomialRing<C>( RingFactory<C> cf, int n, TermOrder t, RelationTable<C> rt)`
- e.g. relation table for Weyl algebras
  - `(new WeylRelations<BigRational>(spfac)). generate()`
- `RelationTable(GenSolvablePolynomialRing<C> r)`
- problem with constructor initialization sequence

Example (cont.)

```
GenSolvablePolynomialRing<BigRational> sfac =
 new GenSolvablePolynomialRing<BigRational>(z,6);

WeylRelations<BigRational> wl =
   new WeylRelations<BigRational>(sfac);
wl.generate();

RelationTable(
( x3 ), ( x0 ), ( x0 * x3 + 1  ),
( x5 ), ( x2 ), ( x2 * x5 + 1  ),
( x4 ), ( x1 ), ( x1 * x4 + 1  )
)
```

Implementation

- 100 classes and interfaces
- plus 50 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

Coefficient implementation

- `BigInteger` based on `java.math.BigInteger`
- implemented like GnuMP library
- using facade 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

Term order implementation

- need comparators for `SortedMap<ExpVector,C>`
- generated from class `TermOrder`
- has methods
  - `getDescendComparator()`
  - `getAscendComparator()`
- implemented all practically used orders
  - (inverse) lexicographical
  - (inverse) graded, i.e. total degree
  - defined by weight matrices
  - elimination orders (split orders)

Solvable polynomial implementation

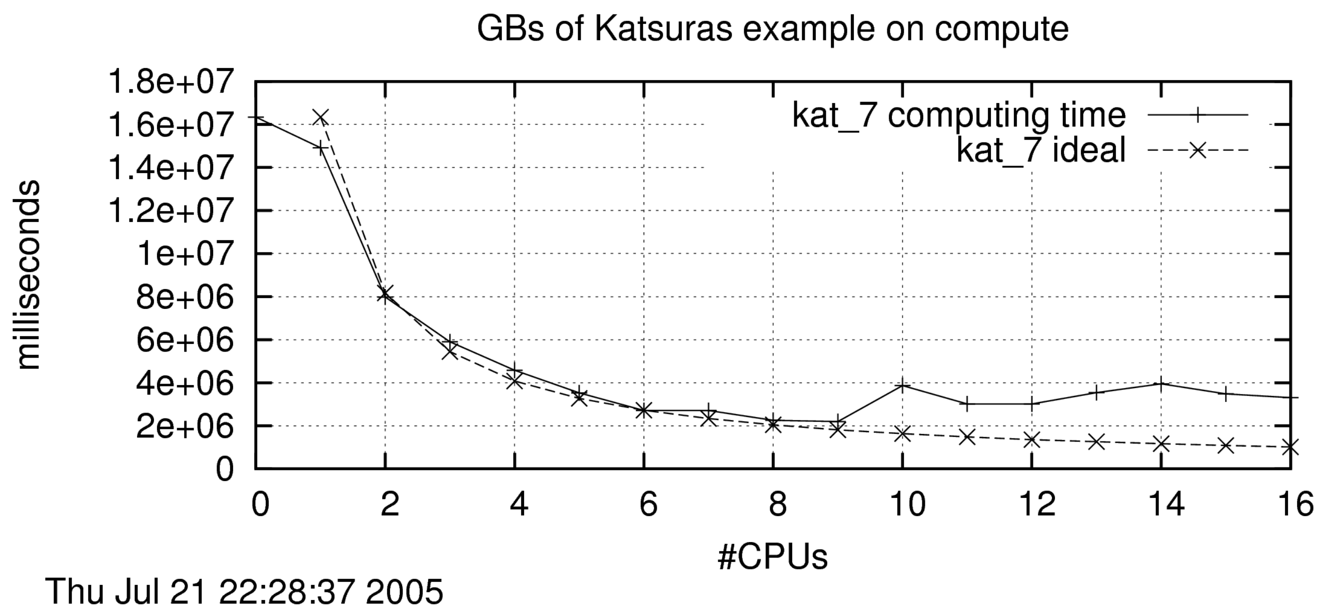
- relations are stored in `RelationTable` object in the factory
- optimized for fast detection of commutative variables $x_j * x_i = x_i x_j$
- overhead to polynomial multiplication is 20%
- table is modified to store relations of powers of variables $x_j^{e_l} * x_i^{e_k} = c_{ikjl} x_i^{e_k} x_j^{e_l} + p_{ikjl}$
- update methods are synchronized
- additive methods are from the superclass

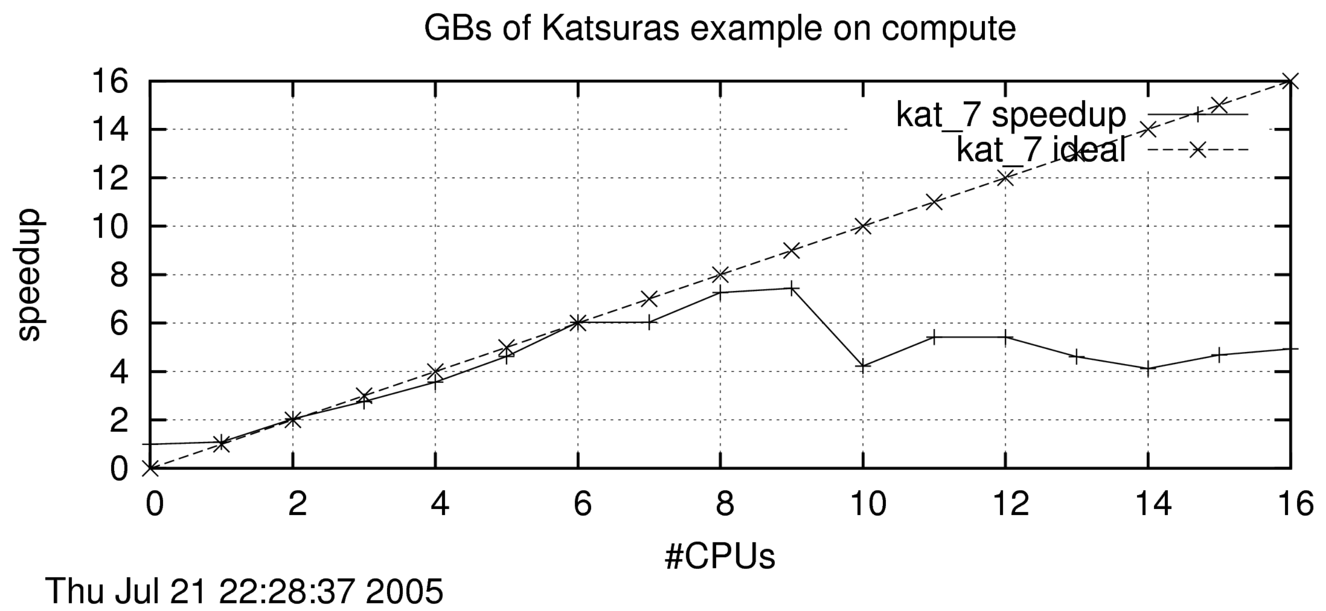
Advanced algorithms

- polynomial reduction (a kind of division with remainder for multivariate polynomials)
- Buchbergers algorithm to compute Groebner bases for sets of polynomials (a kind of Gauss elimination with Euclidean division)
- not much (mathematical) optimization yet, simple structure used also for parallel implementation
- sequential, parallel and distributed versions
- non-commutative left, right and two-sided versions
- modules over polynomial rings and syzygies

Parallel Groebner bases

- work queue of polynomials `CriticalPairList`
- with synchronized methods `get(), put(), removeNext()` to modify data structure
- scales well for 8 CPUs on a well structured problem (see next figures)
- distributed version uses some kind of a distributed list to store polynomials of set (implemented by a DHT)
- use of object serialization for transport of polynomials over the network


GBs of Katsuras example on compute
kat_7 computing time
kat_7 ideal
milliseconds
1.8e+07
1.6e+07
1.4e+07
1.2e+07
1e+07
8e+06
6e+06
4e+06
2e+06
0
0 2 4 6 8 10 12 14 16
#CPUs
Thu Jul 21 22:28:37 2005


GBs of Katsuras example on compute
kat_7 speedup
kat_7 ideal
speedup
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16
#CPUs
Thu Jul 21 22:28:37 2005

Conclusions (1)

- sound object oriented design and implementation of a library for algebraic computations
- type safe trough generic type parameters
- as expressive as categories and domains in Axiom
- multivariate polynomials with multi precision coefficients
- used for a large collection of Groebner base algorithms
- first OO design and implementation of non-commutative polynomials and Groebner bases

Conclusions (2)

- employs various design patterns, e.g. creational patterns (factory), facade pattern
- working horses are from the Java collection framework
- parallel and distributed implementation draw heavily on the Java packages for concurrent programming and networking
- suitability of the design is exemplified by the successful implementation of a large part of `additive ideal theory', e.g. different Groebner base and syzygy algorithms

Conclusions (3)

- Java platform: 64-bit, garbage collection, threads
- Jython wrapper for interactive use
- Problems
  - type erasure in generic interfaces
  - want restrictions on constructors in interfaces
  - quite type safe: polynomials e.g. in 2 and in 3 variables have the same type and at run time an exception will most likely be thrown
- Future
  - more `multiplicative ideal theory', e.g. factorization

Thank you

- Questions?
- Comments?
- http://krum.rz.uni-mannheim.de/jas
- Thanks to
  - Thomas Becker
  - Aki Yoshida
  - the referees
  - all others