On the Design of a Java Computer Algebra System

Heinz Kredel
IT-Center
University of Mannheim
Mannheim, Germany
kredel@rz.uni-mannheim.de

ABSTRACT

This paper considers Java as an implementation language for a starting part of a computer algebra library. It describes a design of basic arithmetic and multivariate polynomial interfaces and classes which are then employed in advanced parallel and distributed Groebner base algorithms and applications. The library is type-safe due to its design with Java's generic type parameters and thread-safe using Java's concurrent programming facilities.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures; G.4 [**Mathematical Software**]: Computer Algebra; I.1 [**Symbolic and Algebraic Manipulation**]: Specialpurpose algebraic systems

General Terms

Design, Algorithms, Type-safe, Thread-safe

Keywords

computer algebra library, multivariate polynomials

1. INTRODUCTION

We describe an object oriented design of a Java Computer Algebra System (called JAS in the following) as type safe and thread safe approach to computer algebra. JAS provides a well designed software library using generic types for algebraic computations implemented in the Java programming language. The library can be used as any other Java software package or it can be used interactively or interpreted through an jython (Java Python) front end. The focus of JAS is at the moment on commutative and solvable polynomials, Groebner bases and applications. By the use of Java as implementation language JAS is 64-bit and multi-core cpu ready. JAS is developed since 2000 (see the weblog in [13]) and was partly described in [12].

Recall form mathematics that a multivariate polynomial p is an element of a polynomial ring R in n variables over some coefficient ring C, i.e. in formal notation $p \in R = C[x_1, \ldots, x_n]$, e.g.

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

is a polynomial in 3 variables over the integers. Note, that the definition is recursive in the sense, that C can be another polynomial ring. More formally a polynomial is a mapping from a monoid T to a ring C, $p = T \longrightarrow C$ where only finitely many elements of T are mapped to non-zero elements of C. In case $R = C[x_1, \ldots, x_n]$ the monoid T, is generated by terms, i.e. (commutative) products of the variables $x_i, i = 1, \ldots, n$. In our example the map is $p =$

$$x_1^2 x_3^4 \mapsto 3, x_2^5 \mapsto 7, x_1^0 x_2^0 x_3^0 \mapsto -61, \text{ else } x_1^{e_1} x_2^{e_2} x_3^{e_3} \mapsto 0.$$

This view is used to implement a polynomial using a `Map` from the Java collection framework. A computer representation of an element of T is used as key and the value is a representation of a (non-zero) element of C, i.e. keys being mapped to zero are not stored in the `Map`. Since some properties of multivariate polynomial rings depend on a certain ordering $<_T$ on the monoid T we actually use a `SortedMap` (with the `TreeMap` implementation). The ordering $<_T$ determines e.g. which monoid in the polynomial is the highest, just as the usual degree does for univariate polynomials. Addition and multiplication of polynomials is defined as usual, the zero polynomial is the empty map and the one polynomial is the map $x_1^0 x_2^0 \ldots x_n^0 \mapsto 1$, where 1 denotes the representation of the one of C. We also consider solvable polynomials which are multivariate polynomials with commutative addition and a non-commutative multiplication $*$ with respect to relations

$$x_j * x_i = c_{ij} x_i x_j + p_{ij},$$

for $1 \le i < j \le n, 0 \ne c_{ij} \in C, x_i x_j >_T p_{ij} \in R$. The (mathematical) class of solvable polynomial rings naturally contains the class of polynomial rings. So a polynomial is always a solvable polynomial with respect to commuting relations. One may then think that a polynomial class (in computer science sense) could extend a solvable polynomial class but it is the other way round.

Based on this sketch of polynomial mathematics the paper describes the basic arithmetic and multivariate polynomial part of a bigger library, which consists of the following additional packages. The package `edu.jas.ring` contains classes for polynomial and solvable polynomial reduction, Groebner bases and ideal arithmetic as well as thread parallel and distributed versions of Buchbergers algorithm like ReductionSeq, GroebnerBaseAbstract, GroebnerBaseSeq, GroebnerBaseParallel and GroebnerBaseDistributed. The package `edu.jas.module` contains classes for module Groebner bases and syzygies over polynomials and solvable polynomials like ModGroebnerBase or SolvableSyzygy. Finally `edu.jas.application` contains classes with applications of Groebner bases such as ideal intersections and ideal quotients implemented in Ideal or SolvableIdeal.

1.1 Related Work

An overview of computer algebra systems and also on design issues can by found in the "Computer Algebra Handbook" [9]. For the scope of this paper the following work was most influential: Axiom [11] and Aldor [22] with their comprehensive type library and category and domain concepts. Sum-It [4] is a type safe library based on Axiom and Weyl [24] presents a concept of an object oriented computer algebra library in Common Lisp. Type systems for computer algebra are proposed by Santas [20] and existing type systems are analyzed by Poll and Thomson [19]. Other library implementations of computer algebra are e.g. LiDIA [5] and Singular [10] in C++ and MAS [14] in Modula-2.

Java for symbolic computation is discussed in [3] with the conclusion, that it fulfills most of the conceptional requirements defined by the authors, but is not suitable because of performance issues (Java up to JDK 1.2 studied). In [1] a package for symbolic integration in Java is presented. A type unsafe algebraic system with Axiom like coercion facilities is presented in [7]. A computer algebra library with maximal use of patterns (object creational patterns, storage abstraction patterns and coercion patterns) are presented by Niculescu [16, 17]. A Java API for univariate polynomials employing a facade pattern to encapsulate different implementations is discussed in [23]. An interesting project is the Orbital library [18], which provides algorithms from (mathematical) logic, polynomial arithmetic with Groebner bases and genetic optimization.

Due to limited space we have not discussed the related mathematical work on solvable polynomials and Groebner base algorithms, see e.g. [2, 8] for some introduction.

1.2 Outline

In section 2 we present an overall view of the design of the central interfaces and classes. We show how part of the Axiom / Aldor basic type hierarchie can be realized. We discuss the usage of creational patterns, such as factory, abstract factory, or prototype in the construction of the library. We do currently not have an explicit storage abstraction and a conversion abstraction to coerce elements from one type to another. However the generic polynomial class `GenPolynomial` applies the facade pattern to hide the user form its complex internal workings. In section 3 we take a closer look at the functionality, i.e. the methods and attributes of the presented main classes. Section 4 treats some aspects of the implementation and discusses the usage of standard Java patterns to encapsulate different implementations of parallel Groebner base algorithms. Finally section 5 draws some conclusions and shows missing parts of the library.

2. DESIGN

One of the first things we have to decide is how we want

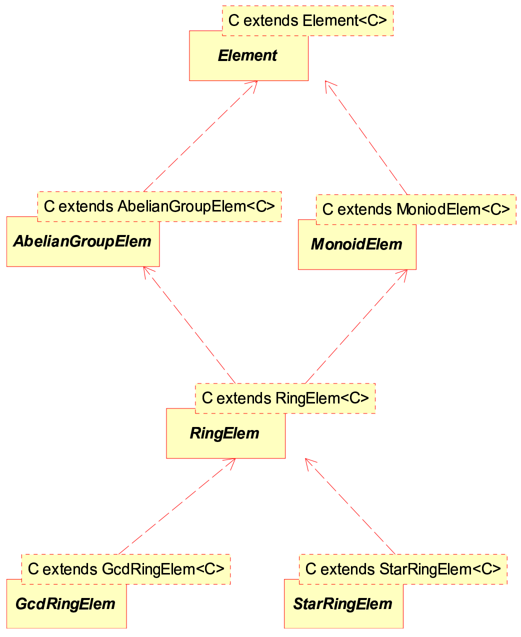


Figure 1: Overview of some algebraic types

to implement algebraic structures and elements of these algebraic structures. Alternatives are i) elements are implemented as (Java or C++) objects with data structure and methods or ii) elements are simple C++ like `structs` or records and algebraic structure functionality is implemented as (static) methods of module like classes. The *second* alternative is more natural to mathematicians, as they perceive algebraic structures as sets (of elements) and maps between such sets. In this view an algebraic structure is a collection of maps (or functions) and a natural implementation is as in FORTRAN as bunches of functions with elements (integers and floats) directly implemented by hardware types. However scientific function libraries implemented in this style are horrible because of the endless parameter lists and the endless repetitions of functions doing the same for other parameter types. The *first* alternative is the approach of computer scientists and it leads to better data encapsulation, context encapsulation and more modular and maintainable code. Since the algebraic elements we are interested in have sufficient internal structure (arbitrary precision integers and multivariate polynomials) we opt for encapsulation with its various software engineering advantages and so choose the first alternative. This reasoning also implies using Java as implementation language, since otherwise we could have used FORTRAN.

2.1 Type structure

Using Java generic types (types as parameters) it is not difficult to specify the interfaces for the most used algebraic types. The interfaces define a type parameter `C` which is required to extend the respective interface. The central interface is `RingElem` (see figures 1 and 3) which extends `AbelianGroupElem` with the additive methods and `MonoidElem` with the multiplicative methods. Both extend `Element` with methods needed by all types. `RingElem` is itself extended by `GcdRingElem` with greatest common divisor methods and `StarRingElem` with methods related to (complex) conjugation. This exemplifies the suitability of Java to implement Axiom / Aldor like type systems, although we do not present such a comprehensive type hierarchy as they do.

2.2 Ring element creation

Figure 2 (see also figure 3) gives an overview of the central classes. The interface `RingElem` defines a recursive type which defines the functionality (see next section) of the polynomial coefficients and is also implemented by the polynomials itself. So polynomials can be taken as coefficients for other polynomials, thus defining a recursive polynomial ring structure.

Since the construction of constant ring elements (e.g. zero and one) has been difficult in our previous designs, we separated the creational aspects of ring elements into ring factories with sufficient context information. The minimal factory functionality is defined by the interface `RingFactory`. Constructors for polynomial rings will then require factories for the coefficients so that the construction of polynomials over these coefficient rings poses no problem. The ring factories are additionally required because of the Java generic type design. I.e. if `C` is a generic type name it is not possible to construct a new object with `new C()`. Even if this would be possible, one can not specify constructor signatures in Java interfaces, e.g. to construct a one or zero constant ring element. Recursion is again achieved by using polynomial factories as coefficient factories in recursive polynomial rings. Constructors for polynomials will always require a polynomial factory parameter which knows all details about the polynomial ring under consideration.

2.3 Polynomials and coefficients

Basic coefficient classes, such as `BigRational` or `BigInteger`, implement both the `RingElem` and `RingFactory` interfaces. This is convenient, since these classes do not need further context information in the factory. In the implementation of the interfaces the type parameter `C extends RingElem<C>` is simultaneously bound to the respective class, e.g. `BigRational`. Coefficient objects can in most cases be created directly via the respective class constructors, but also via the factory methods. E.g. the object representing the number 2 can be created by `new BigRational(2)` or by `fac = new BigRational()`, `fac.fromInteger(2)` and the object representing the rational number 1/2 can be created by `new BigRational(1,2)` or by `fac.parse("1/2")`.

Generic polynomials are implemented in class `GenPolynomial`, which has a type parameter `C` which extends `RingElem<C>` for the coefficient type (see figures 2 and 3). So all operations on coefficients required in polynomial arithmetic and manipulation are guaranteed to exist by the `RingElem` interface. The constructors of the polynomials always require a matching polynomial factory. The generic polynomial factory is implemented in the class `GenPolynomialRing`, again with type parameter `C extends RingElem<C>` (not `RingFactory`). The polynomial factory however implements the interface `RingFactory<C extends RingElem<C>>` so that it can also be used as coefficient factory. The constructors for `GenPolynomialRing` require at least parameters for a coefficient factory and the number of variables of the polynomial ring.

Having generic polynomial and elementary coefficient implementations one can attempt to construct polynomial objects. The type is first created by binding the type parameter `C` to the desired coefficient type, e.g. `BigRational`. So we arrive at the type `GenPolynomial<BigRational>`. Polynomial objects are then created via the respective polynomial factory of type `GenPolynomialRing<BigRational>`, which is created by binding the generic coefficient type of the generic polynomial factory to the desired coefficient type, e.g. `BigRational`. A polynomial factory object is created from a coefficient factory object and the number of variables in the polynomial ring as usual with the new operator via one of its constructors. Given an object `coFac` of type `BigRational`, e.g. created with `new BigRational()`, a polynomial factory object `pf` of the above described type could be created by

```
new GenPolynomialRing<BigRational>(coFac,5).
```

I.e. we specify a polynomial ring with 5 variables over the rational numbers. A polynomial object `p` of the above described type can then be created by any method defined in `RingFactory`, e.g. by `pf.fromInteger(1)`, `pf.getONE()`, `pf.random(3)` or `pf.parse("1")`. See also the example in figure 4.

Since `GenPolynomial` itself implements the `RingElem` interface, they can also be used recursively as coefficients. We continue the polynomial example and are going to use polynomials over the rational numbers as coefficients of a new polynomial. The type is then

```
GenPolynomial<GenPolynomial<BigRational>>
```

and the polynomial factory has type

```
GenPolynomialRing<GenPolynomial<BigRational>>.
```

Using the polynomial coefficient factory `pf` from above a recursive polynomial factory `rfac` could be created by `new`

```
GenPolynomialRing<GenPolynomial<BigRational>>(pf,3)
```

The creation of a recursive polynomial object `r` of the above described type is then as a easy as before e.g. by `rfac.getONE()`, `rfac.fromInteger(1)` or `rfac.random(3)`.

2.4 Solvable polynomials

The generic polynomials are intended as super class for further types of polynomial rings. As one example we take solvable polynomials, which are like normal polynomials but are equipped with a new non-commutative multiplication. From mathematics one would expect that a polynomial class would extend a solvable polynomial class, but it it is the other way, since the multiplication method gets overwritten for non-commutative multiplication. The implementing class `GenSolvablePolynomial` extends `GenPolynomial` (see figures 2 and 3) and inherits all methods except `clone()` and `multiply()`. The class also has a type parameter `C` which extends `RingElem<C>` for the coefficient type. Note, that the inherited methods are in fact creating solvable polynomials since they employ the solvable polynomial factory for the creation of any new polynomials internally. Only the formal method return type is that of `GenPolynomial`, the run-time type is `GenSolvablePolynomial` to which they can be casted as required. The factory for solvable polynomials is implemented by the class `GenSolvablePolynomialRing` which also extends the generic polynomial factory. So this factory can also be used in the constructors of `GenPolynomial` to produce in fact solvable polynomials internally. The data

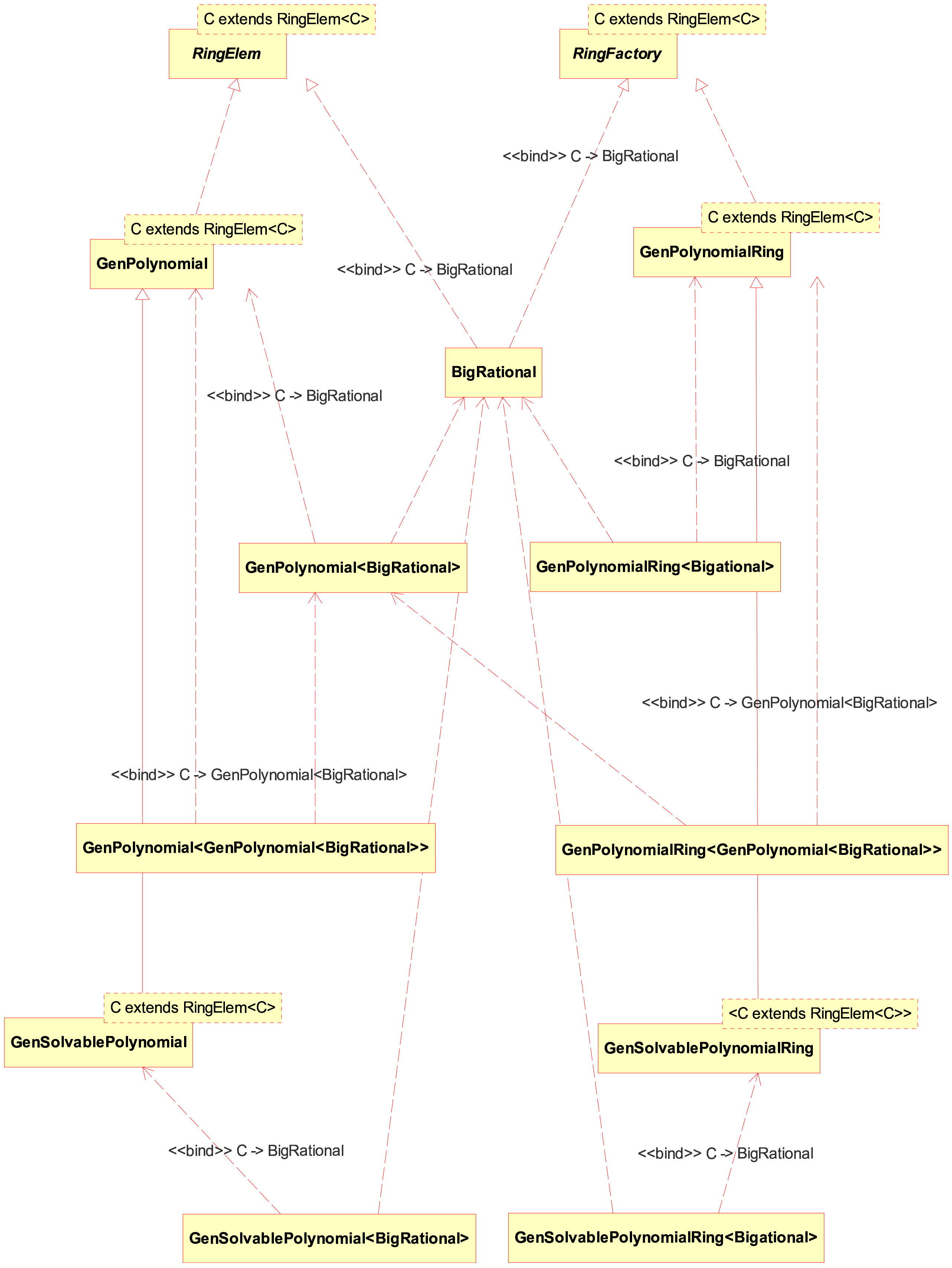

C extends RingElem<C>
RingElem
C extends RingElem<C>
RingFactory
<<bind>> C -> BigRational
C extends RingElem<C>
GenPolynomialRing
C extends RingElem<C>
GenPolynomial
<<bind>> C -> BigRational
BigRational
<<bind>> C -> BigRational
<<bind>> C -> BigRational
GenPolynomial<BigRational>
GenPolynomialRing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

Figure 2: Overview of polynomial types

structure is enhanced by a table of non-commutative relations, called `RelationTable`, defining the new multiplication. The constructors delegate most things to the corresponding super class constructors and additionally have a parameter for the `RelationTable` to be used. Also the methods delegate the work to the respective super class methods where possible and then handle the non-commutative multiplication relations separately.

The construction of solvable polynomial objects follows directly that of polynomial objects. The type is created by binding the type parameter `C` to the desired coefficient type, e.g. `BigRational`. So we have the type `GenSolvablePolynomial<BigRational>`. Solvable polynomial objects are then created via the respective solvable polynomial factory of type

```
GenSolvablePolynomialRing<BigRational>,
```

which is created by binding the generic coefficient type of the generic polynomial factory to the desired coefficient type, e.g. `BigRational`. A solvable polynomial factory object is created as usual from a coefficient factory object, the number of variables in the polynomial ring and a table containing the defining non-commutative relations with the new operator via one of its constructors. Given an object `coFac` of type `BigRational` as before, a polynomial factory object `spfac` of the above described type could be created by `new`

```
GenSolvablePolynomialRing<BigRational>(coFac,5).
```

This defines a solvable polynomial ring with 5 variables over the rational numbers with no commutator relations. A solvable polynomial object of the above described type can then be created by any method defined in RingFactory, e.g. by `spfac.getONE()`, `spfac.fromInteger(1)`, `spfac.parse( "1" )` or `spfac.random(3)`. Some care is needed to create `RelationTable` objects since its constructor requires the solvable polynomial ring which is under construction as parameter (see section 3.3).

3. FUNCTIONALITY OF MAIN CLASSES

In this section we present the methods defined by the interfaces and classes from the proceeding sections. An overview is given in figure 3.

3.1 Ring elements

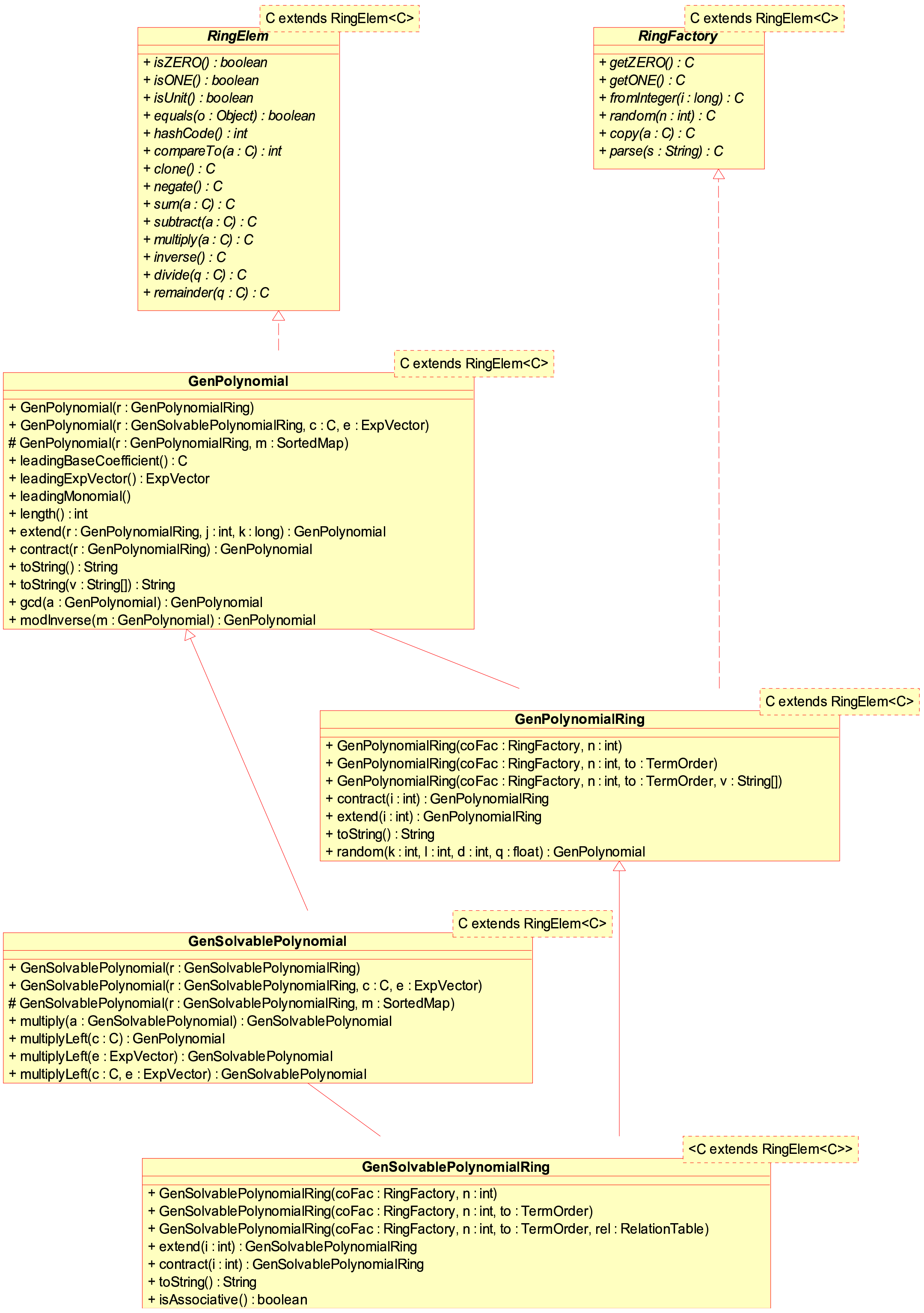
The `RingElem` interface (with type parameter `C`) defines the usual methods required for ring arithmetic such as `C sum(C S)`, `C subtract(C S)`, `C negate()`, `C abs()`, `C multiply(C s)`, `C divide(C s)`, `C remainder(C s)`. `C inverse()`. Although the actual ring may not have inverses for every element or some division algorithm we have included these methods in the definition. In a case where there is no such function, the implementation may deliberately throw a RuntimeException or choose some other meaningful element to return. The method `isUnit()` can be used to check if an element is invertible.

Besides the arithmetic methods there are the following testing methods `boolean isZERO()`, `isONE()`, `isUnit()` and `int signum()`. The first three test if the element is 0, 1 or a unit in the respective ring. The `signum()` method defines the sign of the element (in case of an ordered ring). It is also employed in `toString()` to determine which sign to 'print'. The methods `equals(Object b)`, `int hashCode()` and `int compareTo(C b)` are required to keep Java's object machinery working in our sense. They are used when an element is put into a Java collection class, e.g. `Set`, `Map` or `SortedMap`. The last method `C clone()` can be used to obtain a copy of the actual element. As creational method one should better use the method `C copy(C a)` from the ring factory, but in Java it is more convenient to use the `clone()` method.

As mentioned before, the creational aspects of rings are separated into a ring factory. A ring factory is intended to store all context information known or required for a specific ring. Every ring element should also know its ring factory, so all constructors of ring element implementations require a parameter for the corresponding ring factory. Unfortunately constructors and their signature can not be specified in a Java interface. The `RingFactory` interface also has a generic type parameter `C` which is constrained to a type with the ring element functionality (see figure 3). The defined methods are `C getZERO()`, `C getONE()`, which create 0 and 1 of the ring. The creation of the 1 is most difficult, since for a polynomial it implies the creation of the 1 from the coefficient ring, i.e. we need a factory for coefficients at this point. Then there are methods to embed a natural number into the ring and create the corresponding ring element `C fromInteger(long a)` and `C fromInteger(java.math.BigInteger a)`. The others are `C random (int n)`, `C copy(C c)`, `C parse (String s)`, and `C parse (Reader r)`. The `copy()` method was intended as the main means to obtain a copy of a ring element, but it is now seldom used in our implementation. Instead the `clone()` method is used from the ring element interface. The `random(int n)` method creates a random element of the respective ring. The parameter n specifies an appropriate maximal size for the created element. In case of coefficients it usually means the maximal bit-length of the element, in case of polynomials it influences the coefficient size and the degrees. For polynomials there are `random()` methods with more parameters. The two methods `parse(String s)` and `parse(Reader r)` create a ring element from some external string representation. For coefficients this is mostly implemented directly and for polynomials the class `GenPolynomialTokenizer` is employed internally. In the current implementation the external representation of coefficients may never contain white space and must always start with a digit. In the future the ring factory will be enhanced by methods that test if the ring is commutative, associative or has some other important property or the value of a property, e.g. is an euclidian ring, is a field, an integral domain, a unique factorization domain, its characteristic or if it is Noetherian.

3.2 Polynomials

The `GenPolynomialRing` class has a generic type parameter `C` as already explained (see figure 3). Further the class implements a `RingFactory` over `GenPolynomial<C>` so that it can be used as coefficient factory of a different polynomial ring. The constructors require at least a factory for the coefficients as first parameter of type `RingFactory<C>` and the number of variables in the second parameter. A third parameter can optionally specify a `TermOrder` and a fourth parameter can specify the names for the variables of the polynomial ring. Via `TermOrder` objects the required comparators for the `SortedMap` are produced. Besides the methods required by the `RingFactory` interface there are additional


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
C extends RingElem<C>
GenSolvablePolynomial
+ GenSolvablePolynomial(r : GenSolvablePolynomialRing)
+ GenSolvablePolynomial(r : GenSolvablePolynomialRing, c : C, e : ExpVector)
# GenSolvablePolynomial(r : GenSolvablePolynomialRing, m : SortedMap)
+ multiply(a : GenSolvablePolynomial) : GenSolvablePolynomial
+ multiplyLeft(c : C) : GenPolynomial
+ multiplyLeft(e : ExpVector) : GenSolvablePolynomial
+ multiplyLeft(c : C, e : ExpVector) : GenSolvablePolynomial
<C extends RingElem<C>>
GenSolvablePolynomialRing
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int)
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int, to : TermOrder)
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int, to : TermOrder, rel : RelationTable)
+ extend(i : int) : GenSolvablePolynomialRing
+ contract(i : int) : GenSolvablePolynomialRing
+ toString() : String
+ isAssociative() : boolean

For better readability not all type parameters C are shown.

Figure 3: Overview of class functionality

In this example we show some computations with the polynomial $3x_1^2x_3^4 + 7x_2^5 - 61$ from the introduction.

```
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" );
```

With `toString()` or `toString( ring.getVars() )` the following output is produced. `IGRLEX` is a name for the default term order.

```
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
```

Subtraction and multiplication of polynomials is e.g.

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

with the following output.

```
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
```

Figure 4: Example from the introduction

`random()` methods which provide more control over the creation of random polynomials. They have the following parameters: the bit-size of random coefficients to be used in the `random()` method of the coefficient factory, the number of terms (i.e. the length of the polynomial), the maximal degree in each variable and the density of non-zero exponents, i.e. the ratio of non-zero to zero exponents. The `toString()` method creates a string representation of the polynomial ring consisting of the coefficient factory string representation, the tuple of variable names and the string representation of the term order. The `extend()` and `contract()` methods create 'bigger' respectively 'smaller' polynomial rings. Both methods take a parameter of how many variables are to be added or removed form the actual polynomial ring. `extend()` will setup an elimination term order consisting of two times the actual term order when ever possible.

The `GenPolynomial` class has a generic type parameter `C` as explained above (see figure 3). Further the class implements a `RingElem` over itself `RingElem<GenPolynomial<C>>` so that it can be used for the coefficients of an other polynomial ring. The functionality of the ring element methods has already been explained in the previous section. There are two public and one protected constructors, each requires at least a ring factory parameter `GenPolynomialRing<C> r`. The first creates a zero polynomial `GenPolynomial(r)`, the second creates a polynomial of one monomial with given coefficient and exponent tuple `GenPolynomial(r, C c, ExpVector e)`, the third is protected for internal use only and creates a polynomial from the internal sorted map of an other polynomial `GenPolynomial(r, SortedMap< ExpVector, C > v)`. There is no heavy weight contructor accepting a `Map< ExpVector, C >` parameter. Further there are methods to access parts of the polynomial like leading term, leading coefficient (still called leading base coefficient from some old tradition) and leading monomial. The `toString()` method creates as usual a string representation of the polynomials consisting of exponent tuples and coefficients. One variant of it takes an array of variable names and creates a string consisting of coefficients and products of powers of variables. See the example from the introduction in figure 4. The method `extend()` is used to embed the polynomial into the 'bigger' polynomial ring specified in the first parameter. The embedded polynomial can also be multiplied by a power of a variable. The `contract()` method returns a `Map` of exponents and coefficients. The coefficients are polynomials belonging to the 'smaller' polynomial ring specified in the first parameter. If the polynomial actually belongs to the smaller polynomial ring the map will contain only one pair, mapping the zero exponent vector to the polynomial with variables removed. A last group of methods computes (extended) greatest common divisors. They work correct for univariate polynomials over a field but not for arbitrary multivariate polynomials. These methods will be moved to a new separate class together with a correct implementation for the multivariate case if I find some time.

3.3 Solvable polynomials

The `GenSolvablePolynomial` class also has a generic type parameter `C` as explained above. The class extends the `GenPolynomial` class (see figure 3). It inherits all additive functionality and overwrites the multiplicative functionality with a new non-commutative multiplication method. Unfortunately it cannot implement a `RingElem` over itself

```
RingElem<GenSolvablePolynomial<C>>
```

but can only inherit the implementation of

```
RingElem<GenPolynomial<C>>
```

from its super class. By this limitation a solvable polynomial can still be used as coefficient in another polynomial, but only with the type of its super class. The limitation comes form the erasure of template parameters in `RingElem<...>` to `RingElem` for the code generated. I.e. the generic interfaces become the same after type erasure and it is not allowed to implement the same interface twice. There are two public and one protected constructors as in the super class. Each requires at least a ring factory parameter `GenSolvablePolynomialRing<C> r` which is stored in a variable of type `GenPolynomialRing<C>` shadowing the variable with the same name of the super factory type. Via this mechanism also the super class methods will create solvable polynomials. The rest of the initialization work is delegated to the super class constructor.

The `GenSolvablePolynomialRing` class also has a generic type parameter `C`. It extends `GenPolynomialRing` and overwrites most methods to implement the handling of the `RelationTable`. However it cannot implement a `RingFactory` over `GenSolvablePolynomial<C>` but only a `RingFactory` over `GenPolynomial<C>` by inheritance due to the same reason of type erasure as above. But it can be used as coefficient factory with the type of its super class for a different polynomial ring. One part of the constructors just restate the super class constructors with the actual solvable type. A solvable polynomial ring however must know how to perform the non-commutative multiplication. To this end a data structure with the respective commutator relations is required. It is implemented in the `RelationTable` class. The other part of the constructors additionally takes a parameter of type `RelationTable` to set the initial commutator relation table. Some care is needed to create relation tables and solvable polynomial factories since the relation table requires a solvable polynomial factory as parameter in the constructor. So it is most advisable to create a solvable polynomial factory object with empty relation table and to fill it with commutator relations after the constructor is completed but before the factory will be used. In the above example where `spfac` is a factory for solvable polynomials the relations for a Weyl algebra could be generated as follows

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

There is also a new method `isAssociative()` which tries to check if the commutator relations indeed define an associative algebra. This method should be extracted to the `RingFactory` interface together with a method `isCommutative()`, since both are of general importance and not always fulfilled in our rings. E.g. `BigQuaternion` is not commutative and so a polynomial ring over these coefficients can not be commutative. The same applies to associativity and the class `BigOctonion`.

4. IMPLEMENTATION

Today the implementation consists of about 100 classes and interfaces plus about 50 JUnit classes with unit tests. Logging is provided by the Apache Log4j package. Moreover there are some Jython classes for a more convenient interactive interface.

Basic data types, such as rational numbers, can directly implement both interfaces `RingElem` and `RingFactory` to avoid the separate implementation of factory classes. More complex data types, such as polynomials implement the interfaces in two different classes. Constructors for basic data types can be implemented in any appropriate way. Constructors for more complex data types with separate factory classes should always require one parameter to be of the respective factory type. This is to avoid the creation of elements with no knowledge of is corresponding ring factory. Constructors which require more preconditions, which are only provided by type (internal) methods should not be declared public. It seems best to declare them as protected.

The implementation of basic arithmetic is based on the `java.math.BigInteger` class, which is itself implemented like GnuMP. Multiplication performance was in 2000 approximately 10 to 15 times faster than that of the respective `SACI` module of MAS [14] (see e.g. the the weblog in [13]). Since we require our big integers to implement the `RingElem` interface, we employ the facade pattern for our `BigInteger` class. Beside this, at the moment the following classes are implemented `BigRational`, `ModInteger`, `BigComplex`, `BigQuaternion` and `BigOctonion`. Using (univariate) generic polynomials we provide an `AlgebraicNumber` class, which can be used over `BigRational` or `ModInteger`, i.e. it implements algebraic number rings with zero or finite characteristic.

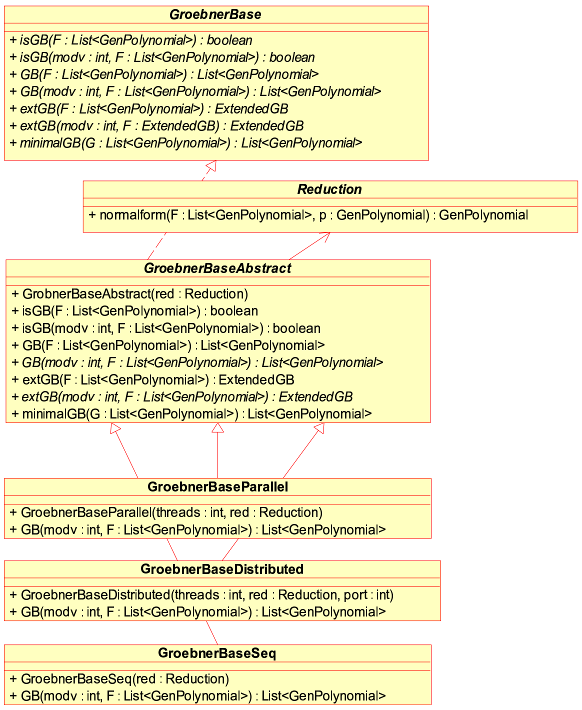

GroebnerBase
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : ExtendedGB) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>
Reduction
+ normalform(F : List<GenPolynomial>, p : GenPolynomial) : GenPolynomial
GroebnerBaseAbstract
+ GrobnerBaseAbstract(red : Reduction)
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : List<GenPolynomial>) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseParallel
+ GroebnerBaseParallel(threads : int, red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseDistributed
+ GroebnerBaseDistributed(threads : int, red : Reduction, port : int)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseSeq
+ GroebnerBaseSeq(red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>

Figure 5: Groebner Base classes

Generic polynomials are implemented as sorted maps from exponent vectors to coefficients. Helper classes are taken from the Java collections framework, i.e. from the package `java.util`. For the implementation of the sorted map the Java class `TreeMap` is taken. An older alternative implementation using `Map`, implemented with `LinkedHashMap`, has been abandoned due to inferior performance. The monoid of terms consists exponent vectors, i.e. the keys of the `Map` are implemented by the class `ExpVector`. There is only one implementation of exponent vectors `ExpVector` as dense Java array of `long`s. Other implementations, e.g. sparse representation or bigger numbers or `int`s are not considered at the moment. The comparators for `SortedMap<ExpVector,C>` are created from a `TermOrder` class, e.g. by method `getDescendComparator()`. `TermOrder` provides `Comparators` for most term orders used in practice: lexicographical, graded and term orders defined by weight matrices. The polynomial objects are intended to be immutable. I.e. the object variables are declared `final` and the map is never modified once it is created. One could also wrap it with `unmodifiableSortedMap()` if desired. This design avoids further synchronization on polynomial methods in parallel algorithms.

As explained above non-commutative polynomials defined with respect to certain commutator relations are extended from `GenPolynomial` respectively `GenPolynomialRing`. The commutator relations are stored in `RelationTable` objects, which are intended to be internal to the `GenSolvablePolynomialRing` since they contain polynomials generated from this factory. The `RelationTable` is optimized for a fast detection of commutative multiplication, i.e. relations of the form $x_j * x_i = x_i x_j$ for some i, j. The overhead of computing commutative polynomials with `GenSolvablePolynomial` objects is approximately 20%. The relation table is eventually modified in synchronized methods if new relations be-

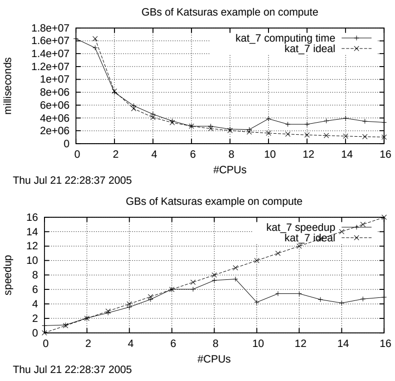


Figure 6: Katsura 7 parallel Groebner base, 16 CPU JDK 1.4, 32 bit JVM, option `UseParallelGC`, Intel XEON 2.7 GHz, 1GB JVM memory, 0 CPUs means sequential

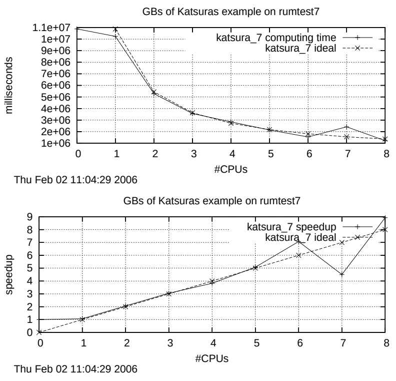


Figure 7: Katsura 7 parallel Groebner base, 8 CPU JDK 1.5, 64 bit JVM, AMD Opteron 2.2 GHz, 1GB JVM memory, 0 CPUs means sequential version

tween powers of variables are computed, e.g. $x_j^{e_j} * x_i^{e_i} = c_{i'j'} x_i^{e_i} x_j^{e_j} + p_{i'j'}$ for some i, j. These new relations are then used to speedup future non commutative multiplications. `GenSolvablePolynomial` implements the non commutative multiplication and uses the additive commutative methods from its super class. As mentioned before, casts are required for the super class methods, e.g.

```
(GenSolvablePolynomial<C>) p.sum(q).
```

The respective objects are however correctly build using the methods from the solvable ring factory.

The class design allows solvable polynomial objects to be used in all algorithms where `GenPolynomials` can be used as parameters as long as no distinction between left and right multiplication is required.

4.1 Groebner bases

As an application of the generic polynomials we have implemented some more advanced algorithms. E.g. polynomial reduction (a kind of multivariate polynomial division

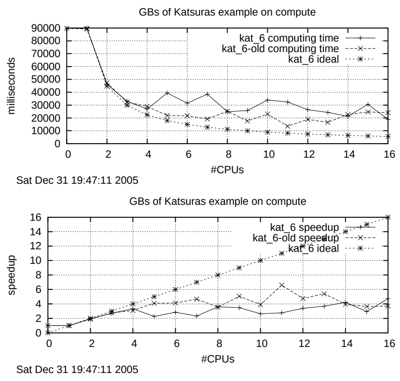


Figure 8: Katsura 6 parallel Groebner base, 16 CPU JDK 1.4, 32 bit JVM, option `UseParallelGC`, Intel XEON 2.7 GHz, 1GB JVM memory, 0 CPUs means sequential

algorithm) or Buchbergers algorithm to compute Groebner bases (a kind of a Gaussian elimination for multivariate polynomials). The algorithms are also implemented for solvable polynomial rings (with left, right and two-sided variants) and modules over these rings. These algorithms are implemented following standard object oriented patterns (see figure 5). There is an interface, e.g. `GroebnerBase`, which specifies the desirable functionality, like `isGB()`, `GB()` or `extGB()`. Then there is an abstract class, e.g. `Groebner-BaseAbstr`, which implements as much methods as possible. It further defines the desirable constructor parameters, e.g. a `Reduction` parameter which sets a polynomial reduction engine with suitable properties. Finally there are concrete classes which extend the abstract class and implement different algorithmic details. E.g. `GroebnerBaseSeq` implements a sequential, `GroebnerBaseParallel` implements a thread parallel and `GroebnerBaseDistributed` implements a network distributed version of the core Groebner base algorithm. In 2003 we compared the Groebner base algorithm to a similar version and implementation of MAS [14]. For the big Trinks example the Java implementation was 8 times faster.

4.2 Parallelism

During the work on [15] we developed the ideas for design of parallel and distributed implementation of core algorithms. The Groebner base computation is done by a variant of the classical Buchberger algorithm. It maintains a data structure, called pair list, for book keeping of the computations (forming S-polynomials and doing reductions). This data structure is implemented by `CriticalPairList` and `OrderedPairList`. Both have synchronized methods `put()` and `getNext()` respectively `removeNext()` to update the data structure. In this way the pair list is used as work queue in the parallel and distributed implementations. The parallel implementations scales well for up to 8 CPUs, for a well structured problem (figures 6 and 7) and up to 4 CPUs on a smaller problem (figure 8). Since the polynomials are implemented as immutable classes no further synchronization is required for the polynomial methods. The distributed implementation makes further use of a distributed list (implemented via a distributed hash table DHT) for the communication of the reduction bases and a distributed thread pool for running the reduction engines in different computers. Java object serialization is used to encode polynomials for network transport. Polynomials are only transfered once to or from a computing node, critical pairs are only transfered using indexes of polynomials in the DHT.

5. CONCLUSIONS

We have provided a sound object oriented design and implementation of a library for algebraic computations in Java. For the first time we have produced a type safe library using generic type parameters. The proposed interfaces and classes are as expressive as the category and domain constructs of Axiom or Aldor, although we have not jet implemented all possible structures. The library provides multivariate polynomials and multiprecision base coefficients which are used for a large collection of Groebner base algorithms. For the first time we have presented an object oriented implementation of non-commutative solvable polynomials and many non-commutative Groebner base algorithms. The library employs various design patterns, e.g. creational patterns (factory and abstract factory) for algebraic element creation. For the main working structures we use the Java collection framework. The parallel and distributed implementation of Groebner base algorithms draws heavily on the Java packages for concurrent programming and internet working. The 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. With the Jython wrapper the library can also be used interactively.

We hope that the problems with type erasure in generic interfaces could be solved in some future version of the Java language. It would also be helpful if there was some way to impose restrictions on constructors in interface definitions.

In the future we will implement more of 'multiplicative ideal theory', i.e. multivariate polynomial greatest common divisors and factorization.

Acknowledgments

I thank Thomas Becker (who is working or thinking about a C++ implementation of a polynomial template library) for discussions on the subject. Further I thank Aki Yoshida from whom I learned much about object oriented programming and design. Thanks also to the anonymous referees for the suggestions to improve the paper.

6. REFERENCES

[1] M. Y. Becker. *Symbolic Integration in Java*. PhD thesis, Trinity College, University of Cambridge, 2001.

[2] T. Becker and V. Weispfenning. *Gröbner Bases - A Computational Approach to Commutative Algebra*. Springer, Graduate Texts in Mathematics, 1993.

[3] L. Bernardin, B. Char, and E. Kaltofen. Symbolic computation in Java: an appraisement. In S. Dooley, editor, *Proc. ISSAC 1999*, pages 237–244. ACM Press, 1999.

[4] M. Bronstein. Sigma$^{\mathrm{it}}$ - a strongly-typed embeddable computer algebra library. In Calmet and Limongelli [6], pages 22–33.

[5] J. Buchmann and T. Pfahler. *LiDIA*, pages 403–408. in Computer Algebra Handbook, Springer, 2003.

[6] J. Calmet and C. Limongelli, editors. *Design and Implementation of Symbolic Computation Systems DISCO '96*, volume 1128 of *Lecture Notes in Computer Science*. Springer, 1996.

[7] M. Conrad. The Java class package com.perisic.ring. Technical report, http://ring.perisic.com/, 2002-2004.

[8] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms*. Springer, Undergraduate Texts in Mathematics, 1992.

[9] J. Grabmaier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook*. Springer, 2003.

[10] G.-M. Greuel, G. Pfister, and H. Schönemann. *Singular - A Computer Algebra System for Polynomial Computations*, pages 445–450. in Computer Algebra Handbook, Springer, 2003.

[11] R. Jenks and R. Sutor, editors. *axiom The Scientific Computation System*. Springer, 1992.

[12] H. Kredel. A systems perspective on A3L. In *Proc. A3L: Algorithmic Algebra and Logic 2005*, pages 141–146. University of Passau, April 2005.

[13] H. Kredel. The Java algebra system. Technical report, http://krum.rz.uni-mannheim.de/jas/, since 2000.

[14] H. Kredel and M. Pesch. *MAS: The Modula-2 Algebra System*, pages 421–428. in Computer Algebra Handbook, Springer, 2003.

[15] H. Kredel and A. Yoshida. *Thread- und Netzwerk-Programmierung mit Java*. dpunkt, 2nd edition, 2002.

[16] V. Niculescu. A design proposal for an object oriented algebraic library. Technical report, Studia Universitatis "Babes-Bolyai", 2003.

[17] V. Niculescu. OOLACA: an object oriented library for abstract and computational algebra. In Vlissides and Schmidt [21], pages 160–161.

[18] A. Platzer. The Orbital library. Technical report, University of Karlsruhe, http://www.functologic.com/, 2005.

[19] E. Poll and S. Thomson. The type system of Aldor. Technical report, Computing Science Institute Nijmegen, 1999.

[20] P. S. Santas. A type system for computer algebra. *J. Symb. Comput.*, 19(1-3):79–109, 1995.

[21] J. M. Vlissides and D. C. Schmidt, editors. *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2004.

[22] S. Watt. *Aldor*, pages 265–270. in Computer Algebra Handbook, Springer, 2003.

[23] C. Whelan, A. Duffy, A. Burnett, and T. Dowling. A Java API for polynomial arithmetic. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 139–144, New York, NY, USA, 2003. Computer Science Press, Inc.

[24] R. Zippel. Weyl computer algebra substrate. In *Proc. DISCO '93*, pages 303–318. Springer-Verlag Lecture Notes in Computer Science 722, 2001.