Comprehensive Gröbner Bases in a Java Computer Algebra System

Heinz Kredel
IT-Center, University of Mannheim, 68131 Mannheim, Germany
kredel@rz.uni-mannheim.de

Abstract

We present an implementation of the algorithms for computing comprehensive Gröbner bases in a Java computer algebra system (JAS). Contrary to approaches to implement comprehensive Gröbner bases with minimal requirements to the computer algebra system, we aim to provide and utilize all necessary algebraic structures occurring in the algorithm. In the implementation of a condition we aim at the maximal semantic exploitation of the occurring algebraic structures: the set of equations equal zero are implemented as an ideal (with Gröbner base computation) and the set of inequalities are implemented as a multiplicative set which is simplified to polynomials of minimal degrees using, for example, square-free decomposition. With our approach we can also make the transition of a comprehensive Gröbner system to a polynomial ring over a (commutative, finite, von Neuman) regular coefficient ring and test or compute Gröbner bases in such polynomial rings.

1 Introduction

In this paper we present an implementation of the algorithms for computing comprehensive Gröbner bases [23, 25, 14, 20] in a Java computer algebra system (JAS) [5, 9, 6, 7].

JAS uses Java to implement a computer algebra library with special emphasis on object oriented programming in an algebraic setting. The emphasis of this paper is also on the library design for comprehensive Gröbner bases. Contrary to approaches to implement comprehensive Gröbner bases with minimal requirements to the computer algebra system, like the one of Suzuki and Sato [21], we aim to provide and utilize all necessary algebraic structures occurring in the algorithm. For example there are parametric polynomials, colored polynomials or coefficients in residue class rings.

In the implementation of a condition we aim at the maximal semantic exploitation of the occurring algebraic structures: the set of equations equal zero are implemented as an ideal (with Gröbner base computation and ideal membership test) and the set of inequalities are implemented as a multiplicative set which is simplified to polynomials of minimal degrees using square-free decomposition or factorization. This approach has partially been taken by [19, 12, 1].

With our approach we can even make the transition of a comprehensive Gröbner system to a polynomial ring over a (commutative, finite, von Neuman) regular coefficient ring and test or compute Gröbner bases in such polynomial rings [22, 24].

1.1 Related Work

Comprehensive Gröbner bases have been introduced by Weispfenning [23] and improved to obtain canonical properties in [25]. Further improvements are achieved by Montes and Manubens [14, 15] and alternative approaches are presented by Sato and Suzuki [20, 21] and [4].

A first implementation comprehensive Gröbner bases was by [19] in ALDES/SAC-2 and MAS, which was improved in [12] and [1]. Newer implementations are presented in [16, 3, 21].

Due to limited space we do not discuss the related mathematical work on Gröbner bases and other computer algebra algorithms, which can be found in standard text books.

1.2 Outline

In the next section 2, we give an an example on using the JAS library. Due to limited space we must assume that you are familiar with the Java programming language, object oriented programming and the JAS type system [9, 11]. Section 3 presents the design of the classes for the implementation of comprehensive Gröbner bases. The topics of the subsections are: conditions and colored polynomials, parametric reduction and colored systems, Gröbner systems and comprehensive Gröbner bases. In section 4 we present some examples with performance measurements and the transition to regular coefficient rings. Finally section 5 draws some conclusions.

2 Introduction to the JAS Library

In this section we discuss an example for the usage of the JAS library. This section contains revised parts of the JAS introduction in [8].

JAS provides a well designed library for algebraic computations implemented with the aid of Java's generic types. 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. JAS implements interfaces and classes for basic arithmetic of arbitrary precision integers, rational numbers and multivariate polynomials with such coefficients. Other packages in JAS are: `edu.jas.ufd` with algorithms for unique factorization domains. `edu.jas.gb` with classes for polynomial and solvable polynomial reduction, Gröbner bases and ideal arithmetic as well as thread parallel and distributed versions of Buchberger's algorithm. `edu.jas.gbmod` contains classes for module Gröbner bases, syzygies for polynomials and solvable polynomials.

For an introduction to the JAS type system see [9, 11]. To get an idea of the interplay of the types, classes and object construction consider the following type

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

of a list of polynomials with coefficients from a direct product of residue class rings modulo some polynomial ideal over the rational numbers. It arises in the computation of Gröbner bases over commutative regular rings $S' = \big(\prod_{\mathfrak{p}\in\mathrm{spec}(R)} R/\mathfrak{p}\big)[y_1,\ldots,y_r]$, where $R = \mathbb{Q}[x_1,\ldots,x_n]$, see [17, 22, 24] and section 4. To keep the example simple we will show how to generate a list `L` of polynomials in the ring

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

The ring S is represented by the object in variable `fac` in the listing in figure 2. Random polynomials of this ring may look like the one shown in figure 1. The coefficients from $(\mathbb{Q}[x_0, x_1, x_2]/\mathrm{ideal}(F))^4$ are shown enclosed in braces `{}` in the form `i=polynomial`. I.e. the index `i` denotes the product component $i = 0, 1, 2, 3$ which reveals that the `Product` class is implemented using a sparse data structure. The list of F is printed after the '`rr =`' together with the indication of the type of the residue class ring `ResidueRing` as polynomial ring in the variables `x0, x1, x2` over the rational numbers `BigRational` with graded lexicographical term order `IGRLEX`. The variables `a, b` are from the 'main' polynomial ring and the rest of figure 1 should be obvious.

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

Figure 1: Random polynomials from ring S

The output in figure 1 is computed by the program from figure 2. Line number 1 defines the variable `L` of our intended type and creates it as an Java `ArrayList`. Lines 2 and 3 show the creation of the base polynomial ring $\mathbb{Q}[x_0, x_1, x_2]$ in variable `pfac`. In lines 4 to 9 the list `F` of random polynomials is constructed which will generate the ideal of the residue class ring. Lines 10 to 13 create a Gröbner basis for the ideal, setup the residue class ring `rr` and print it out. Line 14 constructs the regular ring `pr` as direct product of 4 copies of the residue class ring `rr`. The the final polynomial ring `fac` in the variables `a` and `b` is defined in lines 15 and 16. Lines 17 to 22 then generate the desired random polynomials, put them to the list `L` and print it out.

With this example we see that the software representations of rings snap together like 'LEGO blocks' to build up arbitrary structured rings. This concludes the introduction to JAS, further details can be found, as already mentioned, in [11, 5, 7, 9, 8].

3 Comprehensive Gröbner Bases

Recall some definitions from [23]. Let K be a field, $R = K[U_1, \ldots, U_m]$ a polynomial ring over K in the variables $U_1, \ldots, U_m$. Let $S = R[X_1, \ldots, X_n]$ be a polynomial ring over R in the variables $X_1, \ldots, X_n$ and let $\prec_S$ be a term order on S. S is called a parametric polynomial ring with parameters $U_1, \ldots, U_m$ in the main variables $X_1, \ldots, X_n$. $K[U_1, \ldots, U_m][X_1, \ldots, X_n]$ will be abbreviated by $K[\mathbf{U}][\mathbf{X}]$. For polynomials $f \in S$, the highest term, the leading coefficient, and the leading monomial of f with respect to $\prec_S$ is denoted by $\mathrm{HT}(f)$, $\mathrm{HC}(f)$, and $\mathrm{HM}(f) = \mathrm{HT}(f)\mathrm{HC}(f)$ as usual.

A *specialization* σ of S is a ring homomorphism $\sigma : R \longrightarrow K'$ into some field K'. Let F be a subset of S and let $\mathrm{ideal}(F)$ denote the ideal generated by F. A finite subset $G \subset S$ is a *comprehensive Gröbner base* for $\mathrm{ideal}(F)$ (with respect to $\prec$), if for all fields K' and all specializations $\sigma : R \longrightarrow K'$ of S, $\sigma(G)$ is a Gröbner base for $\mathrm{ideal}(\sigma(F))$ in $K'[X_1, \ldots, X_n]$ (with respect to $\prec$).

```
List<GenPolynomial<Product<Residue<BigRational>>>> L
  = new ArrayList<GenPolynomial<Product<Residue<BigRational>>>>();
BigRational bf = new BigRational(1);
GenPolynomialRing<BigRational> pfac
  = new GenPolynomialRing<BigRational>(bf,3); // no names given
List<GenPolynomial<BigRational>> F
  = new ArrayList<GenPolynomial<BigRational>>();
GenPolynomial<BigRational> pp;
for ( int i = 0; i < 2; i++) {
    pp = pfac.random(5,4,3,0.4f);
    F.add(pp);
}
Ideal<BigRational> id = new Ideal<BigRational>(pfac,F);
id.doGB();
ResidueRing<BigRational> rr = new ResidueRing<BigRational>(id);
System.out.println("rr = " + rr);
ProductRing<Residue<BigRational>> pr
  = new ProductRing<Residue<BigRational>>(rr,4);
String[] vars = new String[] { "a", "b" };
GenPolynomialRing<Product<Residue<BigRational>>> fac
  = new GenPolynomialRing<Product<Residue<BigRational>>>(pr,2,vars);
GenPolynomial<Product<Residue<BigRational>>> p;
for ( int i = 0; i < 3; i++) {
    p = fac.random(2,4,4,0.4f);
    L.add(p);
}
System.out.println("L = " + L);
```

Figure 2: Constructing algebraic objects

Comprehensive Gröbner bases can be computed, for example, via Gröbner systems. A *Gröbner system* $\mathcal{G}$ for an ideal(F), $F \subset S$ is a finite set of pairs (γ, G_γ) where γ is a condition and $G_\gamma \subset S$ is a finite set of polynomials, determined by γ. A comprehensive Gröbner base G for an ideal(F) is then obtained as the union of all G_γ, where each γ also determines F. The meaning of 'condition' and 'determined' is explained next. If in S we have ideal$(F) =$ ideal(G) then G is called a *faithful* comprehensive Gröbner base.

A *condition* γ is a finite set $\{z_i(\mathbf{U}) = 0\} \cup \{n_j(\mathbf{U}) \neq 0\}$ of polynomial equations and inequalities. A *coloring* of the ring R by a condition γ associates a color, namely *green*, *red* and *white*, with each polynomial in R. For $a \in R$, a is colored *green* if $a(\mathbf{U}) = 0$ can be deduced from γ, a is colored *red* if $a(\mathbf{U}) \neq 0$ can be deduced from γ, else a is colored *white*. If a is colored c we write color$(a) = c$. The coloring of R is extended to a coloring of S by the coloring of the coefficients. For $p \in S$ we write $p = p_{green} + p_{red} + p_{white}$ with the restriction $p_{green} \succ p_{red} \succ p_{white}$ for $p_c \neq 0$ (for a color c). Note, that we allow p_{white} to contain *green*, *red* and *white* coefficients, but p_{green} and p_{red} may only contain *green* respectively *red* coefficients, if they are not zero. The wording 'deduced' is left unspecified. It may mean simple inspection of the polynomials in γ or the usage of more sophisticated methods, like ideal membership tests.

A polynomial p is said to be *determined* with respect to a condition γ, if $p_{red} \neq 0$ or if

$p_{red} = 0$ and $p_{white} = 0$. A set F of polynomials is said to be determined wrt. γ, if each $p \in F$ is determined wrt. γ. A polynomial p is said to be determined with respect to a set of conditions Γ, if p is determined wrt. each $\gamma \in \Gamma$.

More on the mathematical background can be found in [23, 25, 24], see also [20, 21, 3, 16].

3.1 Class Layout

We turn now to the algorithms for the computation of comprehensive Gröbner bases in JAS. Due to space restrictions, we must assume some knowledge of Java, object oriented programming and JAS [9, 7, 11] in the following.

The overall layout of the implemented classes is shown in figure 3. The computation of comprehensive Gröbner bases in class `ComprehensiveGroebnerBaseSeq` is done via Gröbner systems, class `GroebnerSystem`. Gröbner systems are implemented as lists of colored systems in class `ColoredSystem`. The colored systems consist of a tuple of a condition in class `Condition`, a list of colored polynomials and data structure `OrderedCPairlist` representing the critical pairs to be considered. Class `ColorPolynomial` implements a polynomial colored with respect to a certain condition.

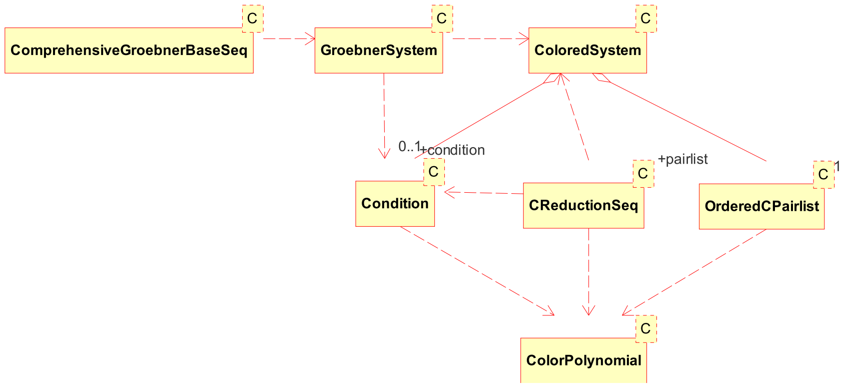

Figure 3: Overview of involved classes

The last class `CReductionSeq` provides methods for parametric reductions relative to conditions and also methods for computing conditions which determine polynomials and sets of polynomials. All classes are parameterized by a type parameter `C` which extends the interface `RingElem<C>`. The implementation is defined for polynomials with polynomial coefficients over a coefficient ring of type `C`, namely `GenPolynomial<GenPolynomial<C>>`.

In the next sub-sections we discuss the functionality of each of the mentioned classes.

3.2 Colored Polynomials and Conditions

Figure 4 shows a class diagram with attributes and methods of the classes `Condition` and `ColorPolynomial`. A condition is defined by a finite set of polynomial equations, polynomials equal to zero $z(\mathbf{U}) = 0$, and a finite set of polynomial inequalities, polynomials not equal to zero $n(\mathbf{U}) \neq 0$. A condition then 'colors' the coefficients of a parametric polynomial in the following way: if a coefficient is contained in the 'equals zero' set, it is colored *green*, if a coefficient is contained in the 'not equals zero' set, it is colored *red*. In case a coefficient is not contained in one of these sets, it is colored *white*. Before we discuss the implementation of these sets, we first explain the rest of the functionality and the colored polynomials.

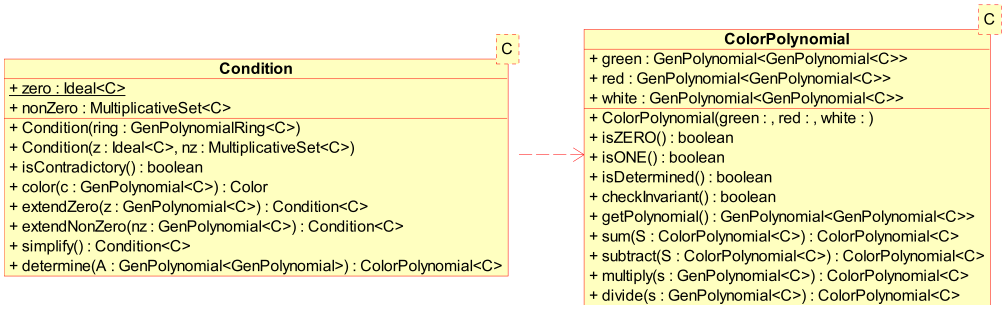
C
Condition
+ zero : Ideal<C>
+ nonZero : MultiplicativeSet<C>
+ Condition(ring : GenPolynomialRing<C>)
+ Condition(z : Ideal<C>, nz : MultiplicativeSet<C>)
+ isContradictory() : boolean
+ color(c : GenPolynomial<C>) : Color
+ extendZero(z : GenPolynomial<C>) : Condition<C>
+ extendNonZero(nz : GenPolynomial<C>) : Condition<C>
+ simplify() : Condition<C>
+ determine(A : GenPolynomial<GenPolynomial>) : ColorPolynomial<C>
C
ColorPolynomial
+ green : GenPolynomial<GenPolynomial<C>>
+ red : GenPolynomial<GenPolynomial<C>>
+ white : GenPolynomial<GenPolynomial<C>>
+ ColorPolynomial(green : , red : , white :)
+ isZERO() : boolean
+ isONE() : boolean
+ isDetermined() : boolean
+ checkInvariant() : boolean
+ getPolynomial() : GenPolynomial<GenPolynomial<C>>
+ sum(S : ColorPolynomial<C>) : ColorPolynomial<C>
+ subtract(S : ColorPolynomial<C>) : ColorPolynomial<C>
+ multiply(s : GenPolynomial<C>) : ColorPolynomial<C>
+ divide(s : GenPolynomial<C>) : ColorPolynomial<C>

Figure 4: Conditions and colored polynomials

The class `Condition` provides the method `color()` to deduce if a given (parametric) coefficient is zero or not with respect to this condition. The method `determine()` takes a parametric polynomial as input an returns a colored polynomial with respect to this condition. The methods `extendZero()` and `extendNonZero()` add a (parametric) coefficient to the set of zero, respectively the set of non-zero, polynomial equations.

A colored polynomial `ColorPolynomial` consists of these three colored parts determined by a condition, with the following restriction on the ordering on the terms. A non-zero green part `green` is greater with respect to the term order of the main variables than a non-zero red part `red`, which is greater than a non-zero white part `white`. In case, one or more of these parts are zero the restriction holds on the remaining non-zero parts. The method `checkInvariant()` provides a test, if these restrictions are fulfilled. The method `isDetermined()` tests if the red part is non-zero or the white part is also zero. Methods `isZERO()` and `isONE()` ignore the green part in performing the respective test. The `getPolynomial()` method returns the sum of all colored parts. The methods `sum()` and `subtract()` compute a colored polynomial which consists of the sum (difference) of the green parts, a zero red part, and a white part computed from the sum (difference) of the given red and white parts. The methods `multiply()` and `divide()` compute a colored polynomial with each colored part multiplied (divided) by a coefficient.

We now turn to the implementation of the sets of equations and inequalities defining a condition. First we do not store the equations them-self, but only the respective polynomials. The implementation is partially inspired by the implementation in [12] which is based on the implementation of [19].

The test if a polynomial is zero, by inspecting a list of polynomials, is not very efficient. For example, the test polynomial might be a linear combination of some polynomials in the list (in other words, it lies in the ideal generated by the polynomials in the list), a fact which is not detected by just inspecting the list. So we replace the list of polynomials by the ideal generated by the list. Then the test if a polynomial is a linear combination of the polynomials is replaced by an ideal membership test. This test can be performed via a normal form computation modulo a Gröbner base of the ideal generated by the polynomials. This functionality is provided by the class `Ideal`. Its method `contains()` lazily computes a Gröbner base if it is required for the ideal membership test. Further we add the square-free part of the polynomials that are put into the ideal, since the test only requires a radical membership test.

Similarly, the test if a polynomial is non-zero can be improved. Instead of just inspecting the list of polynomials if the given polynomial is contained, we can check if the given polynomial is some product of the polynomials in the list. This is done by computing quotients and remainders with respect to polynomials in the list as long as the remainders are zero. If a quotient is constant, the given polynomial was a product of other non-zero polynomials. This algorithms are implemented in class `MultiplicativeSet`. In sub-classes further optimizations are implemented, for example making the polynomials in the set co-prime `MultiplicativeSetCoPrime`, co-prime and square-free `MultiplicativeSet-Squarefree` or irreducible `MultiplicativeSetFactors`. The irreducible factors version relies on the new factorization package, which is not yet in a final state. The default is to use squarefree and co-prime multiplicative sets which are also not too expensive to compute.

The methods `extendZero()` and `extendNonZero()` of `Condition` use the tests just described to avoid adding unnecessary polynomials and to add only maximally reduced polynomials to the respective sets. The methods further try to simplify the condition with method `simplify()` and perform checks for contradictions and return `null` as condition in such a case. Contradictions can show up during the extension operations, as a polynomial in the non-zero list might be contained in the extended ideal generated by the zero polynomials. Similarly a polynomial in the zero polynomial ideal could be a product of polynomials in the extended non-zero polynomials set, again a contradiction. In particular the ideal of zero polynomials might contain 1 at some extension operation. Such contradictory conditions can then be given special treatment in the main part of the algorithm.

3.3 Parametric Reductions and Colored Systems

Class `CReductionSeq` implements parametric reductions with respect to conditions. The class diagram is shown in figure 5. The methods `isNormalform()` and `normalform()` test if a polynomial is in reduced form with respect to a list of polynomials or compute such a reduced form relative to a condition and a list of polynomials. All polynomials are colored polynomials as described above and must be colored consistently and be determined. `isNormalform()` checks if a term with a red coefficient is divisible by a red head term of a polynomial in the list.

The computation of the normal form proceeds by inspecting the first non-green term (of the main variables) in the polynomial to be reduced. If it is actually colored green with respect to the condition, then it is put to the green terms of the result polynomial. If it is colored red or white, the term is reduced with respect to a suitable polynomial in the list. If no such polynomial is found, the process ends for top reduction. For non top-reduction the term is put to the result polynomial and the process continues with the next term. Method `SPolynomial()` computes the S-polynomial of two determined polynomials.

The other methods in class `CReductionSeq` implement the computation of sets of conditions and a list of determined polynomial lists. The method `determine()` takes a list of parametric polynomials as input `List<GenPolynomial<GenPolynomial<C>>>` and returns a list of colored systems `List<ColoredSystem<C>>` (explained further down). The method first computes a set of conditions for the list of input polynomials with method `caseDistinction()` and then determines the polynomials with a method `determine()` which takes a case distinction as input.

A case distinction (a set of conditions) is represented by a list of `Condition` objects. The conditions are constructed in a way, such that every polynomial will have a red head term (or the white part is zero). In the construction of the condition, each (parametric)

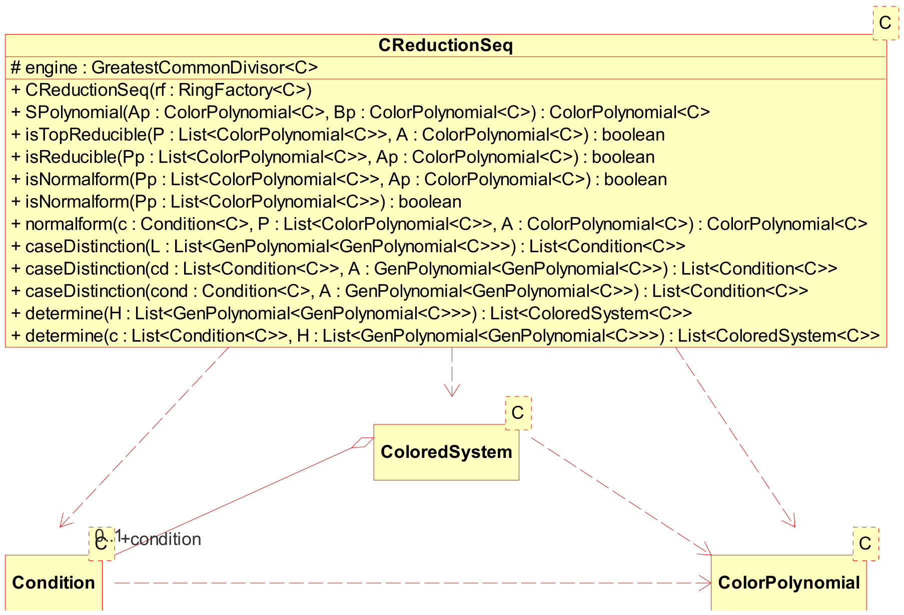
C
CReductionSeq
# engine : GreatestCommonDivisor<C>
+ CReductionSeq(rf : RingFactory<C>)
+ SPolynomial(Ap : ColorPolynomial<C>, Bp : ColorPolynomial<C>) : ColorPolynomial<C>
+ isTopReducible(P : List<ColorPolynomial<C>>, A : ColorPolynomial<C>) : boolean
+ isReducible(Pp : List<ColorPolynomial<C>>, Ap : ColorPolynomial<C>) : boolean
+ isNormalform(Pp : List<ColorPolynomial<C>>, Ap : ColorPolynomial<C>) : boolean
+ isNormalform(Pp : List<ColorPolynomial<C>>) : boolean
+ normalform(c : Condition<C>, P : List<ColorPolynomial<C>>, A : ColorPolynomial<C>) : ColorPolynomial<C>
+ caseDistinction(L : List<GenPolynomial<GenPolynomial<C>>>) : List<Condition<C>>
+ caseDistinction(cd : List<Condition<C>>, A : GenPolynomial<GenPolynomial<C>>) : List<Condition<C>>
+ caseDistinction(cond : Condition<C>, A : GenPolynomial<GenPolynomial<C>>) : List<Condition<C>>
+ determine(H : List<GenPolynomial<GenPolynomial<C>>>) : List<ColoredSystem<C>>
+ determine(c : List<Condition<C>>, H : List<GenPolynomial<GenPolynomial<C>>>) : List<ColoredSystem<C>>
C
ColoredSystem
C
0..1
+condition
Condition
C
ColorPolynomial

Figure 5: Parametric reduction

coefficient of the given polynomial is checked if it is already colored red or green relative to a given condition. If this is not the case, i.e. the coefficient is colored white, the condition is extended two times. First it is extended by adding the coefficient to the set of non-zero polynomials and then it is extended again by adding the coefficient to the set of zero polynomials (as explained in the previous section). If such a newly computed condition is not contradictory and is not already contained in the list of conditions, it is added to the list of conditions.

The two methods `determine()` take a list of parametric polynomials and return a list of `ColoredSystems`, see figure 6. A `ColoredSystem` is a container for a `Condition`, which determines a list of `ColorPolynomials` and a `OrderedCPairlist`. Besides the pair-list which is explained below, the `ColoredSystem` class provides methods similar to the `Color-Polynomial` class. Namely, there are methods to check for the validity of the term order invariants or to check if the list of polynomials is correctly determined. Further there are methods to extract lists of the green or red coefficients, the essential parts or the parametric polynomials them-selfs. Other methods just return respective parts of the condition.

To construct a list of `ColoredSystems`, method `determine()` with a list of `Condition` parameter, uses a list variant of method `determine()` of class `Condition` to compute a list of colored polynomials from the list of the given parametric polynomials. The condition together with the list of determined colored polynomials are then the building parts for the `ColoredSystem` container. The `determine()` method without a `Condition` parameter, first constructs a set of conditions and then constructs a colored system for each condition in the case distinction.

Class `OrderedCPairlist` implements a data structure for the critical pairs to be considered during the curse of the Buchberger algorithm. It encapsulates pair selection strategies and book keeping for criteria to avoid critical pairs.

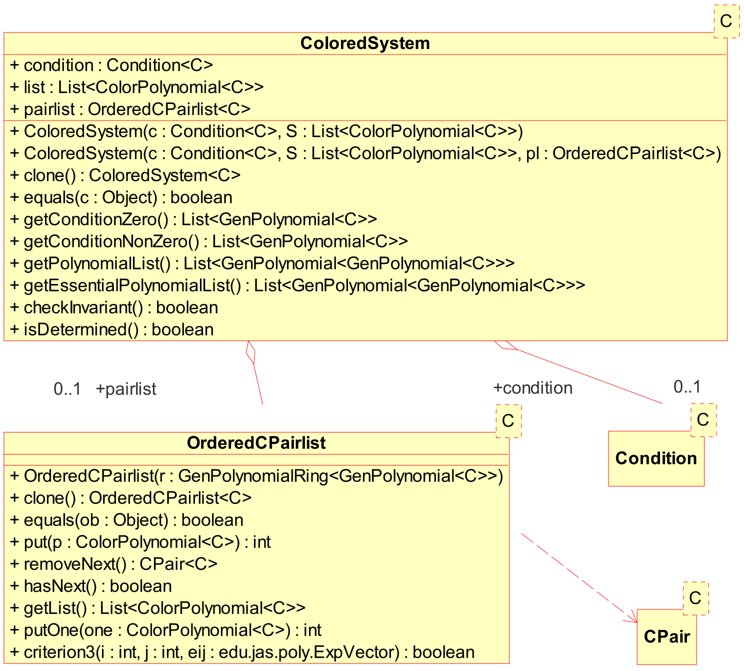
C
ColoredSystem
+ condition : Condition<C>
+ list : List<ColorPolynomial<C>>
+ pairlist : OrderedCPairlist<C>
+ ColoredSystem(c : Condition<C>, S : List<ColorPolynomial<C>>)
+ ColoredSystem(c : Condition<C>, S : List<ColorPolynomial<C>>, pl : OrderedCPairlist<C>)
+ clone() : ColoredSystem<C>
+ equals(c : Object) : boolean
+ getConditionZero() : List<GenPolynomial<C>>
+ getConditionNonZero() : List<GenPolynomial<C>>
+ getPolynomialList() : List<GenPolynomial<GenPolynomial<C>>>
+ getEssentialPolynomialList() : List<GenPolynomial<GenPolynomial<C>>>
+ checkInvariant() : boolean
+ isDetermined() : boolean
0..1 +pairlist
+condition
0..1
C
OrderedCPairlist
+ OrderedCPairlist(r : GenPolynomialRing<GenPolynomial<C>>)
+ clone() : OrderedCPairlist<C>
+ equals(ob : Object) : boolean
+ put(p : ColorPolynomial<C>) : int
+ removeNext() : CPair<C>
+ hasNext() : boolean
+ getList() : List<ColorPolynomial<C>>
+ putOne(one : ColorPolynomial<C>) : int
+ criterion3(i : int, j : int, eij : edu.jas.poly.ExpVector) : boolean
C
Condition
C
CPair

Figure 6: Colored systems and critical pair lists

3.4 Gröbner Systems and Comprehensive Gröbner Bases

A `GroebnerSystem` is a container for a list of `ColoredSystems`, see figure 7. Like class `ColoredSystem`, it has a method `isDetermined()` to test if all contained colored systems are determined and a method `checkInvariant()` to check all invariants of all contained colored polynomials. The method `getConditions()` extracts a list of all `Conditions` from all `ColoredSystems` and stores them for later access in attribute `conds`. The Method `get-CGB()` extracts a list of all parametric polynomials as a union of all parametric polynomials from all colored systems.

The computation of comprehensive Gröbner bases via Gröbner systems is implemented in class `ComprehensiveGroebnerBaseSeq`. This class has methods to test if a given list of parametric polynomials is a comprehensive Gröbner base (method `isGB()`) and to test if a given list of colored systems is a Gröbner system (method `isGBsys()`). Both methods are over-loaded to allow also `GroebnerSystems` as parameters and perform the respective checks. Internally there exist two tests, if a list of parametric polynomials is a comprehensive Gröbner base. `isGBcol()` determines the given list of polynomials and calls method `isGBsys()` on the list of `ColoredSystems`. The second test `isGBsubst()` also determines the given list of polynomials but then maps the polynomials to each residue class modulo the zero polynomial ideal contained in the `Condition` and test if it is a Gröbner base over these coefficient rings. That is, we transform the polynomials from

`GenPolynomial<GenPolynomial<C>>` to `GenPolynomial<Residue<C>>`

and use method `isGB()` from implementation `GroebnerBasePseudoSeq` for the test, that is, we map $K[U_1,\ldots,U_m][X_1,\ldots,X_n] \longrightarrow K[U_1,\ldots,U_m]_{/\mathrm{ideal}(Z_i)}[X_1,\ldots,X_n]$, where Z_i is

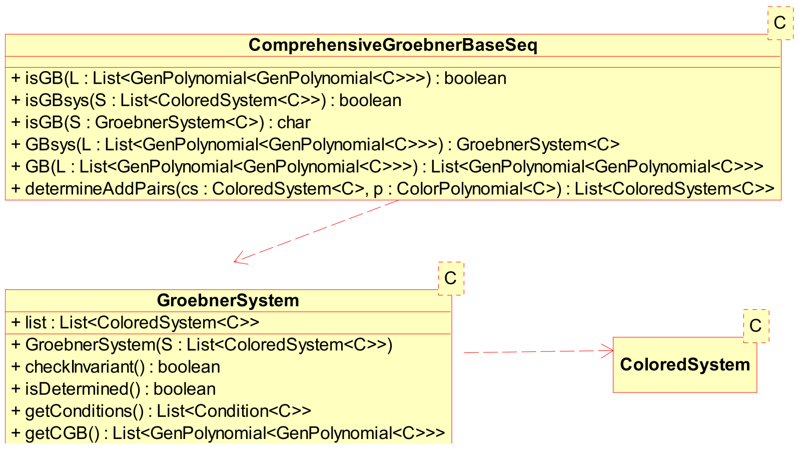
C
ComprehensiveGroebnerBaseSeq
+ isGB(L : List<GenPolynomial<GenPolynomial<C>>>) : boolean
+ isGBsys(S : List<ColoredSystem<C>>) : boolean
+ isGB(S : GroebnerSystem<C>) : char
+ GBsys(L : List<GenPolynomial<GenPolynomial<C>>>) : GroebnerSystem<C>
+ GB(L : List<GenPolynomial<GenPolynomial<C>>>) : List<GenPolynomial<GenPolynomial<C>>>
+ determineAddPairs(cs : ColoredSystem<C>, p : ColorPolynomial<C>) : List<ColoredSystem<C>>
C
GroebnerSystem
+ list : List<ColoredSystem<C>>
+ GroebnerSystem(S : List<ColoredSystem<C>>)
+ checkInvariant() : boolean
+ isDetermined() : boolean
+ getConditions() : List<Condition<C>>
+ getCGB() : List<GenPolynomial<GenPolynomial<C>>>
C
ColoredSystem

Figure 7: Gröbner systems and comprehensive Gröbner bases

the set of polynomials to be treated as zero in condition i. As one last test a random ideal in the coefficient polynomial ring is generated (an ideal generated by random polynomials) and the test is performed modulo this random ideal. Note, that the ideal(Z_i) of the zero conditions might not be a prime ideal. However, the head terms of the polynomials have moreover been colored red by the multiplicative set of non-zero conditions. And, if the condition is not a contradiction, it is guaranteed that they miss all prime ideals which contain the residue class ideal.

To compute a faithful comprehensive Gröbner base, the method `GB()` first computes a `GroebnerSystem` and then extracts the comprehensive Gröbner base with `getCGB()`.

The main work is performed in method `GBsys()`, which takes a list of parametric polynomials as input an returns a `GroebnerSystem` container. In this method, first the method `determine()` of the parametric reduction engine is used to construct a list of determined colored systems. Each `ColoredSystem` is then augmented by a pair list `OrderedCPairlist` containing all critical pairs of the colored polynomials of it. For each `ColoredSystem`, an inner loop iterates over all critical pairs of this system. For each critical pair a parametric S-polynomial and a parametric normalform of it with respect to the list of colored polynomials is computed. If the normal-form polynomial is non-zero, the condition is refined so that it becomes determined with method `determineAddPairs()`. This method also adds new pairs to the critical pair list if required. The method returns a list of `ColoredSystems`, consisting of successors of the actual condition, an updated list of colored polynomials and an updated list of critical pairs. This list of new `ColoredSystems` is then merged with the existing list of `ColoredSystems` and the actual `ColoredSystem` is replaced by a suitable new system. In this way, a depth first search for a `ColoredSystem` with empty pair list is performed. If all critical pairs of the actual `ColoredSystem` are done, it is moved to the result list and the next colored system is taken. By the termination argument for the computation of Gröbner systems only finitely many new colored systems are added at each step and for each colored system only finitely many new critical pairs are generated. So by Königs tree lemma combined with Dickson's lemma the two interleaved loops eventually terminate. Upon termination all critical pairs in a colored system have been processed, so the polynomials form a Gröbner base relative to the given condition. Since the conditions of the colored system cover the empty condition, the list of colored systems form a Gröbner system for the list of given input polynomials.

4 Examples and Gröbner bases over regular rings

In this section we report on some performance measurements and relations to Gröbner bases for regular rings. We first show the performance on the examples from Raksanyi and Hawes2, see [2]. The examples are contained in the `examples` directory of [11].

Table 1: Gröbner system for Raksanyi and Hawes examples

example	MAS time	conditions	JAS time	conditions
Raksanyi, S, Gr	40	3	520 / 229 / 190	5
Raksanyi, Lr	not impl.	–	344 / 134 / 94	3
Raksanyi, L	5630	22	511 / 225 / 175	4
Raksanyi, G	30	3	337 / 147 / 99	3
Hawes2, G	> 20 min	–	1119 / 603 / 578	5

time in milliseconds, Term order: G = graded, L = lexicographical, S = Gr = reverse graded, Lr = reverse lexicographical, timings in slashes are for subsequent runs.

In table 1 we compared the computation with MAS [12] on the same computer. For JAS we compute the same Gröbner system three times in the same instance of a Java virtual machine. The time for the second and third run is separated by a slash. We see, that for the first run there is considerable time spend in JVM code profiling and just-in-time compilation. In subsequent runs we then see performance improvements. In most cases, the computing times for the third run are less than half to one third of the computing for the first run. We see that for small examples the MAS code runs faster, but for bigger examples the JAS code runs faster.

Table 2: Gröbner system for Nabeshima examples

example	from [16]	cond	JAS, AMD, L	cond	JAS, AMD, G	cond
F_1	31	4	285 / 151 / 97	7	270 / 142 / 99	7
F_2	93	6	2299 / 1765 / 1664	12	509 / 281 / 165	10
F_3	2203	22	1186 / 720 / 660	29	1199 / 967 / 681	29
F_4	234	15	1231 / 722 / 674	34	1365 / 845 / 751	34
F_5	109	6	359 / 184 / 126	11	367 / 187 / 125	8
F_6	359	17	95 / 43 / 34	4	90 / 42 / 34	4
F_7	375	7	392 / 194 / 117	6	424 / 242 / 128	6
F_8	133200	458	2548 / 1856 / 1788	32	4883 / 4043 / 3664	32

time in milliseconds, Term order: G = graded, L = lexicographical, cond = number of conditions, timings in slashes are for subsequent runs.

In table 2 we present the JAS computation times of the examples from Nabeshima. The timings of Nabeshima are the original timings from the article [16]. These timings are measured on a Pentium M running at 1.73 GHz. The computing times for JAS are in milliseconds on one (older) AMD 2.1 GHz CPU, with JDK 1.6 and 32-bit server VM. So the timings are not directly comparable, but the CPU influence should not be more than a factor of two. Nevertheless we can draw the conclusion, that the CPU or the system software (C, Risa/Asir versus Java, JAS) is qualitatively in the same speed region and the timing differences seem to be mainly caused by different algorithms. That is, the mathematical optimization to produce a minimal number of conditions, is more important than the relative CPU speed.

Table 3: Gröbner system for Montes examples

example	JAS time	conditions	DISPGB time	conditions
11.1, L	777 / 308 / 327	23	8800	6
11.2, L	490 / 246 / 143	10	5200	6
11.3, L	1013 / 600 / 516	9	115900	7
11.4, L	371939 / 359274 / 355794	7	33000	7
5.1 simpl., L	248 / 95 / 86	3	8400	4

time in milliseconds, Term order: G = graded, L = lexicographical, timings in slashes are for subsequent runs. DISPGB times from [14].

In table 3 we present the JAS computation times of the examples from Montes. The timings of Montes are the original timings from [14]. As in the examples above we conclude, that the different algorithms are most important for the different computing times. As it is not the primary focus of this paper to compare different algorithmic details the timings show that our object oriented approach with Java is not slower than other approaches.

As pointed out in [17, 18, 24] there is some strong relation between comprehensive Gröbner bases and Göbner bases over (von Neumann) regular rings [22]. Since we also have Gröbner bases over (finite) regular rings implemented in JAS, we can check, for example, if a comprehensive Gröbner base is indeed also a Gröbner base over a suitable regular ring.

As with method `isGBsubst()` we take a list of colored polynomials from `GroebnerSystem` (which could be a Gröbner system). From the condition of each colored system we construct a residue class ring modulo the ideal generated from the condition zero polynomials. The (finite) product of these residue class rings are then used as coefficient ring for a polynomial ring with type

```
GenPolynomialRing<Product<Residue<C>>>
```

or in mathematical notation $\left(\prod_{i=1}^{k} K[U_1,\ldots,U_m]_{/\mathrm{ideal}(Z_i)}\right)[X_1,\ldots,X_n]$, where k is the number of colored systems. See section 2 for the construction of the required polynomial rings. Then the union of the parametric polynomials from the colored polynomials is mapped to this polynomial ring. The described conversion is implemented by the (static) method `toProductRes()` of class `PolyUtilApp`. The boolean closure of such a list of polynomials is constructed by method `booleanClosure()` of class `RReductionSeq`. The computation of Gröbner bases for a regular ring is provided via class `RGroebnerBasePseudoSeq<C>` and method `GB()`. The test for a Gröbner base is implemented by method `isGB()`. So if we start with a boolean closed set derived from a comprehensive Gröbner system, the method `isGB()` of the regular coefficient ring Gröbner base will return `true`. An example is contained in the jython file `examples/raksanyi_cr.py` in [11]. Note, as mentioned above, the ideal(Z_i) of the zero conditions might not be a prime ideal. However the head terms of the polynomials have also been colored red by the multiplicative set of non-zero conditions, so it is guaranteed that they miss all prime ideals which contain the residue class ideal, if the condition is not contradictory.

5 Conclusions

We have presented an implementation of the algorithms for computing comprehensive Gröbner bases in a Java computer algebra system (JAS). We provide and utilize all necessary algebraic structures occurring in the comprehensive Gröbner bases algorithm, such as parametric polynomials, colored polynomials, conditions or colored systems. A condition is implemented as an ideal, with normal Gröbner base computations to decide ideal membership and a multiplicative set which is targeted to produce polynomials of minimal degrees using square-free decomposition.

The computing times for our object oriented approach using Java are at least as fast as the times of other implementations. Differences in the computing times are from different mathematical details, which have not been the primary focus of investigation in this paper. With our explicit algebraic types approach we showed how to transform a comprehensive Gröbner system to a polynomial ring over a (commutative, finite, von Neuman) regular coefficient ring and test for Gröbner bases in such polynomial rings.

In the future we will finish the implementation of multivariate polynomial factorization and use it in the handling of the conditions. Further we plan to implement comprehensive Gröbner bases for parametric solvable polynomial rings [13]. There are also many opportunities for utilizing parallelism, see [3] and [10] for a start.

Acknowledgments

I thank Thomas Becker for discussions on the implementation of a polynomial template library and Raphael Jolly for the discussions on the generic type system suitable for a computer algebra system. Thanks also to Markus Aleksy and Hans-Günther Kruse for encouraging and supporting this work. JAS itself has improved by requirements from Axel Kramer and Brandon Barker and by valuable feedback from other colleagues, in particular by Dongming Wang, Thomas Sturm and Wolfgang K. Seiler, to name a few. Finally I thank the anonymous referees for suggestions to improve the paper.

References and Notes

[1] A. Dolzmann and Th. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bull.*, 31(2):2–9, 1997.

[2] Hans-Gert Gräbe. The SymbolicData project. Technical report, see http://www.symbolicdata.org, accessed 2007, June, 2000–2006.

[3] Shutaro Inoue and Yosuke Sato. On the parallel computation of comprehensive Gröbner systems. In *Proc. PASCO'07*, pages 99–101, 2007.

[4] D. Kapur. An approach for solving systems of parametric polynomial equations. In *Principles and Practices of Constraint Programming*, pages 217–244. MIT Press, Cambridge, Mass., 1995.

[5] H. Kredel. On the Design of a Java Computer Algebra System. In *Proc. PPPJ 2006*, pages 143–152. University of Mannheim, 2006.

[6] H. Kredel. Evaluation of a Java Computer Algebra System. In *Proceedings ASCM 2007*, pages 59–62. National University of Singapore, 2007.

[7] H. Kredel. Evaluation of a Java computer algebra system. *Lecture Notes in Computer Science, Springer Berlin / Heidelberg*, 5081:121–138, 2008.

[8] H. Kredel. Multivariate greatest common divisors in the Java Computer Algebra System. In *Proc. Automated Deduction in Geometry (ADG)*, pages 41–61. East China Normal University, Shanghai, 2008.

[9] H. Kredel. On a Java Computer Algebra System, its performance and applications. *Science of Computer Programming*, 70(2-3):185–207, 2008.

[10] H. Kredel. Distributed parallel Gröbner base computation. In *Proc. Workshop on Engineering Complex Distributed Systems (ECDS) at CISIS 2009*, pages on CD–ROM. University of Fukuoka, Japan, 2009.

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

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

[13] Heinz Kredel. *Solvable Polynomial Rings.* Dissertation, Universität Passau, 1993.

[14] A. Montes. An new algorithm for discussing Gröbner basis with parameters. *J. Symb. Comput.*, 33(1-2):183–208, 2002.

[15] A. Montes and M. Manubens. Improving DISPGB algorithm using the discriminant ideal. *J. Symb. Comput.*, 41:1245–1263, 2006.

[16] Katsusuke Nabeshima. A speed-up of the algorithm for computing comprehensive Gröbner systems. In *Proc. ISSAC 2007*, pages 299–306, 2007.

[17] Y. Sato and A. Suzuki. Gröbner bases in polynomial rings over von Neumann regular rings – their applications. In *Proceedings ASCM 2000*, pages 59–62. World Scientific Publications, Lecture Notes Series on Computing, 8, 2000.

[18] Yosuke Sato, Akira Nagai, and Shutaro Inoue. On the computation of elimination ideals of boolean polynomial rings. In *ASCM*, pages 334–348, 2007.

[19] Elke Schönfeld. Parametrische Gröbnerbasen im Computer Algebra System ALDES / SAC-2. Diplomarbeit, Universität Passau, Passau, 1991.

[20] Akira Suzuki and Yosuke Sato. An alternative approach to comprehensive Gröbner bases. *J. Symb. Comput.*, 36(3-4):649–667, 2003.

[21] Akira Suzuki and Yosuke Sato. A simple algorithm to compute comprehensive Gröbner bases using Gröbner bases. In *Proc. ISSAC 2006*, pages 326–331, 2006.

[22] V. Weispfenning. Gröbner bases for polynomial ideals over commutative regular rings. In H.J. Davenport, editor, *Proc. ISSAC'87*, pages 336–347. Springer Verlag, 1987.

[23] V. Weispfenning. Comprehensive Gröbner bases. *J. Symb. Comp.*, 14(1):1–29, 1992.

[24] V. Weispfenning. Comprehensive Gröbner bases and regular rings. *J. Symb. Comput.*, 41:285–296, 2006.

[25] Volker Weispfenning. Canonical comprehensive Gröbner bases. In *ISSAC 2002*, pages 270–276. ACM, 2002.