Modular Methods for Computation of Gröbner Bases

Gerald Slacik
Research Institute for Symbolic Computation (RISC)
Johannes Kepler Universität Linz, Austria

September 23, 1997

Modular Methods for Computation of Gröbner Bases

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur der Technischen Mathematik"

Eingereicht von

Gerald Slacik

September 1997

Angefertigt am Institut für Mathematik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eingereicht bei Univ.-Doz. Dr. Franz Winkler

Acknowledgments

I would to thank Univ.-Doz. Dr. Franz Winkler for the guidance and great patience. Without his kindness to review my work even on weekends I would have been lost. If this work took longer than expected than it is my fault.

Finally, I would like to thank my parents, Christa and Franz Slacik, who enabled me to do my studies and in this way laid the foundation-stone for all my scientific work.

Abstract

We present several attempts to improve the computation speed of Gröbner bases. In particular we give an algorithm for a p-adic computation of Gröbner bases which uses transformation and syzygy matrices for lifting. This algorithm, which was devised by F. Winkler, solves systems of linear equations over polynomials over a Galois field in the lifting step. This is done by computation of a Gröbner basis modulo a prime p. Then we give an improved version of this method which solves linear equations over a Galois field. This replaces the computations of Gröbner bases modulo a prime p by Gaussian elimination modulo p.

Another approach presented is to use the transformation matrix $T^{(0)}$ such that $T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}$. We can compute the normed reduced Gröbner basis G for F in $\mathbb{Q}[x_1, \ldots, x_\nu]$ by solving a system of linear equations over $\mathbb{Q}$ for every polynomial in G.

Finally we present an algorithm which uses the trace of the Gröbner basis computation modulo p to repeat the computation over $\mathbb{Q}$ avoiding all unnecessary S-polynomial computations and reductions to 0. This algorithm was devised by C. Traverso. Again, a slightly modified version is also presented which tracks only the computation of coefficients instead of polynomials avoiding all computations of zero coefficients.

We give the results of practical experiments for these algorithms in the final chapter.

Zusammenfassung

Wir stellen einige Versuche vor, die Berechnung von Gröbner Basen zu beschleunigen. Zuerst stellen wir einen Algorithmus von F. Winkler vor, der mit Hilfe von Transformations und Syzygien Matrizen eine p-adische Näherung berechnet, die auf $\mathbb{Q}$ abgebildet wird. Der Algorithmus löst ein System von linearen Gleichungen von Polynomen über einem Galois Körper um die p-adische Näherung zu berechnen. Dazu werden Gröbner Basen benutzt, die modulo p berechnet werden. Wir beschreiben dann eine verbesserte Version dieses Algorithmus, der ein System von linearen Gleichungen über einem Galois Körper löst. Dadurch wird die Berechnung von Gröbner Basen modulo p durch das Eliminationsverfahren von Gauß ersetzt.

Weiters wird ein Algorithmus beschrieben, der die Transformationsmatrix $T^{(0)}$ benutzt, wobei $T^{(0)}$ die Bedingung $T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}$ erfüllt. Wir können nun die normierte reduzierte Gröbner Basis G in $\mathbb{Q}$, durch Lösen eines linearen Gleichungssystem über $\mathbb{Q}$ für jedes Polynom in G, berechnen.

Zuletzt beschreiben wir einen Algorithmus, der den Pfad, der zur der Gröbner Basis modulo p führt, benutzt um die Berechungen über $\mathbb{Q}$ zu wiederholen. Dabei wird die Berechnung nicht notwendiger S-Polynome (d.h. S-Polynome, die zu 0 reduziert werden können) vermieden. Der Algorithmus stammt von C. Traverso. Weiters wird eine Variante vorgestellt, die nur die Berechnung der Koeffizienten anstatt der Polynome wiederholt und so die Berechnung aller Nullkoeffizienten vermeidet.

Die Ergebnisse der Experimente mit diesen Algorithmen wird im letzten Kapitel präsentiert.

Contents

Chapter 1

Introduction

Practical computing experience on Gröbner bases has shown that computing with rational numbers or integers, very frequently one has very large coefficients in the intermediate computations, and that often the final result is of more moderate size. Trinks remarks in [Trinks, 1984]: "Dealing with $K = \mathbb{Q}$, we usually start with a system $f_1, \ldots, f_m$ having coefficients of modest size, but during the algorithm the size of coefficients tends to increase and often makes results unattainable due to space and time limits."

p-adic methods help to control the enormous growth of coefficients in various problems in computer algebra, such as the computation of greatest common divisors of multivariate polynomials over the integers ([Moses and Yun, 1973], [Miola and Yun, 1974]), and factorization of multivariate integral polynomials ([Zassenhaus, 1969], [Musser, 1975], [Wang and Rothschild, 1975], [Wang, 1978]). For an introduction to p-adic lifting and an overview of applications we refer to [Lauer, 1983].

The analogy to Gcd computations suggest that modular methods may be useful. We pursue several ideas for exploiting the knowledge gained by the computation of a Gröbner basis modulo a prime p to find the Gröbner basis over $\mathbb{Q}$.

In Chapter 2 we give the definition of a Gröbner basis and an algorithm to compute a Gröbner basis from an arbitrary basis. We also give definitions of transformation and syzygy matrices and algorithms to compute them. Furthermore we show how to solve systems of linear equation over $\mathbf{K}[x_1, \ldots, x_\nu]$ with Gröbner bases. This will be used in the following chapters.

Winkler (in [Winkler, 1987b] and [Winkler, 1988]) treated the problem of p-adic computation of Gröbner basis. In Chapter 3 we present his solution. The structure of this chapter is as follows. Section 3.1 deals with the coefficients we want to approximate, the *Farey rationals*. We give an algorithm to obtain the coefficient in $\mathbb{Q}$ of a p-adic approximation of the coefficient. Section 3.2 contains a short discussion of lucky primes, i.e. those primes with respect to which the p-adic approximation of a Gröbner basis is possible. In Section 3.3 existence and uniqueness of such an approximation are investigated, giving rise to the lifting algorithm of Section 3.4. First the normed reduced Gröbner basis $G^{(0)}$ for F in $\mathbb{Z}_p$ and the transformation matrix $R^{(0)}$ and syzygy matrix $S^{(0)}$ are computed. Then the congruences

$$\begin{array}{l} R^{(0)} \cdot G^{(0)} \equiv F \\ S^{(0)} \cdot G^{(0)} \equiv 0 \end{array} \qquad (\text{mod } p)$$

are lifted to the congruences

$$\begin{array}{l} R^{(i)} \cdot G^{(i)} \equiv F \\ S^{(i)} \cdot G^{(i)} \equiv 0 \end{array} \qquad (\text{mod } p^{i+1}).$$

The differences G', R' and S'

$$\begin{aligned} G' &= \frac{1}{p^i} \cdot (G^{(i)} - G^{(i-1)}) \\ R' &= \frac{1}{p^i} \cdot (R^{(i)} - R^{(i-1)}) \\ S' &= \frac{1}{p^i} \cdot (S^{(i)} - S^{(i-1)}) \end{aligned}$$

are obtained by solving a system of linear equations of polynomials over $\mathbb{Z}_p$. Solving such a system is an application of Gröbner bases. Lifting these congruences to congruences modulo p^i for a big enough i and using the algorithm given in Section 3.1 we finally get the normed reduced Gröbner basis for F in $\mathbb{Q}[x_1, \ldots, x_\nu]$.

In Chapter 4 we present an improvement of Winkler's algorithm where the differences G', R' and S' can be obtained by solving a system of linear equations over $\mathbb{Z}_p$. This can be done by using the Gaussian elimination algorithm which has a much better runtime behavior than the Gröbner basis algorithm. The idea is to restrict the notion of luckiness of a prime p so that no new power products are introduced in G', R' and S'. Then we can make an ansatz for G', R' and S' by introducing new indeterminates for the coefficients in $G^{(0)}, R^{(0)}$ and $S^{(0)}$. Using the congruences

$$\begin{array}{lcl} (R^{(i-1)} + p^i \cdot R') \cdot (G^{(i-1)} + p^i \cdot G') & \equiv & F \\ (S^{(i-1)} + p^i \cdot S') \cdot (G^{(i-1)} + p^i \cdot G') & \equiv & 0 \end{array} \qquad (\text{mod } p^{i+1})$$

we obtain a system of linear equations for the new indeterminates by comparing coefficients.

These two chapters try to avoid the growth of the coefficients in the intermediary results. The following chapters do not primarily try to tame the coefficient growth but try to avoid unnecessary computations of S-polynomials (i.e. S-polynomial that can be reduced to 0).

In Chapter 5 we use the transformation matrix $T^{(0)}$ such that

$$T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}$$

to obtain the normed reduced Gröbner basis for F in $\mathbb{Q}[x_1, \ldots, x_\nu]$. Here we again use a prime p such that

$$\left.\begin{array}{rcl} T & = & T^{(0)} \\ G & = & G^{(0)} \end{array}\right. \pmod{p}$$

for some transformation matrix T and the Gröbner basis G over $\mathbb{Q}$ such that no coefficient in T and G vanishes modulo p. We make an ansatz T' and G' for T and G by introducing new indeterminates for the coefficients in $T^{(0)}$ and $G^{(0)}$. Then we get a system of linear equations over $\mathbb{Q}$ for every row in the equation $T' \cdot F = G'$.

Traverso (in [Traverso, 1988]) uses the trace obtained from the Gröbner basis computation modulo p to repeat the computation over $\mathbb{Q}$ avoiding the computation of all unnecessary S-polynomials. In Chapter 6 we present his algorithm and a variant where we trace only the coefficients instead of the polynomials.

In the final Chapter 7 we discuss our implementation in Oberon and the experimental results.

Chapter 2

Gröbner bases

In this chapter we give the definition of a Gröbner basis and an algorithm for constructing one. Our approach to Gröbner bases is based on [Buchberger, 1965] and [Buchberger, 1970]. Furthermore we give the definitions of the transformation matrices of a Gröbner basis and the syzygy matrix of a Gröbner basis. Finally we show how to use Gröbner bases for solving systems of linear equations over $\mathbf{K}[x_1, \ldots, x_\nu]$ and how the notion of Gröbner bases can be generalized for modules.

Throughout this paper we assume that $\mathbf{R}$ is a commutative ring, $\mathbf{R}[x_1, \ldots, x_\nu]$ the ring of polynomials in $x_1, \ldots, x_\nu$ over $\mathbf{R}$. $\mathbf{K}$ is a field, $\mathbf{K}[x_1, \ldots, x_\nu]$ the ring of polynomials in $x_1, \ldots, x_\nu$ over $\mathbf{K}$. We will abbreviate $[x_1, \ldots, x_\nu]$, the monoid of power products of the form $x_1^{i_1} \cdots x_\nu^{i_\nu}$, by $[X]$ and accordingly $\mathbf{R}[x_1, \ldots, x_\nu]$ by $\mathbf{R}[X]$ and $\mathbf{K}[x_1, \ldots, x_\nu]$ by $\mathbf{K}[X]$. If F is any subset of $\mathbf{K}[X]$ we write Ideal(F) for the ideal generated by F in $\mathbf{K}[X]$.

$$\text{Ideal}(F) = \left\{ \sum_{i=1}^{n} h_i \cdot f_i \mid 0 \leq n, h_i \in \mathbf{K}[X], f_i \in F \right\}.$$

In fact, from Hilbert's basis theorem ([Hilbert, 1890], [van der Waerden, 1967]) we know that every ideal in $\mathbf{K}[X]$ has a finite basis.

We will use the following typed variables: $f, g, h \in \mathbf{K}[X]$, $m, n \epsilon \mathbb{N}$, $i, j, k, l \in \mathbb{N} \cup \{0\}$.

2.1 Definition of Gröbner basis

Our definition of a Gröbner basis will be based on the notion of "reduction". For this it is necessary to fix a total ordering $\lhd$ of the power products.

Definition 1 Let $\lhd$ be an ordering on $[X]$ that is compatible with the monoid structure under multiplication, i.e.

$$\begin{aligned} 1 = x_1^0 \cdots x_\nu^0 \lhd t \quad & \text{for all } t \in [X] \setminus \{1\}, \\ s \lhd t \Rightarrow s \cdot u \lhd t \cdot u \quad & \text{for all } s, t, u \in [X]. \end{aligned}$$

We call such an ordering $\lhd$ an *admissible ordering.* □

Example 1 Some examples of admissible orderings are:

1. The *lexicographic ordering* with $x_{\pi(1)} \lhd x_{\pi(2)} \lhd \cdots \lhd x_{\pi(\nu)}$, π a permutation of $\{1, \ldots, \nu\}$:

   $$x_1^{i_1} \cdots x_\nu^{i_\nu} \lhd_{L,\pi} x_1^{j_1} \cdots x_\nu^{j_\nu} \quad \text{iff} \quad \begin{array}{l} \text{there exists a } k \in \{1, \ldots, \nu\} \text{ such that} \\ \text{for all } l < k : i_{\pi(l)} = j_{\pi(l)} \text{ and } i_{\pi(k)} < j_{\pi(k)}. \end{array}$$

   We get the usual *purely lexicographic* ordering by setting $\pi = id$ (which is $1 \lhd x \lhd x^2 \lhd x^3 \lhd \cdots \lhd y \lhd xy \lhd x^2y \lhd x^3y \lhd \cdots \lhd y^2 \lhd xy^2 \lhd \cdots$ in the case of two variables).

2. The *graduated lexicographic ordering* with regard to the permutation π and the weight function $w : \{1, \ldots, \nu\} \to \mathbb{R}^+$:

   $$x_1^{i_1} \cdots x_\nu^{i_\nu} \lhd_{G,\pi,w} x_1^{j_1} \cdots x_\nu^{j_\nu} \quad \text{iff} \quad \begin{array}{l} \displaystyle\sum_{k=1}^{\nu} w(k) \cdot i_k < \sum_{k=1}^{\nu} w(k) \cdot j_k \text{ or} \\ \displaystyle\sum_{k=1}^{\nu} w(k) \cdot i_k = \sum_{k=1}^{\nu} w(k) \cdot j_k \text{ and} \\ x_1^{i_1} \ldots x_\nu^{i_\nu} \lhd_{L,\pi} x_1^{j_1} \ldots x_\nu^{j_\nu}. \end{array}$$

   We get the *total degree ordering* by setting $\pi = id, w = 1_{const}$ (which is $1 \lhd x \lhd y \lhd x^2 \lhd xy \lhd y^2 \lhd x^3 \lhd x^2y \lhd xy^2 \lhd y^3 \lhd \cdots$ in the case of two variables).

□

Definition 2 Let $f \in \mathbf{K}[X] \setminus \{0\}$, $F \subseteq \mathbf{K}[X]$, $\lhd$ an admissible ordering.

1. The *set of power products* of f:

   $$\mathrm{pp}(f) := \{t \in [X] \mid \mathrm{coeff}(f, t) \neq 0\}.$$

2. The *leading power product* of f:

   $$\mathrm{lpp}(f) := \max_{\lhd} \mathrm{pp}(f),$$

3. The *trailing power product* of f:

$$\mathrm{tpp}(f) := \min_{\triangleleft} \mathrm{pp}(f),$$

4. The *leading coefficient* of f:

$$\mathrm{lc}(f) := \mathrm{coeff}(f, \mathrm{lpp}(f)),$$

5. The *trailing coefficient* of f:

$$\mathrm{tc}(f) := \mathrm{coeff}(f, \mathrm{tpp}(f)),$$

6. The *leading term* of f:

$$\mathrm{lt}(f) := \mathrm{lc}(f) \cdot \mathrm{lpp}(f),$$

7. The *reductum* of f:

$$\mathrm{red}(f) := f - \mathrm{lt}(f),$$

8. The *set of power products* of F:

$$\mathrm{pp}(F) := \bigcup_{f \in F} \mathrm{pp}(f).$$

9. The *set of leading power products* of F:

$$\mathrm{lpp}(F) := \{\mathrm{lpp}(f) \mid f \in F\}.$$

10. The *set of leading coefficients* of F:

$$\mathrm{lc}(F) := \{\mathrm{lc}(f) \mid f \in F\}.$$

□

Definition 3 Any admissible ordering $\triangleleft$ on $[X]$ gives rise to a partial ordering $\ll$ on $\mathbf{K}[X]$ in the following way:

$$\begin{array}{lll} f \ll g & \text{iff} & f = 0 \text{ and } g \neq 0 \text{ or} \\ & & f \neq 0, g \neq 0 \text{ and } \mathrm{lpp}(f) \triangleleft \mathrm{lpp}(g) \text{ or} \\ & & f \neq 0, g \neq 0 \text{ and } \mathrm{lpp}(f) = \mathrm{lpp}(g) \text{ and } \mathrm{red}(f) \ll red(g). \end{array}$$

□

Lemma 1 *$\ll$ is a Noetherian partial ordering on $\mathbf{K}[X]$.*

One of the central notions of the theory of Gröbner bases is the concept of polynomial reduction, which we introduce now.

Definition 4 Let $F \subseteq \mathbf{K}[X]$, $g, h \in \mathbf{K}[X]$. We say that g *is reducible using* c, t, f ($c \in \mathbf{K}, t \in [X], f \in \mathbf{K}[X]$), written $g \rightarrow_{f,c,t}$ iff

$$\mathrm{coeff}(g, t \cdot \mathrm{lpp}(f)) \neq 0 \text{ and } c = \frac{\mathrm{coeff}(g, t \cdot \mathrm{lpp}(f))}{\mathrm{lc}(f)}.$$

We say that g *reduces to* h *w.r.t.* F, written $g \rightarrow_F h$, iff

$$\text{there are } f \in F, t \in [X], c \in \mathbf{K}, \text{ such that } g \rightarrow_{f,c,t} \text{ and } h = g - c \cdot t \cdot f.$$

If we want to indicate which polynomial, power product and coefficient are used in the reduction, we write $g \rightarrow_{f,c,t} h$. We say that g *is reducible w.r.t.* F, written $g \rightarrow_F$ iff

$$\text{there is a } h \in \mathbf{K}[X] \text{ such that } g \rightarrow_F h$$

We denote the reflexive transitive closure of $\rightarrow_F$ by $\rightarrow_F^*$, i.e.: $g \rightarrow_F^* h$ iff

$$\begin{gathered}\text{there is a finite sequence } g_0, g_1, \ldots, g_m \in \mathbf{K}[X] \text{ such that} \\ g = g_0 \rightarrow_F g_1 \rightarrow_F \ldots \rightarrow_F g_m = h.\end{gathered}$$

□

Hence, roughly, g reduces to h w.r.t. F iff a monomial in g can be deleted by the subtraction of an appropriate multiple $c \cdot t \cdot f$ of a polynomial f in F yielding h. Thus, the reduction may be viewed as one step in a generalized division.

Example 2 Consider a set of polynomials $F := \{f_1, f_2, f_3\}$ in $\mathbb{Q}[x, y, z]$, where

$$\begin{aligned} f_1 &:= -4z^3 + 4z^2x + 16x^2 + 1, \\ f_2 &:= 2y^2x + 4x + 1, \\ f_3 &:= -2zx^2 + 2y^2 + x. \end{aligned}$$

Let $\lhd$ be the graduated lexicographical ordering with $x \lhd y \lhd z$. The leading power products are

$$\begin{aligned} \mathrm{lpp}(f_1) &= z^3, \\ \mathrm{lpp}(f_2) &:= y^2x, \\ \mathrm{lpp}(f_3) &:= zx^2. \end{aligned}$$

The leading coefficients are

$$\begin{aligned} \mathrm{lc}(f_1) &= -4, \\ \mathrm{lc}(f_2) &:= 2, \\ \mathrm{lc}(f_3) &:= -2. \end{aligned}$$

Consider

$$g := 3zy^2x^2 + z^2.$$

W.r.t. F, g reduces, for example, to

$$h := -6zx^2 + z^2 - 3/2zx.$$

Namely,

$$g \to_{f_2, \frac{3}{2}, zt} h.$$

□

Definition 5 Let F be a finite subset of $\mathbf{K}[X]$ and $f, h \in \mathbf{K}[X]$. We say that h is in *normal form* (or *reduced form*) *w.r.t.* F iff

$$\text{there is no } h' \in \mathbf{K}[X] \text{ such that } h \to_F h'.$$

We say that h is a *normal form of* f *w.r.t.* F iff

$$\begin{gathered} \textit{there is a sequence of reductions} \\ f = g_0 \to_F g_1 \to_F g_2 \ldots \to_F g_m = h \\ \textit{and } h \textit{ is a normal form w.r.t. } F. \end{gathered}$$

An algorithm S is called a *normal form algorithm* (or *simplifier*) iff *S(F, g) is a normal form of g w.r.t. F for all finite subsets F of* $\mathbf{K}[X]$ *and for all* $g \in \mathbf{K}[X]$. □

Lemma 2 *The following algorithm is a normal form algorithm:*

Algorithm 1 (NormalForm)

$h \leftarrow$ NormalForm(F, g)
  input: g, a polynomial in $\mathbf{K}[X]$, and F, a finite subset of $\mathbf{K}[X]$.
  output: h, a polynomial in $\mathbf{K}[X]$ such that
      h is a normal form of g w.r.t. F.
BEGIN
  $h := g$
  WHILE there exist $f \in F, c \in \mathbf{K}$, and $t \in [X]$ such that $h \to_{f,c,t}$ DO
    choose f, c, t such that $h \to_{f,c,t}$ and $t \cdot \mathrm{lpp}(f)$ is maximal (w.r.t. $\lhd$)
    $h := h - c \cdot t \cdot f$
  END
END NormalForm.

The correctness of this algorithm should be clear. For the correctness, the selection of the maximal product $t \cdot \mathrm{lpp}(f)$ is not mandatory. However, this choice is of crucial importance for efficiency. The termination of the algorithm is guaranteed by the following lemma:

Lemma 3 *Let F be a finite subset of* $\mathbf{K}[X]$*. Then $\rightarrow_F$ is a noetherian relation (i.e. there is no infinite sequence $g_0 \rightarrow_F g_1 \rightarrow_F \cdots$).*

Proof: see [Buchberger, 1976]. □

Example 3 The polynomial h in the Example 2 can be further reduced:

$$g_1 := h \rightarrow_{f_3,3,1} = z^2 - 3/2zx - 6y^2 - 3x.$$

g_1 is in normal form w.r.t. F: no power product occurring in h is a multiple of the leading power product of one of the polynomials in F. Thus, no reduction is possible. For g another reduction is possible:

$$g_2 := g \rightarrow_{f_3,-3/2,y^2} = 3y^4 + 3/2y^2x + z^2.$$

g_2 is already in normal form w.r.t. F. □

From the example one can see that, in general, it is possible for g to have normal forms g_1 and g_2 w.r.t. F and $g_1 \neq g_2$. Those sets F, for which such a situation does not occur, play a crucial role for many algorithmic solutions of problems in polynomial ideal theory:

Definition 6 [Buchberger, 1965], [Buchberger, 1976] Let $G \subset \mathbf{K}[X]$. G is called a *Gröbner basis* (for the ideal generated by G) iff it satisfies the following condition: Given $g, h_1, h_2 \in \mathbf{K}[X]$ and h_1 and h_2 are normal forms of g w.r.t. F, then $h_1 = h_2$. □

If $F \subset \mathbf{K}[X]$ and G is a Gröbner basis for Ideal(F), then we will also say "G is a Gröbner basis for F". A Gröbner basis of an ideal I is by no means uniquely defined. In fact, whenever F is a Gröbner basis for I and $f \in I$, then also $F \cup \{f\}$ is a Gröbner basis for I.

2.2 Algorithmic Construction of Gröbner bases

We first show how it may be decided whether a given basis F is a Gröbner basis. With the above definition it is not practicable to test if F is a Gröbner basis. However, Buchberger in [Buchberger, 1965] has been able to reduce this test to just testing a finite number of critical situations. For that purpose he has introduced the notion of "S-polynomials".

Definition 7 Let $f, g \in \mathbf{K}[X] \setminus \{0\}$. The *S-polynomial* of f and g is

$$\text{S-pol}(f,g) := \text{lc}(g)\frac{\text{lcm}(\text{lpp}(f),\text{lpp}(g))}{\text{lpp}(f)}f - \text{lc}(f)\frac{\text{lcm}(\text{lpp}(f),\text{lpp}(g))}{\text{lpp}(g)}g.$$

□

Example 4 For f_2 and f_3 in Example 2, the S-pol(f_2, f_3) is

$$zxf_2 + y^2 f_3 = 2y^4 + 4zx^2 + y^2x + zx.$$

□

Note that the least common multiple of lpp(f_1) and lpp(f_2) is the minimal power product that is reducible both modulo f_1 and modulo f_2. The algorithmic criterion for Gröbner bases is formulated in the following theorem, which forms the core of the method for the construction of Gröbner basis:

Theorem 1 (Buchberger's Theorem) *Let F be a finite subset of $\mathbf{K}[X]$ and S an arbitrary normal form algorithm. The following properties are equivalent:*

1. *F is a Gröbner basis.*
2. *For all $f_1, f_2 \in F$: $S(F, \text{S-pol}(f_1, f_2)) = 0$.*

Proof: See [Buchberger, 1965]. □

Buchberger's Theorem gives an algorithm for checking whether a given finite basis is a Gröbner basis: one only has to consider the finitely many pairs f_1, f_2 of polynomials in F, compute the corresponding S-polynomials and see whether they reduce to 0 by application of the normal form algorithm S. In fact, by a simple extension we get (a first and primitive version of) an algorithm for constructing a Gröbner basis.

Algorithm 2 (GröbnerBasis)

$G \leftarrow$ GröbnerBasis(F)
    input: F, a finite subset of $\mathbf{K}[X]$.
    output: G, a finite subset of $\mathbf{K}[X]$ such that
        Ideal(F) = Ideal(G) and G is a Gröbner basis.
BEGIN
    $G := F \setminus \{0\}$
    $B := \{\{g_1, g_2\} \mid g_1, g_2 \in G, g_1 \neq g_2\}$
    WHILE $B \neq \emptyset$ DO

$\{g_1, g_2\} :=$ a pair in B
$B := B \setminus \{\{g_1, g_2\}\}$
$h :=$ S-pol(g_1, g_2)
$h :=$ NormalForm(G, h)
IF $h \neq 0$ THEN
$B := B \cup \{\{g, h\} \mid g \in G\}$
$G := G \cup \{h\}$
END
END
RETURN G
END GröbnerBasis.

Every polynomial h constructed in Algorithm 2 (GröbnerBasis) is in Ideal(F), so Ideal$(F) =$ Ideal(G) throughout GröbnerBasis. Thus, it is obvious from Buchberger's Theorem that GröbnerBasis yields a correct result if it stops. The termination of GröbnerBasis is a consequence of Dickson's Lemma, which says that in $[X]$ there is no infinite chain of elements $s_1, s_2, \ldots$ such that $s_i \not| s_j$ for all $j > 1, 1 \leq i < j$. The leading power products of the polynomials added to the basis form such a sequence in $[X]$, so this sequence must be finite.

From GröbnerBasis it is obvious that every ideal in $\mathbf{K}[X]$ has a Gröbner basis.

Example 5 Let $F = \{f_1, f_2\}$, with $f_1 = x^2y^2 + y - 1, f_2 = x^2y + x$. We compute a Gröbner basis of Ideal(F) in $\mathbb{Q}[x, y]$, w.r.t. the total degree ordering with $x \lhd y$. The following describes one way in which the algorithm could execute (recall that there is a free choice of pairs in the loop):

1. S-pol$(f_1, f_2) = f_1 - yf_2 = -xy + y - 1$, so $G = \{f_1, f_2, f_3\}$, with $f_3 = xy - y + 1$.
2. S-pol$(f_2, f_3) = f_2 - xf_3 = xy \rightarrow_{f_3} y - 1$, so $G = \{f_1, f_2, f_3, f_4\}$, with $f_4 = y - 1$.
3. S-pol$(f_3, f_4) = f_3 - xf_4 = -y + x + 1 \rightarrow_{f_4} x$, so $G = \{f_1, f_2, f_3, f_4, f_5\}$, with $f_5 = x$.

All the other S-polynomials now reduce to 0, so GröbnerBasis terminates with

$$G = \{x^2y^2 + y - 1, x^2y + x, xy - y + 1, y - 1, x\}.$$

□

The Gröbner basis in Example 5 is quite complicated. In fact, $\{y - 1, x\}$ is a Gröbner basis for the ideal considered in Example 5.

There are many different characterizations of Gröbner bases. The following theorem lists a few of them.

Theorem 2 *Let I an ideal in $\mathbf{K}[X]$ and G a finite subset of I. The following conditions are equivalent.*

1. *G is a Gröbner basis for I.*
2. *$f \to_G^* 0$ for all $f \in I$.*
3. *$f \to_G$ for all $f \in I \setminus \{0\}$.*
4. *For all $g \in I$, $h \in \mathbf{K}[X]$: if $g \to_G^* h$ and h is in normal form w.r.t. G then $h = 0$.*
5. *For all $g, h_1, h_2 \in \mathbf{K}[X]$: if $g \to_G^* h_1$ and $g \to_G^* h_2$ and h_1 and h_2 are in normal form w.r.t. G then $h_1 = h_2$.*
6. *For all $g_1, g_2 \in G$: $\text{S-pol}(g_i, g_j) \to_G^* 0$.*

Proof: see for instance [Winkler, 1996]. □

Theorem 3 *Let G be a Gröbner basis for an ideal I in $\mathbf{K}[X]$. Let $g, h \in G$ with $g \neq h$.*

1. *If $\mathrm{lpp}(g) | \mathrm{lpp}(h)$ then $G' = G \setminus \{h\}$ is also a Gröbner basis for I.*
2. *If $h \to_g h'$ then $G' = (G \setminus \{h\}) \cup \{h'\}$ is also a Gröbner basis for I.*

Proof: see [Winkler, 1996]. □

This leads us to the following definition:

Definition 8 Let G be a Gröbner basis in $\mathbf{K}[X]$.

G is *minimal* iff $\mathrm{lpp}(g) \not| \mathrm{lpp}(h)$ for all $g, h \in G$ with $g \neq h$.

G is *reduced* iff for all $g, h \in G$ with $g \neq h$, h cannot be reduced by g.

G is *normed* iff $\mathrm{lc}(g) = 1$ for all $g \in G$. □

Theorem 4 *Every ideal in $\mathbf{K}[X]$ has a unique normed reduced Gröbner basis.*

Proof: see [Winkler, 1996]. □

The most expensive operations in the algorithm GröbnerBasis are the reductions of h w.r.t. to G in the WHILE -loop. In Example 5 there were 10 S-polynomials, but in order to arrive at the Gröbner basis we considered only 3 of them. So one obvious question seems to be: is there a way of testing in advance whether an S-polynomial will be *necessary*?

Theorem 5 (Product Criterion) *Let F a finite subset of* $\mathbf{K}[X]$, $f_1, f_2 \in F$. *If* $\mathrm{lpp}(f_1) \cdot \mathrm{lpp}(f_2) = \mathrm{lcm}(\mathrm{lpp}(f_1), \mathrm{lpp}(f_2))$ *then* $\text{S-pol}(f_1, f_2) \to_F^* 0$.

Proof: see [Buchberger, 1965] or [Buchberger, 1970]. □

Theorem 6 (Chain Criterion) *Let $G \subset \mathbf{K}[X]$. G is a Gröbner basis iff for all $\{g_1, g_2\} \in G \times G$ there exists $h_0, \ldots, h_m \in G$ such that*

$$\begin{aligned}
&g_1 = h_0, g_2 = h_m, \\
&\mathrm{lcm}(\mathrm{lpp}(h_0), \ldots, \mathrm{lpp}(h_m)) | \mathrm{lcm}(\mathrm{lpp}(g_1), \mathrm{lpp}(g_2)), \\
&\textit{for all } 0 \leq i < m : \text{S-pol}(h_i, h_{i+1}) \to_F^* 0.
\end{aligned}$$

Proof: see [Winkler and Buchberger, 1979]. □

Testing for long chains is costly. So usually one tests only for short chains with one intermediate polynomial.

This leads to an improved version of GröbnerBasis algorithm.

Algorithm 3 (Improved GröbnerBasis)

$G \leftarrow$ GröbnerBasis(F)
  input: F, a finite subset of $\mathbf{K}[X]$.
  output: G, a finite subset of $\mathbf{K}[X]$ such that
      Ideal(F) = Ideal(G) and G is a Gröbner basis.
BEGIN
  $G := F \setminus \{0\}$
  $B := \{\{g_1, g_2\} \mid g_1, g_2 \in G, g_1 \neq g_2\}$
  WHILE $B \neq \emptyset$ DO
    $\{g_1, g_2\} :=$ a pair in B such that $\mathrm{lcm}(\mathrm{lpp}(g_1), \mathrm{lpp}(g_2))$ is minimal w.r.t. $\triangleleft$
    $B := B \setminus \{\{g_1, g_2\}\}$
    IF $\neg$ProductCriterion(g_1, g_2) & $\neg$ChainCriterion(g_1, g_2, G, B) THEN
      $h :=$ S-pol(g_1, g_2)
      $h :=$ NormalForm(G, h)
      IF $h \neq 0$ THEN
        $B := B \cup \{\{g, h\} \mid g \in G\}$
        $G := G \cup \{h\}$
      END
    END
  END
  RETURN G
END GröbnerBasis.

Algorithm 4 (ProductCriterion)

```
prod ← ProductCriterion(g1, g2)
   input: g1, g2 ∈ K[X].
   output: prod a flag indicating if the product criterion is true for g1, g2.
BEGIN
   RETURN lpp(g1) · lpp(g2) = lcm(lpp(g1), lpp(g2))
END ProductCriterion.
```

Algorithm 5 (ChainCriterion)

```
chain ← ChainCriterion(g1, g2, G, B)
   input: G, a finite subset of K[X],
          g1, g2 ∈ G,
          B, a set of pairs of polynomials in G.
   output: chain a flag indicating if the chain criterion is true for g1, g2, G, B.
BEGIN
   FOR all h ∈ G DO
      IF
            h ≠ g1 & h ≠ g2
         &
            lpp(h)|lcm(lpp(g1), lpp(g2))
         &
            {h, g1} ∉ B & {h, g2} ∉ B
      THEN
         RETURN TRUE
      END
   END
   RETURN FALSE
END ChainCriterion.
```

2.3 Transformation and Syzygy Matrices

For convenience we will often write a basis F for a polynomial ideal as a vector $(f_1, \ldots, f_m)^T$ rather than a set $\{f_i, \ldots, f_m\}$.

Whenever $g_1 \to_F g_2$, $F = (f_1, \ldots, f_m)^T$, then $g_1 - g_2 = (h_1, \ldots, h_m) \cdot F$ for some polynomials $h_1, \ldots, h_m \in \mathbf{K}[X]$. But not every vector $(h_1, \ldots, h_m)$ corresponds to a reduction w.r.t. F.

Definition 9 A vector $(h_1, \ldots, h_m)$ such that $(h_1, \ldots, h_m) \cdot F = 0$ is called a *syzygy* of F. □

The set of syzygies form a module over $\mathbf{K}[X]$, the module of syzygies of F.

Definition 10 Let $F = (f_1, \ldots, f_m)^T$ a finite sequence of polynomials in $\mathbf{K}[X]$ and $G = (g_1, \ldots, g_n)^T$ a Gröbner basis for F. Then there exist *transformation matrices* T and R over $\mathbf{K}[X]$ such that

$$\begin{aligned} G &= T \cdot F \\ F &= R \cdot G \end{aligned}$$

and a *syzygy matrix* S such that

$$S \cdot G = 0$$

and the rows of S constitute a basis for the module of syzygies of G. □

T can be derived by bookkeeping during the computation of the Gröbner basis G. The rows of R can be derived from the reduction of the polynomials of F to 0. The rows of S can be derived from the reduction of the S-polynomials of G to 0.

Problem 1 Given a Gröbner basis $G = \{g_1, \ldots, g_n\} \subset \mathbf{K}[X]$ and a polynomial $f \in \mathrm{Ideal}(G)$. Find $h_1, \ldots, h_n \in \mathbf{K}[X]$ such that $f = \sum_{i=1}^{n} h_i \cdot g_i$ (and $\mathrm{lpp}(h_i \cdot g_i) \lhd \mathrm{lpp}(f)$ for $1 \leq i \leq n$).

Roughly, reduce f to zero w.r.t. G and collect the multiples of the g_i necessary in the reduction. In more detail: take Algorithm 1 (NormalForm) and insert instructions that collect the multiplies of the g_i used in the reduction.

Algorithm 6 (ReductionVector)

$(h_1, \ldots, h_n) \leftarrow$ ReductionVector(f, G)
  input: f, a polynomial in Ideal(G), and
    $G = \{g_1, \ldots, g_n\}$, a finite subset of $\mathbf{K}[X]$ and Gröbner basis.
  output: $(h_1, \ldots, h_n)$, a vector of polynomials in $\mathbf{K}[X]$ such that
    $f = \sum_{i=1}^{n} h_i \cdot g_i$.
BEGIN
  $(h_1, \ldots, h_n) := (0, \ldots, 0)$
  $h := g$
  WHILE $h \neq 0$ DO
    choose i, c, t such that $h \rightarrow_{g_i,c,t}$ and $t \cdot \mathrm{lpp}(g_i)$ is maximal (w.r.t. $\lhd$)
    $h := h - c \cdot t \cdot g_i$
    $h_i := h_i + c \cdot t$
  END
  RETURN $(h_1, \ldots, h_n)$
END ReductionVector.

Problem 2 Given $F = \{f_1, \ldots, f_m\} \subset \mathbf{K}[X]$ and $G = \{g_1, \ldots, g_n\}$ such that $G = \text{GröbnerBasis}(F)$. Find a transformation matrix R with n rows and m columns over $\mathbf{K}[X]$ such that

$$f_j = \sum_{i=1}^{n} R_{i,j} \cdot g_i \text{ for } 1 \leq j \leq m.$$

The j-th row of R consists of $h_1, \ldots, h_n$ that are obtained by ReductionVector for the representation of f_j.

Algorithm 7 (ReductionMatrix)

$R \leftarrow \text{ReductionMatrix}(F, G)$
  input: $F = \{f_1, \ldots, f_m\}$, a finite subset of $\mathbf{K}[X]$, and
    $G = \{g_1, \ldots, g_n\}$, a finite subset of $\mathbf{K}[X]$ and
    $G = \text{GröbnerBasis}(F)$
  output: R, transformation matrix with m rows and n columns over $\mathbf{K}[X]$
    such that $f_j = \sum_{i=1}^{n} R_{i,j} \cdot g_i$ for $1 \leq j \leq m$.
BEGIN
  FOR $1 \leq j \leq n$ DO
    $(R_{1,j}, \ldots, R_{n,j}) := \text{ReductionVector}(f_j, G)$
  END
  RETURN R
END ReductionMatrix.

Problem 3 Given $F = \{f_1, \ldots, f_m\} \subset \mathbf{K}[X]$. Find $G = \{g_1, \ldots, g_n\}$ and T such that $G = \text{GröbnerBasis}(F)$, T is a transformation matrix with n rows and m columns over $\mathbf{K}[X]$ and

$$g_i = \sum_{j=1}^{m} T_{i,j} \cdot f_j \text{ for } 1 \leq i \leq n.$$

Augment Algorithm 2 or Algorithm 3 (GröbnerBasis) by instructions that keep track of the multiplies of f_j that are used in the reduction of those polynomials whose normal form is adjoined to the basis G (compare Algorithm 6 (ReductionVector)).

Problem 4 Given a reduced Gröbner basis $G = \{g_1, \ldots, g_n\}$. Find a syzygy matrix S with n columns over $\mathbf{K}[X]$ such that the finitely many rows of S constitute a set of generators for the linear homogeneous equation

$$0 = \sum_{i=1}^{n} h_i \cdot g_i, \text{ with } h_i \in \mathbf{K}[X] \ \text{ for all } 1 \leq j \leq n,$$

i.e. S should consist of n-tuples $(s_{1,1},\dots,s_{1,n}),\dots,(s_{k,1},\dots,s_{k,n})$ of polynomials such that

$$\sum_{i=1}^{n} s_{j,i}\cdot g_i = 0 \text{ for } 1\le j\le k,$$

and for all $(h_1,\dots,h_n)$ for which

$$\sum_{i=1}^{n} h_i\cdot g_i = 0$$

there exist polynomials $p_1,\dots,p_k\in\mathbf{K}[X]$ such that

$$(h_1,\dots,h_n)=\sum_{j=1}^{k} p_j\cdot(s_{j,1},\dots,s_{j,n}).$$

The rows of S can be obtained by applying Algorithm 6 (ReductionVector) to the S-polynomials of G and extending the result vector by the terms used for the computation of S-polynomial. Using all pairs will give us $\binom{n}{2}$ rows.

Algorithm 8 (SyzygyMatrix)

$S \leftarrow$ SyzygyMatrix(G)
  input: $G=\{g_1,\dots,g_n\}\subset\mathbf{K}[X]$, a reduced Gröbner basis.
  output: S a syzygy matrix with $\binom{n}{2}$ rows and n columns over $\mathbf{K}[X]$
    such that $S\cdot G=0$ and for all $h_1,\dots,h_n\in\mathbf{K}[X]$ for which
    $\sum_{i=1}^{n} h_i\cdot g_i=0$ there exist polynomials $p_1,\dots,p_k\in\mathbf{K}[X]$
    such that $(h_1,\dots,h_n)=\sum_{j=1}^{k} p_j\cdot(s_{j,1},\dots,s_{j,n})$.
BEGIN
  $S :=$ empty matrix
  FOR all pairs (i,j) with $1\le i<j\le n$ DO
    $h := \text{S-pol}(g_i,g_j) = c_j\cdot t_i\cdot g_i - c_i\cdot t_j\cdot g_j$,
      where $c_i=\mathrm{lc}(g_i)$, $c_j=\mathrm{lc}(g_j)$,
        $t=\mathrm{lcm}(\mathrm{lpp}(g_i),\mathrm{lpp}(g_j))$,
        $t_i=t/\mathrm{lpp}(g_i)$, $t_j=t/\mathrm{lpp}(g_j)$.
    $(h_1,\dots,h_n) :=$ ReductionVector(h,G)
    Add $(0,\dots,0,\underbrace{c_j\cdot t_i}_{i},0,\dots,0,\underbrace{-c_i\cdot t_j}_{j},0,\dots,0)-(h_1,\dots,h_n)$ as last row in S
  END
END SyzygyMatrix.

2.4 Solution of Algebraic Equations

This section is based on [Winkler, 1992] and [Winkler, 1996].

We consider a system of equations

$$\begin{array}{rcl} f_1(x_1,\ldots,x_\nu) & = & 0, \\ \vdots & & \vdots \\ f_m(x_1,\ldots,x_\nu) & = & 0 \end{array} \tag{2.1}$$

where $f_1,\ldots,f_m \in \mathbf{K}[X]$. First let us decide whether (2.1) has any solutions in $\bar{\mathbf{K}}^\nu$, $\bar{\mathbf{K}}$ the algebraic closure of $\mathbf{K}$.

Theorem 7 *Let $F = \{f_1,\ldots,f_m\} \subseteq \mathbf{K}[X]$ and G a normed Gröbner basis of F. (2.1) is unsolvable (in $\bar{\mathbf{K}}$) if and only if $1 \in G$.*

Proof: If $1 \in G$ then $1 \in \mathrm{Ideal}(F)$, so every solution of (2.1) is a solution of $1 = 0$, i.e. there is no solution.

On the other hand assume that (2.1) is unsolvable. Then 1 vanishes on every root of (2.1). So by Hilbert's Nullstellensatz $1 \in \mathrm{Radical}(\mathrm{Ideal}(F))$, i.e. $1^k = 1 \in \mathrm{Ideal}(F)$ for some k. Thus $1 \in G$. □

Now suppose that (2.1) is solvable. We want to determine whether there are finitely or infinitely many solutions of (2.1).

Theorem 8 *Let $F = \{f_1,\ldots,f_m\} \subseteq \mathbf{K}[X]$ and G a Gröbner basis of F. (2.1) has finitely many solutions (i.e. $\mathrm{Ideal}(F)$ is 0-dimensional) if and only if for every $i, 1 \leq i \leq \nu$ there is a polynomial $g_i \in G$ such that $\mathrm{lpp}(g_i)$ is a power of x_i.*

Proof: $I = \mathrm{Ideal}(F)$ is 0-dimensional if and only if $\mathbf{K}[X]_{/I}$ has finite vector space dimension over K (see [Gröbner, 1949]). That is the case if and only if the number of irreducible power products w.r.t. G is finite. □

For actually computing the solutions of (2.1) we use the elimination property of Gröbner bases. It states that if G is a Gröbner basis of F w.r.t. the purely lexicographic ordering with $x_1 < \ldots < x_\nu$, then the i-th elimination ideal of $\mathrm{Ideal}(F)$ is generated by the polynomials in G that depend only on the variables $x_1,\ldots,x_\nu$.

Theorem 9 (Elimination Property) *Let $F = \{f_1,\ldots,f_m\} \subseteq \mathbf{K}[X]$ and G a Gröbner basis for F w.r.t. the purely lexicographic ordering with $x_1 < \ldots < x_\nu$. Then*

$$\mathrm{Ideal}(F) \cap \mathbf{K}[x_1,\ldots,x_i] = \mathrm{Ideal}(G \cap \mathbf{K}[x_1,\ldots,x_i])_{\mathbf{K}[x_1,\ldots,x_i]}.$$

Proof: Obviously the right hand side is contained in the left hand side.

On the other hand, assume that $f \in \mathrm{Ideal}(F) \cap \mathbf{K}[x_1, \ldots, x_i]$. Then $f \rightarrow_G^* 0$. All the polynomials in this reduction depend only on $x_1, \ldots, x_i$. So we get a representation of f as a linear combination $\sum_{i=1}^k h_i \cdot g_i$, where $g_i \in G \cap \mathbf{K}[x_1, \ldots, x_i]$ and $h_i \in \mathbf{K}[x_1, \ldots, x_i]$. □

Example 6 Let $F = \{f_1, f_2, f_3\} \subset \mathbb{Q}[x, y, z]$ be the set of polynomials

$$\begin{aligned} f_1 &= zy^2 + 2x + \frac{1}{2}, \\ f_2 &= zx^2 + y^2 + \frac{1}{2}x, \\ f_3 &= zx - y^2x - 4x^2 - \frac{1}{4}. \end{aligned}$$

Let $<$ be the purely lexicographic ordering with $x < y < z$. The Gröbner basis algorithm applied to F generates the normed reduced Gröbner basis $G = \{g_1, g_2, g_3\}$, where

$$\begin{aligned} g_1 &= z + \frac{64}{65}x^4 - \frac{432}{65}x^3 + \frac{168}{65}x^2 - \frac{354}{65}x + \frac{8}{5} \\ g_2 &= y^2 - \frac{8}{13}x^4 + \frac{54}{13}x^3 - \frac{8}{13}x^2 + \frac{17}{26}x \\ g_3 &= x^5 - \frac{27}{4}x^4 + 2x^3 - \frac{21}{16}x^2 + x + \frac{5}{32}. \end{aligned}$$

Applying Theorem 7 we see that the system of equations $f_1 = 0, f_2 = 0, f_3 = 0$ is solvable in the algebraic closure of $\mathbb{Q}$ (the field of algebraic numbers). Furthermore, by Theorem 8, this system of equations has finitely many solutions. The variables in the Gröbner basis G are totally separated. An approximation of a root of g_3 up to $\pm\frac{1}{100000}$ is -0.128475. This solution of $g_3 = 0$ can be continued to solutions of $g_2 = 0$ and $g_1 = 0$, yielding the approximation $(-0.128475, 0.321145, -2.356718)$ for the original system of equations. □

Example 7 The same method can be applied to algebraic equations with symbolic coefficients. For example let $F = \{f_1, f_2, f_3, f_4\} \subset \mathbb{Q}(a, b, c, d)[x_1, x_2, x_3, x_4]$ be the set of polynomials

$$\begin{aligned} f_1 &= x_4 + (b - d), \\ f_2 &= x_4 + x_3 + x_2 + x_1 - (a + c + d), \\ f_3 &= x_3x_4 + x_1x_4 + x_2x_3 - (ad + ac + cd), \\ f_4 &= x_1x_3x_4 - (acd).\} \end{aligned}$$

Let $<$ be the purely lexicographic ordering with $x_1 < x_2 < x_3 < x_4$. The normed reduced Gröbner basis for $\mathrm{Ideal}(F)$ is $G = \{g_1, g_2, g_3, g_4\}$, where

$$g_1 = x_4 + (b - d),$$

$$\begin{aligned}
g_2 &= x_3 - \frac{b^2-2bd+d^2}{acd}x_1^2 - \frac{abc+abd-acd-ad^2+bcd-cd^2}{acd}x_1 - (a+c+d),\\
g_3 &= x_2 + \frac{b^2-2bd+d^2}{acd}x_1^2, + \frac{abc+abd-ad^2+bcd-cd^2}{acd}x_1 - (b-d),\\
g_4 &= x_1^3 + \frac{ac+ad+cd}{b-d}x_1^2 + \frac{a^2cd+ac^2d+acd^2}{b^2-2bd+d^2}x_1 + \frac{a^2c^2d^2}{b^3-3b^2d+3bd^2-d^3}.
\end{aligned}$$

Thus, the system has finitely many solutions. A particular solution of $g_4 = 0$ is $(-ad)/(b-d)$, which can be continued to the solution

$$\left(\frac{-ad}{b-d}, \frac{ab+b^2-bd}{b-d}, c, -b+d\right).$$

□

Every Gröbner basis G of a 0-dimensional ideal I w.r.t. $<_{L,id}$ has the form

$$\begin{array}{c}
g_{1,1}(x_1) \in \mathbf{K}[x_1]\\[1em]
\left.\begin{array}{r} g_{2,1}(x_1,x_2)\\ \vdots\\ g_{2,k_2}(x_1,x_2)\end{array}\right\} \in \mathbf{K}[x_1,x_2]\\
\vdots\\
\left.\begin{array}{r} g_{\nu,1}(x_1,x_2,\ldots,x_\nu)\\ \vdots\\ g_{\nu,k_\nu}(x_1,x_2,\ldots,x_\nu)\end{array}\right\} \in \mathbf{K}[x_1,x_2,\ldots,x_\nu]
\end{array}$$

Kalkbrener has proved in [Kalkbrener, 1987] that if $b = (b_1,\ldots,b_{l-1}) \in \bar{\mathbf{K}}^{l-1}$ is a solution of $g_{1,1},\ldots,g_{l-1,k_{l-1}}$, then there is an index $j, 1 \le j \le k_l$, such that $\mathrm{lc}_{x_l}(g_{l,j})(b) \neq 0$ and

$$g_{l,j}(b,x_l) = \gcd(g_{l,1}(b,x_l),\ldots,g_{l,k_l}(b,x_l)).$$

In particular, that means, that every solution b can be continued to a solution $\bar{b} = (b_1,\ldots,b_{l-1},b_l) \in \bar{\mathbf{K}}^l$ of $g_{1,1}.\ldots,g_{l,k_l}$.

2.5 Linear Equations over $\mathbf{K}[X]$

This section is based on [Winkler, 1992] and [Winkler, 1996].

For $f_1,\ldots,f_m,f \in \mathbf{K}[X]$ we consider the linear equation

$$f_1\cdot z_1 + \ldots + f_m\cdot z_m = f. \tag{2.2}$$

The general solution of (2.2) is to be sought in $\mathbf{K}[X]^m$.

Theorem 10 *If $G = \{g_1, \ldots, g_n\}$ is a Gröbner basis, then a basis for the module of all solutions of the homogeneous system*

$$g_1 \cdot z_1 + \ldots + g_n \cdot z_n = 0 \tag{2.3}$$

can be given as

$$S = \{\overbrace{p_{i,j} \cdot e_i - q_{i,j} \cdot e_j - (k^1_{i,j}, \ldots, k^n_{i,j})}^{S_{i,j}} \mid 1 \le i < j \le n\},$$

where for $i \in \{1, \ldots, n\}$ $e_i = (0, \ldots, 0, \underbrace{1}_{i}, 0, \ldots, 0,)$ (with 1 at the i-th position) and for $1 \le i < j \le n$ $p_{i,j}, q_{i,j}$, are such that

$$\text{S-pol}(g_i, g_j) = p_{i,j} \cdot g_i - q_{i,j} \cdot g_j = \sum_{l=1}^{n} k^l_{i,j} \cdot g_l$$

and $k^1_{i,j}, \ldots, k^n_{i,j}$ are the polynomials extracted from the reduction of S-pol(g_i, g_j) *to 0. (A solution $z = (z_1, \ldots, z_n)$ of (2.3) is called* syzygy *of $g_1, \ldots, g_n$.)*

Proof: Obviously, every element of S is a syzygy of G.

On the other hand, let $z = (z_1, \ldots, z_n) \neq (0, \ldots, 0)$ be an arbitrary syzygy of G. Let

$$p = \max_{\lhd}\{t \in [X] \mid \text{coeff}(g_i \cdot z_i) \neq 0, 1 \le i \le n\}$$

and let $i_1 \lhd \ldots i_k$ be those indices such that $\text{lpp}(g_{i_j} \cdot z_{i_j}) = p$. We have $k \ge 2$. Suppose $k > 2$. By subtracting a suitable multiple of S_{i_{k-1}, i_k} from z, we can reduce the number of positions in y that contribute to the highest power product p in (2.3). Iterating this process at most $k-2$ times, we finally reach a situation, where only two positions in the syzygy contribute to the power product p. Now the highest power product in (2.3) can be decreased by subtracting a suitable multiple of S_{i_1, i_2}. Since $\lhd$ is Noetherian, this process terminates, leading to an expression of z as a linear combination of elements of S. $\square$

Now that we are able to solve homogeneous linear equations for Gröbner bases, we want to transform the result to an arbitrary basis.

Theorem 11 *Let $F = \{f_1, \ldots, f_m\} \subseteq \mathbf{K}[X]$ and $G = \{g_1, \ldots, g_n\}$ a Gröbner basis for F. We think of F and G as column vectors, i.e. $F = (f_1, \ldots, f_m)^T, G = (g_1, \ldots, g_n)^T$. Let the rows of the matrix S be a basis for the syzygies of G, and let the matrices R and T be such that $G = T \cdot F$ and $F = R \cdot G$. Then the rows of Q are a basis for the syzygies of F, where*

$$Q = \left(\begin{array}{c} I_m - R \cdot T \\ \hline S \cdot T \end{array} \right).$$

Proof: Let r the number of rows of S, $b_1, \ldots, b_{m+r} \in \mathbf{K}[X], b = (b_1, \ldots, b_{m+r})$.

$$\begin{aligned}(b \cdot Q) \cdot F &= ((b_1, \ldots, b_m) \cdot (I_m - R \cdot T) + (b_{m+1}, \ldots, b_{m+r}) \cdot S \cdot T) \cdot F \\ &= (b_1, \ldots, b_m) \cdot (F - \underbrace{R \cdot T \cdot F}_{F}) + (b_{m+1}, \ldots, b_{m+r}) \cdot S \cdot \underbrace{T \cdot F}_{G} \\ &= 0.\end{aligned}$$

So every linear combination of the rows of Q is a syzygy of F.

On the other hand, let $H = (h_1, \ldots, h_m)$ be a syzygy of F. Then $H \cdot R$ is a syzygy of G. So for some H', $H \cdot R = H' \cdot S$, and therefore $H \cdot R \cdot T = H' \cdot S \cdot T$. Thus,

$$H = H \cdot (I_m - R \cdot T) + H' \cdot S \cdot T = (H, H') \cdot Q,$$

i.e. H is a linear combination of the rows of Q. □

What we still need is a particular solution of (2.2). (2.2) has a solution if and only if $f \in \mathrm{Ideal}(F) = \mathrm{Ideal}(G)$. If the reduction of f to normal form w.r.t. G yields $f' \neq 0$, then (2.2) has no solution. Otherwise one can extract from this reduction polynomials $h'_1, \ldots, h'_m$ such that

$$g_1 \cdot h'_1 + \cdots + g_n \cdot h'_n = f.$$

So $h' \cdot T$ is a particular solution of (2.2). This leads to the following algorithm for solving (2.2).

Algorithm 9 (LinSolve1)

$(\vec{a}, A) \leftarrow$ LinSolve1$(f_1, \ldots, f_m, f)$
  input: $(f_1, \ldots, f_m, f) \in \mathbf{K}[X]$.
  output: $\vec{a}$, a polynomial vector of length m and
    A, a polynomial matrix with n rows and m columns,
    for some n, such that the set of solutions of
    $f_1 \cdot z_1 + \ldots + f_m \cdot z_m = f$ is
    $\{\vec{a} + A \cdot (c_1, \ldots, c_{m+r})^T \mid c_1, \ldots, c_{m+r} \in \mathbf{K}[X]\}$
    or unsolvable.

BEGIN
  $(G, T) :=$ GröbnerBasis(F) and corresponding transformation matrix T such that
    $T \cdot F = G$
  $R :=$ ReductionMatrix(F, G)
  $S :=$ SyzygyMatrix(G)

$$A = \left(\begin{array}{c} I_m - R \cdot T \\ \hline S \cdot T \end{array} \right)^T$$

```
    f' := NormalForm(G, f)
    IF f' ≠ 0 THEN
        RETURN unsolvable
    ELSE
        a' := ReductionVector(f, G)
        a := a' · T^T
        RETURN (a, A)
    END
END LinSolve1.
```

Example 8 Let F and G be as in Example 6. We want to compute a basis for the syzygies of F, i.e. a basis for the solutions of the linear equation

$$f_1 \cdot z_1 + f_2 \cdot z_2 + f_3 \cdot z_3 = 0.$$

From the Gröbner basis algorithm we can extract the transformation matrix

$$T = \begin{pmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,1} & t_{3,2} & t_{3,3} \end{pmatrix}.$$

where

$$\begin{aligned}
t_{1,1} &= \frac{32}{35}z^2x^3 + \frac{32}{35}z^2x - \frac{256}{455}zx^7 + \frac{1728}{455}zx^6 - \frac{512}{455}zx^5 + \frac{256}{65}zx^4 - \frac{256}{455}zx^3 + \\
&\quad \frac{64}{455}zx^2 + \frac{104}{35}z + \frac{128}{65}x^6 - \frac{864}{65}x^5 + \frac{192}{91}x^4 - \frac{432}{455}x^3 + \frac{64}{455}x^2 - \frac{348}{35}x, \\
t_{1,2} &= -\frac{32}{35}z^2y^2x - \frac{16}{5}z^2 + \frac{256}{455}zy^2x^5 - \frac{1728}{455}zy^2x^4 + \frac{256}{455}zy^2x^3 - \frac{64}{455}zy^2x^2 + \\
&\quad \frac{128}{65}zx^4 - \frac{864}{65}zx^3 + \frac{64}{455}zx^2 + \frac{5392}{455}zx - \frac{128}{65}y^2x^4 + \frac{864}{65}y^2x^3 - \frac{128}{65}y^2x^2 + \\
&\quad \frac{512}{455}x^6 - \frac{6912}{455}x^5 + \frac{4768}{91}x^4 - \frac{544}{65}x^3 + \frac{2376}{455}x^2 - \frac{448}{65}x, \\
t_{1,3} &= -\frac{32}{35}z^2y^2 + \frac{16}{5}z^2x + \frac{256}{455}zy^2x^4 - \frac{1728}{455}zy^2x^3 + \frac{256}{455}zy^2x^2 - \frac{64}{455}zy^2x - \\
&\quad \frac{128}{65}zx^5 + \frac{864}{65}zx^4 - \frac{128}{65}zx^3 + \frac{32}{65}zx^2 - \frac{64}{35}zx + \frac{68}{35}z - \frac{128}{65}y^2x^3 + \\
&\quad \frac{864}{65}y^2x^2 - \frac{128}{65}y^2x + \frac{512}{455}x^5 - \frac{3776}{455}x^4 + \frac{2672}{455}x^3 - \frac{64}{65}x^2 + \frac{864}{455}x - \frac{32}{5}, \\
t_{2,1} &= \frac{4}{13}x^3 - \frac{4}{13}x, \\
t_{2,2} &= \frac{4}{13}y^2x + \frac{14}{13}, \\
t_{2,3} &= \frac{4}{13}y^2 - \frac{14}{13}x, \\
t_{3,1} &= \frac{1}{7}zx^3 + \frac{1}{7}zx + \frac{1}{2}x^4 + \frac{1}{2}x^2 + \frac{13}{28},
\end{aligned}$$

$$
\begin{aligned}
t_{3,2} &= -\frac{1}{7}zy^2x - \frac{1}{2}z - \frac{1}{2}y^2x^2 - \frac{2}{7}x^2 + \frac{5}{28}x,\\
t_{3,3} &= -\frac{1}{7}zy^2 + \frac{1}{2}zx - \frac{1}{2}y^2x + \frac{7}{4}x^2 - \frac{2}{7}x + \frac{17}{56}.
\end{aligned}
$$

Reduction of the polynomials of F to 0 w.r.t. G yields the entries for the matrix

$$
R := \begin{pmatrix} \frac{8}{13}x^4 - \frac{54}{13}x^3 + \frac{8}{13}x^2 - \frac{17}{26}x & z & -\frac{512}{845}x^3 + \frac{3456}{845}x^2 - \frac{64}{65}x + \frac{16}{5} \\ x^2 & 1 & -\frac{64}{65}x \\ x & -x & -\frac{8}{5} \end{pmatrix}.
$$

Reducing the S-polynomials of G to 0 we get the rows of S, which form a basis for the syzygies of G.

$$
S := \begin{pmatrix} 0 & s_{1,2} & s_{1,3} \\ s_{2,1} & 0 & s_{2,3} \\ s_{3,1} & s_{3,2} & 0 \end{pmatrix},
$$

where

$$
\begin{aligned}
s_{1,2} &= x^5 - \frac{27}{4}x^4 + 2x^3 - \frac{21}{16}x^2 + x + \frac{5}{32},\\
s_{1,3} &= -y^2 + \frac{8}{13}x^4 - \frac{54}{13}x^3 + \frac{8}{13}x^2 - \frac{17}{26}x,\\
s_{2,1} &= x^5 - \frac{27}{4}x^4 + 2x^3 - \frac{21}{16}x^2 + x + \frac{5}{32},\\
s_{2,3} &= -z - \frac{64}{65}x^4 + \frac{432}{65}x^3 - \frac{168}{65}x^2 + \frac{354}{65}x - \frac{8}{5},\\
s_{3,1} &= y^2 - \frac{8}{13}x^4 + \frac{54}{13}x^3 - \frac{8}{13}x^2 + \frac{17}{26}x,\\
s_{3,2} &= -z - \frac{64}{65}x^4 + \frac{432}{65}x^3 - \frac{168}{65}x^2 + \frac{354}{65}x - \frac{8}{5}.
\end{aligned}
$$

Now according to Theorem 11 the rows of

$$
Q = \begin{pmatrix} I_3 - R \cdot T \\ \hline S \cdot T \end{pmatrix}.
$$

form a basis for the syzygies of F. E.g.

$$
\begin{aligned}
Q_1 = & -\frac{125}{182}zx^4 - \frac{17}{182}zx^2 + \frac{8}{91}zx^7 + \frac{8}{91}zx^3 - \frac{1383}{728}x^3 - \\
& \frac{54}{91}zx^6 + \frac{16}{91}zx^5 - \frac{13}{28}y^2 - \frac{2}{91}x^2 - \frac{32}{91}x + \frac{1}{13}x^5 - \\
& \frac{30}{91}x^4 - \frac{4}{13}x^6 - \frac{1}{7}zy^2x^3 - \frac{1}{7}zy^2x - \frac{1}{2}y^2x^4 - \frac{1}{2}y^2x^2.
\end{aligned}
$$

□

Of course, once we are able to solve single linear equations over $\mathbf{K}[X]$, we can also solve system of linear equations by dealing with the equations recursively. However, it is also possible to extend the concept of Gröbner bases from ideals to modules and solve a whole system of linear equations by a single computation of a Gröbner basis for a submodule of $\mathbf{K}[X]^m$.

2.6 Gröbner Bases for Submodule of $\mathbf{K}[X]^m$

This section is based on [Möller and Mora, 1986] and [Winkler, 1992].

A module $\mathbf{A}$ over a ring $\mathbf{R}$ is basically the same as a vector space, except that we do not presuppose that $\mathbf{R}$ is a field. So, in a module $\mathbf{A}$ we can add elements and we can multiply them by scalars from $\mathbf{R}$, with the usual laws that hold in a vector space. In this section we are dealing with the module $\mathbf{K}[X]^m$ over $\mathbf{K}[X]$, i.e. the module of (column) vectors of polynomials, where the ring of scalars is $\mathbf{K}[X]$. Addition and scalar multiplication are defined componentwise.

Definition 11 Let $\mathbf{A}$ be a module over $\mathbf{R}$, $\mathbf{R}$ a commutative ring with 1. A subset $\mathbf{A}'$ of $\mathbf{A}$ is a *submodule* iff it is closed under addition and under multiplication by elements of $\mathbf{R}$. Let B be a subset of $\mathbf{A}$. The *submodule generated by* B is

$$< B >= \left\{ \sum_{i=1}^{l} r_i \cdot b_i \mid r_i \in \mathbf{R}, b_i \in B, l \in \mathbb{N}_0 \right\}.$$

In this case B is called a *basis* (or *generating set*) of $< B >$. □

Like for ideals in $\mathbf{K}[X]$ one can show that every submodule of $\mathbf{K}[X]^m$ has a finite basis. In this section we generalize the notion of Gröbner bases for ideals to Gröbner bases for submodules of $\mathbf{K}[X]^m$, so that we can solve systems of linear questions.

In order to be able to generalize the notions of reduction, S-polynomials and Gröbner bases we first need to define admissible orderings. An admissible ordering will be an ordering of the set $[\vec{X}]$, which consists of all row vectors

$$(0, \ldots, 0, u, 0, \ldots, 0)^T$$

of length m, where $u \in [X]$. We call an element of $[\vec{X}]$ a power product tuple.

The partial divisibility ordering $<_d$ on $[\vec{X}]$ is defined as $\vec{p_1} <_d \vec{p_2}$ iff there exists a power product $q \in [X] \setminus \{1\}$ such that $\vec{p_1} = q \cdot \vec{p_2}$.

Definition 12 An *admissible ordering* $\lhd_v$ on $[\vec{X}]$ is any total ordering which satisfies

1. $\vec{p_1} <_d \vec{p_2} \Rightarrow \vec{p_1} \lhd_v \vec{p_2}$ for all $\vec{p_1}, \vec{p_2} \in [\vec{X}]$,
2. $\vec{p_1} \lhd_v \vec{p_2} \Rightarrow q \cdot \vec{p_1} \lhd_v q \cdot \vec{p_2}$ for all $\vec{p_1}, \vec{p_2} \in [\vec{X}], q \in [X]$.

$\square$

Again, every admissible ordering on $[\vec{X}]$ turns out to be Noetherian.

Example 9 Some examples for admissible orderings are:

1. The *term-first-ordering* based on the admissible ordering $\lhd$ of $[X]$: for $\vec{p_1} = (0, \ldots, 0, u_1, 0, \ldots, 0)$, u_1 at the i-th position, and $\vec{p_2} = (0, \ldots, 0, u_2, 0, \ldots, 0)$, u_2 at the j-th position

$$\vec{p_1} \lhd_v \vec{p_2} \text{ iff } u_1 \lhd u_2 \text{ or } (u_1 = u_2 \text{ and } i > j).$$

For instance, if $\lhd$ is the graduated lexicographic ordering on $[x, y]$ with $x \lhd y$, then in the term-first-ordering based on $\lhd$ we have

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \lhd_v \begin{pmatrix} 0 \\ x \end{pmatrix} \lhd_v \begin{pmatrix} x \\ 0 \end{pmatrix} \lhd_v \begin{pmatrix} 0 \\ y \end{pmatrix} \lhd_v \cdots \lhd_v \begin{pmatrix} 0 \\ xy \end{pmatrix} \lhd_v \cdots \lhd_v \begin{pmatrix} 0 \\ y^2 \end{pmatrix} \lhd_v \cdots$$

2. The *index-first-ordering* based on the admissible ordering $\lhd$ of $[X]$: for $\vec{p_1}, \vec{p_2}$ as above,

$$\vec{p_1} \lhd_v \vec{p_2} \text{ iff } i > j \text{ or } (i = j \text{ and } u_1 \lhd u_2).$$

For instance, if $\lhd$ is the graduated lexicographic ordering on $[x, y]$ with $x \lhd y$, then in the index-first-ordering based on $\lhd$ we have

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \lhd_v \begin{pmatrix} 0 \\ x \end{pmatrix} \lhd_v \begin{pmatrix} 0 \\ y \end{pmatrix} \lhd_v \cdots \lhd_v \begin{pmatrix} 0 \\ xy^2 \end{pmatrix} \lhd_v \cdots \lhd_v \begin{pmatrix} 1 \\ 0 \end{pmatrix} \lhd_v \begin{pmatrix} x \\ 0 \end{pmatrix} \lhd_v \cdots$$

$\square$

Definition 13 Let $\lhd_v$ be an admissible ordering on $[X]$. Then every nonzero vector of polynomials $\vec{f}$ in $\mathbf{K}[X]^m \setminus \{0\}$ has a unique representation

$$\vec{f} = \sum_{i=1}^{n} \operatorname{coeff}(\vec{f}, p_i) \cdot \vec{p_i},$$

where $\operatorname{coeff}(\vec{f}, p_i) \in \mathbf{K} \setminus \{0\}$, $\vec{p_i} \in [\vec{X}]$ and $\vec{p_1} \lhd_v \vec{p_2} \lhd_v \cdots \lhd_v \vec{p_n}$. $\vec{p_n}$ is the *leading power product tuple* ($\operatorname{lppt}(\vec{f})$) of $\vec{f}$, the nonzero component of $\vec{p_n}$ is the *leading power product* ($\operatorname{lpp}(\vec{f})$) of $\vec{f}$, $\operatorname{coeff}(\vec{f}, p_i)$ is the *coefficient* of $\vec{p_i}$ in $\vec{f}$, $\operatorname{coeff}(\vec{f}, p_1)$ is the *leading coefficient* ($\operatorname{lc}(\vec{f})$) of $\vec{f}$, the index k, such that the nonzero component of $\vec{p_1}$ occurs at the k-th position, is the *head position* ($\operatorname{hp}(\vec{f})$) of $\vec{f}$. $\square$

As in the case of polynomials, an admissible ordering $\lhd_v$ on $[X]$ can be extended to a partial Noetherian ordering $\ll$ on $\mathbf{K}[X]^m$. From now on let $\lhd_v$ be a fixed admissible ordering on $[\vec{X}]$.

Definition 14 Let $\vec{F} \subseteq \mathbf{K}[X]^m$, $\vec{g}, \vec{h} \in \mathbf{K}[X]^m$. We say that $\vec{g}$ *reduces to* $\vec{h}$ *w.r.t.* $\vec{F}$, abbreviated $\vec{g} \rightarrow_{\vec{F}} \vec{h}$, iff there are $\vec{f} \in \vec{F}$, $t \in [X]$, such that

$$c := \frac{\operatorname{coeff}(\vec{g}, t \cdot \operatorname{lpp}(\vec{f}))}{\operatorname{lc}(\vec{f})} \neq 0 \text{ and } \vec{h} = \vec{g} - c \cdot t \cdot \vec{f}.$$

If we want to indicate which polynomial, power product and coefficient are used in the reduction, we write $\vec{g} \rightarrow_{\vec{f},c,t} \vec{h}$. □

The notion of normal form and the normal form algorithm can be generalized accordingly to modules.

Example 10 Consider $\vec{F} = \{\vec{f_1}, \vec{f_2}, \vec{f_3}\}$, where

$$\begin{aligned}
\vec{f_1} &= \begin{pmatrix} 2y^3 - x \\ xy - 1 \end{pmatrix}, \\
\vec{f_2} &= \begin{pmatrix} \frac{1}{2}xy \\ x^2 - x \end{pmatrix}, \\
\vec{f_3} &= \begin{pmatrix} -\frac{1}{2}x^2 + 1 \\ y^3 - 1 \end{pmatrix}.
\end{aligned}$$

The pairs of polynomials are elements of $\mathbb{Q}[x, y]^2$. In this example let us use the term-first-ordering $\lhd_v$ based on the graduated lexicographic ordering $x \lhd y$ on $\mathbb{Q}[x, y]$. The leading power product tuples are

$$\begin{aligned}
\operatorname{lppt}(\vec{f_1}) &= \begin{pmatrix} y^3 \\ 0 \end{pmatrix}, \\
\operatorname{lppt}(\vec{f_2}) &= \begin{pmatrix} xy \\ 0 \end{pmatrix}, \\
\operatorname{lppt}(\vec{f_3}) &= \begin{pmatrix} 0 \\ y^3 \end{pmatrix}.
\end{aligned}$$

The leading power products are

$$\begin{aligned}
\operatorname{lpp}(\vec{f_1}) &= y^3, \\
\operatorname{lpp}(\vec{f_2}) &= xy, \\
\operatorname{lpp}(\vec{f_3}) &= y^3.
\end{aligned}$$

The leading coefficients are

$$\begin{aligned} \mathrm{lc}(\vec{f_1}) &= 2, \\ \mathrm{lc}(\vec{f_2}) &= \frac{1}{2}, \\ \mathrm{lc}(\vec{f_3}) &= 1. \end{aligned}$$

The head positions are

$$\begin{aligned} \mathrm{hp}(\vec{f_1}) &= 1, \\ \mathrm{hp}(\vec{f_2}) &= 1, \\ \mathrm{hp}(\vec{f_3}) &= 2. \end{aligned}$$

We now consider the reduction of an element $\vec{g} \in \mathbb{Q}[x,y]^2$ w.r.t. F.

$$\begin{aligned} \vec{g} &:= \begin{pmatrix} xy^3+1 \\ y^3 \end{pmatrix} \to_{\vec{f_1}} \begin{pmatrix} \frac{1}{2}x^2+1 \\ y^3 - \frac{1}{2}x^2y + \frac{1}{2}x \end{pmatrix} =: \vec{g'} \\ \vec{g'} &= \vec{g} - \frac{\mathrm{coeff}(\vec{g}, x \cdot \mathrm{lppt}(\vec{f_1}))}{\mathrm{lc}(\vec{f_1})} \cdot x \cdot \vec{f_1} = \vec{g} - \frac{1}{2} x \vec{f_1}. \\ \vec{g'} &\to_{\vec{f_3}} \begin{pmatrix} x^2 \\ -\frac{1}{2}x^2y + \frac{1}{2}x + 1 \end{pmatrix} =: \vec{g_1}. \end{aligned}$$

$\vec{g_1}$ is irreducible w.r.t. $\vec{F}$.

$$\vec{g} \to_{\vec{f_2}} \begin{pmatrix} 2y^2+1 \\ -2x^2y^2+y^3+2xy^2 \end{pmatrix} \to_{\vec{f_3}} \begin{pmatrix} 2y^2+\frac{1}{2}x^2 \\ -2x^2y^2+2xy^{2+1} \end{pmatrix} =: \vec{g_2}.$$

$\vec{g_2}$ is irreducible w.r.t. $\vec{F}$, too. Hence, $\vec{g_1}$ and $\vec{g_2}$ are normal forms of $\vec{g}$ w.r.t. $\vec{F}$. □

Definition 15 Let $\vec{f}, \vec{g} \in \mathbf{K}[X]^m \setminus \{0\}, \vec{t} = \mathrm{lcm}(\mathrm{lpp}(\vec{f}), \mathrm{lpp}(\vec{g}))$. The *S-vector* of $\vec{f}$ and $\vec{g}$, S-vector$(\vec{f}, \vec{g})$, is defined as

$$\text{S-vector}(\vec{f}, \vec{g}) = \begin{cases} \mathrm{lc}(\vec{g}) \cdot \frac{\vec{t}}{\mathrm{lpp}(\vec{f})} \cdot \vec{f} - \mathrm{lc}(\vec{f}) \cdot \frac{\vec{t}}{\mathrm{lpp}(\vec{g})} \cdot \vec{g} & \text{if } \mathrm{hp}(\vec{f}) = \mathrm{hp}(\vec{g}) \\ 0 & \text{otherwise} \end{cases}$$

□

Definition 16 Let $\vec{F} \subseteq \mathbf{K}[X]^m$, $\vec{g}, \vec{h} \in \mathbf{K}[X]^m$. We say that $\vec{F}$ is a *Gröbner basis* (for $< \vec{F} >$) iff it satisfies the following condition: Given $\vec{g}, \vec{h_1}, \vec{h_2} \in \vec{F}$ and $\vec{h_1}$ and $\vec{h_2}$ are normal forms of $\vec{g}$ w.r.t. $\vec{F}$, then $\vec{h_1} = \vec{h_2}$. □

Theorem 12 *Let M be a submodule of $\mathbf{K}[X]^m$, $\vec{G}$ a finite subset of M. The following conditions are equivalent.*

1. *$\vec{G}$ is a Gröbner basis for M.*

2. *$\vec{f} \to^*_{\vec{G}} 0$ for all $\vec{f} \in M$.*

3. *$\vec{f} \to_{\vec{G}}$ for all $\vec{f} \in M \setminus \{0\}$.*

4. *For all $\vec{g} \in M$, $\vec{h} \in \mathbf{K}[X]^m$: if $\vec{g} \to^*_{\vec{G}} \vec{h}$ and $\vec{h}$ is in normal form w.r.t. $\vec{G}$ then $\vec{h} = 0$.*

5. *For all $\vec{g}, \vec{h_1}, \vec{h_2} \in \mathbf{K}[X]^m$: if $\vec{g} \to^*_{\vec{G}} \vec{h_1}$ and $\vec{g} \to^*_{\vec{G}} \vec{h_2}$ and $\vec{h_1}$ and $\vec{h_2}$ are in normal form w.r.t. $\vec{G}$ then $\vec{h_1} = \vec{h_2}$.*

6. *For all $\vec{g_1}, \vec{g_2} \in \vec{G}$:* S-vector$(\vec{g_1}, \vec{g_1}) \to^*_{\vec{G}} 0$.

Proof: See [Furukawa et al., 1986] or [Möller and Mora, 1986]. □

So we get an algorithm for computing Gröbner bases for submodules of $\mathbf{K}[X]^m$.

Algorithm 10 (GröbnerBasis-M)

$\vec{G} \leftarrow$ GröbnerBasis-M$(\vec{F})$
    input: $\vec{F}$, a finite subset of $\mathbf{K}[X]^m$.
    output: $\vec{G}$, a finite subset of $\mathbf{K}[X]^m$ such that
        $< \vec{F} >=< \vec{G} >$ and G is a Gröbner basis.
BEGIN
    $\vec{G} := \vec{F} \setminus \{0\}$
    $B := \{\{\vec{g}_1, \vec{g}_2\} \mid \vec{g}_1, \vec{g}_2 \in \vec{G}, \vec{g}_1 \neq \vec{g}_2\}$
    WHILE $B \neq \emptyset$ DO
        $\{\vec{g}_1, \vec{g}_2\} :=$ a pair in B
        $B := B \setminus \{\{\vec{g}_1, \vec{g}_2\}\}$
        $\vec{h} :=$ S-pol$(\vec{g}_1, \vec{g}_2)$
        $\vec{h} :=$ normal form of $\vec{h}$ w.r.t. $\vec{G}$
        IF $\vec{h} \neq 0$ THEN
            $B := B \cup \{\{\vec{G}, \vec{h}\} \mid \vec{G} \in \vec{G}\}$
            $\vec{G} := \vec{G} \cup \{\vec{h}\}$
        END
    END
    RETURN $\vec{G}$
END GröbnerBasis-M.

By a suitable generalization of Dickson's Lemma one can show that the algorithm GröbnerBasis-M terminates for every input.

The notions of reduced and normed Gröbner basis can be generalized in a straightforward way to $\mathbf{K}[X]^m$, and one can show that every submodule M of $\mathbf{K}[X]^m$ has a uniquely determined normed reduced Gröbner basis.

Gröbner bases for modules are important for the solution of linear systems over $\mathbf{K}[X]$ in two ways:

1. once a basis $\{z^{(1)}, \ldots, z^{(l)}\}$ for the syzygies of a linear system is computed, we want to somehow simplify this basis. One way of doing that is computing a normed reduced Gröbner basis for the module of syzygies;
2. instead of dealing with the equations in a linear system

$$\begin{pmatrix} f_{1,1} & \cdots & f_{1,s} \\ \vdots & & \vdots \\ f_{t,1} & \cdots & f_{t,s} \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ \vdots \\ z_s \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_t \end{pmatrix} \tag{2.4}$$

one by one in a recursive way, one can look at (2.4) as

$$\begin{pmatrix} f_{1,1} \\ \vdots \\ f_{t,1} \end{pmatrix} \cdot z_1 + \cdots + \begin{pmatrix} f_{1,s} \\ \vdots \\ f_{t,s} \end{pmatrix} \cdot z_s = \begin{pmatrix} f_1 \\ \vdots \\ f_t \end{pmatrix}$$

and solve it in one stroke by an application of GröbnerBasis-M. This is possible, since Theorems 10, 11 and Algorithm LinSolve1 can be generalized exactly as they are to this situation.

Example 11 This example is from [Winkler, 1992]. We apply the method of Gröbner basis for modules in order to solve the following linear system over $\mathbb{Q}[x_1, x_2, x_3, x_4]$:

$$\begin{pmatrix} x_3x_1 - x_2^2 & x_4x_1 - x_3x_2 & x_4x_2 - x_3^2 \\ x_2^2 - x_2x_1 & x_4x_1 - x_3x_2 & x_4x_3 - x_4x_1 \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

where

$$\begin{aligned} f_1 &= 2x_4x_1 + x_3^3x_1 - x_3^2x_2^2 - 2x_3x_2, \\ f_2 &= 2x_4x_1 + x_3^3x_2^2x_1 - 2x_3^2x_2^4 - 2x_3^2x_2^3x_1 - 2x_3^2x_2^2x_1^2 + x_3^2x_2^2 - \\ &\quad x_3^2x_2x_1 - 2x_3x_2^5 + 5x_3x_2^4x_1 - 2x_3x_2^3x_1^2 - 2x_3x_2. \end{aligned}$$

The first step is to compute a Gröbner basis for the module generated by h_1, h_2, h_3, where

$$\begin{aligned} h_1 &= \begin{pmatrix} x_3x_1 - x_2^2 \\ x_2^2 - x_2x_1 \end{pmatrix}, \\ h_2 &= \begin{pmatrix} x_4x_1 - x_3x_2 \\ x_4x_1 - x_3x_2 \end{pmatrix}, \\ h_2 &= \begin{pmatrix} x_4x_2 - x_3^2 \\ x_4x_3 - x_4x_1 \end{pmatrix}. \end{aligned}$$

Using the index-first-ordering based on the purely lexicographic ordering with $x_1 < x_2 < x_3 < x_4$ we get, after the reduction of seven S-vectors, the following result $G = (h_1, \ldots, h_7)$, where

$$\begin{aligned}
h_4 &= \begin{pmatrix} 0 \\ x_4x_3x_2 + x_4x_2^2 - 3x_4x_2x_1 - x_4x_1^2 + x_3^2x_2 + 2x_3x_2^2 - x_3x_2x_1 \end{pmatrix}, \\
h_5 &= \begin{pmatrix} 0 \\ x_4x_3x_1 - x_4x_2x_1 - x_4x_1^2 + 2x_3x_2^2 - x_3x_2x_1 \end{pmatrix}, \\
h_6 &= \begin{pmatrix} 0 \\ x_4x_2^2x_1 - x_4x_2x_1^2 - \frac{1}{2}x_4x_1^3 + \frac{1}{2}x_3^2x_2x_1 - x_3x_2^3 + \frac{3}{2}x_3x_2^2x_1 - \frac{1}{2}x_3x_2x_1^2 \end{pmatrix}, \\
h_7 &= \begin{pmatrix} 0 \\ x_3^3x_2x_1 - 2x_3^2x_2^3 + 2x_3^2x_2^2x_1 - 2x_3^2x_2x_1^2 - 2x_3x_2^4 + 5x_3x_2^3x_1 - 2x_3x_2^2x_1^2 \end{pmatrix}
\end{aligned}$$

together with the transformation matrix

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ x_4 + x_3 & -x_3 - x_2 & x_2 + x_1 \\ x_3 & -x_2 & x_1 \\ \frac{1}{2}x_4x_1 - \frac{1}{2}x_3x_2 + \frac{1}{2}x_3x_1 & -\frac{1}{2}x_3x_1 + \frac{1}{2}x_2^2 - \frac{1}{2}x_2x_1 & \frac{1}{2}x_1^2 \\ t_{7,1} & t_{7,2} & t_{7,3} \end{pmatrix}$$

where

$$\begin{aligned}
t_{7,1} &= x_4x_3x_1 - x_4x_2x_1 - x_4x_1^2 - x_3^2x_2 + x_3^2x_1 - x_3x_2^2 + 2x_3x_2x_1, \\
t_{7,2} &= -x_3^2x_1 + x_3x_2^2 + x_3x_1^2 + x_2^3 - 2x_2^2x_1, \\
t_{7,3} &= x_3x_1^2 - 2x_2^2x_1 + x_2x_1^2.
\end{aligned}$$

A transformation matrix R from $h_1, \ldots, h_7)^T$ to $h_1, \ldots, h_3)^T$ is

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A basis for the module of syzygies for $h_1, \ldots, h_3)$ can be computed as the rows of

$$\begin{pmatrix} I_3 - R \cdot T \\ \hline S \cdot T \end{pmatrix},$$

where the rows of S generate the module of syzygies of $(h_1, \ldots, h_7)$.

$I_3 - R \cdot T$ is the 0-matrix.

After linear transformation, the matrix S contains one non-vanishing row derived from

$$\begin{aligned}
\text{S-vector}(h_4, h_7) &= x_3^2 x_1 h_4 - x_4 h_7 \\
&= (2x_3x_2^2 - x_3x_2x_1 - x_3x_1^2 + 2x_2^2x_1 + 2x_2x_1^2 - 5x_1^3)h_4 + \\
&\quad (-x_3x_1^2 - 2x_1^3)h_5 + \\
&\quad (-2x_2^2 + 2x_2x_1 + 14x_1^2)h_6 + \\
&\quad (x_{3+}x_2 + 2x_1)h_7
\end{aligned}$$

The corresponding row of S is:

$$\begin{pmatrix}
0 \\
0 \\
0 \\
-x_3^2x_1 + 2x_3x_2^2 - x_3x_2x_1 - x_3x_1^2 + 2x_2^2x_1 + 2x_2x_1^2 - 5x_1^3 \\
-x_3x_1^2 - 2x_1^3 \\
-2x_2^2 + 2x_2x_1 + 14x_1^2 \\
x_4 + x_3 + x_2 + 2x_1
\end{pmatrix}^T .$$

After multiplication with the matrix T we get the basis

$$\begin{pmatrix}
x_4^2x_3x_1 - x_4^2x_2x_1 - x_4^2x_1^2 - x_4x_3^2x_2 + x_4x_3^2x_1 + x_4x_3x_2^2 + x_4x_3x_2x_1 - x_3^3x_2^2 \\
-x_4x_3^2x_1 + x_4x_3x_2^2 + x_4x_3x_1^2 + x_4x_2^3 - 2x_4x_2^2x_1 - x_3^2x_2^2 + x_3^2x_2x_1 \\
x_4x_3x_1^2 - 2x_4x_2^2x_1 + x_4x_2x_1^2 - x_3^2x_2x_1 + 2x_3x_2^3 - x_3x_2^2x_1
\end{pmatrix}^T .$$

for the solutions of the homogeneous system. A particular solution of the inhomogeneous solutions is

$$\begin{pmatrix}
x_4x_3x_2x_1 - x_4x_2^2x_1 - x_4x_2x_1^2 - x_3^2x_2^2 + x_3^2x_2x_1 + x_3^2 - x_3x_2^3 + 2x_3x_2^2x_1 \\
-x_3^2x_2x_1 + x_3x_2^3 + x_3x_2x_1^2 + x_2^4 - 2x_2^3x_1 + 2 \\
x_3x_2x_1^2 - 2x_2^3x_1 + x_2^2x_1^2
\end{pmatrix}^T .$$

□

Chapter 3

p-adic Lifting

In this chapter we present the algorithm for lifting a Gröbner basis as first presented by Winkler in [Winkler, 1987b] and [Winkler, 1988]. The structure of this chapter is as follows. Section 3.1 deals with the coefficients we want to approximate, the *Farey rationals*. Section 3.2 contains a short discussion of lucky primes, i.e. those primes with respect to which the p-adic approximation of a Gröbner basis is possible. In Section 3.3 existence and uniqueness of such an approximation are investigated, giving rise to the lifting algorithm of Section 3.4.

3.1 N-Farey rationals

In the following we will be dealing primarily with Gröbner bases in the ring $\mathbb{Q}[X]$. By $\mathbb{Z}_m$ we mean the ring of integers modulo m. The aim of of this chapter is to approximate the Gröbner basis G of a polynomial ideal in $\mathbb{Q}[X]$ by a basis G' in $\mathbb{Z}_{p^k}[X]$, p a prime. If p^k is sufficiently large, then we will be able to recover the "true" coefficients of G from their approximations in G'. The coefficients we want to approximate will be *Farey rationals*.

Definition 17 The *N-Farey rationals*, $N \in \mathbb{N}$, are defined as

$$\mathcal{F}_N = \{\frac{n}{d} \mid n, d \in \mathbb{Z}, -N \leq n \leq N, 1 \leq d \leq N, \gcd(n, d) = 1\}$$

Furthermore, for a prime p, $\mathcal{F}_{p,N} = \mathcal{F}_N \cap \{n/d \in \mathbb{Q} \mid \gcd(d, p) = 1\}$. □

The elements of $\mathcal{F}_{p,N}$ can be encoded uniquely in the integers modulo m for a suitable m.

Theorem 13 *Let p a prime, $m = p^k$, N such that $N \leq \sqrt{(m-1)/2}$. Then for any $a \in \mathbb{Z}_m$ there exists at most one $n/d \in \mathcal{F}_{p,N}$ such that $n \equiv a \cdot d \pmod{m}$.*

Proof: see [Trinks, 1984]. □

The usual canonical mapping is used for mapping $\mathcal{F}_{p,N}$ into $\mathbb{Z}_{p^k}$.

$$\begin{aligned} f : \mathcal{F}_{p,N} &\rightarrow \mathbb{Z}_{p^k} \\ n/d &\mapsto n \cdot d^{-1} \pmod{p^k} \end{aligned}$$

For the inverse mapping from $\mathbb{Z}_{p^k}$ to $\mathcal{F}_{p,N}$ one can use a suitably extended Euclidean algorithm (see [Kornerup and Gregory, 1983]). In more detail:

$$\begin{aligned} f^{-1} : \mathbb{Z}_{p^k} &\rightarrow \mathcal{F}_{p,N} \\ a = \sum_{i=1}^{k} a_i \cdot p^i &\mapsto n/d \end{aligned}$$

by using the extended Euclidean algorithm with the seed matrix

$$\begin{pmatrix} p^k & a \\ 0 & 1 \end{pmatrix}.$$

Algorithm 11 (FareyToRational)

$n/d \leftarrow$ FareyToRational(a, p, k)
  input: $p \in \mathbb{Z}$, a prime, $k \in \mathbb{N}$, $a \in \mathbb{Z}_{p^k}$.
  output: n/d the uniquely defined rational such that $n \equiv a \cdot d \pmod{p^k}$ and
    $|n| < N, d < N$ for $N = \lfloor \sqrt{(p^k - 1)/2} \rfloor$.

BEGIN
  $N := \lfloor \sqrt{(p^k - 1)/2} \rfloor$
  $\begin{pmatrix} n' & n \\ d' & d \end{pmatrix} := \begin{pmatrix} p^k & a \\ 0 & 1 \end{pmatrix}$
  WHILE $n > N$ or $|d| > N$ DO
    $q := \lfloor n'/n \rfloor$
    $\begin{pmatrix} n' & n \\ d' & d \end{pmatrix} := \begin{pmatrix} n & n' - q \cdot n \\ d & d' - q \cdot d \end{pmatrix}$
  END
  RETURN n/d
END FareyToRational.

Whenever we say that p does not divide q for a prime p and a rational number q, we mean that p divides neither the numerator nor the denominator of q, i.e. the p-adic norm of q is one, $|q|_p = 1$, and when we say that q' is the image of $q = n/d \in \mathcal{F}_{p,N}$ modulo $m = p^k$, we mean that $n \equiv d \cdot q' \pmod{m}$.

3.2 Lucky Primes

In [Brown, 1971] a prime p is called lucky for the computation of the greatest common divisor of two integral polynomials f and g, if p does not divide the leading coefficient of f and g and the degree of the gcd of f and g modulo p equals the degree of the gcd of f and g over the integers (the gcd of f and g modulo p cannot be less than the degree of the gcd over the integers, as long as p does not divide any of the leading coefficients of f and g). Only such lucky primes can be used in the modular computation of the polynomial greatest common divisor. We need a similar condition on the prime p.

Example 12

1. Let $F = \{y+3, 2yx+3y-5x\} \subset \mathbb{Q}[x, y]$. Let the power products be ordered according to the total degree ordering with $x < y$. The normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$ is $G = \{x + 9/11, y + 3\}$.

   11 divides the second coefficient of the first polynomial in Gröbner basis , so 11 is not "lucky" for F. The normed reduced Gröbner basis for Ideal(F) in $\mathbb{Z}_{11}[x, y]$ is $\{1\}$.

   Even though 5 divides the last coefficient of the second polynomial of F the results over $\mathbb{Z}_5$ and $\mathbb{Q}$ correspond.

2. Let $F = \{y^2 - 7, yx^3 + 3x^2, y^3x - 2y^2\} \subset \mathbb{Q}[x, y]$. Let the power products be ordered according to the total degree ordering with $x < y$. The normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$ is $G = \{1\}$.

   7 divides the second coefficient of the first polynomial in F. $G \bmod 7 = \{1\}$, but that is not a Gröbner basis for Ideal(F) in $\mathbb{Z}_7[x, y]$. Actually, the normed reduced Gröbner basis for Ideal(F) in $\mathbb{Z}_7[x, y]$ is $\{x^2, y^2\}$. So 7 is not "lucky" for F.

   $G \bmod 5 = \{1\}$, but that is not a Gröbner basis for Ideal(F) in $\mathbb{Z}_5[x, y]$. Actually, the normed reduced Gröbner basis for Ideal(F) in $\mathbb{Z}_5[x, y]$ is $\{y - x, x^2 - 2\}$. So 5 is not "lucky" for F.

   From this example we see that it is not sufficient to just require that p does not divide any coefficient of F and G. In this case the transformation matrix T is

   $$\begin{pmatrix} \frac{1}{140}y^3x^3 + \frac{3}{140}y^2x^2 - \frac{1}{7} & \frac{1}{20}y^2 & -\frac{1}{140}y^2x^2 - \frac{1}{28}yx - \frac{1}{14} \end{pmatrix}$$

   and T has no representation in $\mathbb{Z}_5[x, y]$ or $\mathbb{Z}_7[x, y]$.

3. Let $F = \{yx^2 + 3x^2 - 7, y^2x - 2y^2, y^2x^2 - 2yx\} \subset \mathbb{Q}[x, y]$. Let the power products be ordered according to the total degree ordering with $x < y$. The normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$ is $G = \{y, x^2 - 7/3\}$.

   In this case the transformation matrix T is

   $$\begin{pmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \end{pmatrix}$$

with

$$
\begin{aligned}
t_{1,1} &= -\frac{4}{63}y^2x^2 + \frac{11}{378}y^2x + \frac{37}{189}y^2 + \frac{8}{63}yx - \frac{1}{7}y, \\
t_{1,2} &= \frac{8}{63}x^2 - \frac{47}{189}x - \frac{37}{54}, \\
t_{1,3} &= \frac{4}{63}yx^2 - \frac{11}{378}yx + \frac{4}{21}x^2 - \frac{37}{189}y - \frac{3}{14}x - \frac{4}{9}, \\
t_{2,1} &= \frac{1}{27}y^2x^2 - \frac{5}{162}y^2x - \frac{7}{81}y^2 - \frac{2}{27}yx + \frac{1}{3}, \\
t_{2,2} &= -\frac{2}{27}x^2 + \frac{14}{81}x + \frac{49}{162}, \\
t_{2,3} &= -\frac{1}{27}yx^2 + \frac{5}{162}yx - \frac{1}{9}x^2 + \frac{7}{81}y + \frac{1}{6}x + \frac{7}{27}.
\end{aligned}
$$

Although 11, 37 and 47 divide coefficients in the transformation matrix T, the results over $\mathbb{Z}_{11}, \mathbb{Z}_{37}, \mathbb{Z}_{47}$ and $\mathbb{Q}$ correspond.

4. (Example (c) from [Winkler, 1988]) Let $F = \{4x + 2y^2z + 1, 16x^2 + 4xy^2 - 4z + 1, -2x^2z + x + 2y^2\} \subset \mathbb{Q}[x, y, z]$. Let the power products be ordered according to the purely lexicographic ordering with $z < y < x$. The normed reduced Gröbner basis for F in $\mathbb{Q}[x, y, z]$ has the leading power products $\{z^7, y^2, x\}$ (see [Winkler et al., 1985]).

   The normed reduced Gröbner basis for Ideal(F) in $\mathbb{Z}_7[x, y, z]$ has the leading power products $\{z^6, zy^2, y^4, x\}$. So 7 is not "lucky" for F.

5. (Example (d) from [Winkler, 1988]) Let $F = \{yx + 4x^2 + 3x, yx^2 + 9x^2 - y\} \subset \mathbb{Q}[x, y]$. Let the power products be ordered according to the total degree ordering with $x < y$. The normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$ is $G = \{yx + 4x^2 + 3x, y^2 - 16x^2 + 3y - 12x, x^3 - 3/2x^2 + 1/4y\}$.

   The normed reduced Gröbner basis for Ideal(F) in $\mathbb{Z}_5[x, y]$ is $\{yx - x^2 - 2x, y^2 - x^2 - 2y - 2x, x^3 + x^2 - y\}$. So 5 is "lucky" for F.

□

[Ebert, 1983] observes that in general the number of polynomials in the normed reduced Gröbner basis G for some $F \subset \mathbb{Q}[X]$ can be greater than, equal to, or less than the number of polynomials in the normed reduced Gröbner basis G_p for F in $\mathbb{Z}_p[X]$, for some prime p. In Example 12 we see that also the leading power products of G can be greater than, equal to, less than, or incomparable to the leading power products of G_p. From Example 12 we also see, that it is not sufficient to just require that p does not divide any coefficient of F and G. Fortunately, there are only finitely many such "unlucky" primes.

Theorem 14 (Theorem 1 in [Winkler, 1988]) *Let $F = \{f_1, \ldots, f_m\}$ be a finite sequence of polynomials in $\mathbb{Q}[X]$, $G = \{g_1, \ldots, g_n\}$ the normed reduced Gröbner basis* *for F in $\mathbb{Q}[X]$. For almost all primes p the images $\bar{F} = F \bmod p$, $\bar{G} = G \bmod p$ exist and $\bar{G}$ is the normed reduced Gröbner basis for $\bar{F}$ in $\mathbb{Z}_p[X]$.*

Proof: Let T, R be transformation matrices over $\mathbb{Q}[X]$ such that

$$\begin{aligned} G &= T \cdot F, \\ F &= R \cdot G, \end{aligned}$$

and S the syzygy matrix over $\mathbb{Q}[X]$ such that

$$S \cdot G = 0$$

and the rows of S are the syzygies generated from reductions of all S-polynomials of G to 0 w.r.t. G. Let p be such that it does not divide any coefficient of F, G, T, R and S. Then $\bar{F} = F \bmod p$, $\bar{G} = G \bmod p$, $\bar{T} = T \bmod p$, $\bar{R} = R \bmod p$ exist and

$$\left.\begin{aligned} \bar{G} &\equiv \bar{T} \cdot \bar{F} \\ \bar{F} &\equiv \bar{R} \cdot \bar{G} \end{aligned}\right. \quad \pmod{p}.$$

So $\mathrm{Ideal}(\bar{F}) = \mathrm{Ideal}(\bar{G})$ in $\mathbb{Z}_p[X]$.

Furthermore, $\bar{S} = S \bmod p$ exists,

$$\bar{S} \cdot \bar{G} \equiv 0 \pmod{p},$$

and the rows of the syzygy matrix $\bar{S}$ correspond to reductions of the S-polynomials of $\bar{G}$ to 0 w.r.t. $\bar{G}$ in $\mathbb{Z}_p[X]$. So by Theorem 2 $\bar{G}$ is the normed reduced Gröbner basis for $\bar{F}$ in $\mathbb{Z}_p[X]$.

For a given F there are only finitely many primes that divide some coefficient of F, G, T, R or S. □

Definition 18 ["luckiness"] Let $F \subset \mathbb{Q}[X]$, G the normed reduced Gröbner basis for F. Let p be a rational prime. We call p *lucky* iff there are transformation matrices T, R and a syzygy matrix S such that p does not divide any coefficient in F, G, T, R and S. □

3.3 The Lifting of a Gröbner Basis

From now on we assume that p is a rational prime. Let $F = (f_1, \ldots, f_m)^T$ be a finite sequence of polynomials in $\mathbb{Q}[X]$. Let $G^{(0)}$ be a Gröbner basis for F in $\mathbb{Z}_p[X]$, and $T^{(0)}, R^{(0)}$ transformation matrices over $\mathbb{Z}_p[X]$, and $S^{(0)}$ a syzygy matrix for $G^{(0)}$ over $\mathbb{Z}_p[X]$ such that

$$\left.\begin{aligned} T^{(0)} \cdot F &\equiv G^{(0)} \\ R^{(0)} \cdot G^{(0)} &\equiv F \\ S^{(0)} \cdot G^{(0)} &\equiv 0 \end{aligned}\right. \quad \pmod{p} \tag{3.1}$$

where the rows of $S^{(0)}$ correspond to reductions of the S-polynomials of $G^{(0)}$ to 0 w.r.t. $G^{(0)}$.

The first two congruences in (3.1) guarantee that F and $G^{(0)}$ generate the same ideal in $\mathbb{Z}_p[X]$. The third congruence and the special form of the rows guarantee that $G^{(0)}$ is a Gröbner basis. Our aim is now to lift the congruences (3.1) to congruences modulo p^i for large enough i. In more detail: we want a sequence $(G^{(i)},T^{(i)},R^{(i)},S^{(i)})$ for $0 \leq i < t$ with t a positive integer or ∞ such that

$$\begin{array}{rcl} T^{(i)} \cdot F & \equiv & G^{(i)} \\ R^{(i)} \cdot G^{(i)} & \equiv & F \\ S^{(i)} \cdot G^{(i)} & \equiv & 0 \end{array} \quad \pmod{p^{i+1}},$$

$$\begin{array}{rcl} G^{(i)} & \equiv & G^{(i-1)} \\ T^{(i)} & \equiv & T^{(i-1)} \\ R^{(i)} & \equiv & R^{(i-1)} \\ S^{(i)} & \equiv & S^{(i-1)} \end{array} \quad \pmod{p^i} \text{ if } i \geq 1.$$

We want $G^{(i)}$ to be a vector of polynomials over $\mathbb{Z}_{p^{i+1}}[X]$, $T^{(i)}$, $R^{(i)}$ the transformation matrices and $S^{(i)}$ the syzygy matrix over $\mathbb{Z}_{p^{i+1}}[X]$ where the rows correspond to the "reductions of S-polynomial of $G^{(i)}$ to 0 w.r.t. $G^{(i)}$". Since $\mathbb{Z}_{p^{i+1}}[X]$ is no longer a field we don't have the notion of reductions, S-polynomials and Gröbner bases. We now extend the definition of the reduction relation to a commutative ring. From now on we assume that $\mathbf{R}$ is a commutative ring, $\mathbf{R}[X]$ the ring of polynomials in $x_1, \ldots, x_\nu$ over $\mathbf{R}$.

Definition 19 Let $F \subseteq \mathbf{R}[X]$, $g, h \in \mathbf{R}[X]$. We say that g *reduces to* h *w.r.t.* F, abbreviated $g \rightarrow_F h$, iff there are $f \in F$, $t \in [X]$, such that

$$c := \frac{\operatorname{coeff}(g, t \cdot \operatorname{lpp}(f))}{\operatorname{lc}(f)} \neq 0 \text{ and } h = g - c \cdot t \cdot f.$$

□

For the existence of the coefficient $c = \operatorname{coeff}(g, \operatorname{lpp}(f) \cdot t)/\operatorname{lc}(f) \in \mathbf{R}$ it is necessary that $\operatorname{lc}(f)$ is invertible in $\mathbf{R}$.

The definition of S-polynomial was chosen so that it can be extended to a commutative ring without modification.

Definition 20 Let $f, g \in \mathbf{R}[X] \setminus \{0\}$. The *S-polynomial* of f and g is

$$\text{S-pol}(f, g) := \operatorname{lc}(g) \frac{\operatorname{lcm}(\operatorname{lpp}(f), \operatorname{lpp}(g))}{\operatorname{lpp}(f)} f - \operatorname{lc}(f) \frac{\operatorname{lcm}(\operatorname{lpp}(f), \operatorname{lpp}(g))}{\operatorname{lpp}(g)} g.$$

□

Note however that the notion of Gröbner basis can not be extended so easily to polynomials over rings. For further information look at [Buchberger, 1985].

The work required in the lifting process depends, of course, on the number of congruences that have to be lifted. So we first present two methods to reduce this work before we present the lifting algorithm itself.

The transformation matrix $T^{(0)}$ in 3.1 can contain very high power products, since F is not a Gröbner basis and therefore the representation of $G^{(0)}$ in terms of F is not just a reduction. This problem does not occur with $R^{(0)}$ and $S^{(0)}$, since $G^{(0)}$ is a Gröbner basis. The following theorem shows that actually we don't really have to consider the equivalence $T^{(0)}.F \equiv G^{(0)} \pmod p$.

Theorem 15 *Let G and G' be Gröbner bases in* $\mathbf{K}[X]$, $\mathrm{lpp}(G) = \mathrm{lpp}(G')$ *and* $G \subseteq \mathrm{Ideal}(G')$. *Then* $\mathrm{Ideal}(G) = \mathrm{Ideal}(G')$.

Proof: From $G \subseteq \mathrm{Ideal}(G')$ we know that $\mathrm{Ideal}(G) \subseteq \mathrm{Ideal}(G')$. We now show that the other inclusion direction $\mathrm{Ideal}(G) \supseteq \mathrm{Ideal}(G')$ is true, too. Let $f \in \mathrm{Ideal}(G')$ be a nonzero polynomial. f is reducible w.r.t. G' and, since $\mathrm{lpp}(G) = \mathrm{lpp}(G')$, it is also reducible w.r.t. G. Let f' be the reduction result w.r.t. G. $f' \in \mathrm{Ideal}(G')$ since $G \subseteq \mathrm{Ideal}(G')$. This step can be repeated for nonzero f' and since the reduction w.r.t. G is Noetherian, we finally get that every nonzero $f \in \mathrm{Ideal}(G')$ can be reduced to 0 w.r.t. G. Therefore, $\mathrm{Ideal}(G) \supseteq \mathrm{Ideal}(G')$. □

Corollary 1 Let $F \subset \mathbb{Q}[X]$, , G the normed reduced Gröbner basis for F. Let p be a lucky prime for F and $G^{(0)}$ the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$. Let G', R', S' be the result of a lifting process of the congruences $R^{(0)} \cdot G^{(0)} \equiv F$ and $S^{(0)} \cdot G^{(0)} \equiv 0$ such that $R' \cdot G' = F$ and $S' \cdot G' = 0$. If G' is a normed reduced Gröbner basis and $\mathrm{lpp}(G) = \mathrm{lpp}(G')$ then $G = G'$.

Proof: G' is a Gröbner basis and $\mathrm{lpp}(G) = \mathrm{lpp}(G')$. From $R' \cdot G' = F$ we know that $G \subseteq \mathrm{Ideal}(F) \subseteq \mathrm{Ideal}(G')$. So by Theorem 15 and the uniqueness of normed reduced Gröbner bases(Theorem 4) we get $G = G'$. □

Provided that in the lifting process the leading power products of the approximations to the basis G' remain unchanged (3.1) is reduced to

$$\begin{array}{rcl} R^{(0)} \cdot G^{(0)} & \equiv & F \\ S^{(0)} \cdot G^{(0)} & \equiv & 0 \end{array} \pmod p \tag{3.2}$$

where the rows of $S^{(0)}$ correspond to reductions of the S-polynomials of $G^{(0)}$ to 0 w.r.t. $G^{(0)}$.

Another method to reduce the work required in the lifting process is to eliminate rows in the syzygy matrix. The second congruence and the special form of the rows of $S^{(0)}$ in (3.2) guarantee that $G^{(0)}$ is a Gröbner basis. In fact various criteria can be used to eliminate unnecessary S-polynomials or rows of $S^{(0)}$, respectively the Product Criterion (Theorem 5) and the Chain Criterion (Theorem6).

When reducing the number of rows in $S^{(0)}$, one has to make sure that this works uniformly for coefficient domains $\mathbb{Z}_{p^i}$ for $i \geq 1$. In addition to the usual requirement for such a criterion, namely that checking only the necessary S-polynomials for reducibility to 0 suffices to ensure the reducibility to 0 for all S-polynomials, we demand that it should work uniformly for coefficient domains $\mathbb{Z}_{p^i}[X], i \geq 1$. I.e. if $G^{(0)}$ is a sequence of n polynomials over $\mathbb{Z}_p$ and $G^{(i)}$ is a sequence of polynomials over $\mathbb{Z}_{p^i}$ such that

$$G^{(i)} \equiv G^{(0)} \pmod{p}$$

and

$$\mathrm{pp}(G_j^{(i)}) = \mathrm{pp}(G_j^{(0)}) \text{ for } 1 \leq j \leq n$$

then S-pol$(G_j^{(i)}, G_k^{(i)})$ is necessary if and only if S-pol$(G_j^{(0)}, G_k^{(0)})$ is necessary for $1 \leq j < l \leq n$. The Product Criterion and the Chain Criterion satisfy our requirements. If $R^{(0)}$ is such that it contains only rows corresponding to necessary S-polynomials of $G^{(0)}$, then it is also called a syzygy matrix. In the following we will use this modified version of the SyzygyMatrix algorithm:

Algorithm 12 (SyzygyMatrixFromNecessarySpols)

$S \leftarrow$ SyzygyMatrixFromNecessarySpols(G)
  input: $G = \{g_1, \ldots, g_n\} \subset \mathbf{K}[X]$, a reduced Gröbner basis.
  output: S a syzygy matrix with l rows and n columns over $\mathbf{K}[X]$
    such that $S \cdot G = 0$ where each row corresponds to the syzygy derived
    from the reduction of a necessary S-pol(g_j, g_k) to 0 w.r.t. G.

BEGIN
  $S :=$ empty matrix
  FOR all pairs (i, j) with $1 \leq i < j \leq n$ DO
    IF $\neg$ProductCriterion(g_i, g_j) & $\neg$ChainCriterion$(g_i, g_j, G, \emptyset)$ THEN
      $h :=$ S-pol$(g_i, g_j) = c_j \cdot t_i \cdot g_i - c_i \cdot t_j \cdot g_j$,
        where $c_i = \mathrm{lc}(g_i)$, $c_j = \mathrm{lc}(g_j)$,
          $t = \mathrm{lcm}(\mathrm{lpp}(g_i), \mathrm{lpp}(g_j))$,
          $t_i = t/\mathrm{lpp}(g_i)$, $t_j = t/\mathrm{lpp}(g_j)$.
      $(h_1, \ldots, h_n) :=$ ReductionVector(h, G)
      Add $(0, \ldots, 0, \underbrace{c_j \cdot t_i}_{i}, 0, \ldots, 0, \underbrace{-c_i \cdot t_j}_{j}, 0, \ldots, 0) - (h_1, \ldots, h_n)$ as last row in S
    END
  END
END SyzygyMatrixFromNecessarySpols.

The following technicality will be used in subsequent proofs.

Lemma 4 *Let p be a rational prime, $\mathcal{A}$ be the set of polynomials in $\mathbb{Q}[X]$ no coefficient of which has a denominator divisible by p. Let I be an ideal in $\mathbb{Q}[X]$, $I' = I \cap \mathcal{A}$ and I_p the set of polynomials f in $\mathbb{Z}_p$ such that $f \equiv g \pmod{p}$ for some $g \in I'$. I_p is an ideal in $\mathbb{Z}_p$. Let $h \in I', h' \in \mathbb{Z}_p$ be such that $p^{i-1} \cdot h' \equiv h \pmod{p^i}$. Then $h' \in I_p$.*

Proof: Since h is a multiple of p^{i-1} there exists some $g \in \mathcal{A}$ such that $h = p^{i-1} \cdot g$ and $h' = g \bmod p$. Since $h \in I'$, also $g \in I'$, so $h \; \in I_p$. □

Definition 21 Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, $S^{(0)}$ a syzygy matrix for $G^{(0)}$; specifically, let the jth row of $S^{(0)}$ be the syzygy of $G^{(0)}$ derived from the reduction of S-pol$(G^{(0)}_{l(j)}, G^{(0)}_{r(j)})$ to 0 w.r.t. $G^{(0)}$. Let t be a positive integer or ∞, and for $0 \le i < t$ let $G^{(i)}, R^{(i)}, S^{(i)}$ be matrices over $\mathbb{Z}_{p^{i+1}}[X]$ such that

$$\begin{array}{rcl} R^{(i)} \cdot G^{(i)} & \equiv & F \\ S^{(i)} \cdot G^{(i)} & \equiv & 0 \end{array} \qquad \pmod{p^{i+1}},$$

$$\begin{array}{rcl} G^{(i)} & \equiv & G^{(i-1)} \\ R^{(i)} & \equiv & R^{(i-1)} \\ S^{(i)} & \equiv & S^{(i-1)} \end{array} \qquad \pmod{p^i} \text{ if } i \ge 1,$$

$$\left.\begin{array}{rcl} \mathrm{lc}(G^{(i)}_j) & = & 1 \\ \mathrm{pp}(G^{(i)}_j) & = & \mathrm{pp}(G^{(0)}_j) \end{array}\right\} \text{ for } 1 \le j \le n.$$

Then $L = ((G^{(i)}, R^{(i)}, S^{(i)}))_{0 \le i < t}$ is a *lifting sequence for F* modulo p.

For $0 \le j < t$ the lifting sequence L is called *reducing up to j* iff for all $0 \le i \le j$ the rows of $S^{(i)}$ are the syzygies of $G^{(i)}$ corresponding to reductions of the necessary S-polynomials of $G^{(i)}$ to 0 in $\mathbb{Z}_{p^{i+1}}[X]$; specifically, the jth row of $S^{(i)}$ is the syzygy of $G^{(i)}$ corresponding to the reduction of S-pol$(G^{(i)}_{l(j)}, G^{(i)}_{r(j)})$ to 0 w.r.t. $G^{(i)}$. □

Theorem 16 (existence of a lifting sequence) (Theorem 3 in [Winkler, 1988]) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$. If*

$$\begin{array}{rcl} R^{(0)} \cdot G^{(0)} & \equiv & F \\ S^{(0)} \cdot G^{(0)} & \equiv & 0 \end{array} \qquad \pmod{p}$$

holds for some (m, n)-matrix $R^{(0)}$ over $\mathbb{Z}_p[X]$, and some (l, n)-matrix $S^{(0)}$ over $\mathbb{Z}_p[X]$, then for every $i \in \mathbb{N}$ there exist $G^{(i)}, R^{(i)}, S^{(i)}$ over $\mathbb{Z}_{p^i}[X]$ such that

$$\begin{array}{rcl} R^{(i-1)} \cdot G^{(i-1)} & \equiv & F \\ S^{(i-1)} \cdot G^{(i-1)} & \equiv & 0 \end{array} \qquad \pmod{p^i} \tag{3.3}$$

and

$$\begin{array}{rcl} R^{(i-1)} & \equiv & R^{(i-2)} \\ S^{(i-1)} & \equiv & S^{(i-2)} \end{array} \quad (\text{mod } p^{i-1}) \text{ } \textit{if } i > 1 \tag{3.4}$$

and

$$G^{(i-1)} = G \text{ mod } p^i. \tag{3.5}$$

The increments

$$\tfrac{1}{p^{i-1}}(G^{(i-1)} - G^{(i-2)}), \quad \tfrac{1}{p^{i-1}}(R^{(i-1)} - R^{(i-2)}), \quad \tfrac{1}{p^{i-1}}(S^{(i-1)} - S^{(i-2)})$$

can be chosen as a solution of a system of linear equations in $\mathbb{Z}_p[X]$ *and*

$$\left.\begin{array}{rcl} \text{lc}(G_j^{(i)}) & = & 1 \\ \text{pp}(G_j^{(i)}) & = & \text{pp}(G_j^{(0)}) \end{array}\right\} \textit{ for } 1 \leq j \leq n.$$

Proof: By induction on i. For $i = 1$ the statements (3.3) and (3.5) obviously hold. (3.4) is void. Now let $i > 1$. By the induction hypothesis there exist $G^{(i-2)}, R^{(i-2)}, S^{(i-2)}$ over $\mathbb{Z}_{p^{i-1}}[X]$ such that (3.3) and (3.5) hold for $i-1$. Let $G' \in \mathbb{Z}_p[X]^n$, $G^{(i-1)} \in \mathbb{Z}_{p^i}[X]^n$ be such that

$$G^{(i-1)} = G^{(i-2)} + p^{i-1} \cdot G' = G \text{ mod } p^i.$$

So (3.5) obviously holds. We need to construct matrices R', S' over $\mathbb{Z}_p[X]$ such that for

$$\begin{array}{rcl} R^{(i-1)} & = & R^{(i-2)} + p^{i-1} \cdot R' \\ S^{(i-1)} & = & S^{(i-2)} + p^{i-1} \cdot S' \end{array}$$

the congruence (3.3) hold (the definition of $R^{(i-1)}$ and $S^{(i-1)}$ directly implies that (3.4) holds). So we have to solve

$$\begin{array}{rcl} F & \equiv & R^{(i-1)} \cdot G^{(i-1)} \\ & \equiv & R^{(i-2)} \cdot G^{(i-1)} + p^{i-1} \cdot R' \cdot G^{(i-1)} \\ & \equiv & R^{(i-2)} \cdot G^{(i-2)} + p^{i-1} \cdot R^{(i-2)} \cdot G' + p^{i-1} \cdot R' \cdot G^{(0)} \quad (\text{mod } p^i). \end{array}$$

Rewriting (3.6) as

$$F - R^{(i-2)} \cdot G^{(i-2)} - p^{i-1} \cdot R^{(i-2)} \cdot G' \equiv p^{i-1} \cdot R' \cdot G^{(0)} \quad (\text{mod } p^i), \tag{3.6}$$

we see that the image modulo p^i of the left hand side of (3.6) is a vector

$$\begin{pmatrix} h_1^{(i-1)} \\ \vdots \\ h_m^{(i-1)} \end{pmatrix}$$

where each component $h_k^{(i-1)}$ is the image modulo p^i of a polynomial $h_k := F_k - R^{(i-2)} \cdot G$ in $\mathrm{Ideal}(F) = \mathrm{Ideal}(G)$, no coefficient of which has a denominator divisible by p. Because of (3.3) for $i-1$, the left hand side is divisible by p^{i-1}. So we have to solve

$$\frac{1}{p^{i-1}} \cdot \begin{pmatrix} h_1^{(i-1)} \\ \vdots \\ h_m^{(i-1)} \end{pmatrix} \equiv R' \cdot G^{(0)} \pmod{p}.$$

For every $k, 1 \leq k \leq m, h_k \in \mathrm{Ideal}(G)$, so by Lemma 4 $h_k^{i-1}/p^{i-1} \in \mathrm{Ideal}(G^{(0)})$ and therefore we can take the elements of R' as the multiplicands used in the reductions of $h_1^{i-1}/p^{i-1}, \ldots, h_m^{i-1}/p^{i-1}$ to 0 w.r.t. $G^{(0)}$.

We still have to find a matrix S' such that

$$(S^{(i-2)} + p^{i-1} \cdot S') \cdot (G^{(i-2)} + p^{i-1} \cdot G') \equiv 0 \pmod{p^i}.$$

So we have to solve

$$S^{(i-2)} \cdot G^{(i-2)} + p^{i-1} \cdot S^{(i-2)} \cdot G' \equiv p^{i-1} \cdot S' \cdot G^{(0)} \pmod{p^i}. \tag{3.7}$$

The image modulo p^i of the left hand side of (3.7) is a vector of images of polynomial in $\mathrm{Ideal}(F) = \mathrm{Ideal}(G)$. Because of (3.3) for $i-1$, the left hand side is divisible by p^{i-1}. So we have to solve

$$\frac{1}{p^{i-1}} \cdot (S^{(i-2)} \cdot G^{(i-2)} + p^{i-1} \cdot S^{(i-2)} \cdot G') \equiv S' \cdot G^{(0)} \pmod{p}.$$

All the images modulo p of the polynomials on the left hand side are in $\mathrm{Ideal}(G^{(0)})$ (by Lemma 4), so the reduction to 0 w.r.t. $G^{(0)}$ yields the matrix S'.

We have to solve

$$\begin{aligned} F &\equiv (R^{(i-2)} + p^{i-1} \cdot R') \cdot (G^{(i-2)} + p^{i-1} \cdot G') \\ &\equiv R^{(i-2)} \cdot G^{(i-2)} + p^{i-1} \cdot R' \cdot G^{(i-2)} + p^{i-1} \cdot R^{(i-2)} \cdot G' \\ &\equiv R^{(i-2)} \cdot G^{(i-2)} + p^{i-1} \cdot (R' \cdot G^{(0)} + R^{(0)} \cdot G') \pmod{p^i}. \end{aligned}$$

Collecting terms and doing the same for $S^{(i-1)} \cdot G^{(i-1)} \equiv 0 \pmod{p^i}$ we get the system of linear equations for $G' = (G'_1, \ldots, G'_n)^T, R' = (R'_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$ and $S' = (S'_{ij})_{1 \leq i \leq l, 1 \leq j \leq n}$

$$\begin{aligned} R' \cdot G^{(0)} + R^{(0)} \cdot G' &\equiv \frac{1}{p^{i-1}} \cdot (F - R^{(i-2)} \cdot G^{(i-2)}) \pmod{p} \\ S' \cdot G^{(0)} + S^{(0)} \cdot G' &\equiv \frac{1}{p^{i-1}} \cdot (-S^{(i-2)} \cdot G^{(i-2)}) \pmod{p}. \end{aligned}$$

Moreover,

$$\left. \begin{aligned} \mathrm{lc}(G_j^{(i)}) &= 1 \\ \mathrm{pp}(G'_j) &\subseteq \mathrm{pp}(G_j^{(0)}) \setminus \mathrm{lpp}(G_j^{(0)}) \end{aligned} \right\} \text{ for } 1 \leq j \leq n.$$

□

We have shown in Theorem 16 that for every lucky prime p for F there is indeed a lifting sequence. Now we have to deal with the uniqueness of such a lifting sequence. It turns out that the components $G^{(i)}$ are indeed uniquely determined, whereas the components $R^{(i)}, S^{(i)}$ are usually not. Starting from the normed reduced Gröbner basis $G^{(0)}$ modulo p we get $G^{(i)}$, a sequence of polynomials in $\mathbb{Z}_{p^{i+1}}[X]$. $\mathbb{Z}_{p^{i+1}}$ is not a field, and since we have not introduced the notion of a Gröbner basis over a ring we have to prove some theorems before we can give a proof for the uniqueness of a lifting sequence.

Theorem 17 (Theorem 4(a) in [Winkler, 1988]) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting sequence for F modulo p. Then all the S-polynomials of $G^{(i)}$ can be reduced to 0 w.r.t. $G^{(i)}$.*

Proof: Since $G^{(0)}$ is a Gröbner basis, we know that all the S-polynomials $G^{(0)}$ can be reduced to 0 w.r.t. $G^{(0)}$, and actually the rows of $S^{(0)}$ are just syzygies derived from the reductions of the necessary S-polynomials of $G^{(0)}$.

If L is reducing up to i, then obviously the S-polynomial of $G^{(i)}$ can be reduced to 0 w.r.t. $G^{(i)}$. Otherwise let $j, 0 < j < i$, be the smallest index such that the rows of $S^{(j)}$ do not correspond to the syzygies derived from the reductions of the necessary S-polynomials of $G^{(j)}$ to 0 w.r.t. $G^{(j)}$.

The proof consists of two steps. First we will show how to construct a lifting sequence $\bar{L} = ((G^{(k)}, R^{(k)}, \bar{S}^{(k)}))_{0 \leq k \leq j}$ that is reducing up to j from the lifting sequence L that is reducing up to $j-1$. And then we will give a way to continue the lifting sequence $\bar{L}$ from j to i.

Let S' be the (l, n)-matrix over $\mathbb{Z}_p[X]$ such that

$$S^{(j)} = S^{(j-1)} + p^j \cdot S'.$$

The elements of $S^{(j-1)}$ can be considered as the multiplicands used in an "incomplete" reduction of the necessary S-polynomials of $G^{(j)}$ w.r.t. $G^{(j)}$. A step in this incomplete reduction consists of partially reducing an occurring term, possibly leaving a coefficient which is a multiple of p^j. The result of this incomplete reduction is a vector of the form

$$S^{(j-1)} \cdot G^{(j)} = p^j \cdot h = p^j \cdot \begin{pmatrix} h_1 \\ \vdots \\ h_l \end{pmatrix}$$

for some $h_1, \ldots, h_l \in \mathbb{Z}_p[X]$. Since L is a lifting sequence we have

$$0 \equiv S^{(j)} \cdot G^{(j)}$$

$$\begin{aligned} &\equiv S^{(j-1)} \cdot G^{(j)} + p^j \cdot S' \cdot G^{(j)} \\ &\equiv p^j \cdot h + p^j \cdot S' \cdot G^{(0)} \pmod{p^{j+1}}, \end{aligned}$$

or equivalently,

$$h \equiv -S' \cdot G^{(0)} \pmod{p}.$$

Obviously $h_k \in \mathrm{Ideal}(G^{(0)})$ for $1 \leq k \leq l$, so h_k can be reduced to 0 w.r.t. $G^{(0)}$. Actually, whenever a term has to be reduced which is also reduced in the incomplete reduction above, then the corresponding basis polynomials can be used in corresponding reduction steps. Let the elements of the matrix $\bar{S}'$ be the multiplicands used in this reduction of h to 0, so that $h \equiv -\bar{S}' \cdot G^{(0)} \pmod{p}$. For the matrix

$$P^{(j)} := S' - \bar{S}'$$

we have

$$P^{(j)} \cdot G^{(0)} \equiv \pmod{p}.$$

If we now let

$$\bar{S}^{(j)} = S^{(j)} - p^j \cdot P^{(j)},$$

then the rows of $\bar{S}^{(j)}$ correspond to the syzygies from reductions of the necessary S-polynomial of $G^{(j)}$ to 0 w.r.t. $G^{(j)}$. I.e.

$$\bar{L} := ((G^{(0)}, R^{(0)}, S^{(0)}), \ldots, (G^{(j-1)}, R^{(j-1)}, S^{(j-1)}), (G^{(j)}, R^{(j)}, \bar{S}^{(j)}))$$

is reducing up to j. And

$$(\bar{S}^{(j)} - S^{(j)}) \cdot G^{(j)} \equiv p^j \cdot \bar{S}' \cdot G^{(j)} \equiv 0 \pmod{p^{j+1}}.$$

In the second step we construct matrices $\bar{S}^{(j+1)}, \ldots, \bar{S}^{(i)}$ such that

$$\begin{aligned} \bar{L} := ((G^{(0)}, R^{(0)}, S^{(0)}), \ldots, (G^{(j-1)}, R^{(j-1)}, S^{(j-1)}), \\ (G^{(j)}, R^{(j)}, \bar{S}^{(j)}), \ldots, , (G^{(i)}, R^{(i)}, \bar{S}^{(i)})) \end{aligned}$$

is a lifting sequence for F modulo p. Actually we prove the following: for every k with $j \leq k \leq i$ there is a matrix $P^{(k)}$ over $\mathbb{Z}_p[X]$ such that for

$$\begin{aligned} \bar{S}^{(k)} &:= S^{(k)} - \sum_{r=j}^{k} p^r \cdot P^{(r)}, \\ \bar{S}^{(j-1)} &:= S^{(j-1)} \end{aligned}$$

we have

$$\bar{S}^{(k)} \cdot G^{(k)} \equiv 0 \pmod{p^{k+1}}, \tag{3.8}$$

$$\bar{S}^{(k)} \equiv \bar{S}^{(k-1)} \pmod{p^k}, \tag{3.9}$$

and

$$\left(\sum_{r=j}^{k} p^r \cdot P^{(r)}\right) \cdot G^{(k-j)} \equiv 0 \pmod{p^{k+1}}. \tag{3.10}$$

By induction on k.

Let $k = j$. Then we have

$$\bar{S}^{(j)} \cdot G^{(j)} = S^{(j)} \cdot G^{(j)} - p^j \cdot P^{(j)} \cdot G^{(j)} \equiv 0 \pmod{p^{j+1}},$$

$$\bar{S}^{(j)} = S^{(j)} - p^j \cdot P^{(j)} \equiv S^{(j-1)} \equiv \bar{S}^{(j-1)} \pmod{p^j},$$

and

$$p^j \cdot P^{(j)} \cdot G^{(0)} \equiv 0 \pmod{p^{j+1}}.$$

Now consider k such that $j < k \leq i$. Let S'' be a (l, n)-matrix over $\mathbb{Z}_p[X]$ such that

$$S^{(k)} = S^{(k-1)} + p^k \cdot S''.$$

By the induction hypothesis we have

$$(\bar{S}^{(k-1)} + p^k \cdot S'') \cdot G^{(k)} \equiv (S^{(k-1)} + p^k \cdot S'') \cdot G^{(k)} - \left(\sum_{r=j}^{k-1} p^r \cdot P^{(r)}\right) \cdot G^{(k)} \equiv 0 \pmod{p^k}.$$

So

$$-\left(\sum_{r=j}^{k-1} p^r \cdot P^{(r)}\right) \cdot G^{(k-j)} \equiv p^k \cdot \begin{pmatrix} h'_1 \\ \vdots \\ h'_l \end{pmatrix} =: p^k \cdot h' \pmod{p^{k+1}}$$

for some $h'_1, \ldots, h'_l \in \mathbb{Z}_p[X]$. Each $h'_t, 1 \leq t \leq l$ is the homomorphic image of a polynomial in Ideal(G) modulo p, no coefficient of which has a denominator divisible by p. So by Lemma 4 $h'_t \in$ Ideal$(G^{(0)})$ and therefore can be reduced to 0 w.r.t. $G^{(0)}$ for every $t, 1 \leq t \leq l$. Collecting the multiplicands used in these reductions in the matrix $P^{(k)}$, we get

$$h' \equiv P^{(k)} \cdot G^{(0)} \pmod{p}.$$

For this definition we get

$$\begin{aligned}
\bar{S}^{(k)} \cdot G^{(k)} &\equiv \left(S^{(k)} - \sum_{r=j}^{k} p^r \cdot P^{(r)}\right) \cdot G^{(k)} \\
&\equiv S^{(k)} \cdot G^{(k)} - \left(\sum_{r=j}^{k-1} p^r \cdot P^{(r)}\right) \cdot G^{(k)} - p^k \cdot P^{(k)} \cdot G^{(k)} \\
&\equiv S^{(k)} \cdot G^{(k)} + p^k \cdot h' - p^k \cdot h' \\
&\equiv 0 \pmod{p^{k+1}},
\end{aligned}$$

so (3.8) holds. Regarding (3.9) we have

$$\bar{S}^{(k)} = S^{(k)} - \sum_{r=j}^{k} p^r \cdot P^{(r)} \equiv S^{(k-1)} - \sum_{r=j}^{k-1} p^r \cdot P^{(r)} = \bar{S}^{(k-1)} \pmod{p^k}.$$

And finally,

$$\begin{aligned}\left(\sum_{r=j}^{k} p^r \cdot P^{(r)}\right) \cdot G^{(k-j)} &= \left(\sum_{r=j}^{k-1} p^r \cdot P^{(r)}\right) \cdot G^{(k-j)} + p^k \cdot P^{(k)} \cdot G^{(k-j)} \\ &\equiv -p^k \cdot h' + p^k \cdot h' \\ &= 0 \pmod{p^{k+1}}\end{aligned}$$

proves (3.10).

So we have constructed a lifting sequence $\bar{L}$ that is reducing up to j. Repeating this process, we finally get a lifting sequence which is reducing up to i. Thus, all the necessary S-polynomial of $G^{(i)}$ are reducible to 0 w.r.t. $G^{(i)}$, and therefore all S-polynomial are reducible to 0 w.r.t. $G^{(i)}$.

Theorem 18 *Let* $\mathbf{R}$ *a commutative ring and* $G = \{g_1, \ldots, g_n\} \subset \mathbf{R}[X]$ *such that all S-polynomials of* G *can be reduced to 0 w.r.t.* G *and all* g_i *monic. Then every polynomial* $h \in \mathrm{Ideal}(G)$ *can be reduced to 0 w.r.t.* G.

Proof: A polynomial h in $\mathrm{Ideal}(G)$ can be written as

$$h = \sum_{i=1}^{n} h_i \cdot g_i. \tag{3.11}$$

for some $h_i \in \mathbf{R}[X]$. Let

$$p := \max_{\lhd}\{\mathrm{lpp}(h_i \cdot g_i) \mid 1 \leq i \leq n\}$$

be the highest power product w.r.t. $\lhd$ occurring in some summand on the right hand side of (3.11). Assume $\mathrm{lpp}(h) \lhd p$. Then the coefficients of u on the right hand side of (3.11) cancel.

Let $i_1 < \cdots < i_k$ be those indices such that $\mathrm{lpp}(h_{i_j} \cdot g_{i_j}) = p$. We have $k \geq 2$. Let $f_l \in \mathbf{R}[X], 1 \leq l \leq n$ be the polynomials extracted from the reduction of the S-polynomial $\text{S-pol}(g_{i_1}, g_{i_2})$ to 0 w.r.t. G. With $t := \mathrm{lcm}(\mathrm{lpp}(g_{i_1}), \mathrm{lpp}(g_{i_2}))$ we have

$$\text{S-pol}(g_{i_1}, g_{i_2}) = \mathrm{lc}(g_{i_2}) \cdot \frac{t}{\mathrm{lpp}(g_{i_1})} \cdot g_{i_1} - \mathrm{lc}(g_{i_1}) \cdot \frac{t}{\mathrm{lpp}(g_{i_2})} \cdot g_{i_2} = \sum_{l=1}^{n} f_l \cdot g_l,$$

and $\mathrm{lpp}(f_l \cdot g_l) \lhd \mathrm{lcm}(\mathrm{lpp}(g_{i_1}), \mathrm{lpp}(g_{i_2}))$ for $1 \leq l \leq n$.

Then for
$$c := \frac{\mathrm{lc}(h_{i_1} \cdot g_{i_1})}{\mathrm{lc}(g_{i_1}) \cdot \mathrm{lc}(g_{i_2})}$$
we have
$$\begin{aligned} h &= \sum_{l=1}^{n} h_l \cdot g_l \\ &= \sum_{l=1}^{n} h_l \cdot g_l - c \cdot \frac{p}{t}. \left(\text{S-pol}(g_{i_1}, g_{i_2}) - \sum_{l=1}^{n} f_l \cdot g_l\right) \\ &= \sum_{l=1}^{n} \left(h_l - c \cdot \frac{p}{t} \cdot f_l\right) \cdot g_l - \frac{\mathrm{lc}(h_{i_1} \cdot g_{i_1})}{\mathrm{lc}(g_{i_1})} \cdot \frac{p}{\mathrm{lpp}(g_{i_1})} \cdot g_{i_1} + \frac{\mathrm{lc}(h_{i_1} \cdot g_{i_1})}{\mathrm{lc}(g_{i_2})} \cdot \frac{p}{\mathrm{lpp}(g_{i_2})} \cdot g_{i_2} \end{aligned}$$

Since $\mathrm{lpp}(f_l \cdot g_l) \lhd \mathrm{lcm}(\mathrm{lpp}(g_{i_1}), \mathrm{lpp}(g_{i_2})) = t$ we have eliminated at least i_1 from the positions that contribute to the highest power product p in the summands on the right hand side of (3.11). We can iterate this process until we finally reach a situation, where only two positions contribute to the power product p. Now $\mathrm{lc}(h_{i_1} \cdot g_{i_1}) = \mathrm{lc}(h_{i_2} \cdot g_{i_2})$ and by repeating the elimination process once more we can decrease the power product p. So, finally, we have
$$h = \sum_{i=1}^{n} \bar{h}_i \cdot g_i.$$
where the leading power products on both sides of will be the same. Then the terms in the polynomials $\bar{h}_i$ can be used as the multiplicands in the reduction of h to 0 w.r.t. G. □

Theorem 19 (Theorem 4(b) in [Winkler, 1988]) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting sequence for F modulo p. Then every polynomial $h \in G^{(i)} (\subseteq \mathbb{Z}_{p^{i+1}}[X])$ can be reduced to 0 w.r.t. $G^{(i)}$.*

Proof: By Theorem 17 and Theorem 18. □

Theorem 20 (Theorem 4(c) in [Winkler, 1988]) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting sequence for F modulo p. Then* $\mathrm{Ideal}(F) = \mathrm{Ideal}(G^{(i)})$ *as ideals in $\mathbb{Z}_{p^{i+1}}[X]$.*

Proof: Since p is lucky, $\bar{G}^{(i)} := G \bmod p^{i+1}$ exists and $\mathrm{Ideal}(\bar{G}^{(i)}) = \mathrm{Ideal}(F)$ as ideals in $\mathbb{Z}_{p^i}[X]$. Certainly $\mathrm{Ideal}(F) \subseteq \mathrm{Ideal}(G^{(i)})$, since $R^{(i)} \cdot G^{(i)} \equiv F \pmod{p^{i+1}}$. Assume that $\mathrm{Ideal}(G^{(i)}) \not\subseteq \mathrm{Ideal}(F)$. Let h be a polynomial in $\mathrm{Ideal}(G^{(i)}) \setminus \mathrm{Ideal}(F)$. By Theorem 19, h is reducible w.r.t. $G^{(i)}$. But every polynomial which is reducible w.r.t. $G^{(i)}$ is also reducible w.r.t. $\bar{G}^{(i)}$, since the leading power products in both cases are the same and the basis elements are monic. Say $h \rightarrow_{\bar{G}^{(i)}} g$. Then $g \in \mathrm{Ideal}(G^{(i)})$, since $\mathrm{Ideal}(\bar{G}^{(i)}) = \mathrm{Ideal}(F) \subseteq \mathrm{Ideal}(G^{(i)})$. But $g \notin \mathrm{Ideal}(G^{(i)})$, for otherwise $h \in \mathrm{Ideal}(F)$. So $g \in \mathrm{Ideal}(G^{(i)}) \setminus \mathrm{Ideal}(F)$, and by the same reasoning as for h we get that g can be reduced w.r.t. $\bar{G}^{(i)}$. That process can be repeated indefinitely, leading to an infinite chain of reductions. This, however, is impossible because the reduction relation $\rightarrow$ is Noetherian. □

We finally show the uniqueness of the $G^{(i)}$ in the lifting sequence.

Theorem 21 (uniqueness of lifting sequences) (Theorem 4(d) in [Winkler, 1988]) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting sequence for F modulo p. Then $G^{(i)} = G \bmod p^{i+1}$.*

Proof: Since p is lucky, $\bar{G}^{(i)} = G \bmod p^{i+1}$ exists. Assume that there exists a $k, 1 \leq k \leq n$, such that $G_k^{(i)} \neq \bar{G}_k^{(i)}$. Since $\mathrm{lpp}(G_k^{(i)}) = \mathrm{lpp}(\bar{G}_k^{(i)})$, $\bar{G}_k^{(i)}$ can be reduced by $G_k^{(i)}$ to some nonzero polynomial $h \in \mathbb{Z}_p[X]$. h is irreducible w.r.t. $G^{(i)}$ because the same power product occur in $G^{(i)}$ and $\bar{G}^{(i)}$ and $G^{(i)}$ is reduced. So, by Theorem 19, $h \notin \mathrm{Ideal}(G^{(i)})$, and therefore $\mathrm{Ideal}(G^{(i)}) \neq \mathrm{Ideal}(\bar{G}^{(i)}) = \mathrm{Ideal}(F)$ in $\mathbb{Z}_{p^{i+1}}[X]$. This, however, is a contradiction to Theorem 20. So our assumption must be wrong and $G^{(i)} = G \bmod p^{i+1}$. □

If p is a lucky prime for F, then by Theorem 16 the congruence (3.2) can be extended to a lifting sequence of arbitrary length and by Theorem 21 such a lifting sequence guarantees that we get the correct approximation of the normed reduced Gröbner basis G for F. In order to compute $(G^{(i)}, R^{(i)}, S^{(i)})$ from $(G^{(i-1)}, R^{(i-1)}, S^{(i-1)})$ we have to solve the system of linear equations

$$\begin{aligned} R' \cdot G^{(0)} + R^{(0)} \cdot G' &\equiv \frac{1}{p^i} \cdot (F - R^{(i-1)} \cdot G^{(i-1)}) \pmod{p} \\ S' \cdot G^{(0)} + S^{(0)} \cdot G' &\equiv \frac{1}{p^i} \cdot (-S^{(i-1)} \cdot G^{(i-1)}) \pmod{p}. \end{aligned}$$

for $G' = (g'_1, \ldots, g'_n)^T, R' = (r'_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$ and $S' = (s'_{ij})_{1 \leq i \leq l, 1 \leq j \leq n}$ over $\mathbb{Z}_p[X]$. Or written in matrix form we have to solve the system

$$U \cdot c = v^{(i)} \tag{3.12}$$

where U is the $(m+l, n\cdot(m+l+1))$-matrix

$$\left(\begin{array}{ccc|ccc|c}
G_1^{(0)} \dots G_n^{(0)} & & & & & & \\
0 & \ddots & 0 & & 0 & & R^{(0)} \\
 & & G_1^{(0)} \dots G_n^{(0)} & & & & \\
\hline
 & & & G_1^{(0)} \dots G_n^{(0)} & & & \\
 & 0 & & 0 & \ddots & 0 & S^{(0)} \\
 & & & & & G_1^{(0)} \dots G_n^{(0)} &
\end{array}\right)$$

and $v^{(i)}$ the vector of length $(m+l)$

$$v^{(i)} = \frac{1}{p^i}\cdot\left(\begin{array}{c} F - R^{(i-1)}\cdot G^{(i-1)} \\ \hline -S^{(i-1)}\cdot G^{(i-1)} \end{array}\right).$$

If the vector of length $(n\cdot(m+l+1))$

$$(r'_{11},\dots,r'_{1n},\dots,r'_{m1},\dots,r'_{mn},s'_{11},\dots,s'_{1n},\dots,s'_{m1},\dots,s'_{mn},g'_1,\dots,g'_n)^T$$

is a solution such that $\mathrm{pp}(g'_j)\subseteq \mathrm{pp}(G_j^{(0)})\setminus\{\mathrm{lpp}(G_j^{(0)})\}$ for $1\le j\le n$, then

$$\begin{aligned}
G^{(i)} &= G^{(i)} + p^i\cdot\begin{pmatrix} g'_1 \\ \vdots \\ g'_n \end{pmatrix}, \\
R^{(i)} &= R^{(i)} + p^i\cdot\begin{pmatrix} r'_{11} & \dots & r'_{1n} \\ \vdots & & \vdots \\ r'_{m1} & \dots & r'_{mn} \end{pmatrix}, \\
S^{(i)} &= S^{(i)} + p^i\cdot\begin{pmatrix} s'_{11} & \dots & s'_{1n} \\ \vdots & & \vdots \\ s'_{l1} & \dots & s'_{ln} \end{pmatrix}.
\end{aligned}$$

For the computation of a basis

$$C' = \{\begin{pmatrix} \vdots \\ g'^{(1)}_1 \\ \vdots \\ g'^{(1)}_n \end{pmatrix},\dots,\begin{pmatrix} \vdots \\ g'^{(1)}_q \\ \vdots \\ g'^{(q)}_n \end{pmatrix}\}$$

for the module of solutions of the homogeneous system

$$U\cdot c = 0 \tag{3.13}$$

and a solution to the inhomogeneous system (3.12) we refer to [Winkler, 1992], [Furukawa et al., 1986] and [Möller and Mora, 1986]. Observe that for every $1 \leq j, k \leq n$ there exists a solution $c' = (\ldots, c'_{n(m+l)+1}, \ldots, c'_{n(m+l)+n})^T$ of (3.13) such that $c'_{n(m+l)+j} = G_k^{(0)}$ and $c'_{n(m+l)+t} = 0$ for $1 \leq t \leq n, t \neq j$.

The only remaining complication is that during the lifting process no new power products are allowed to be introduced in the basis and the basis polynomials should stay monic. In order to satisfy this requirement, the basis C' of the solution module of the homogeneous system is transformed to a basis

$$C = \left\{ \begin{pmatrix} \vdots \\ g_1^{(1)} \\ \vdots \\ g_n^{(1)} \end{pmatrix}, \ldots, \begin{pmatrix} \vdots \\ g_q^{(1)} \\ \vdots \\ g_n^{(q)} \end{pmatrix} \right\}$$

in which the last n components constitute a Gröbner basis for the module generated by these components (see [Möller and Mora, 1986]; [Winkler, 1987a]).

Lemma 5 *Let $i \in \mathbb{N}, G^{(0)} = (G_1^{(0)}, \ldots, G_n^{(0)})^T$ the normed reduced Gröbner basis for $F = (F_1, \ldots, F_m)^T$ in $\mathbb{Z}_p[X]$, p lucky for F. Let C be a basis for the module of solutions of (3.13) such that the last n components of the vectors in C constitute a Gröbner basis for the module generated by these components. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j < i}$ be a lifting sequence for F modulo p.*

(a) *There exists a vector $g' = (g'_1, \ldots, g'_n)^T$ over $\mathbb{Z}_p[X]$ such that the last n components of every solution*

$$\bar{c} = (\bar{c}_1, \ldots, \bar{c}_{n(m+l)}, \bar{c}_{n(m+l)+1}, \ldots, \bar{c}_{n(m+l)+n})^T$$

of (3.12) satisfying the additional requirement

$$\mathrm{pp}(\bar{c}_{n(m+l)+k}) \subseteq \mathrm{pp}(G_k^{(0)}) \setminus \{\mathrm{lpp}(G_k^{(0)})\} \text{ for } 1 \leq k \leq n \tag{3.14}$$

are equal to g'.

(b) *From an arbitrary solution $\hat{c}$ of (3.12) a solution $\bar{c}$ of (3.12) which satisfies the additional requirement (3.14) can be computed.*

Proof: (a) By Theorems 16 and 21.

(b) Let $\hat{c} = (\ldots, h_1, \ldots, h_n)^T$. Then $h - g' = (h_1 - g'_1, \ldots, h_n - g'_n)^T$ are the last n components of a solution of (3.13). So they can be reduced to 0 w.r.t. the last n

components of

$$C = \{ \underbrace{\begin{pmatrix} \vdots \\ g_1^{(1)} \\ \vdots \\ g_n^{(1)} \end{pmatrix}}_{c_1}, \ldots, \underbrace{\begin{pmatrix} \vdots \\ g_q^{(1)} \\ \vdots \\ g_n^{(q)} \end{pmatrix}}_{c_r} \}.$$

If we let the possible coefficients in g' be parameters and reduce $h - g'$ w.r.t. the last n components of C, we get linear equations for these parameters that have a unique solution (by (a)), and we get a representation of $h - g'$ as a linear combination of the vectors consisting of the last n components of C,

$$g' = h + \sum_{j=1}^{r} a_j . \begin{pmatrix} g_1^{(1)} \\ \vdots \\ g_n^{(1)} \end{pmatrix} .$$

Then

$$\bar{c} = \hat{c} + \sum_{j=1}^{r} a_j \cdot C_j$$

is a solution of (3.12) which satisfies the additional requirement (3.14). $\square$

The left hand side U of (3.12) remains the same throughout the lifting process. The right hand side varies and it can be efficiently computed as

$$\begin{aligned} v^{(i+1)} &= \frac{1}{p^{i+1}} \cdot \begin{pmatrix} F - R^{(i)} \cdot G^{(i)} \\ \hline -S^{(i)} \cdot G^{(i)} \end{pmatrix} \\ &= \frac{1}{p^{i+1}} \cdot \begin{pmatrix} F - R^{(i-1)} \cdot G^{(i-1)} \\ \hline S^{(i-1)} \cdot G^{(i-1)} \end{pmatrix} - \frac{1}{p} \cdot \begin{pmatrix} R' \cdot G^{(i-1)} + R^{(i-1)} \cdot G' + p^i \cdot R' \cdot G' \\ \hline S' \cdot G^{(i-1)} + S^{(i-1)} \cdot G' + p^i \cdot R' \cdot G' \end{pmatrix} \\ &= \frac{1}{p} \cdot \left(v^{(i)} - \begin{pmatrix} R' \cdot G^{(i)} + R^{(i-1)} \cdot G' \\ \hline S' \cdot G^{(i)} + S^{(i-1)} \cdot G' \end{pmatrix} \right), \end{aligned}$$

where G', R', S' are the matrices over $\mathbb{Z}_p[X]$ such that

$$\begin{aligned} G^{(i)} &= G^{(i-1)} + p^i \cdot G', \\ R^{(i)} &= R^{(i-1)} + p^i \cdot R', \\ S^{(i)} &= S^{(i-1)} + p^i \cdot S'. \end{aligned}$$

3.4 The Lifting Algorithm

Suppose we knew how to determine a reasonable bound N for the coefficients of the normed reduced Gröbner basis G for a given F and how to select a lucky prime p. Then the following algorithm could be used to compute G by lifting the corresponding Gröbner basis modulo p just high enough so that every coefficient in G has a unique representation. If such a bound N is not known, then the algorithm Lift can be considered as p-adically approximating G.

Algorithm 13 (Lift)

$G \leftarrow \mathrm{Lift}(F, p, N)$
  input: $F \in \mathbb{Q}[X]^m$,
    p, a lucky prime for F,
    N, a bound on the coefficients in G,
      the normed reduced Gröbner basis G for F,
      i.e. every coefficient of G is in $\mathcal{F}_{p,N}$.
  output: G, the normed reduced Gröbner basisfor F.
BEGIN
  // length of lifting sequence
  $K :=$ such that $2N^2 + 1 \le p^K$
  // initialization
  $G^{(0)} = (G_1^{(0)}, \ldots, G_n^{(0)}) := \mathrm{GröbnerBasis}(F)$ modulo p
  $R^{(0)} := \mathrm{ReductionMatrix}(F, G^{(0)})$ modulo p
  $S^{(0)} := \mathrm{SyzygyMatrixFromNecessarySpols}(G^{(0)})$ modulo p
  // solution of homogeneous system
  $U :=$ the matrix as in (3.13)
  Compute a basis

$$C := \{ \begin{pmatrix} \vdots \\ g_1^{(1)} \\ \vdots \\ g_n^{(1)} \end{pmatrix}, \ldots, \begin{pmatrix} \vdots \\ g_q^{(1)} \\ \vdots \\ g_n^{(q)} \end{pmatrix} \}$$

  for the module of solutions of $U \cdot c = 0$ in $\mathbb{Z}_p[X]$, such that
  the last n components of the basis vectors in C constitute a Gröbner basis
  for the module generated by these components.

$$v^{(1)} := \frac{1}{p} \cdot \begin{pmatrix} F - R^{(0)} \cdot G^{(0)} \\ \hline -S^{(0)} \cdot G^{(0)} \end{pmatrix}$$

  FOR $i := 1$ TO K DO
    // lift to congruence modulo p^{i+1}
    Compute a particular solution

$(r'_{11}, \ldots, r'_{1n}, \ldots, r'_{m1}, \ldots, r'_{mn}, s'_{11}, \ldots, s'_{1n}, \ldots, s'_{m1}, \ldots, s'_{mn}, g'_1, \ldots, g'_n)^T$
of $U \cdot C = v^{(i)}$ over $\mathbb{Z}_p[X]$ such that
$\mathrm{pp}(\bar{c}_{n(m+l)+k}) \subseteq \mathrm{pp}(G_k^{(0)}) \setminus \{\mathrm{lpp}(G_k^{(0)})\}$ for $1 \leq k \leq n$.
// plug in the solution
$G' := (g'_1, \ldots, g'_n)^T$
$R' := (r'_{kj})_{1 \leq k \leq m, 1 \leq j \leq n}$
$S' := (s'_{kj})_{1 \leq k \leq l, 1 \leq j \leq n}$
$G^{(i)} := G^{(i-1)} + p^i \cdot G'$
$R^{(i)} := R^{(i-1)} + p^i \cdot R'$
$S^{(i)} := S^{(i-1)} + p^i \cdot S'$

$$v^{(i+1)} := \frac{1}{p} \cdot \left(v^{(i)} - \left(\begin{array}{c} R' \cdot G^{(i)} + R^{(i-1)} \cdot G' \\ \hline S' \cdot G^{(i)} + S^{(i-1)} \cdot G' \end{array} \right) \right)$$

END
// convert the coefficients back to $\mathbb{Q}$
$G :=$ Apply FareyToRational(a, p, K) to every coefficient a in $G^{(K)}$
RETURN G
END Lift.

Example 13 We carry out the algorithm Lift for computing a p-adic approximation of a Gröbner basis in $\mathbb{Q}[x, y]$. As the ordering $<$ of the power products we choose the lexicographic ordering with $x < y$. Let the input basis F be

$$F = \begin{pmatrix} x^2y^2 - \frac{14}{3}x^3y \\ xy^3 - \frac{12}{7}x^2y^2 + 8x \\ y^4 + 8y - 24x^3 \end{pmatrix}.$$

The normed reduced Gröbner basis for F over $\mathbb{Q}$ is

$$G = \begin{pmatrix} x^2 \\ xy^3 + 8x \\ y^4 + 8y \end{pmatrix}.$$

So the coefficients of both F and G are relatively small, whereas the highest coefficient appearing in the computation of G is 1098247/1190896.

As the prime we choose 5, which is indeed a lucky prime for F. The Gröbner basis for F modulo 5 is

$$G^{(0)} = \begin{pmatrix} x^2 \\ xy^3 - 2x \\ y^4 - 2y \end{pmatrix},$$

the transformation matrix from $G^{(0)}$ to F modulo 5 is

$$R^{(0)} = \begin{pmatrix} y^2 + 2yx & 0 & 0 \\ -y^2 & 1 & 0 \\ x & 0 & 1 \end{pmatrix}$$

and the rows of $S^{(0)}$ are the syzygies derived from the necessary S-polynomials S-pol$(G_1^{(0)}, G_2^{(0)})$, S-pol$(G_2^{(0)}, G_3^{(0)})$ of $G^{(0)}$

$$S^{(0)} = \begin{pmatrix} y^3-2 & -x & 0 \\ 0 & y & -x \end{pmatrix}.$$

In lifting the Gröbner basis $G^{(0)}$ modulo 5 to a basis modulo some higher power of 5 we have to solve a system of inhomogeneous linear equations. The left hand side of this system is the matrix U, whose transposed is

$$\begin{pmatrix}
x^2 & 0 & 0 & 0 & 0 \\
xy^3-2x & 0 & 0 & 0 & 0 \\
y^4-2y & 0 & 0 & 0 & 0 \\
0 & x^2 & 0 & 0 & 0 \\
0 & xy^3-2x & 0 & 0 & 0 \\
0 & y^4-2y & 0 & 0 & 0 \\
0 & 0 & x^2 & 0 & 0 \\
0 & 0 & xy^3-2x & 0 & 0 \\
0 & 0 & y^4-2y & 0 & 0 \\
0 & 0 & 0 & x^2 & 0 \\
0 & 0 & 0 & xy^3-2x & 0 \\
0 & 0 & 0 & y^4-2y & 0 \\
0 & 0 & 0 & 0 & x^2 \\
0 & 0 & 0 & 0 & xy^3-2x \\
0 & 0 & 0 & 0 & y^4-2y \\
y^2+2yx & -y^2 & x & y^3-2 & 0 \\
0 & 1 & 0 & -x & y \\
0 & 0 & 1 & 0 & -x
\end{pmatrix}.$$

We compute a basis C for the module of solutions of $U \cdot c = 0$ in $\mathbb{Z}_5[x,y]$, such that the last 3 components of the basis vectors in C constitute a Gröbner basis for the module generated by these components. C includes the vectors

$$\begin{pmatrix} \vdots \\ x^2 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \vdots \\ xy^3-2x \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \vdots \\ y^4-2y \\ 0 \\ 0 \end{pmatrix},$$

$$\begin{pmatrix} \vdots \\ 0 \\ x^2 \\ 0 \end{pmatrix}, \begin{pmatrix} \vdots \\ 0 \\ xy^3-2x \\ 0 \end{pmatrix}, \begin{pmatrix} \vdots \\ 0 \\ y^4-2y \\ 0 \end{pmatrix},$$

$$\begin{pmatrix} \vdots \\ 0 \\ 0 \\ x^2 \end{pmatrix}, \begin{pmatrix} \vdots \\ 0 \\ 0 \\ xy^3-2x \end{pmatrix}, \begin{pmatrix} \vdots \\ 0 \\ 0 \\ y^4-2y \end{pmatrix}.$$

In the first lifting step we compute a particular solution to the system

$$U \cdot c = \frac{1}{5} \cdot \begin{pmatrix} F - R^{(0)} \cdot G^{(0)} \\ \hline -S^{(0)} \cdot G^{(0)} \end{pmatrix} = \begin{pmatrix} 2x^3y \\ 2x^2y^2 \\ 0 \\ 2x^2 \\ 0 \end{pmatrix} \tag{3.15}$$

over $\mathbb{Z}_5[x,y]$. The basis vectors in C are used to reduce $\hat{c}$ to a solution $\bar{c}$ of (3.15) satisfying the requirement that none of the last 3 components of $\bar{c}$ contains any power product that does not already appear in the corresponding elements of $G^{(0)}$ and, moreover, the coefficient of $\mathrm{lpp}(G_j^{(0)})$ in $\bar{c}_{15+j}$ is 0, for $1 \le j \le 3$. As the vector $\bar{c}$ we get

$$\bar{c} = \begin{pmatrix} 2xy \\ 0 \\ 0 \\ -xy^4+2x^2y^3+2y^2+2xy+x^2 \\ x^2y-2x^3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ -xy^5+2x^2y^4+2xy^2+x^2y \\ 0 \\ x^3y-2x^4 \\ 0 \\ 2x \\ 2y \end{pmatrix}.$$

Now we use the components of $\bar{c}$ to update the approximations to $G^{(1)}, R^{(1)}, S^{(1)}$, getting

$$G^{(1)} = \begin{pmatrix} x^2 \\ xy^3+8x \\ y^4+8y \end{pmatrix},$$

$$R^{(0)} = \begin{pmatrix} y^2+12yx & 0 & 0 \\ -5xy^4+10x^2y^3+9y^2+10xy+5x^2 & 5x^2y-10x^3+1 & 0 \\ x & 0 & 1 \end{pmatrix},$$

$$S^{(0)} = \begin{pmatrix} y^3 + 8 & -x & 0 \\ -5xy^5 + 10x^2y^4 + 10xy^2 + 5x^2y & y & 5x^3y - 10x^2 - x \end{pmatrix}.$$

Two further lifting steps will not change the coefficients in $G^{(1)}$ but are needed for the back mapping to $\mathbb{Q}$. $G^{(1)} = G^{(3)}$ is then the normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$. $\square$

Chapter 4

p-adic Lifting of the Coefficients

Now we want to present an improved version of the lifting algorithm Lift presented in Chapter 3. The algorithm presented there lifts a Gröbner basis $G^{(0)}$ in $\mathbb{Z}_p[X]$ for F by solving linear equations over $\mathbb{Z}_p[X]$. These systems of linear equation are solved by Gröbner basis computation over $\mathbb{Z}_p[X]$ or the module $\mathbb{Z}_p[X]^r$ and these computations are very costly in terms of time and space. In this chapter we lift the coefficients of the polynomials instead of the polynomials themselves. And as we will see this can be done by solving systems of linear equations over $\mathbb{Z}_p$. This can be done by Gaussian elimination, which has a much better complexity than the Gröbner basis algorithm.

One of the requirements in the lifting process was that $\mathrm{pp}(G_j^{(i)}) = \mathrm{pp}(G_j^{(0)})$. The idea is now to expand this requirement to the transformation matrices $R^{(i)}$ and syzygy matrices $S^{(i)}$. So what we need is that for a given Gröbner basis $G^{(0)}$ and matrices $R^{(0)}, S^{(0)}$ over $\mathbb{Z}_p$ there are a Gröbner basis G and matrices R, S such that all power products occurring in polynomials in $G^{(0)}, R^{(0)}, S^{(0)}$ also occur in their respective polynomial in G, R, S. This can be guaranteed by a suitable restriction of the notion of luckiness of a prime p.

In this chapter we describe a more practical approach to computing p-adic approximations of Gröbner bases. For this reason we will deal only with transformation matrices resulting from the Buchberger algorithm for the computation of a Gröbner basis. This implies that we have to slightly modify our definition of luckiness.

4.1 Lucky Primes

Definition 22 ["luckiness"] Let F be a finite sequence of polynomials in $\mathbb{Q}[X]$, G the normed reduced Gröbner basis for F. Let p be a rational prime. p is *lucky* for F if and only if for all transformation matrices T resulting from an application of the Buchberger algorithm to F, all transformation matrices R resulting from an application of algorithm ReductionMatrix and all syzygy matrices S for G resulting from an application of algorithm SyzygyMatrix, p does not divide any coefficient in F, G, T, R and S. □

We need the following theorem by D. König to proof that there are only finitely many unlucky primes.

Lemma 6 (The "infinity lemma") *In any infinite oriented tree for which every vertex has finite degree, there is an "infinite path from the root," i.e., an infinite sequence of vertices $V_0, V_1, V_2, \ldots$ in which V_0 is the root and $\mathrm{fin}(e[V_{j+1}]) = V_j$ for all $j \geq 0$.*

Proof: See [Knuth, 1973]. □

Corollary 2 Let F be a finite sequence of polynomials in $\mathbb{Q}[X]$. There are only finitely many unlucky primes for F.

Proof: Let G be a finite sequence of polynomials in $\mathbb{Q}[X]$, the normed reduced Gröbner basis for F in $\mathbb{Q}[X]$. In every step of the Gröbner basis computation for a given F (and admissible ordering $\lhd$) there are only a finite number of choices to continue, i.e. every vertex has finite degree. Ultimately everyone of these choices terminates ([Buchberger, 1965]). So there is no "infinite path from the root". Therefore the tree has to be finite. So there are only finitely many transformation matrices T resulting from an application of the Buchberger algorithm to F such that $T \cdot F = G$. R and S are traces of reductions to 0, w.r.t. G. Again in every reduction step of the reduction there are only finitely many choices how to reduce further and finally every one of these reduction processes stops with result 0. So there are only finitely many transformation matrices R resulting from an application of algorithm ReductionMatrix and syzygy matrices S resulting from an application of algorithm SyzygyMatrix such that $R \cdot G = F$ and $S \cdot G = 0$. From Theorem 14 we know that for arbitrary but fixed transformation matrices T, R and syzygy matrix S there are only finitely many unlucky primes p. Therefore the number of primes which divide a coefficient in any of the transformation or syzygy matrices is finite. □

4.2 The Lifting of the Coefficients of a Gröbner Basis

As in Section 3.3 we can avoid the lifting of the transformation matrix T and the reductions of unnecessary S-polynomials in S, because the primes lucky by Definition 22 include all primes lucky by Definition 18.

We give a new definition of a lifting sequence including the additional requirement that no new power product are introduced in a polynomial in the lifting process.

Definition 23 Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, $R^{(0)}$ a transformation matrix for $G^{(0)}$ with m rows and n columns resulting from an application of algorithm ReductionMatrix, $S^{(0)}$ a syzygy matrix for $G^{(0)}$ with l rows and n columns resulting from an application of algorithm SyzygyMatrix; specifically, let the jth row of $S^{(0)}$ be the syzygy of $G^{(0)}$ derived from the reduction of S-pol$(G_{j_1}^{(0)}, G_{j_2}^{(0)})$ to 0 w.r.t. $G^{(0)}$ for some $1 \leq j_1 < j_2 \leq n$. Let t be a positive integer or ∞, and for $0 \leq i < t$ let $G^{(i)}, R^{(i)}, S^{(i)}$ be matrices over $\mathbb{Z}_{p^{i+1}}[X]$ such that

$$\begin{array}{rcl} R^{(i)} \cdot G^{(i)} & \equiv & F \\ S^{(i)} \cdot G^{(i)} & \equiv & 0 \end{array} \quad (\text{mod } p^{i+1}), \tag{4.1}$$

$$\begin{array}{rcl} G^{(i)} & \equiv & G^{(i-1)} \\ R^{(i)} & \equiv & R^{(i-1)} \\ S^{(i)} & \equiv & S^{(i-1)} \end{array} \quad (\text{mod } p^{i+1}) \text{ if } i \geq 1, \tag{4.2}$$

$$\left.\begin{array}{rcl} \text{lc}(G_j^{(i)}) & = & 1 \\ \text{pp}(G_j^{(i)}) & = & \text{pp}(G_j^{(0)}) \end{array}\right\} \; for 1 \leq j \leq n, \tag{4.3}$$

$$\begin{array}{rcl} \text{pp}(R_{jk}^{(i)}) & \subseteq & \text{pp}(R_{jk}^{(0)}) \; for 1 \leq j \leq m, 1 \leq k \leq n, \\ \text{pp}(S_{jk}^{(i)}) & \subseteq & \text{pp}(S_{jk}^{(0)}) \; for 1 \leq j \leq l, 1 \leq k \leq n, \end{array} \tag{4.4}$$

and

$$\left.\begin{array}{rcl} \text{lc}(S_{jj_1}^{(i)}) & = & 1 \\ \text{lc}(S_{jj_2}^{(i)}) & = & -1 \\ \text{lpp}(S_{jj_1}^{(i)}) & = & \dfrac{\text{lcm}(\text{lpp}(G_{j_1}^{(0)}), \text{lpp}(G_{j_2}^{(0)}))}{\text{lpp}(G_{j_1}^{(0)})} \\ \text{lpp}(S_{jj_2}^{(i)}) & = & \dfrac{\text{lcm}(\text{lpp}(G_{j_1}^{(0)}), \text{lpp}(G_{j_2}^{(0)}))}{\text{lpp}(G_{j_2}^{(0)})} \end{array}\right\} \text{ for } 1 \leq j \leq l, 1 \leq j_1 < j_2 \leq n. \tag{4.5}$$

Then $L = ((G^{(i)}, R^{(i)}, S^{(i)}))_{0 \leq i < t}$ is a *lifting sequence for* F modulo p. □

Note that a lifting sequence by Definition 23 is also a lifting sequence in the sense of Definition 21.

Theorem 22 (existence of a lifting sequence) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G \in \mathbb{Q}[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Q}[X]$. $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$. $R^{(0)}$ a transformation matrix for $G^{(0)}$ with m rows and n columns, $S^{(0)}$ a syzygy matrix for $G^{(0)}$ with l rows and n columns over $\mathbb{Z}_p[X]$ such that*

$$\begin{array}{rcl} R^{(0)} \cdot G^{(0)} & \equiv & F \\ S^{(0)} \cdot G^{(0)} & \equiv & 0 \end{array} \quad (\text{mod } p)$$

and the jth row of $R^{(0)}$ corresponds to the reduction of F to 0 w.r.t. $G_j^{(0)}$ and the jth row of $S^{(0)}$ is the syzygy of $G^{(0)}$ derived from the reduction of $\text{S-pol}(G_{j_1}^{(0)}, G_{j_2}^{(0)})$ *to 0 w.r.t. $G^{(0)}$ for some $1 \leq j_1 < j_2 \leq n$.*

Then there exist R a transformation matrix for G with m rows and n columns, S a syzygy matrix for G with l rows and n columns over $\mathbb{Q}[X]$ such that

$$\begin{aligned} R \cdot G &= F \\ S \cdot G &= 0 \end{aligned}$$

and the jth row of R corresponds to the reduction of G_j to 0 and the jth row of S is the syzygy of G derived from the reduction of $\text{S-pol}(G_{j_1}, G_{j_2})$ *to 0 w.r.t. G. And for all $i \in \mathbb{N}$*

$$\begin{aligned} G^{(i)} &= G \\ R^{(i)} &= R \mod p^{i+1}, \\ S^{(i)} &= S \end{aligned}$$

$$\begin{aligned} \text{pp}(R_{jk}) &\subseteq \text{pp}(R_{jk}^{(0)}) \ for 1 \leq j \leq m, 1 \leq k \leq n, \\ \text{pp}(S_{jk}) &\subseteq \text{pp}(S_{jk}^{(0)}) \ for 1 \leq j \leq l, 1 \leq k \leq n, \end{aligned}$$

and

$$\left.\begin{aligned} \text{lc}(S_{jj_1}) &= 1 \\ \text{lc}(S_{jj_2}) &= -1 \\ \text{lpp}(S_{jj_1}) &= \frac{\text{lcm}(\text{lpp}(G_{j_1}^{(0)}), \text{lpp}(G_{j_2}^{(0)}))}{\text{lpp}(G_{j_1}^{(0)})} \\ \text{lpp}(S_{jj_2}) &= \frac{\text{lcm}(\text{lpp}(G_{j_1}^{(0)}), \text{lpp}(G_{j_2}^{(0)}))}{\text{lpp}(G_{j_2}^{(0)})} \end{aligned}\right\} \ for\ 1 \leq j \leq l, 1 \leq j_1 < j_2 \leq n.$$

Proof: The steps used in the computation of $R^{(0)}, S^{(0)}$ can be carried out over $\mathbb{Q}[X]$ and yield the sought matrices R, S. Suppose that this can not be done. Then somewhere in the computation a coefficient $c \neq 0$ in a polynomial vanishes modulo p; i.e. $c \equiv 0 \pmod p$. But that can not occur because of the definition of luckiness for the prime p. □

From Theorem 22 we know that for every lucky prime p for F there is indeed a lifting sequence. Again the components $G^{(i)}$ are uniquely determined, whereas the components $R^{(i)}, S^{(i)}$ are usually not. For the uniqueness of the lifting sequence it suffices to show that all S-polynomial of $G^{(i)}$ can be reduced to 0 w.r.t. $G^{(i)}$. Then the Theorems 19, 20 and 21 apply and we get the uniqueness of the lifting sequence.

Theorem 23 *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting* *sequence for F modulo p. Then all the S-polynomials of $G^{(i)}$ can be reduced to 0 w.r.t. $G^{(i)}$.*

Proof: Since $G^{(0)}$ is a Gröbner basis, we know that all the S-polynomials $G^{(0)}$ can be reduced to 0 w.r.t. $G^{(0)}$, and actually the rows of $S^{(0)}$ are just syzygies derived from the reductions of the necessary S-polynomials of $G^{(0)}$.

From the definition of a lifting sequence we know that in every row of $S^{(i)}$ there are two columns with polynomials such that their leading terms are the multiplicands of the computation of the corresponding S-polynomial. Furthermore no new power products are introduced in a polynomial of $S^{(i)}$ compared to $S^{(0)}$. So the other terms from the polynomials of a row are the multiplicands used in the reduction of the S-polynomial to 0 w.r.t. $S^{(i)}$. The reductions are the same as in the case with $i = 0$; just with different coefficients. □

Theorem 24 (uniqueness of lifting sequences) *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j \leq i}$ be a lifting sequence for F modulo p. Then*

(a) *every polynomial $h \in G^{(i)} (\subseteq \mathbb{Z}_{p^{i+1}}[X])$ can be reduced to 0 w.r.t. $G^{(i)}$,*

(b) $\mathrm{Ideal}(F) = \mathrm{Ideal}(G^{(i)})$ *as ideals in $\mathbb{Z}_{p^{i+1}}[X]$,*

(c) $G^{(i)} = G \bmod p^{i+1}$.

Proof: Since every lifting sequence of Definition 23 is also a lifting sequence in the sense of Definition 21,

(a) is just a special case of Theorem 19,

(b) is just a special case of Theorem 20 and

(c) is just a special case of Theorem 21.

□

So by Theorem 22 and 24 we know that for a given F and Gröbner basis $G^{(0)}$ modulo a lucky prime p there exists a lifting sequence exists and it is unique. Furthermore in the lifting process no new power products are introduced in $G^{(i)}$, $R^{(i)}$ and $S^{(i)}$. We now show that we can compute the coefficients of the matrices added in the lifting step by a system of linear equations over $\mathbb{Z}_p$.

Theorem 25 *Let $F \in \mathbb{Q}[X]^m$, p lucky for F, $G^{(0)} \in \mathbb{Z}_p[X]^n$ and the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$, and $i \geq 0$. Let $L = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0 \leq j < i}$ be a* *lifting sequence for F modulo p. Then we can compute $G^{(i)}, R^{(i)}, S^{(i)}$ such that $L' = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0\le j<i+1}$ is a lifting sequence by solving a system of linear equations over $\mathbb{Z}_p$.*

Proof: We need to construct matrices G', R', S' over $\mathbb{Z}_p$ such that for

$$\begin{aligned} G^{(i)} &= G^{(i-1)} + p^i \cdot G' \\ R^{(i)} &= R^{(i-1)} + p^i \cdot R' \\ S^{(i)} &= S^{(i-1)} + p^i \cdot S' \end{aligned}$$

$L' = ((G^{(j)}, R^{(j)}, S^{(j)}))_{0\le j<i+1}$ is a lifting sequence. I.e. (4.1), (4.3), (4.4) and (4.5) of the definition of a lifting sequence are true for $G^{(i)}, R^{(i)}$ and $S^{(i)}$. The definition of $G^{(i)}, R^{(i)}$ and $S^{(i)}$ directly implies that (4.2) holds.

We have to show that the coefficients of the polynomials in the increments G', R' and S' can be chosen as a solution of a system of linear equations in $\mathbb{Z}_p$. We make an ansatz for G', R' and S' by replacing every nonzero coefficient in every polynomial in $G^{(i)}, R^{(i)}$ and $S^{(i)}$ by a new unbound variable $u_j, 1 \le\ j \le t$ for some $t \in \mathbb{N}$ except those leading coefficients mentioned in (4.3) and (4.5). We replace these leading coefficients by zero. With the definitions above we have

$$\left.\begin{aligned} F &\equiv R^{(i)} \cdot G^{(i)} \\ &\equiv R^{(i-1)} \cdot G^{(i-1)} + p^{i+1} \cdot R' \cdot G^{(i-1)} + p^{i+1} \cdot R^{(i-1)} \cdot G' \\ 0 &\equiv S^{(i)} \cdot G^{(i)} \\ &\equiv S^{(i-1)} \cdot G^{(i-1)} + p^{i+1} \cdot S' \cdot G^{(i-1)} + p^{i+1} \cdot S^{(i-1)} \cdot G' \end{aligned}\right. \pmod{p^{i+1}}. \tag{4.6}$$

We rewrite 4.6 as

$$\left.\begin{aligned} \tfrac{1}{p^{i+1}} \cdot \left(F - R^{(i-1)} \cdot G^{(i-1)}\right) &\equiv R' \cdot G^{(0)} + R^{(0)} \cdot G' \\ \tfrac{1}{p^{i+1}} \cdot \left(-S^{(i-1)} \cdot G^{(i-1)}\right) &\equiv S' \cdot G^{(0)} + S^{(0)} \cdot G' \end{aligned}\right. \pmod{p^{i+1}}. \tag{4.7}$$

The coefficients of the polynomials of the right hand sides of (4.7) are linear polynomials in $\mathbb{Z}_p[u_1, \ldots, u_t]$. Since no cancellations of terms take place in polynomials of the right hand sides of (4.7) every power product with a nonzero coefficient in a polynomial of a left hand side of (4.7) appears also in the corresponding polynomial of the right hand side with a nonzero coefficient. We get a system of linear equations by setting coefficients of both sides equal.

We know that this system of linear equations is solvable because of Theorem 22 and that every u_j in a polynomial of G' is uniquely determined by Theorem 24. So every solution of this system of linear equations plugged into G', R', S' would yield $G^{(i)}, R^{(i)}, S^{(i)}$ that fulfill our requirements (4.1), (4.3), (4.4) and (4.5). □

So G' is uniquely defined, whereas the matrices R' and S' are usually not. I.e. there are some variables which can be free chosen; they correspond to coefficients in

R' and S'; variables corresponding to coefficients in G always have a unique solution. We have to solve the system

$$A \cdot \vec{u} = v^{(i)} \tag{4.8}$$

over $\mathbb{Z}_p$, where A is the matrix

$$A = \left(\begin{array}{ccc} \mathrm{coeff}(\mathrm{lc}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1), u_1) & \ldots & \mathrm{coeff}(\mathrm{lc}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1), u_t) \\ \vdots & & \vdots \\ \mathrm{coeff}(\mathrm{tc}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m), u_1) & \ldots & \mathrm{coeff}(\mathrm{tc}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m), u_t) \\ \hline \mathrm{coeff}(\mathrm{lc}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1), u_1) & \ldots & \mathrm{coeff}(\mathrm{lc}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1), u_t) \\ \vdots & & \vdots \\ \mathrm{coeff}(\mathrm{tc}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l), u_1) & \ldots & \mathrm{coeff}(\mathrm{tc}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l), u_t) \end{array} \right),$$

$\vec{u}$ is the vector

$$\vec{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_t \end{pmatrix},$$

and $v^{(i)}$ is the vector

$$v^{(i)} = \left(\begin{array}{c} \mathrm{coeff}(\frac{1}{p^{i+1}} \cdot \left(F - R^{(i-1)} \cdot G^{(i-1)}\right), \mathrm{lpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p^{i+1}} \cdot \left(F - R^{(i-1)} \cdot G^{(i-1)}\right), \mathrm{tpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m)) \\ \hline \mathrm{coeff}(\frac{1}{p^{i+1}} \cdot \left(-S^{(i-1)} \cdot G^{(i-1)}\right), \mathrm{lpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p^{i+1}} \cdot \left(-S^{(i-1)} \cdot G^{(i-1)}\right), \mathrm{tpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l)) \end{array} \right)$$

The left hand side A of (4.8) remains the same throughout the lifting process. The right hand side varies and it can be efficiently computed as

$$v^{(i+1)} = \left(\begin{array}{c} \mathrm{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(R" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \mathrm{lpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(R" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \mathrm{tpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m)) \\ \hline \mathrm{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(S" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \mathrm{lpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(S" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \mathrm{tpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l)) \end{array} \right),$$

where $G", R", S"$ are the matrices over $\mathbb{Z}_p[X]$ such that

$$\begin{aligned} G^{(i)} &= G^{(i-1)} + p^i \cdot G", \\ R^{(i)} &= R^{(i-1)} + p^i \cdot R", \\ S^{(i)} &= S^{(i-1)} + p^i \cdot S". \end{aligned}$$

Note, however, that there is usually no A^{-1} such that for every vector b over $\mathbb{Z}_p$

$$A^{-1} \cdot b$$

is a solution for (4.8).

4.3 The Lifting Algorithm

Provided that we know how to determine a bound B for the coefficients of the normed reduced Gröbner basis G for a given F and how to select a lucky prime p then the following algorithm computes G by lifting the coefficients of the corresponding Gröbner basis modulo p.

Algorithm 14 (LiftCoeff)

$G \leftarrow \mathrm{LiftCoeff}(F, p, N)$
  input: $F \in \mathbb{Q}[X]^m$,
    p, a lucky prime for F,
    N, a bound on the coefficients in G,
      the normed reduced Gröbner basis G for F,
      i.e. every coefficient of G is in $\mathcal{F}_{p,N}$.
  output: G, the normed reduced Gröbner basis for F.
BEGIN
  // length of lifting sequence
  $K :=$ such that $2N^2 + 1 \leq p^K$
  // initialization
  $G^{(0)} = (G_1^{(0)}, \ldots, G_n^{(0)}) := \mathrm{GröbnerBasis}(F)$ modulo p
  $R^{(0)} := \mathrm{ReductionMatrix}(F, G^{(0)})$ modulo p
  $S^{(0)} := \mathrm{SyzygyMatrixFromNecessarySpols}(G^{(0)})$ modulo p
  We make an ansatz for G', R' and S' by replacing every nonzero coefficient
    in every polynomial in $G^{(0)}$, $R^{(0)}$ and $S^{(0)}$ by a new unbound variable u_j
    except those leading coefficients mentioned in (4.3) and (4.5).
  $A :=$ the matrix as in (4.8)

$$v^{(1)} := \left(\begin{array}{c} \mathrm{coeff}(\frac{1}{p} \cdot \left(F - R^{(0)} \cdot G^{(0)}\right), \mathrm{lpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p} \cdot \left(F - R^{(0)} \cdot G^{(0)}\right), \mathrm{tpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m)) \\ \hline \mathrm{coeff}(\frac{1}{p} \cdot \left(-S^{(0)} \cdot G^{(0)}\right), \mathrm{lpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1)) \\ \vdots \\ \mathrm{coeff}(\frac{1}{p} \cdot \left(-S^{(0)} \cdot G^{(0)}\right), \mathrm{tpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l)) \end{array} \right)$$

  FOR $i := 1$ TO K DO

// lift to congruence modulo p^{i+1}
Compute a solution $c^{(i)}$ for $A \cdot \vec{u} = v^{(i)}$
// plug in the solution
$G'' :=$ Replace every u_j in G' by $c_j^{(i)}$
$R'' :=$ Replace every u_j in G' by $c_j^{(i)}$
$S'' :=$ Replace every u_j in G' by $c_j^{(i)}$
$G^{(i)} := G^{(i-1)} + p^i \cdot G'$
$R^{(i)} := R^{(i-1)} + p^i \cdot R'$
$S^{(i)} := S^{(i-1)} + p^i \cdot S'$

$$v^{(i+1)} := \left(\begin{array}{c} \text{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(R" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \text{lpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_1)) \\ \vdots \\ \text{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(R" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \text{tpp}((R' \cdot G^{(0)} + R^{(0)} \cdot G')_m)) \\ \hline \text{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(S" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \text{lpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_1)) \\ \vdots \\ \text{coeff}(\frac{1}{p} \cdot \left(v^{(i)} - \left(S" \cdot G^{(i)} + R^{(i-1)} \cdot G"\right)\right), \text{tpp}((S' \cdot G^{(0)} + S^{(0)} \cdot G')_l)) \end{array} \right)$$

END
// convert the coefficients back to $\mathbb{Q}$
$G :=$ Apply FareyToRational(a, p, K) to every coefficient a in $G^{(K)}$
RETURN G
END LiftCoeff.

We carry out the algorithm LiftCoeff for computing a Gröbner basis in $\mathbb{Q}[x, y]$. We choose the total degree ordering with $x < y$. Let the input basis F be

$$F = \begin{pmatrix} 2yx^2 - yx - x^2 + y \\ yx + 2x^2 + 1 \end{pmatrix}.$$

As the prime we choose 7, which is indeed a lucky prime for F. The Gröbner basis for F modulo 7 is

$$G^{(0)} = \begin{pmatrix} x^3 - 2x^2 - 2y - 3x - 2 \\ y^2 + y + x \\ yx + 2x^2 + 1 \end{pmatrix},$$

the transformation matrix $R^{(0)}$ is

$$R^{(0)} = \begin{pmatrix} 3 & 0 & 2x - 1 \\ 0 & 0 & 1 \end{pmatrix},$$

and the syzygy with the necessary S-polynomials $S^{(0)}$ is

$$S^{(0)} = \begin{pmatrix} 3 & x & -y + 2x - 1 \\ y + 2x & 2 & -x^2 + 2x \end{pmatrix}.$$

We make an ansatz for G', R' and S' by replacing every nonzero coefficient in every polynomial in $G^{(0)}, R^{(0)}$ and $S^{(0)}$ by a new unbound variable u_j except those leading coefficients mentioned in (4.3) and (4.5). Then we get

$$G' = \begin{pmatrix} u_7x^2 + u_6y + u_5x + u_4 \\ u_3y + u_2x \\ u_1x^2 + u_0 \end{pmatrix}$$

and

$$\begin{pmatrix} R' \\ \hline S' \end{pmatrix} = \begin{pmatrix} u_{17} & 0 & u_{16}x + u_{15} \\ 0 & 0 & u_{14} \\ \hline u_{13} & 0 & u_{12}x + u_{11} \\ u_{10}x & u_9 & u_8x \end{pmatrix}.$$

Then we compute the right hand sides of (4.7) and get

$$\begin{pmatrix} R^{(0)} \cdot G' + R' \cdot G^{(0)} \\ \hline S^{(0)} \cdot G' + S' \cdot G^{(0)} \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ \hline h_3 \\ h_4 \end{pmatrix}.$$

with

$$\begin{aligned}
h_1 &= u_{16}yx^2 + (u_{17} + 2u_{16} + 2u_1)x^3 + u_{15}yx + (-2u_{17} + 2u_{15} + 3u_7 - u_1)x^2 + \\
&\quad (-2u_{17} + 3u_6)y + (-3u_{17} + u_{16} + 3u_5 + 2u_0)x + (-2u_{17} + u_{15} + 3u_4 - u_0) \\
h_2 &= u_{14}yx + (2u_{14} + u_1)x^2 + (u_{14} + u_0) \\
h_3 &= (u_{12} - u_1)yx^2 + (u_{13} + 2u_{12} + 2u_1)x^3 + (u_{11} + u_3)yx + \\
&\quad (-2u_{13} + 2u_{11} + 3u_7 + u_2 - u_1)x^2 + (-2u_{13} + 3u_6 - u_0)y + \\
&\quad (-3u_{13} + u_{12} + 3u_5 + 2u_0)x + (-2u_{13} + u_{11} + 3u_4 - u_0) \\
h_4 &= (u_{10} - u_1)x^4 + (u_8 + u_7)yx^2 + (-2u_{10} + 2u_8 + 2u_7 + 2u_1)x^3 + \\
&\quad (u_9 + u_6)y^2 + (-2u_{10} + 2u_6 + u_5)yx + (-3u_{10} + 2u_5 - u_0)x^2 + \\
&\quad (u_9 + u_4 + 2u_3)y + (-2u_{10} + u_9 + u_8 + 2u_4 + 2u_2 + 2u_0)x
\end{aligned}$$

Then left hand sides of 4.7 are

$$v^{(1)} = \frac{1}{p} \cdot \begin{pmatrix} F - R^{(0)} \cdot G^{(0)} \\ \hline -S^{(0)} \cdot G^{(0)} \end{pmatrix} = \begin{pmatrix} -x^3 + x^2 + y + x + 1 \\ 0 \\ \hline -x^3 + x^2 + y + x + 1 \\ yx + x^2 \end{pmatrix}.$$

We write the equation $A \cdot \vec{u} = v^{(1)}$ as polynomials for easier reading

$$\begin{aligned}
u_{16} &= 0, \\
u_{17} + 2u_{16} + 2u_1 &= -1, \\
u_{15} &= 0, \\
-2u_{17} + 2u_{15} + 3u_7 - u_1 &= 1, \\
-2u_{17} + 3u_6 &= 1, \\
-3u_{17} + u_{16} + 3u_5 + 2u_0 &= 1, \\
-2u_{17} + u_{15} + 3u_4 - u_0 &= 1, \\
u_{14} &= 0, \\
2u_{14} + u_1 &= 0, \\
u_{14} + u_0 &= 0, \\
u_{12} - u_1 &= 0, \\
u_{13} + 2u_{12} + 2u_1 &= -1, \\
u_{11} + u_3 &= 0, \\
-2u_{13} + 2u_{11} + 3u_7 + u_2 - u_1 &= 1, \\
-2u_{13} + 3u_6 - u_0 &= 1, \\
-3u_{13} + u_{12} + 3u_5 + 2u_0 &= 1, \\
-2u_{13} + u_{11} + 3u_4 - u_0 &= 1, \\
u_{10} - u_1 &= 0, \\
u_8 + u_7 &= 0, \\
-2u_{10} + 2u_8 + 2u_7 + 2u_1 &= 0, \\
u_9 + u_6 &= 0, \\
-2u_{10} + 2u_6 + u_5 &= 1, \\
-3u_{10} + 2u_5 - u_0 &= 1, \\
u_9 + u_4 + 2u_3 &= 0, \\
-2u_{10} + u_9 + u_8 + 2u_4 + 2u_2 + 2u_0 &= 0.
\end{aligned}$$

The solution $c^{(1)}$ for the first lifting step is

$$\begin{aligned}
u_{17} &= -1, \\
u_{16} &= 0, \\
u_{15} &= 0, \\
u_{14} &= 0, \\
u_{13} &= -1, \\
u_{12} &= 0, \\
u_{11} &= 0, \\
u_{10} &= 0,
\end{aligned}$$

$$\begin{aligned} u_9 &= -2, \\ u_8 &= -2, \\ u_7 &= 2, \\ u_6 &= 2, \\ u_5 &= -3, \\ u_4 &= 2, \\ u_3 &= 0, \\ u_2 &= 0, \\ u_1 &= 0, \\ u_0 &= 0. \end{aligned}$$

Plugging in the solution we get

$$G'' = \begin{pmatrix} 2x^2 + 2y - 3x + 2 \\ 0 \\ 0 \end{pmatrix}$$

and

$$G^{(1)} = \begin{pmatrix} x^3 + 12x^2 + 12y - 24x + 12 \\ y^2 + y + x \\ yx + 2x^2 + 1 \end{pmatrix}.$$

Mapping back the coefficients of $G^{(1)}$ to $\mathbb{Q}$ we get G

$$G = \begin{pmatrix} x^3 - \frac{1}{4}x^2 - \frac{1}{4}y + \frac{1}{2}x - \frac{1}{4} \\ y^2 + y + x \\ yx + 2x^2 + 1 \end{pmatrix},$$

which is the normed reduced Gröbner basis for F in $\mathbb{Q}[x, y]$.

Chapter 5

Using Transformation Matrix T

In this chapter we present an algorithm that exploits the knowledge gained in the computation of the transformation matrix T. The motivation here is not to control the growth of the coefficients but to avoid unnecessary computation. Although the product criterion and chain criterion help to detect S-polynomials that can be reduced to 0, Algorithm GröbnerBasis still computes many S-polynomials which can be reduced to 0. By using the transformation matrix $T^{(0)}$ and Gröbner basis $G^{(0)}$ over $\mathbb{Z}_p[X]$ to compute a transformation matrix T and Gröbner basis G over $\mathbb{Q}[X]$ we try to circumvent the computation of all these unnecessary S-polynomials. Another positive observation is that although $T^{(0)}$ can contain very high power products we usually find a matrix T which does not contain as high power products as $T^{(0)}$.

The idea is to compute the coefficients from the equation $T \cdot F = G$. We get a system of linear equations over $\mathbb{Q}$ by replacing the coefficients of T and G by new variables. Solving these linear equations over $\mathbb{Q}$ gives us the coefficients of T and G.

We define primes as lucky as in Definition 22.

5.1 Computing the Coefficients

Theorem 26 (existence) *Let $F \in \mathbb{Q}[X]^m$, $G \in \mathbb{Q}[X]^n$ the normed reduced Gröbner basis for F, p a lucky prime for F and $G^{(0)} \in \mathbb{Z}_p[X]^n$ the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$. For every transformation matrix $T^{(0)}$ over $\mathbb{Z}_p[X]$ resulting from an application of the Buchberger algorithm to F such that $T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}$ there exists a transformation matrix T over $\mathbb{Q}[X]$ such that $T \cdot F = G$ and*

$$\mathrm{pp}(T_{i,j}) \subseteq \mathrm{pp}({T^{(0)}}_{i,j}) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m.$$

Proof: A row in $T^{(0)}$ is the trace of a computation of a polynomial of $g_k \bmod p$ with a GröbnerBasis algorithm. The same computation can be carried out over $\mathbb{Q}$ with

F instead of F mod p yielding a row of T. A difference in the power products could only occur if somewhere in the computation a term vanishes modulo p. But this case is excluded by the definition of a lucky prime. □

Theorem 27 (uniqueness) *Let $F \in \mathbb{Q}[X]^m$, p a lucky prime for F, $G \in \mathbb{Q}[X]^n$ the normed reduced Gröbner basis for F, and $G^{(0)} \in \mathbb{Z}_p[X]^n$ the normed reduced Gröbner basis for F in $\mathbb{Z}_p[X]$. Let $T^{(0)}$ be a transformation matrix over $\mathbb{Z}_p[X]$ such that*

$$T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}.$$

Let T' be a matrix over $\mathbb{Q}[X]$ with m rows and n columns, $G' \in \mathbb{Q}[X]^n$ such that

$$T' \cdot F = G'$$

and

$$\left.\begin{array}{rcl} \mathrm{lc}(G'_i) & = & 1 \\ \mathrm{lpp}(G'_i) & = & \mathrm{lpp}(G_i) \\ \mathrm{pp}(G'_i) & \subseteq & \mathrm{pp}(G_i) \end{array}\right\} \text{ for } 1 \le i \le n.$$

Then $G' = G$.

Proof: We show that $G'_i = G_i$ for $1 \le i \le n$. Let $g' := G'_i, g := G_i$ for some arbitrary but fixed $i \in \{1, \ldots, n\}$. We know $\mathrm{Ideal}(G') \subseteq \mathrm{Ideal}(F) = \mathrm{Ideal}(G)$ from $T' \cdot F = G'$. So $(g' - g) \in \mathrm{Ideal}(G)$. So $g' - g$ can be reduced to 0 w.r.t. G. Since the leading coefficients of g' and g are both 1, the leading power products are equal and all power products of g' also occur in g with a non-zero coefficient we have $\mathrm{pp}(g' - g) \subseteq \mathrm{pp}(g) \setminus \{\mathrm{lpp}(g)\}$. Since G is reduced $g' - g$ can not be reduced any further. Since G is a Gröbner basis we have $g' - g = 0$. □

So G is uniquely defined, whereas the transformation matrix T is usually not. I.e. there are some variables which are not uniquely determined by $t'_i \cdot F = g'_i$; they correspond to coefficients in T; variables corresponding to coefficients in G always have a unique solution. In Algorithm LiftTrans we take the pragmatic approach and set those variables to 0.

5.2 Algorithm LiftTrans

Algorithm 15 (LiftTrans)

$S \leftarrow \mathrm{LiftTrans}(F)$
  input: $F = \{f_1, \ldots, f_m\} \subset \mathbf{K}[X]$, p a lucky prime for F.
  output: $G = \{g_1, \ldots, g_n\} \subset \mathbf{K}[X]$, the reduced Gröbner basis for F.
BEGIN

$(G^{(0)}, T^{(0)}) :=$ GröbnerBasis(F mod p) over $\mathbb{Z}_p$ and corresponding
transformation matrix T such that $T^{(0)} \cdot F \equiv G^{(0)} \pmod{p}$

FOR all rows i of $T^{(0)}$ DO
// Make an ansatz with indeterminate coefficients:
$g'_i :=$ Replace each coefficient except the leading in $G_i^{(0)}$
with a new indeterminate u_j.
$t'_i :=$ Replace each coefficient in $T_i^{(0)}$ with a new indeterminate u_k.
$p := t'_i \cdot F - g'_i \in (\mathbb{Q}[u_1, \ldots, u_l])[X]$
$e := \{c = 0 \mid c = \text{coeff}(p, t) \in \mathbb{Q}[u_1, \ldots, u_l], c \neq 0 \text{ for some } t \in [X]\}$.
Solve the system of linear equations e in $\mathbb{Q}$ for the indeterminates $u_1, \ldots, u_l$.
$g_i :=$ Plug the results in g'_i.
END
RETURN $\{g_1, \ldots, g_n\}$
END LiftTrans.

One advantage of this algorithm is that we can check afterwards if the prime p was lucky. If any system of linear equations is not solvable the choice of p was not lucky. If the algorithm returns with a result G for F we are sure that Ideal$(G) \subseteq$ Ideal(F). And we can check if G is a Gröbner basis for F simply by reducing all polynomials of F and all necessary S-polynomials of G. If all these reductions yield 0 we have a Gröbner basis for F.

Another advantage is that we don't need a bound for the coefficients of the Gröbner basis G, because we compute the coefficients directly over q and not by a lifting process.

Example 14 We carry out the algorithm LiftTrans for computing a Gröbner basis in $\mathbb{Q}[x, y]$. We choose the total degree ordering with $x < y$. Let the input basis F be

$$F = \begin{pmatrix} y^2x^2 - 2yx \\ y^2x - 2y^2 \\ yx^2 + 3x^2 - 7 \end{pmatrix}.$$

As the prime we choose 13, which is indeed a lucky prime for F. The Gröbner basis for F modulo 13 is

$$G^{(0)} = \begin{pmatrix} x^2 + 2 \\ y \end{pmatrix},$$

the transformation matrix $T^{(0)}$ is

$$T^{(0)} = \begin{pmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \end{pmatrix}$$

with

$$t_{1,1} = -yx^2 + 3yx - 3x^2 - 2y - 2x - 6,$$

$$\begin{aligned}
t_{1,2} &= -2x^2 - 4x + 6,\\
t_{1,3} &= y^2x^2 - 3y^2x + 2y^2 - 2yx - 4,\\
t_{2,1} &= -2yx^2 + 2yx - 6x^2 + 4y - 3x + 1,\\
t_{2,2} &= -4x^2 - 3x + 1,\\
t_{2,3} &= 2y^2x^2 - 2y^2x - 4y^2 - 4yx - 2y.
\end{aligned}$$

We compute the coefficients over $\mathbb{Q}$ for $G_1^{(0)}$. First we replace each coefficient except the leading in $G_1^{(0)}$ with a new indeterminate:

$$G_1^{(0)} = x^2 + u_0,$$

replace each coefficient in $T_1^{(0)}$ with a new indeterminate:

$$T_1^{(0)} = \begin{pmatrix} u_5y^2x^2 + u_4y^2x + u_3y^2 + u_2yx + u_1 \\ u_8x^2 + u_7x + u_6 \\ u_{14}yx^2 + u_{13}yx + u_{12}x^2 + u_{11}y + u_{10}x + u_9 \end{pmatrix}^T,$$

and compute

$$\begin{aligned}
T_1^{(0)} \cdot F - G_1^{(0)} = {} & (u_{14} + u_5)y^3x^4 + (u_{13} + u_4)y^3x^3 + \\
& (u_{12} + 3u_5)y^2x^4 + (u_{11} + u_3)y^3x^2 + \\
& (-2u_{14} + u_{10} + u_8 + 3u_4 + u_2)y^2x^3 + \\
& (-2u_{13} + u_9 - 2u_8 + u_7 - 7u_5 + 3u_3)y^2x^2 + \\
& (-2u_{12} + 3u_2)yx^3 + (-2u_{11} - 2u_7 + u_6 - 7u_4)y^2x + \\
& (-2u_{10} + u_1)yx^2 + (-2u_6 - 7u_3)y^2 + (-2u_9 - 7u_2)yx + \\
& (3u_1 - 1)x^2 + (-7u_1 - u_0).
\end{aligned}$$

We get a system of linear equations from the coefficients of $T_1^{(0)} \cdot F - G_1^{(0)}$

$$\begin{aligned}
0 &= u_{14} + u_5,\\
0 &= u_{13} + u_4,\\
0 &= u_{12} + 3u_5,\\
0 &= u_{11} + u_3,\\
0 &= -2u_{14} + u_{10} + u_8 + 3u_4 + u_2,\\
0 &= -2u_{13} + u_9 - 2u_8 + u_7 - 7u_5 + 3u_3,\\
0 &= -2u_{12} + 3u_2,\\
0 &= -2u_{11} - 2u_7 + u_6 - 7u_4,\\
0 &= -2u_{10} + u_1,\\
0 &= -2u_6 - 7u_3,\\
0 &= -2u_9 - 7u_2,\\
0 &= 3u_1 - 1,\\
0 &= -7u_1 - u_0.
\end{aligned}$$

The solution for this system of linear equations is

$$
\begin{aligned}
u_{14} &= \frac{1}{2}u_2, \\
u_{13} &= \frac{1}{2}u_3 + \frac{2}{27}, \\
u_{12} &= \frac{3}{2}u_2, \\
u_{11} &= -u_3, \\
u_{10} &= \frac{1}{6}, \\
u_9 &= -\frac{7}{2}u_2, \\
u_8 &= \frac{3}{2}u_3 + \frac{1}{18}, \\
u_7 &= u_3 + \frac{7}{27}, \\
u_6 &= -\frac{7}{2}u_3, \\
u_5 &= -\frac{1}{2}u_2, \\
u_4 &= -\frac{1}{2}u_3 - \frac{2}{27}, \\
u_1 &= \frac{1}{3}, \\
u_0 &= -\frac{7}{3}.
\end{aligned}
$$

where u_2, u_3 can be freely chosen. We set them 0 and get the particular solution

$$
\begin{aligned}
u_{14} &= 0, \\
u_{13} &= \frac{2}{27}, \\
u_{12} &= 0, \\
u_{11} &= 0, \\
u_{10} &= \frac{1}{6}, \\
u_9 &= 0, \\
u_8 &= \frac{1}{18}, \\
u_7 &= \frac{7}{27}, \\
u_6 &= 0, \\
u_5 &= 0, \\
u_4 &= -\frac{2}{27}, \\
u_3 &= 0,
\end{aligned}
$$

$$\begin{aligned} u_2 &= 0, \\ u_1 &= \frac{1}{3}, \\ u_0 &= -\frac{7}{3}. \end{aligned}$$

Plugging in the results gives us

$$G_1^{(0)} = x^2 - \frac{7}{3}$$

and

$$T_1^{(0)} = (-\frac{2}{27}y^2x + \frac{1}{3}, \frac{1}{18}x^2 + \frac{7}{27}x, \frac{2}{27}yx + \frac{1}{6}x).$$

Repeating this steps for $G_2^{(0)}$ and $T_2^{(0)}$ we get

$$G = \begin{pmatrix} x^2 - \frac{7}{3} \\ y \end{pmatrix},$$

and

$$T = \begin{pmatrix} \frac{2}{27}yx + \frac{1}{6}x & \frac{1}{18}x^2 + \frac{7}{27}x & -\frac{2}{27}y^2x + \frac{1}{3} \\ -\frac{8}{63}yx - \frac{3}{14}x & -\frac{1}{6}x^2 - \frac{4}{9}x & \frac{8}{63}y^2x - \frac{1}{7}y \end{pmatrix}.$$

Compared to

$$T' = \begin{pmatrix} t'_{1,1} & t'_{1,2} & t'_{1,3} \\ t'_{2,1} & t'_{2,2} & t'_{2,3} \end{pmatrix}$$

with

$$\begin{aligned} t'_{1,1} &= -\frac{1}{27}yx^2 + \frac{5}{162}yx - \frac{1}{9}x^2 + \frac{7}{81}y + \frac{1}{6}x + \frac{7}{27}, \\ t'_{1,2} &= -\frac{2}{27}x^2 + \frac{14}{81}x + \frac{49}{162}, \\ t'_{1,3} &= \frac{1}{27}y^2x^2 - \frac{5}{162}y^2x - \frac{7}{81}y^2 - \frac{2}{27}yx + \frac{1}{3}, \\ t'_{2,1} &= \frac{4}{63}yx^2 - \frac{11}{378}yx + \frac{4}{21}x^2 - \frac{37}{189}y - \frac{3}{14}x - \frac{4}{9}, \\ t'_{2,2} &= \frac{8}{63}x^2 - \frac{47}{189}x - \frac{37}{54}, \\ t'_{2,3} &= -\frac{4}{63}y^2x^2 + \frac{11}{378}y^2x + \frac{37}{189}y^2 + \frac{8}{63}yx - \frac{1}{7}y \end{aligned}$$

which is the transformation matrix obtained from the Gröbner basis computation over $\mathbb{Q}[X]$ we see that the T does not contain as high power products as T'. □

Chapter 6

Gröbner Trace

In this chapter we present an algorithm that exploits the knowledge gained in the computation of a Gröbner basis modulo p by keeping a trace of the algorithm. I.e. we memorize the computation of S-polynomials and reductions of polynomials and the order of these operations. By repeating these computations over $\mathbb{Q}$ instead of $\mathbb{Z}_p$ we avoid then all computations of S-polynomials that can be reduced to 0. This idea was published in [Traverso, 1988].

6.1 Lucky Primes

We could use the Definition 22 of luckiness. But since we do not need the transformation matrix R or syzygy matrix S we can skip the conditions on the coefficients of these matrices.

Definition 24 ["luckiness"] Let F be a finite sequence of polynomials in $\mathbb{Q}[X]$, G the normed reduced Gröbner basis for F. Let p be a rational prime. p is *lucky* for F if and only if for all transformation matrices T resulting from an application of the Buchberger algorithm to F, p does not divide any coefficient in F, G and T. □

Since the unlucky primes of Definition 24 are a subset of the unlucky primes of Definition 22 we know that there are only finitely many unlucky primes.

It would even suffice to avoid the primes in the coefficients of the particular transformation matrix T we use in the computation. This would reduce the number of unlucky primes further. With the definition given above luckiness is independent of the implementation of the Gröbner basis algorithm. And in practise we have to deal with unlucky primes very rarely anyway.

6.2 Tracing Polynomials

The algorithm is a natural by-product of the algorithm for the transformation matrix T. The bookkeeping done there is a trace. We just use this trace to compute the Gröbner basis again over $\mathbb{Q}$ instead of computing T.

We need to keep track of every decision made during the Gröbner basis computation where the algorithm has a freedom of choice. In the NormalForm algorithm where we can choose which term to eliminate in the polynomial to reduce and which polynomial we use for the reduction. In the GröbnerBasis algorithm we can choose which S-polynomial to consider next. So if we keep track of these choices we are able to reconstruct the computation of the Gröbner basis.

We first discuss the implementation of the algorithm for computing the transformation matrix T to see the similarities between the computation of T and the Gröbner trace algorithm.

In Definition 4 we said that $g \rightarrow_F h$, iff

$$\text{there are } f \in F, t \in [X], c \in \mathbf{K}, \text{ such that } g \rightarrow_{f,c,t} \text{ and } h = g - c \cdot t \cdot f.$$

for $F \subseteq \mathbf{K}[X]$, $g, h \in \mathbf{K}[X]$. In a practical implementation for a quotient field K over an integral domain I we would use: $g \rightarrow_F h$, iff

$$\text{there are } f \in F, t \in [X], c_1, c_2 \in I \text{ such that } g \rightarrow_{f,\frac{c_1}{c_2},t} \text{ and } c_1 \cdot h = g - c_2 \cdot t \cdot f.$$

The two definitions are equal up to a coefficient in K. This coefficient has no influence on the ideals we are dealing with. But the advantage of the second definition is that we can stay in the integral domain. And this helps to reduce the number of gcd-computations. In the definition of S-polynomials we already avoided a division of coefficients. For quotient fields we compute the standardized reduced Gröbner basis instead of the normed reduced Gröbner basis.

Definition 25 Let G be a Gröbner basis in $\mathbf{K}[X]$, $\mathbf{K}$ a quotient field of some integral domain. G is *standardized* iff $\text{content}(g) = 1$ for all $g \in G$. □

So we have two slightly different ways of computing Gröbner bases depending on K being a quotient field or not. The best way to handle such slight variations in algorithms is by using object oriented programming. Figure 6.1 is a diagram of the class hierarchy of the polynomials used in Gröbner basis computation.

As one can see we have defined an abstract basis class AbsGPoly which has abstract methods for all operations that we want to vary. Derived from GPoly are two concrete classes IntGPoly and GPoly which implement the exact behavior over a quotient field (e.g. $\mathbb{Q}$) or any arbitrary field (e.g. $\mathbb{Z}_p$ for a prime p). Norm is

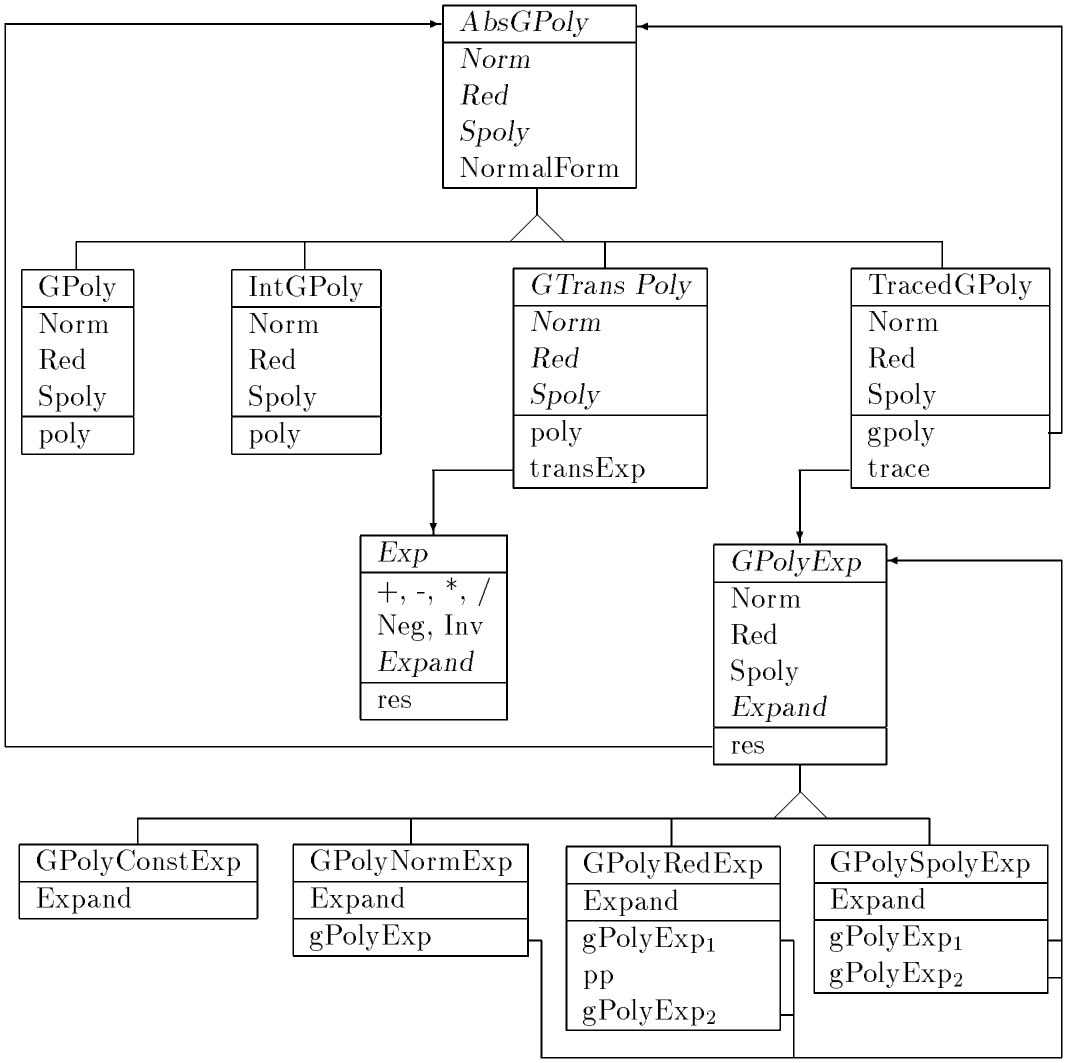

Figure 6.1: Class hierarchy of multivariate polynomials for Gröbner basis computation and expressions.

used to normalize a polynomial somehow. That is the polynomial is multiplied by some $c \in \mathbf{K}$ such that either the content or the leading coefficient of the polynomial becomes 1. Red is used to reduce a polynomial p_1 by a polynomial p_2. Again either we multiply both p_1 and p_2 by suitable $c_1, c_2 \in \mathbf{K}$ such that all coefficients of the result polynomial are in the integral domain or just multiply p_2 by coefficient $c \in \mathbf{K}$ such that the desired term vanishes. Spol is used to compute the S-polynomial of two polynomials p_1 and p_2 with the same variations as for Red. The possibility to vary the implementation of the NormalForm algorithm is not taken into consideration in this class hierarchy.

GTransPoly memorizes the transformation used in its computation. It does the actual transformation on the polynomial poly and uses the class Exp and its derivations to store the arithmetic operation and operands (i.e. coefficients, power products, polynomials). The class hierarchy of Exp is not shown in detail here. When we are done with the Gröbner basis computation we call Expand of the polynomials in the result. Expand calls Expand recursively on the stored operands and uses these results to carry out the stored operation. And of course these results are cached so that we have to compute them just once. As starting points we use vectors of unity with 1 in the i-th position for the polynomial f_i of the basis F. Calling Expand on these ConstExps just returns the corresponding vector of unity. We get the rows of the transformation matrix T by calling Expand for all members transExp of GTransPoly objects in the result basis G and collecting the results. GTransPoly is abstract class with two concrete derived classes for arbitrary coefficient and coefficients of an quotient field similar to GPoly and InGPoly. They are not shown in the diagram.

TracedGPoly does not keep track of the coefficients and power products used in the reductions and S-polynomial computations. It just memorizes the polynomials used for the computation. The coefficients are of no use if we compute the Gröbner basis over $\mathbb{Z}_p$ first keeping the polynomials normed and then repeat the computation over $\mathbb{Q}$ where we keep the polynomials standardized. The power products can be easily extracted from the polynomials. We just have to remember which power product we were trying to eliminate. This can usually done by an index.

GPolyExp is an abstract class which is used as a memory for TracedGPoly. It implements the methods Norm, Red and Spoly. When one of these methods is called the method returns an object of the corresponding derived class and initializes the member variables of this object with the parameters of the called method. Expand returns either a GPoly of IntGPoly object depending on what we started with. The first call to expand stores the result in the variable res. Subsequent calls just return the cached result. GPolyExp has the following derived classes:

GPolyConstExp keeps a polynomial of F wrapped in a GPoly or IntGPoly object and Expand just returns it.

GPolyNormExp stores the polynomial (or more exactly the way to compute this polynomial) to normalize. Expand just calls Norm for the expanded gPolyExp member and returns the result.

GPolyRedExp stores the polynomial to reduce, which power product to eliminate and the polynomial used in the reduction. Expand just expands the two stored polynomials and reduces the first by eliminating the power product with a suitable multiple of the second. If this reduction is not possible because either the power product has a zero coefficient in the first polynomial or the leading power product of the second does not divide the power product we report an unlucky prime.

GPolySpolyExp stores the two polynomials from which the S-polynomial is computed. Expand computes the S-polynomial of the two expanded polynomials.

So we either keep track of the coefficients and power products used in the Gröbner basis computation to construct the transformation matrix T. Or we memorize the sequence of reductions and S-polynomials and repeat this sequence over the desired quotient field. With these classes the algorithm for computing a Gröbner basis over $\mathbb{Q}$ by using the trace obtained via computation of the Gröbner basis over $\mathbb{Z}_p$ is quite simple.

Algorithm 16 (Trace)

$G \leftarrow \mathrm{Trace}(F, p)$
  input: F, a finite subset of $\mathbb{Q}[X]$,
      p, a lucky prime for F.
  output: G, a finite subset of $\mathbb{Q}[X]$ such that
      $\mathrm{Ideal}(F) = \mathrm{Ideal}(G)$ and G is a Gröbner basis.
BEGIN
  $F' := \{ \mathrm{TracedGPoly}(f \bmod p, \mathrm{GPolyConstExp}(f)) \mid f \in F\}$
  $G' := \mathrm{GröbnerBasis}(F') \bmod p$
  $G := \{g.\mathrm{trace.Expand}() \mid g \in G'\}$
  RETURN G
END Trace.

In the method Expand of GPolyRedExp we check if the reductions are possible, so if we get a result G from Trace for F, we know that $\mathrm{Ideal}(G) \subseteq \mathrm{Ideal}(F)$. We can then check if G is the Gröbner basis for F by reducing all polynomials in F and all S-polynomial of G to zero. If any of these reductions yields a nonzero result p was unlucky. This gives the possibility to test the result and prime p.

6.3 Tracing Coefficients

In the previous section we did the essential steps of the Gröbner basis computation again with polynomials over a different field. But we already knew the power products of the result. All we needed were the coefficients in the new field. So we present another variant where we keep track of all operations on the coefficients. We can no longer keep the coefficients in the integral domain, but we avoid all polynomial computations and what seems more important all computations of zero coefficients.

We show the relevant parts of the class hierarchy in Figure 6.2.

Coefficient is an abstract class defining the operations possible on coefficients. Coefficients can be added, subtracted, multiplied and divided. A coefficient can be negated (i.e. return its additive inverse) and inverted (i.e. return its multiplicative inverse). They can return a representation for zero and one in the corresponding domain. And they can be tested whether they are zero or one.

TracedCoeff is used to keep track of the operations on coefficients. It executes the operation on its member coeff and stores the operation in its member trace, which is an object of class Exp.

Exp is an abstract basis class used as memory for tracked operations on coefficients. The arithmetic operations return an instance of the corresponding class. The abstract method Expand is called to repeat the operations used to compute the coefficient. Expand uses the member res as a cache so that we have to compute the desired coefficient just once.

ConstExp is used as a starting point for our tracked coefficients. It stores the coefficient in res and Expand just returns it.

UnaryExp is an abstract basis class for the unary operations negate and inverse. It defines the member variable exp which stores the expression we want to negate or invert.

NegExp represents the negation of a coefficient. Expand calls the Expand on the inherited member exp and returns the additive inverse of it.

InvExp represents the inverse of a coefficient. Expand calls the Expand on the inherited member exp and returns the multiplicative inverse of it.

BinaryExp is an abstract basis class for the binary operations. It defines the member variables exp_1 and exp_2 which store the expressions we want to add, subtract, multiply or divide.

AddExp represents the addition of two coefficients. Expand calls Expand on the inherited members exp_1 and exp_2 and returns the sum of them.

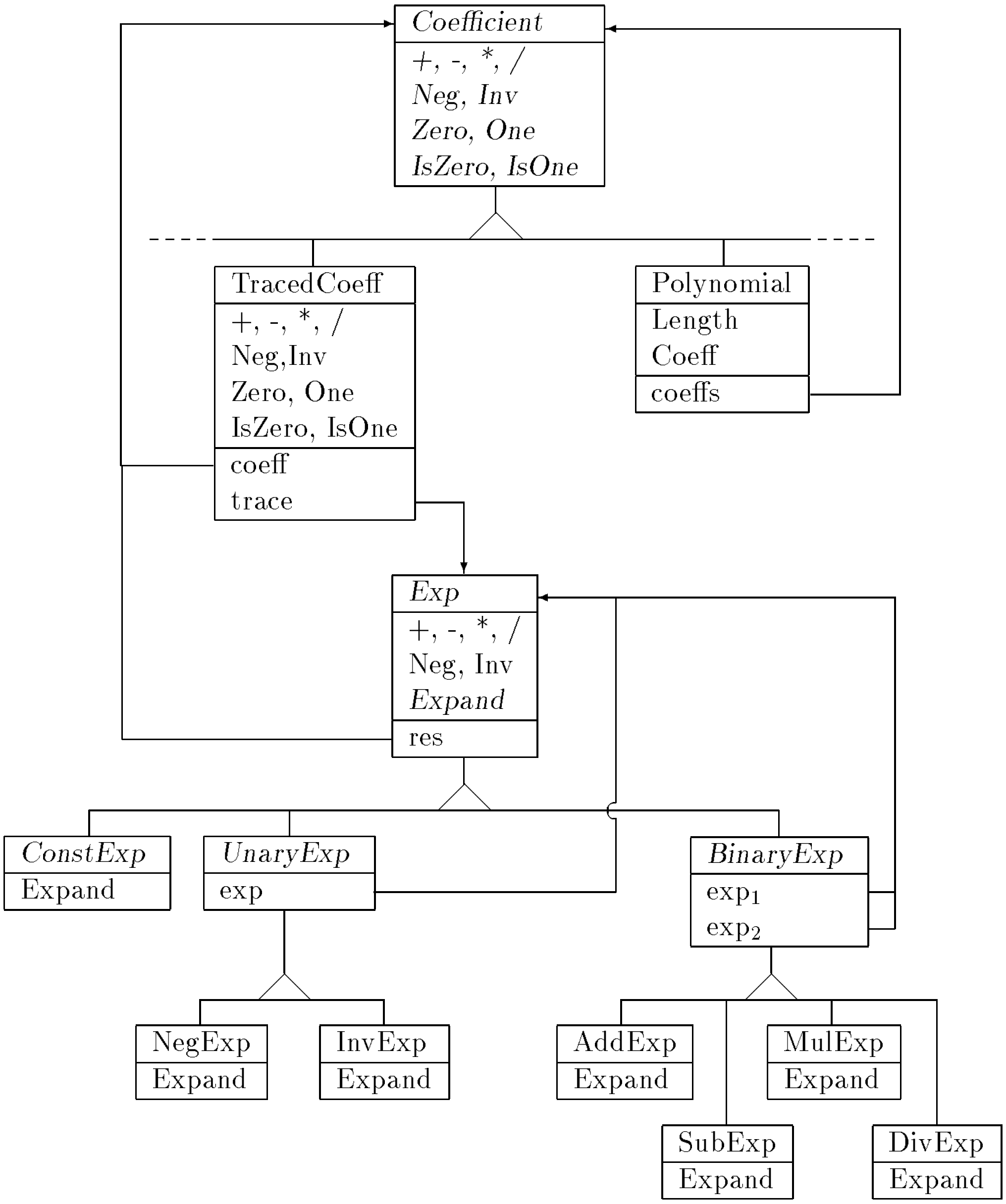

Figure 6.2: Class hierarchy of coefficients and expressions.

SubExp represents the subtraction of two coefficients. Expand calls Expand on the inherited members $\exp_1$ and $\exp_2$ and returns the difference of them.

MulExp represents the multiplication of two coefficients. Expand calls Expand on the inherited members $\exp_1$ and $\exp_2$ and returns the product of them.

DivExp represents the division of two coefficients. Expand calls Expand on the inherited members $\exp_1$ and $\exp_2$ and returns the quotient of them.

And again the algorithm for computing a Gröbner basis over $\mathbb{Q}$ by keeping a trace of the operations on the coefficients of $\mathbb{Z}_p$ and repeating this operations on coefficients of $\mathbb{Q}$, is simple.

Algorithm 17 (TraceCoeff)

$G \leftarrow$ TraceCoeff(F, p)
input: F, a finite subset of $\mathbb{Q}[X]$,
p, a lucky prime for F.
output: G, a finite subset of $\mathbb{Q}[X]$ such that
Ideal(F) = Ideal(G) and G is a Gröbner basis.
BEGIN
F' := Replace every coefficient c in F by TracedCoeff($c \bmod p$, ConstExp(c))
G' := GröbnerBasis$(F') \bmod p$
G := Expand every coefficient in G'
RETURN G
END TraceCoeff.

We were able to test the result of the algorithm Trace. This is no longer possible since we do not compute the terms which should vanish. We only know that if a coefficient is expanded to zero the prime p must be unlucky. We can check if Ideal$(F) \subseteq$ Ideal(G) and if G is a Gröbner basis by testing whether all all polynomials in F and all S-polynomial of G can be reduced to zero w.r.t. G. If any of these reductions yields a nonzero result p was unlucky. But we can not test if Ideal$(G) \subseteq$ Ideal(F).

Chapter 7

Implementation and Experimental results

7.1 Implementation

The first implementation of LiftCoeff was done in Maple. But soon it became obvious that this was not the best way to do it. For example the Gröbner basis implementation of Maple is tailored to the computation of Gröbner bases over $\mathbb{Q}$ and in order to implement the Gröbner basis algorithm over $\mathbb{Z}_p$ you had to duplicate nearly half of the subroutines, because the code for keeping the polynomials standardized was littered all over the code. Just to keep the polynomials normed one had to change many subroutines, which needed no other modification. Also it was nearly impossible to obtain information about coefficient growth during the Gröbner basis computation. So we decided to implement it from scratch.

We chose Oberon-2 as a programming language. It was favored over C++ because Oberon-2 comes with a garbage collector and is very fast to program in. Oberon is a descendant of the Pascal language family from N. Wirth. It was designed to simplify Modula-2 and adds constructs for object-oriented programming. It was shortly followed by Oberon-2 which added some minor enhancements and better support for object-oriented programming constructs. Oberon is an operating system, too. And it is a great programming environment. For more information on Oberon we refer to [Reiser, 1991], [Reiser and Wirth, 1992] and [Wirth and Gutknecht, 1992]. A very good introduction to object oriented programming and Oberon-2 can be found in [Mössenböck, 1993]. Oberon is available on many hardware platforms (Intel PCs, Apple Macintosh, HP, Sun SPARC, IBM RS6000) and programs written in Oberon are truly cross-platform.

We give an overview over the class hierarchies. In Figure 7.1 the class hierarchy for the basic types for computation (i.e. Integers, Polynomials, etc.) is shown.

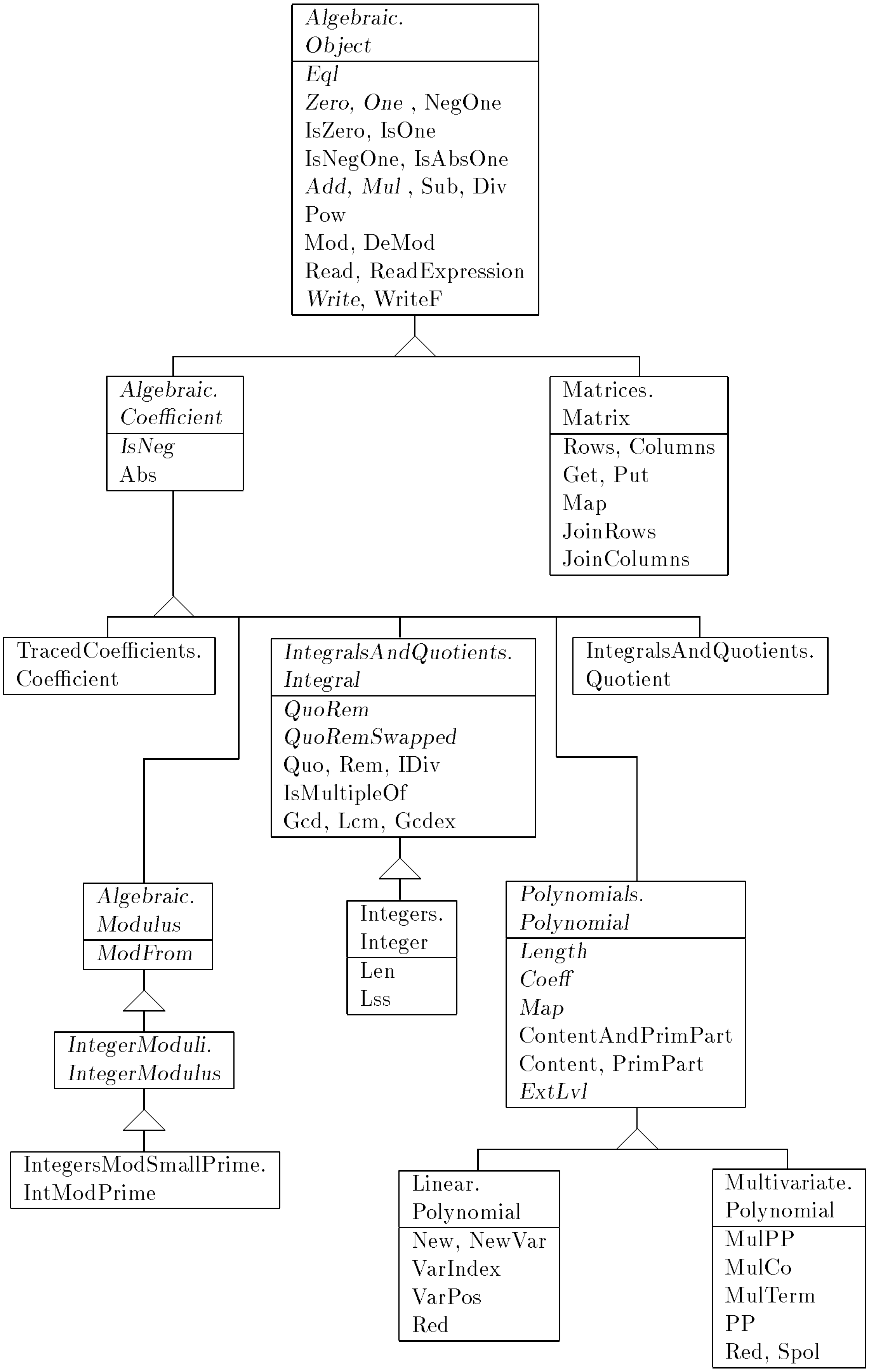

Figure 7.1: Class hierarchy of algebraic objects.

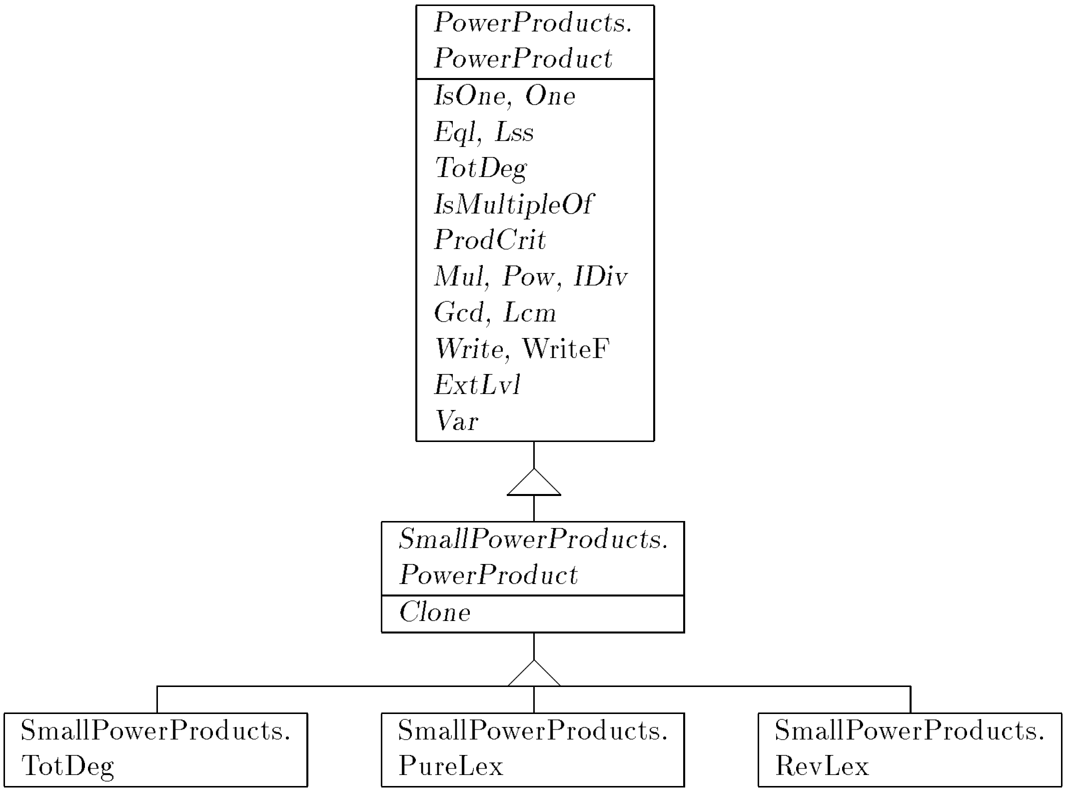

Figure 7.2: Class hierarchy of power products.

Multivariate polynomials use an abstract class PowerProduct to represent the power products. In Figure 7.2 the class hierarchy for power products is shown.

For the implementation of the Gaussian elimination algorithm we choose the representation as linear polynomials, because our systems of linear equations are sparse. We again implemented the algorithm on an abstract class and the derived classes define the behavior for quotient fields or arbitrary fields. The class hierarchy is shown in Figure 7.3.

We give a short description of all modules and classes.

Algebraic Module Algebraic defines the base classes for all computations.

Object is an abstract class defining the possible arithmetic operations. For the dyadic operations $+, -, *, /, \mathrm{mod}$ we need a multiple dispatch mechanism since it depends not only on the receiver which method is called. For commutative operations this is easy. If the receiver does not know how to deal with the argument it dispatches the method again with the argument

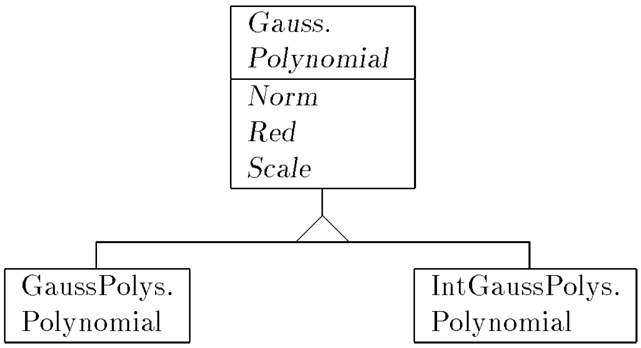

Figure 7.3: Class hierarchy of linear polynomials for Gaussian elimination.

as receiver. If e.g. an integer receives the message to add a polynomial to itself it just tells the polynomial to add the integer to itself. And polynomials have to know how to add coefficients to themself. This works also fine for noncommutative operations that can be expressed in terms of commutative operations (e.g. $a - b \rightarrow (-b) + a$). For mod there is the artificial operation ModFrom which is just the modulo operation with receiver and argument swapped.

Coefficient is an abstract class which is used to form polynomials and matrices over arbitrary domains.

Modulus is an abstract class and root for all domains where we compute modulo something. So we can e.g. compute with matrices of polynomials modulo some prime p and the operations on the matrices and polynomials do not need to be aware of that. Maple would first compute the result in $\mathbb{Q}$ and then take the remainder. This can lead to quite big intermediary results and we to avoid that. If we write a.Mod(p) in usual form than the receiver is wrong so it has the default implementation p.ModFrom(a). Of course matrices and polynomials overwrite this default behavior and call Mod for all their coefficients.

Matrices Module Matrices implements matrices over arbitrary coefficients. Map applies a function to all coefficients of the matrix.

TracedCoefficients Module TracedCoefficients is used to keep track of the operations on coefficients. It is explained in Chapter 6.

IntegralsAndCoefficients Module IntegralsAndCoefficients define classes for integral domains and quotient fields over integral domains.

Integral is an abstract class for members of integral domains. It defines the quotient and remainder operations. QuoRemSwapped is used for multiple dispatch if the receiver does not know how to compute quotient and remainder for a certain argument. Gcd and Lcm compute the greatest common divisor and least common multiple respectively. Gcdex computes the greatest common divisor and its cofactors u, v such that $\gcd(a, b) = u.a + v.b$. IDiv is used for divisions where we know the remainder to be 0. It returns an object of type Integral whereas Div would return an Object and it enables a faster implementation by overwriting.

Quotient concrete class for quotient fields over integral domains. Its denominator is always positive and never zero or one. Numerator and denominator are normalized i.e. their greatest common divisor is one.

Integers Module Integers implements arithmetic for arbitrary long integers. They are represented by an array of 16 bit words, their length and a sign flag. Length gives you the number of bits of an integer.

IntegerModuli Module IntegerModuli defines the abstract class IntegerModulus which is used to represent members of Galois fields GF(p) with $p \in \mathbb{Z}$.

IntegersModSmallPrime Module IntegersModSmallPrime defines the concrete class IntModPrime derived from IntegerModulus. It is used to represent members of Galois fields for small primes (fitting in a computer word). The biggest usable prime in our implementation is 46337.

Polynomials Module Polynomials defines an abstract class for polynomials. For polynomials with coefficients of an integral domain or quotient field you can compute the content and the primitive part. Map applies a function to all coefficients of a polynomial. Since coefficients of a polynomial can be polynomials too, we order polynomials by their extension level. Let p_1 be a polynomial with extension level l_1 and p_2 a polynomial with extension level l_2. If $l_1 < l_2$ then p_1 is in the coefficient domain of p_2. This is used in multiple dispatch situations to decide whether we are dealing with two polynomial or a polynomial and a coefficient which just happens to be a polynomial.

Linear Module Linear defines an concrete class for linear polynomials. A polynomial $\sum_{i=1}^{n} c_i \cdot x_i$ is represented with an array of coefficients c_i and variable indices i and the length n. The coefficients are ordered with decreasing variable indices. New and NewVar are two helper methods for easy creation of linear polynomials representing indeterminates. VarIndex(j) returns the j-th variable index. VarIndex(0) returns the highest variable index. VarPos(j) returns the position of the variable with index j. $\mathrm{Red}(p_1, c_1, n, p_2, c_2) = c_1.p_1 - c_2.p_2$, where $p_1.\mathrm{VarIndex}(n) = p_2.\mathrm{VarIndex}(0)$. It is used by the Gaussian elimination algorithm to eliminate a variable in a polynomial p_1 by subtraction a suitable multiple of another polynomial p_2. Red is optimized regarding space consumption compared to computing this expression with multiplications and subtractions.

Multivariate Module Multivariate defines an concrete class for multivariate polynomials. A polynomial is represented with an array of coefficients c_i and power products and the length n. The coefficients are ordered with decreasing order of the power products. A variant with a linked list instead of an array was also tested. It exhibited no advantage over the implementation with arrays. So the simpler implementation with arrays was chosen. MulPP, MulCo, MulTerm are just short cuts for the multiplication with a power product, coefficient and both. PP(j) returns the j-th power product. PP(0) returns the leading power product as defined by the admissible ordering of the power products. Red is used by the Gröbner basis algorithm to eliminate a variable in a polynomial p_1 by subtraction a suitable multiple of another polynomial p_2. Spol is used by the Gröbner basis algorithm to compute S-polynomials. Red and Spol are optimized regarding space consumption compared to computing this expressions with multiplications and subtractions.

PowerProducts Module PowerProducts defines an abstract class for power products, which is used for the representation of multivariate polynomials.

SmallPowerProducts Module SmallPowerProducts defines a concrete class PowerProduct for power products with degrees less than 127 and a maximal number of variables of 7. Derived from PowerProduct are the concrete classes TotDeg, PureLex and RevLex which have the total degree ordering, purely lexicographic ordering and the reverse lexicographic ordering as their admissible ordering.

Gauss Module Gauss implements the Gaussian elimination algorithm. We choose the representation as linear polynomials, because our systems of linear equations are sparse. We again implemented the algorithm on an abstract class and the derived classes define the behavior for quotient fields or arbitrary fields.

GaussPolys Module GaussPolys implements the class Polynomial which extends Gauss.Polynomial for the case where the leading coefficient of normalized polynomials is one.

IntGaussPolys Module IntGaussPolys implements the class Polynomial which extends Gauss.Polynomial for coefficients of an integral domain. Normalized polynomials have a content of one.

Grobner Module Grobner implements the Gröbner basis algorithm. The selection strategy for critical pairs is to choose the S-polynomial with the minimal least common multiple of leading power products according to the admissible ordering of the power products. After adding a new polynomial to the basis we reduce all polynomials in the basis with regard to the others as proposed by Buchberger in [Buchberger, 1985]. We normalize a polynomial after it can not be reduced further and not after every reduction. This minimizes the number of gcd computations and avoids it completely in the case of a reduction to 0. Our tests showed that this improves the run time. As was explained in Chapter 6 the actual reduction and normalization is handled by a derived class.

GrobnerPolys Module GrobnerPolys implements the class Polynomial which extends Grobner.Polynomial for the case where the leading coefficient of normalized polynomials is one.

IntGrobnerPolys Module IntGrobnerPolys implements the class Polynomial which extends Grobner.Polynomial for coefficients of an integral domain. Normalized polynomials have a content of one.

Expressions Module Expressions implements the class Exp. For details see Chapter 6.

GrobnerTrace Module GrobnerTrace implements the algorithms for computing transformation and syzygy matrices. For details of the implementation see Chapter 6. The algorithm for the computation of the syzygy matrix allows you to choose between all rows or only those that correspond to necessary S-polynomials.

GrobnerTransPolys Module GrobnerTransPolys implements the class Polynomial which extends GrobnerTrans.Polynomial for the case where the leading coefficient of normalized polynomials is one.

IntGrobnerTransPolys Module IntGrobnerTransPolys implements the class Polynomial which extends GrobnerTrans.Polynomial for coefficients of an integral domain. Normalized polynomials have a content of one.

TracedCoefficients Module TracedCoefficients implements the class TracedCoeff. For details see Chapter 6.

GrobnerPolyExpressions Module GrobnerPolyExpressions implements the class GPolyExp. For details see Chapter 6.

TracedGrobnerPolys Module TracedGrobnerPolys implements the class GPolyTrace. For details see Chapter 6.

ModularGB Module ModularGB implements the algorithms FareyToRational, LiftCoeff, LiftTrans, Trace and TraceCoeffas described throughout this paper.

7.2 Experimental results

Times were measured on a Power Macintosh with 160MHz and 48MB RAM. Our implementation is roughly three times faster than Maple V Release 4 Student Edition. We have found examples where our implementation is 10 and even 100 times faster and one example where Maple is 20 times faster. But these big differences result from the different strategies of Gröbner basis computation. Maple does not reduce all polynomials after a new polynomial is added to the basis.

We tested over 100 examples and we list some typical or interesting one and the result for them.

1. The Caprasse example from [Faugere et al., 1986] with 4 variables and purely lexicographic ordering with $z > y > x > t$.

$$\begin{aligned}\{\ & 2zyt - 2z + xt^2 - x,\\ & zy^2 - z + 2yxt - 2x,\\ & 4zy^2x - zx^3 + 4zx + 2y^3t - 10y^2 + 4yx^2t - 10yt + 4x^2 + 2,\\ & -z^3x + 4z^2yt + 4z^2 + 4zxt^2 + 4zx + 2yt^3 - 10yt - 10t^2 + 2\}\end{aligned}$$

2. The Katsura example from [Boege et al., 1986] with 4 variables and purely lexicographic ordering with $u_0 > u_1 > u_2 > u_3$.

$$\begin{aligned}\{\ & u_0 + 2u_1 + 2u_2 + 2u_3 - 1,\\ & 2u_0u_2 + u_1^2 + 2u_1u_3 - u_2,\\ & 2u_0u_1 + 2u_1u_2 - u_1 + 2u_2u_3,\\ & u_0^2 - u_0 + 2u_1^2 + 2u_2^2 + 2u_3^2\}\end{aligned}$$

3. The Katsura example from [Boege et al., 1986] with 5 variables and purely lexicographic ordering with $u_0 > u_1 > u_2 > u_3 > u_4$.

$$\begin{aligned}\{\ & u_0 + 2u_1 + 2u_2 + 2u_3 + 2u_4 - 1,\\ & 2u_0u_3 + 2u_1u_2 + 2u_1u_4 - u_3,\\ & 2u_0u_1 + 2u_1u_2 - u_1 + 2u_2u_3 + 2u_3u_4,\\ & u_0^2 - u_0 + 2u_1^2 + 2u_2^2 + 2u_3^2 + 2u_4^2\}\end{aligned}$$

4. The symmetry example from [Gräbe, 1993] with 3 variables and purely lexicographic ordering with $x > y > z$.

$$\begin{aligned}\{\ & x + y^3 + z^2 - 3,\\ & x^2 + y + z^3 - 3,\\ & x^3 + y^2 + z - 3\}\end{aligned}$$

5. The Trinks example from [Boege et al., 1986] with 6 variables and purely lexicographic ordering with $w > p > z > t > s > b$.

$$\begin{aligned}\{\ & 45p + 35s - 165b - 36,\\ & 35p + 40z + 25t - 27s,\\ & 15w + 25ps + 30z - 18t - 165b^2,\\ & -9w + 15pt + 20zs,\\ & 99w - 11sb + 3b^2,\\ & wp + 2zt - 11b^2\}\end{aligned}$$

6. The Trinks example from [Boege et al., 1986] with 6 variables and purely lexi-

cographic ordering with $w > p > z > t > s > b$.

$$\begin{aligned}\{\quad & 10000b^2 + 6600b + 2673,\\ & 45p + 35s - 165b - 36,\\ & 35p + 40z + 25t - 27s,\\ & 15w + 25ps + 30z - 18t - 165b^2,\\ & -9w + 15pt + 20zs,\\ & 99w - 11sb + 3b^2,\\ & wp + 2zt - 11b^2\}\end{aligned}$$

7. The example 1 from [Traverso, 1988] with 3 variables and purely lexicographic ordering with $x > y > z$

$$\begin{aligned}\{\quad & y^2 - 11z + 23,\\ & xy - y^2 + 17z^2,\\ & x^2 - 5yz + 7\}\end{aligned}$$

8. The example 2 from [Traverso, 1988] with 2 variables and purely lexicographic ordering with $x > y$

$$\begin{aligned}\{\quad & x^2y^3 - 7xy^4 + 21x - 11,\\ & 41x^3 + xy^5 - 101\}\end{aligned}$$

9. A randomly generated example with 2 polynomials in 2 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\quad & -4075x_1^2x_2 - 9162x_1x_2^2 - 3208x_2^3 + 9193x_1x_2 + 3141x_1 + 8565x_2,\\ & 4529x_1^3 + 1156x_1^2x_2 - 5443x_1x_2^2 + 4452x_2^3 + 7180x_1^2 + 3123x_2^2\}\end{aligned}$$

10. A randomly generated example with 2 polynomials in 2 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\quad & -44131x_1^3 + 79610x_1x_2^2 + 67044x_2^3 + 55556x_1^2 - 83322x_1 + 95185,\\ & -48237x_1^3 - 57341x_1^2x_2 - 96486x_1^2 + 25367x_1x_2 - 19542x_2^2 + 88168\}\end{aligned}$$

11. A randomly generated example with 3 polynomials in 2 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\quad & -91x_1^3x_2 + 78x_1^2x_2^2 - 87x_2^4 - 14x_2^3 - 49x_1x_2 - 1,\\ & 51x_1^4 - 92x_1^2x_2^2 - 63x_1^3 + 76x_2^3 - 65x_1 - 70x_2\}\end{aligned}$$

12. A randomly generated example with 3 polynomials in 2 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\quad & -16081x_1^2x_2 - 85348x_1x_2^2 + 38054x_2^3 - 38438x_1x_2 + 98597x_2^2 - 35909x_1,\\ & 53857x_1^2x_2 + 347x_1x_2^2 - 71550x_2^3 - 22739x_1x_2 - 63422x_2 + 83624,\\ & 18075x_1^3 - 89066x_1^2x_2 + 43995x_1x_2^2 - 20322x_2^3 - 96499x_1^2 + 81542x_2^2\}\end{aligned}$$

13. A randomly generated example with 3 polynomials in 2 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\ & 9598x_1^4 - 6078x_2^4 - 8336x_1x_2^2 + 6768x_2^3 - 7622x_1 + 1862x_2,\\ & -1990x_1^4 - 9334x_1x_2^3 - 3035x_2^4 + 391x_1^2x_2 - 7012x_1x_2 + 2507x_2,\\ & 1415x_1^4 + 8873x_1^2x_2^2 + 2036x_1^2x_2 - 839x_2^3 - 2662x_1^2 - 2914x_1x_2\}\end{aligned}$$

14. A randomly generated example with 3 polynomials in 2 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\ & -85644x_1x_2^3 + 17298x_2^4 - 65636x_1^3 + 40049x_2^3 + 45375x_2^2 - 31499x_1,\\ & -37123x_1^4 + 31969x_1^3x_2 - 45309x_1^2x_2^2 + 16903x_1x_2^3 - 95274x_1^2x_2 + 3332x_1x_2^2,\\ & -61417x_1^4 - 97008x_1^2x_2^2 - 69582x_1x_2^3 + 87758x_1^2x_2 - 10699x_1x_2 - 66547\}\end{aligned}$$

15. A randomly generated example with 4 polynomials in 2 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\ & 57445x_1^2x_2 + 23485x_1x_2^2 - 4839x_2^3 - 16457x_1^2 + 20642x_1x_2 - 3518x_2,\\ & -98427x_1^3 + 60091x_1^2x_2 + 24495x_2^3 + 31278x_1,\\ & -36816x_1^3 - 83498x_1x_2^2 - 68467x_2^3 - 86223x_2^2 - 65460x_1 - 6448x_2,\\ & 29878x_1^3 + 11095x_1^2x_2 - 6574x_2^3 + 13041x_1^2 + 4389x_2 + 29486\}\end{aligned}$$

16. A randomly generated example with 4 polynomials in 2 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2$.

$$\begin{aligned}\{\ & -78162x_1^2x_2^2 + 25907x_2^4 + 61674x_1^3 + 81391x_1^2x_2 - 76523x_1x_2^2 + 20770x_2^2,\\ & -61538x_1^4 + 71895x_2^4 + 59726x_2^2 + 30230x_1 + 65182x_2 + 50398,\\ & 64899x_1^4 - 26439x_1^2x_2 + 21554x_2^3 + 75363x_1^2 + 95720x_1x_2 + 8448x_2,\\ & 11536x_1^4 - 99528x_1^2x_2 - 18501x_1^2 + 62822x_1x_2 + 19474x_2^2 - 15544x_2\}\end{aligned}$$

17. A randomly generated example with 2 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.

$$\begin{aligned}\{\ & 4781x_1^2x_3 + 3662x_1x_2x_3 - 778x_1x_3^2 - 7831x_2^2 + 3836x_3^2 - 567x_2,\\ & 1979x_1^3x_2 - 243x_1x_2x_3^2 + 1936x_2x_3^3 - 2576x_2^2x_3 + 4955x_2x_3 - 7576x_2\}\end{aligned}$$

18. A randomly generated example with 2 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.

$$\begin{aligned}\{\ & -640x_1^4 - 64591x_1x_2^2 - 57308x_2^3 - 27225x_2x_3^2 + 62317x_2x_3 - 47439x_3^2,\\ & -43302x_1^4 + 37798x_1^3x_3 + 26033x_1^2x_3^2 + 85798x_1x_2 + 85166x_1x_3 - 42449x_1\}\end{aligned}$$

19. A randomly generated example with 3 polynomials in 3 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ 55x_1x_2^2 + 48x_1x_2x_3 + 19x_2x_3^2 + 73x_3^3 - 89x_2^2 + 2x_1x_3,\ -4x_1^2x_2 + 46x_2^3 - 66x_2^2x_3 - 15x_2^2 + 9x_1 + 64,\ 34x_1^2x_2 - 81x_1x_2^2 + 78x_2^3 - 85x_1^2x_3 + 3x_3^2 + 48x_2\}$$

20. A randomly generated example with 3 polynomials in 3 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ 8907x_1^2x_3 - 9200x_2x_3^2 - 5141x_3^3 + 4472x_1^2 + 3596x_2x_3 + 3298x_3,\ 1654x_1^3 + 9408x_1x_2^2 + 865x_3^3 + 1953x_1x_3 + 9082x_2 + 968,\ 8601x_1^3 - 5526x_1^2x_2 + 2072x_1x_2^2 + 6401x_1x_3^2 - 3976x_1x_2 + 5197x_1\}$$

21. A randomly generated example with 3 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ 61x_2^2x_3^2 + 52x_1x_2^2 - 76x_2x_3^2 - 72x_3^3 - 92x_2^2 - 3x_3,\ 77x_1^3x_3 - 91x_1x_2^2x_3 - 90x_1x_2x_3 - 68x_1^2 + 32x_1 + 97x_3,\ 57x_1x_2^3 + 97x_1x_2^2x_3 + 100x_2^3x_3 - 98x_1x_3 + 88x_3^2 + 85x_2\}$$

22. A randomly generated example with 3 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ -590x_1^2x_2x_3 - 93x_1x_3^3 - 304x_1^3 + 522x_2^3 + 927x_1^2x_3 + 61,\ 555x_2^4 + 119x_1^2x_2x_3 - 610x_3^4 - 114x_1^3 - 729x_1x_2^2 + 780x_1x_2x_3,\ 68x_1^3x_2 - 392x_2^3x_3 + 570x_2^3 + 620x_2^2x_3 - 464x_2x_3^2 + 511x_2\}$$

23. A randomly generated example with 4 polynomials in 3 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ -139x_1x_2^2 + 595x_2^2x_3 - 642x_2x_3^2 - 144x_1x_3 - 228x_1 + 80,\ -475x_1^2x_2 + 803x_2^2x_3 - 831x_2^2 - 723x_3^2 - 485x_1 + 133x_2,\ -843x_1^3 + 702x_1x_2^2 - 301x_2^2x_3 + 581x_1x_3^2 - 132x_2x_3^2 + 656x_2^2,\ -918x_1^3 + 526x_1x_2^2 - 190x_2^3 + 860x_1x_3^2 - 897x_1^2 - 401x_1x_2\}$$

24. A randomly generated example with 4 polynomials in 3 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.
$$\{\ -14520x_2^3 + 71804x_1x_3^2 + 13922x_2x_3^2 + 7059x_1^2 + 66336x_2 - 22567,\ -45931x_1x_2^2 + 16349x_1x_3^2 + 72546x_2x_3^2 + 34967x_3^3 - 20047x_2^2 + 99947x_2,\ -37084x_1^2x_2 + 63475x_2^3 - 46400x_1^2x_3 + 25668x_1x_3^2 - 56422x_1^2 + 81098x_2x_3,\ -90153x_1^3 - 32465x_1x_2^2 - 24533x_2^3 - 61015x_1^2x_3 - 59072x_1x_3^2 + 88307x_2x_3\}$$

25. A randomly generated example with 4 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.

$$\begin{aligned}\{\quad & -5x_2^3x_3 - 5x_1^2x_3 - 7x_2^2x_3 - 5x_1^2 - 3x_1x_2 - x_2,\\ & 8x_1^2x_2x_3 + 8x_1x_2x_3^2 - 5x_1^3 + 3x_2^3 + 3x_1x_2,\\ & 7x_1x_2^3 - 5x_1^3x_3 - 5x_2^3x_3 + 6x_1x_2x_3^2 + 7x_1^2x_2,\\ & 9x_1^4 + 8x_2^2x_3^2 + 6x_2x_3^3 + 5x_1^2x_3 + 2x_2\}\end{aligned}$$

26. A randomly generated example with 4 polynomials in 3 variables and maximal degree 4. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3$.

$$\begin{aligned}\{\quad & 78x_1x_2^2x_3 - 94x_1x_3^3 + 57x_1^2x_2 - 58x_2x_3^2 - 39x_1x_2 - 95x_2^2,\\ & 27x_2^4 + 29x_1^2x_2x_3 - 94x_1^2x_3^2 - 43x_1x_3^2 + 11x_1x_2,\\ & -40x_1^2x_2^2 - 59x_1x_2^3 + 100x_1^2x_3 + 91x_1x_2x_3 - 54x_2^2,\\ & 44x_1^4 - 81x_1x_2^3 + 4x_1x_3^3 + 73x_2x_3^3 - 55x_1^2x_2 + 93x_2^2\}\end{aligned}$$

27. A randomly generated example with 3 polynomials in 4 variables and maximal degree 3. The variable are ordered according to the reverse lexicographic ordering with $x_1 > x_2 > x_3 > x_4$.

$$\begin{aligned}\{\quad & 676x_1^2x_2 + 415x_1x_2x_3 + 481x_2^2x_3 + 948x_3^3 - 398x_2^2 + 589x_4,\\ & 666x_1^2x_2 + 22x_2x_3^2 + 599x_3^3 + 715x_1^2x_4 + 813x_2^2x_4 - 446x_1x_2,\\ & 856x_1^3 + 112x_1^2x_4 - 659x_3^2x_4 + 287x_3x_4^2 + 262x_1x_4 - 413x_2\}\end{aligned}$$

We give now several tables of computation times and interesting parameters for the modular variants of Gröbner basis computation presented in this paper. As a prime we used 46337 which was a lucky prime for all these examples.

In Table 7.1 we present interesting parameters and computation times of the Gröbner basis computation with the following columns

1. The number of the example.
2. The number of polynomials in the basis.
3. The number of polynomials in the resulting Gröbner basis.
4. The maximal total degree of the power products in the basis.
5. The maximal total degree of a power products during the Gröbner basis computation.
6. The maximal total degree of the power products in the resulting Gröbner basis. A zero in this column means that the resulting Gröbner basis is $\{1\}$.

7. The maximal length of an integer in the basis.

8. The maximal length of an integer during the Gröbner basis computation.

9. The maximal length of an integer in the resulting Gröbner basis.

10. The number of gcd computations during the Gröbner basis computation. Together with the maximal length of the integers this is a good measure for the time spent in computing coefficients.

11. The number of S-polynomials computations during the Gröbner basis computation.

12. The number of reductions during the Gröbner basis computation.

13. The time in milli seconds needed to compute the Gröbner basis over $\mathbb{Q}$. More specifically the variant were we keep the coefficients in $\mathbb{Z}$. This is the time which all modular variants of Gröbner basis computation are measured by.

In Table 7.2 we present interesting parameters and computation times of the LiftCoeff algorithm with the following columns

1. The number of the example.

2. The maximal length of an integer in the basis.

3. The maximal length of an integer during the Gröbner basis computation.

4. The maximal length of an integer in the resulting Gröbner basis.

5. The number of lifting steps.

6. The number of equations for the coefficients.

7. The number of variables in the equations i.e. the number of coefficients to lift.

8. The time for the Gröbner basis computation.

9. The time for the Gröbner basis computation modulo p.

10. The time for the LiftCoeff algorithm.

We took as a coefficient bound the maximal coefficient in the Gröbner basis. So that we computed only the minimal number of lifting steps. As you can see LiftCoeff is much slower in nearly all cases except those were we do not need to lift because the coefficients can be mapped to $\mathbb{Q}$ from the Gröbner basis modulo p. These case are usually those where the Gröbner basis is $\{1\}$. Example 13 and 25 are the exceptions and have $\{x_2, x_1\}$ and $\{x_2^2, x_1x_2, 5x_1^2x_3+5x_1^2+x_2, x_1^3, 6x_2x_3x_3^3-5x_1^2+x_2\}$ as Gröbner bases. Example 27 is the only one we found where LiftCoeff is faster than the usual

	#polys		max toddeg			max int len						time
#	in	out	in	dur	out	in	dur	out	#gcd	#spol	#red	gb
1	4	7	4	69	20	4	236	29	14643	102	4771	3484
2	4	4	2	18	8	2	171	39	399	3	150	67
3	4	8	2	34	10	2	16918	202	29247	55	8323	242200
4	3	3	3	100	27	2	12500	117	3206	1	446	66467
5	6	6	2	20	10	8	1488	334	422	1	122	284
6	7	1	2	4	0	14	381	1	210	2	83	33
7	3	3	2	14	8	5	110	55	152	4	75	50
8	2	2	6	30	16	7	281	139	178	1	56	83
9	2	4	3	7	5	14	207	117	41	3	15	17
10	2	4	3	7	5	17	382	193	37	3	16	17
11	2	5	4	10	7	7	150	76	66	4	21	17
12	3	1	3	6	0	17	1372	1	163	2	56	66
13	3	2	4	8	1	14	2424	1	357	3	99	383
14	3	1	4	8	0	17	5225	1	435	3	114	1150
15	4	1	3	5	0	17	1741	1	153	1	51	66
16	4	1	4	6	0	17	3363	1	242	1	67	300
17	2	5	4	11	8	13	258	224	105	5	61	50
18	2	9	4	12	9	17	675	391	543	12	214	467
19	3	11	3	9	6	7	728	264	823	18	378	583
20	3	12	3	9	6	14	960	401	711	19	271	534
21	3	13	4	12	6	7	433	188	2021	22	672	784
22	3	11	4	13	9	10	589	439	1225	18	871	966
23	4	1	3	7	0	10	25650	1	1817	7	333	49267
24	4	1	3	7	0	17	21715	1	1914	7	344	46984
25	4	5	4	11	4	4	138224	3	3659	16	604	1324716
26	3	11	3	11	7	10	500	237	1623	17	926	1550
27	4	9	4	10	4	7	19545	25	5458	24	810	138533

Table 7.1: Times and parameters for Gröbner basis computation.

#	max int len: in	dur	out	#steps	#eqns	#vars	time in ms: GB	GB $\mathbb{Z}_p$	LiftCoeff
1	4	236	29	3	853	813	3484	1783	26550
2	2	171	39	4	113	112	67	17	667
3	2	16918	202	25	2415	1599	242200	15734	1670600
4	2	12500	117	14	379	374	66467	283	75800
5	8	1488	334	42	155	155	284	33	16383
6	14	381	1	0	0	0	33	0	0
7	5	110	55	7	82	82	50	17	867
8	7	281	139	17	130	125	83	17	10050
9	14	207	117	14	58	53	17	0	550
10	17	382	193	24	62	56	17	0	1133
11	7	150	76	9	100	88	17	0	733
12	17	1372	1	0	0	0	66	17	17
13	14	2424	1	0	0	0	383	17	17
14	17	5225	1	0	0	0	1150	16	16
15	17	1741	1	0	0	0	66	0	0
16	17	3363	1	0	0	0	300	17	17
17	13	258	224	28	150	123	50	16	3783
18	17	675	391	49	558	476	467	100	61000
19	7	728	264	33	614	529	583	100	69017
20	14	960	401	49	485	401	534	67	101767
21	7	433	188	23	1292	953	784	183	484183
22	10	589	439	55	1703	1224	966	283	394350
23	10	25650	1	0	0	0	49267	50	50
24	17	21715	1	0	0	0	46984	50	50
25	4	138224	3	0	0	0	1324716	116	116
26	10	500	237	29	1829	1253	1550	450	246584
27	7	19545	25	3	68	70	138533	167	250

Table 7.2: Times and parameters for algorithm LiftCoeff.

Gröbner basis computation. At least it proves the concept. For practical purposes these cases are to rare especially if you look at the penalty you pay if there is no speed up.

In Table 7.3 we present interesting parameters and computation times of the LiftTrans algorithm with the following columns

1. The number of the example.
2. The maximal length of an integer during the Gröbner basis computation.
3. The maximal length of an integer during the LiftTrans algorithm.
4. The number of polynomials in the resulting Gröbner basis.
5. The number of equations for the coefficients. We get a system of linear equations for every polynomial in the Gröbner basis and we give the sum of all equations.
6. The number of variables in the equations i.e. the number of coefficients to lift. We get a system of linear equations for every polynomial in the Gröbner basis and we give the sum of all variables.
7. The time for the Gröbner basis computation.
8. The time for the Gröbner basis computation modulo p and the corresponding transformation matrix.
9. The time for the LiftTrans algorithm.

We were not able to obtain results for the examples 1, 3, 4 and 27 because we ran out of memory. For all examples we examined LiftTrans was slower than the usual Gröbner basis computation. For most examples the coefficient growth is even worse than for the Gröbner basis computation. But then the intent of LiftTrans was not to avoid coefficient growth but to compute the coefficients "faster". As you can see this idea does not work.

In Table 7.4 we present interesting parameters of the Trace algorithm with the following columns

1. The number of the example.
2. The maximal total degree of a power products during the Gröbner basis computation.
3. The maximal total degree of a power products during the Trace algorithm.
4. The maximal length of an integer during the Gröbner basis computation.

#	max int len		#polys	#eqns	#vars	time in ms		
	GB	Trans				GB	GB $\mathbb{Z}_p$	LiftTrans
2	171	1870	4	6100	7323	67	1016	191300
5	1488	5015	6	3926	3993	284	667	42284
6	381	1895	1	265	298	33	50	2333
7	110	88	3	251	278	50	34	234
8	281	347	2	132	116	83	34	167
9	207	248	4	45	45	17	0	50
10	382	252	4	51	51	17	0	67
11	150	90	5	84	84	17	0	50
12	1372	2286	1	36	36	66	17	417
13	2424	6334	2	193	191	383	66	6850
14	5225	5585	1	100	112	1150	84	5134
15	1741	3484	1	40	52	66	17	1300
16	3363	4576	1	65	72	300	34	1817
17	258	274	5	134	132	50	17	184
18	675	698	9	643	639	467	117	1667
19	728	5848	11	799	879	583	150	19650
20	960	2655	12	457	452	534	83	2850
21	433	11588	13	2123	2332	784	383	154567
22	589	4465	11	781	777	966	367	6267
23	25650	22073	1	462	545	49267	750	938083
24	21715	40271	1	408	446	46984	700	1089250
25	138224	11811	5	4887	6897	1324716	3000	3724583
26	500	508	11	985	986	1550	567	2733

Table 7.3: Times and parameters for algorithm LiftTrans.

5. The maximal length of an integer during the Trace algorithm.
6. The number of gcd computations during the Gröbner basis computation.
7. The number of gcd computations during the Trace algorithm.
8. The number of S-polynomials computations during the Gröbner basis computation.
9. The number of S-polynomials computations during the Trace algorithm.
10. The number of reductions during the Gröbner basis computation.
11. The number of reductions during the Trace algorithm.

In Table 7.5 we present the computation times of the Trace algorithm with the following columns

1. The number of the example.
2. The time for the Gröbner basis computation.
3. The time for the Gröbner basis computation modulo p and keeping a trace of S-polynomial and reductions.
4. The time for the Trace algorithm. This time does not include the time for testing if the result is indeed a Gröbner basis for the original basis.
5. The time for the checking the result. Checking include the reduction of all polynomial of the original basis and all necessary S-polynomials to 0. This time is not included in the time for the Trace algorithm.

We see that the times for the Trace algorithm range from 50% faster to 20% slower. The biggest speed up is in those case where the final check of the Gröbner basis is costly. Including the time needed for the check we get times from 25% faster to 50% slower. Looking at the number of S-polynomials and reductions one would expect more improvement. But looking at the maximal total degree, maximal integer length and number of gcd computations one sees that the polynomials with the high power products and big coefficients have to be computed anyway. This seems to be the only algorithm which is suited for practical purposes.

In Table 7.6 we present interesting parameters of the TraceCoeff algorithm with the following columns

1. The number of the example.
2. The maximal length of an integer during the Gröbner basis computation.

#	max toddeg		max int len		#gcd		#spoly		#red	
	GB	Trace	GB	Trace	GB	Trace	GB	Trace	GB	Trace
1	69	67	236	236	14643	12844	102	38	4771	2383
2	18	18	171	171	399	466	3	2	150	135
3	34	34	16918	16918	29247	25619	55	21	8323	3530
4	100	100	12500	12500	3206	3502	1	1	446	444
5	20	20	1488	1488	422	515	1	1	122	120
6	4	4	381	381	210	171	2	2	83	50
7	14	14	110	110	152	167	4	3	75	35
8	30	31	281	277	178	244	1	1	56	24
9	7	7	207	207	41	55	3	2	15	8
10	7	7	382	382	37	52	3	2	16	9
11	10	10	150	150	66	86	4	3	21	12
12	6	4	1372	1372	163	144	2	2	56	37
13	8	6	2424	2424	357	308	3	3	99	72
14	8	6	5225	5225	435	374	3	3	114	81
15	5	4	1741	1741	153	148	1	1	51	33
16	6	4	3363	3363	242	220	1	1	67	47
17	11	11	258	224	105	104	5	3	61	23
18	12	12	675	532	543	507	12	7	214	71
19	9	9	728	731	823	869	18	9	378	127
20	9	9	960	960	711	695	19	9	271	80
21	12	11	433	433	2021	2216	22	10	672	327
22	13	13	589	589	1225	1398	18	8	871	226
23	7	5	25650	25650	1817	1423	7	7	333	237
24	7	5	21715	21715	1914	1508	7	7	344	246
25	11	6	138224	138224	3659	3130	16	10	604	497
26	11	11	500	500	1623	1656	17	8	926	254
27	10	7	19545	19545	5458	4809	24	11	810	690

Table 7.4: Parameters for algorithm Trace.

	time in ms			
#	GB	GB $\mathbb{Z}_p$	Trace	check
1	3484	1800	3850	284
2	67	17	83	34
3	242200	15933	224383	4067
4	66467	300	66767	300
5	284	16	300	133
6	33	0	16	0
7	50	17	50	17
8	83	17	83	17
9	17	0	17	0
10	17	0	17	0
11	17	0	16	0
12	66	17	67	0
13	383	0	333	0
14	1150	16	983	0
15	66	17	67	0
16	300	16	250	0
17	50	17	17	17
18	467	100	283	183
19	583	100	317	383
20	534	83	250	317
21	784	183	683	450
22	966	300	533	883
23	49267	50	36633	0
24	46984	50	35633	0
25	1324716	116	1034333	0
26	1550	467	883	1150
27	138533	183	117133	0

Table 7.5: Times for algorithm Trace.

3. The maximal length of an integer during the TraceCoeff algorithm.
4. The number of gcd computations during the Gröbner basis computation.
5. The number of gcd computations during the TraceCoeff algorithm.
6. The number of additions and subtractions of coefficients during the Gröbner basis computation.
7. The number of additions and subtractions of coefficients during the TraceCoeff algorithm.
8. The number of multiplications and divisions of coefficients during the Gröbner basis computation.
9. The number of multiplications and divisions of coefficients during the TraceCoeff algorithm.

In Table 7.7 we present the computation times of the TraceCoeff algorithm with the following columns

1. The number of the example.
2. The time for the Gröbner basis computation.
3. The time for the Gröbner basis computation modulo p and keeping a trace of the operations on the coefficients.
4. The time for the TraceCoeff algorithm. This time does not include the time for testing if the result is indeed a Gröbner basis for the original basis.

We were not able to obtain results for example 3 because we ran out of memory. We have to compute all the big coefficients like for LiftTrans. But we do not deal with polynomials. And looking at the number of operations we see that we avoid quite a lot of them. Despite of that all computation times are much worse than for the usual Gröbner basis computation. And the number of gcd computations tell us why. Since our coefficients are no longer in $\mathbb{Z}$ but in $\mathbb{Q}$ we need a gcd computation for every arithmetic operation. This emphasizes the importance of avoiding gcd computations. Therefore it is a good idea to normalize the polynomials in the Gröbner basis computation not after every reduction step but only if they are in normal form w.r.t. to the actual basis in the computation.

7.3 Conclusion

We tried several variants of modular Gröbner basis computation. The only interesting variant seems to be the Trace algorithm by Traverso. Although the speed up is not

	max int len		#gcd		#add		#mul	
#	GB	TraceC	GB	TraceC	GB	TraceC	GB	TraceC
1	236	236	14643	164502	86534	42387	89764	44785
2	171	162	399	4119	1094	988	1281	1163
4	12500	12500	3206	59885	16153	15613	16677	16110
5	1488	1481	422	4126	982	946	1204	1165
6	381	381	210	594	154	123	249	208
7	110	99	152	778	426	173	515	239
8	281	251	178	708	582	157	703	242
9	207	130	41	228	95	45	129	80
10	382	287	37	286	115	58	150	95
11	150	112	66	544	218	109	274	171
12	1372	1370	163	656	170	156	220	196
13	2424	2413	357	1719	437	410	525	480
14	5225	5225	435	2232	567	527	676	621
15	1741	1744	153	531	137	129	186	169
16	3363	3368	242	879	226	202	300	263
17	258	222	105	951	549	177	717	306
18	675	459	543	6982	5312	1553	5811	1952
19	728	560	823	7974	5714	1865	6077	2144
20	960	586	711	4785	4075	1076	4381	1332
21	433	426	2021	17446	8954	4016	9804	4723
22	589	439	1225	11302	12052	2498	13213	3168
23	25650	25650	1817	10121	2656	2431	2979	2696
24	21715	21717	1914	11013	2901	2646	3234	2919
25	138224	138226	3659	21381	5749	5181	6355	5739
26	500	361	1623	19208	19206	4369	20715	5250
27	19545	19544	5458	35729	9260	8572	10130	9374

Table 7.6: Parameters for algorithm TraceCoeff.

#	time in ms		
	GB	GB $\mathbb{Z}_p$	TraceC
1	3484	3250	15433
2	67	50	467
4	66467	584	656584
5	284	34	3984
6	33	17	67
7	50	17	67
8	83	17	117
9	17	0	16
10	17	17	50
11	17	0	50
12	66	17	250
13	383	17	1584
14	1150	33	4400
15	66	17	200
16	300	17	934
17	50	16	116
18	467	183	1817
19	583	200	2600
20	534	134	1450
21	784	333	4216
22	966	500	2417
23	49267	100	185250
24	46984	100	187683
25	1324716	216	4188566
26	1550	783	4716
27	138533	334	660300

Table 7.7: Times for algorithm TraceCoeff.

that much as one would expect form the avoided S-polynomial computations and reductions. Further advantages of this algorithm are that we can check the result. Even if we choose an unlucky prime we can use the computation done up to the point where we recognize that p is unlucky as a starting point for another try.

LiftCoeff is considerably slower in most cases and although there are examples where we gain a speed up there is still the open problem of finding a reasonable bound for the coefficients of the result.

LiftTrans and TraceCoeff are slower in all cases.

Since unlucky primes are seldom if we choose large primes we can use the Gröbner basis modulo p to check if the basis contains 1. This is useful for checking if a system of equations is solvable.

Appendix A

Program Listing

```
MODULE Algebraic;
(**
   Abstract classes for objects for computing.
   Algebraic defines Objects for algebraic computations (+, -, *, /, ^, neg, inv, =, 0, 1, ..). The argument of an dyadic operator can
   be an arbitrary algebraic object . Therefore implementations handle the cases they know about and delegate the others. For a
   commutative operator this is simply done by swapping the argument and receiver object (e.g.: a+b → b+a). Argument
   swapping works also fine for noncommutative operators that can be expressed in terms of commutative operators
   (e.g.: a-b := a+(-b )).
   If two operands are in incomptabible domains (i.e. there is no natural way to imbed one domain in the other) a trap
   DomainMismatch is generated. e.g.:
      (Integer)7 + (Integer)3 ⇒ (Integer)10
      (Integer)7 + (PrimeNum)(3 mod 5) ⇒ Trap(DomainMismatch)
      (PrimeNum)(7 mod 11) + (PrimeNum)(3 mod 5) ⇒ Trap(DomainMismatch)
      (PrimeNum)(7 mod 11) + (PrimeNum)(3 mod 11) ⇒ (PrimeNum)(10 mod 11)
      (Polynomial)(7x + 2) + (Integer)3 ⇒ (Polynomial)(7x + 5)

   Coefficients and modulus are in the same module to define object.mod(modulus).

   Author: Gerald Slacik (gs)
*)

   IMPORT S := Scanners, F := Formatters;

   CONST
      abstractMethod* = 127; (** Trap code for call of an abstract method. **)
      notYetImplemented* = 126; (** Trap code for call of a procedure not yet implemented. **)
      obsolete* = 125; (** Trap code for call of an obsolete procedure. **)
      divideByZero* = 124; (** Trap code for divsion by zero. **)
      domainMismatch* = 123; (** Trap code for an artihmetic operations on incompatible domains. **)

   TYPE
      Object* = POINTER TO ObjectDesc;
      ObjectDesc* = RECORD
      (**
         Abstract.
         Root of all algebraic objects.
      **)
      END;

      Coefficient* = POINTER TO CoefficientDesc;
      CoefficientDesc* = RECORD (ObjectDesc)
      (**
         Abtsract, Extension.
         Abtsract class for Coefficients of polynomials, matrices etc.
      **)
      END;

      Modulus* = POINTER TO ModulusDesc;
      ModulusDesc* = RECORD (CoefficientDesc)
      (**
         Abstract, Extension
         Abstract class for Integrals modulo a prime, prime polynomials etc.
         A modulus is not just a representation for the result of "a mod p". p is always stored together with a mod p.
         So operations on moduli are always "mod p".
         Example for Integers:
               Modulus a := 3 mod 7 (* a is represented as integer 3 AND integer 7 *)
```

```
            Modulus b := 4 mod 7 (* a is represented as integer 4 AND integer 7 *)
            Modulus c := a + b (* c is represented as integer 0 AND integer 7 *)
            Modulus d := a.Inv() (* d is represented as integer 5 AND integer 7 *)

      Modulus is in the module Algebraic to define a.Mod(b) on an arbitrary Object a.
    **)
    END;

    Param* = RECORD
    (**
      Parameter type used by MapFunc.
      See also: MapFunc.
    **)
    END;

    MapFunc* = PROCEDURE (co: Coefficient; VAR param(*inout*): Param): Coefficient;
    (**
      Function type used by Map.
      For composed objects (e.g. polynomials, matrices etc.) function f is applied to all coefficients of a.
    **)

    Write* = RECORD (F.Write)
    (** Procedure Object for WriteFormat. **)
    END;

  PROCEDURE (b: Modulus) ModFrom* (a: Object): Modulus;
  (**
    Abstract method. Must be overwritten.

    Semantically equivalent to a.Mod(b).
    If a does not know how to handle a.Mod(b) it calls b.ModFrom(a).
    So b can not call a.Mod(b)!

    If the receiver of a message does not know the type of its parameter
    receiver and parameter are swapped. this works fine for commutative
    operators such as '*' and '+' or operators which can be defined with a
    commutative operator such as '/' [* inv] and '-' [+ neg]
    but for noncommutative we have to define
    a message where receiver and parameter are swapped.

    See also Mod.
  **)
  END ModFrom;

  (** Object **)

  PROCEDURE (a: Object) Write* (VAR f(*inout*): F.Formatter);
  (**
    Abstract method. Must be overwritten.
    Writes a textual representation of a with f.
  **)
  END Write;

  PROCEDURE (VAR w(*inout*): Write) Write* (VAR f(*inout*): F.Formatter);
  END Write;

  PROCEDURE (o: Object) WriteF* (VAR f(*inout*): F.Formatter; str: ARRAY OF CHAR);
  (**
    Write formatted object o.
    See also: WriteFormat.
  **)
  END WriteF;

  PROCEDURE (a: Object) Eql* (b: Object): BOOLEAN;
  (**
    Abstract method. Must be overwritten.

    result = (a = b).
  **)
  END Eql;

  PROCEDURE (a: Object) Zero* (): Object;
  (**
    Abstract method. Must be overwritten.

    result = 0.
  **)
  END Zero;

  PROCEDURE (a: Object) One* (): Object;
  (**
    Abstract method. Must be overwritten.

    result = 1.
  **)
```

```
END One;

PROCEDURE (a: Object) Neg* (): Object;
(**
  Abstract method. Must be overwritten.
  The type of the result is the same as the type of a.

  result = -a.
**)
END Neg;

PROCEDURE (a: Object) NegOne* (): Object;
(**
  Default method.
  The type of the result is the same as the type of a.

  result = -1.

  Except for performace, equivalent to:
    RETURN a.One().Neg().
**)
END NegOne;

PROCEDURE (a: Object) Inv* (): Object;
(**
  Abstract method. Must be overwritten.

  result = 1 / a.

  Pre
  ~a.IsZero() divideByZero
**)
END Inv;

PROCEDURE (a: Object) Add* (b: Object): Object;
(**
  Abstract method. Must be overwritten.

  result = a + b.
**)
END Add;

PROCEDURE (a: Object) Mul* (b: Object): Object;
(**
  Abstract method. Must be overwritten.

  result = a * b.
**)
END Mul;

PROCEDURE (a: Object) IsZero* (): BOOLEAN;
(**
  Default method.

  result = (a = 0).

  Except for performace, equivalent to:
    a.Eql(a.Zero()).
**)
END IsZero;

PROCEDURE (a: Object) IsOne* (): BOOLEAN;
(**
  Default method.

  result = (a = 1).

  Except for performace, equivalent to:
    a.Eql(a.One()).
**)
END IsOne;

PROCEDURE (a: Object) IsNegOne* (): BOOLEAN;
(**
  Default method.

  result = (a = -1).

  Except for performace, equivalent to:
    a.Eql(a.NegOne()) .
**)
END IsNegOne;

PROCEDURE (a: Object) IsAbsOne* (): BOOLEAN;
(**
```

```
    Default method.

    result = (|a| = 1).

    Except for performace, equivalent to:
        a.IsOne() OR a.IsNegOne()
**)
END IsAbsOne;

PROCEDURE (a: Object) Sub* (b: Object): Object;
(**
    Default method.

    result = a - b.

    Except for performace, equivalent to:
        a.Add(b.Neg()).
**)
END Sub;

PROCEDURE (a: Object) Div* (b: Object): Object;
(**
    Default method.

    result = a / b.

    Pre
    ~a.IsZero() divideByZero

    Except for performace, equivalent to:
        a.Add(b.Inv()).
**)
END Div;

PROCEDURE (a: Object) Pow* (exp: INTEGER): Object;
(**
    Default method.

    result = a ^ exp.
    The type of the result is the same as the type of a.

    Pre
    exp >= 0 20

    Except for performace, equivalent to:
        VAR c: Object;
        c := a.One();
        WHILE exp > 0 DO
                IF ODD(exp) THEN c := c.Mul(a) END;
                a := a.Mul(a); exp := exp DIV 2
        END;
        RETURN c
**)
END Pow;

PROCEDURE (a: Object) Mod* (b: Modulus): Object;
(**
    Default method.

    Let b = x mod p.
    result = a mod p.

    Except for performace, equivalent to:
        RETURN b.ModFrom(a)

    See also ModFrom.
**)
END Mod;

PROCEDURE (a: Object) Demod*(): Object;
(**
    Default method.


    Let a = x mod p.
    result = x.
    Demod is defined for Objects (and not only for Modulus) to "demod" polynomials, matrices etc.

    Example for Integers:
        Modulus a := 3 mod 7 ( a is represented as integer 3 AND integer 7 )
        Object res := a.Demod() ( res is represented as integer 3 )

    Default implementation calls HALT(domainMismatch)
**)
END Demod;
```

```
PROCEDURE (a: Object) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): Object);
(**
   Abstract method. Must be overwritten.

   Reads a textual representation of a with s. Read does not read operators.
   So Read can not read polynomials, powerproducts, quotients etc.
   In normal use ReadExpression should be called instead of Read.

   Read is used in ReadExpression to convert scanned numbers & names into Objects.
   a is used to guard the recognition of names.
   e.g.: a is a polynomial over integer mod 7 with variables x,y,z. Then
      a.ReadExpression("12y") yields (5 mod 7)*y,
      a.ReadExpression("2a+b") gives an "unknown variable" error.
**)
END Read;

PROCEDURE (a: Object) ReadExpression* (VAR s(*inout*): S.Scanner; VAR res(*out*): Object);
(**
   Default method.

   Reads an Expression of the form:
   Expression = ["+" | "-"] Term {AddOperator Term}.
   AddOperator = "+" | "-".
   Term = Factor {[MulOperator] Factor}.
   MulOperator = "*" | "/".
   Factor = (Object | "(" Expression ")" ) ["^" Number] .
   Object = as defined by a.Read(s)

   Two consecutive factors without a MulOperator between are multiplied!
   Attention: If s does not allow whiteSpace in numbers!
**)
END ReadExpression;

(** Coefficient **)

PROCEDURE (a: Coefficient) IsNeg* (): BOOLEAN;
(**
   Abstract method. Must be overwritten.
   Defined for coefficients to write polnomials.

   result = (a < 0).
**)
END IsNeg;

PROCEDURE (a: Coefficient) Abs* (): Coefficient;
(**
   Default method.

   result = |a|.

   Except for performace, equivalent to:
      IF a.IsNeg() THEN a := a.Neg() END;
      RETURN a.
**)
END Abs;

(** Modulus **)

PROCEDURE (a: Modulus) Mod* (b: Modulus): Object;
(**
   Extension.
   Overwrites Coefficient.Mod.

   Since a is already "x mod p" you should not call Mod in the first place.
   HALT(domainMismatch).
**)
END Mod;


PROCEDURE (a: Modulus) Demod*(): Object;
(**
   Abstract method. Must be overwritten. Extension.
   Overwrites Coefficient.Demod.

   Let b = x mod p.
   result = x.
**)
END Demod;

END Algebraic.
```

```
MODULE IntegralsAndQuotients;
(**
   Abstract classes for Integral Domains, QuotientFields.
   They are implented in one module to avoid cyclic import. Quotients are based on Integrals.

   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic, S := Scanners, F := Formatters;

   TYPE
      Integral* = POINTER TO IntegralDesc;
      IntegralDesc* = RECORD (A.CoefficientDesc)
      (**
         Abstract.

         Root of all elements of integral domains.
         Integral.ReadExpression returns quotients if the occur in the input (e.g. "1/2").
      **)
      END;

      Quotient* = POINTER TO QuotientDesc;
      QuotientDesc* = RECORD (A.CoefficientDesc)
      (**
         Class for elements of quotient fields = (quotients over integral domains).
         Quotients are stored in normal form a/b where:
         ~a.IsZero() & ~b.IsNeg() & a.Gcd(b).IsOne()

         No Init(quotient: Quotient; numerator, denominator: Integral): Quotient
         function is available since n/d may be an Integral. Use num.Div(den) instead.
      **)
         num-: Integral; (** numerator # 0 **)
         den-: Integral (** denominator > 0 **)
      END;

   (** Integrals **)

   PROCEDURE (a: Integral) Inv* (): A.Object;
   (**
      Extension.
      Overwrites Coefficient.Inv.

      result = 1/a.
      Pre
      ~a.IsZero() divideByZero
   **)
   END Inv;

   PROCEDURE (a: Integral) QuoRem* (b: Integral; VAR q(*out*), r(*out*): Integral);
   (**
      Abstract method. Must be overwritten.

      Calculate q, r s.t.
         a = b*q + r &
         0 <= abs(r) < abs(b) &
         sgn(r) = sgn(a) &
         sgn(q) = sgn(a*b)
         (i.e. rounds towards zero).

      Pre
      ~b.IsZero() divideByZero
   **)
   END QuoRem;

   PROCEDURE (a: Integral) QuoRemSwapped* (b: Integral; VAR quo(*out*), rem(*out*): Integral);
   (**
      Abstract method. Must be overwritten.

      Semantically equivalent to b.QuoRem(a).
      If a is an extension of Integral that b can not handle (e.g. b is an Integer and a is an integral polynom
      (i.e. a polynom with integral coefficients)) then b calls a.QuoRemSwapped(b, q, r).
      So you can not call b.QuoRem(a).

      Pre
      ~a.IsZero() divideByZero

      See also ModFrom, QuoRem.
   **)
   END QuoRemSwapped;

   PROCEDURE (a: Integral) IDiv* (b: Integral): Integral;
   (**
      Default method.

      result = a/b, for division where the remainder is known to be 0.
```

```
    Pre
    ~b.IsZero() divideByZero
    a.Rem.b.IsZero() 20

    Except for performace, equivalent to:
        VAR r: Integral;
        RETURN a.QuoRem(b, r).

    See also QuoRem.
**)
END IDiv;

PROCEDURE (a: Integral) Quo* (b: Integral): Integral;
(**
    Default method.

    result = a/b.

    Pre
    ~b.IsZero() divideByZero

    Except for performace, equivalent to:
        VAR q, r: Integral;
        a.QuoRem(b, q, r);
        RETURN q

    See also QuoRem.
**)
END Quo;

PROCEDURE (a: Integral) Rem* (b: Integral): Integral;
(**
    Default method.

    result = a mod b.

    Pre
    ~b.IsZero() divideByZero

    Except for performace, equivalent to:
        VAR q, r: Integral;
        a.QuoRem(b, q, r);
        RETURN r

    See also QuoRem.
**)
    VAR q, r: Integral;
END Rem;

PROCEDURE (a: Integral) IsMultipleOf* (b: Integral): BOOLEAN;
(**
    Default method.

    result = (b | a).

    Pre
    ~a.IsZero() divideByZero

    Except for performace, equivalent to:
        a.Rem(b).IsZero().

    See also Rem.
**)
END IsMultipleOf;

PROCEDURE (a: Integral) Gcd* (b: Integral): Integral;
(**
    Default method.

    result = Greatest common divisor of a and b.

    Post
    ~result.IsNeg() 60.
    a.IsZero() & b.IsZero()
        result.IsZero() 61
    a.IsZero() & ~b.IsZero()
        result.Eql(b.Abs()) 62
    ~a.IsZero() & b.IsZero()
        result.Eql(a.Abs()) 63

    Except for performace, equivalent to:
        VAR c: Integral;
        a := a.Abs(); b := b.Abs();
        WHILE ~b.IsZero() DO c := a.Rem(b); a := b; b := c; END;
```

```
        RETURN a.

    See also Rem.
  **)
  END Gcd;

  PROCEDURE (a: Integral) Lcm* (b: Integral): Integral;
  (**
    Default method.

    result = Least common multiple of a and b.

    Except for performace, equivalent to:
      a.Mul(b.IDiv(a.Gcd(b))).

    See also Gcd, IDiv.
  **)
  END Lcm;

  PROCEDURE (a: Integral) Gcdex* (b: Integral; VAR gcd(*out*), u(*out*), v(*out*): Integral);
  (**
    Default method.
    Greatest common divisor of a and b with cofactors.

    Post
    result = a.Gcd(b) 60
    result = u.Mul(a).Add(v.Mul(b)) 61

    Except for performace, equivalent to:
      VAR q, r, aa, bb: Integral;
      aa := a; bb := b; u := a.One(); v := a.Zero();
      WHILE ~bb.IsZero() DO
          q := aa.QuoRem(bb, r); aa := bb; bb := r;
          r := u.Sub(v.Mul(q)); u := v; v := r;
      END;
      IF aa.IsNeg() THEN aa := aa.Neg(); u := u.Neg(); v := v.Neg(); END;
      v := aa.Sub(u.Mul(a)).IDiv(bb);
      RETURN aa.

    See also Gcd.
  **)
  END Gcdex;

  PROCEDURE (a: Integral) Div* (b: A.Object): A.Object;

  (** Quotients **)

  PROCEDURE (a: Quotient) IsZero* (): BOOLEAN;
  PROCEDURE (a: Quotient) IsOne* (): BOOLEAN;
  PROCEDURE (a: Quotient) IsNegOne* (): BOOLEAN;
  PROCEDURE (a: Quotient) IsAbsOne* (): BOOLEAN;
  PROCEDURE (a: Quotient) IsNeg* (): BOOLEAN;
  PROCEDURE (a: Quotient) Zero* (): A.Object;
  PROCEDURE (a: Quotient) One* (): A.Object;
  PROCEDURE (a: Quotient) NegOne* (): A.Object;
  PROCEDURE (a: Quotient) Neg* (): A.Object;
  PROCEDURE (a: Quotient) Inv* (): A.Object;
  PROCEDURE (a: Quotient) Eql* (b: A.Object): BOOLEAN;
  PROCEDURE (a: Quotient) Add* (b: A.Object): A.Object;
  PROCEDURE (a: Quotient) Mul* (b: A.Object): A.Object;
  PROCEDURE (a: Quotient) Mod* (b: A.Modulus): A.Object;
  PROCEDURE (a: Quotient) Demod* (): A.Object;
  PROCEDURE (a: Quotient) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): A.Object);
  PROCEDURE (a: Quotient) Write* (VAR f(*inout*): F.Formatter);

END IntegralsAndQuotients.
```

```
MODULE Integers;
(**
   Arithmetic for arbitrary long integers.

   Author: Gerald Slacik (gs)
*)

   IMPORT IQ := IntegralsAndQuotients, A := Algebraic, S := Scanners, F := Formatters;

   TYPE
      Integer* = POINTER TO IntegerDesc;
      IntegerDesc* = RECORD (IQ.IntegralDesc)
      (** Abitrary long integers. **)
      END;

   VAR
      zero*, one*, negOne*: Integer;
      maxLen-: LONGINT; (** The length of the absolute greatest integer during the computation. Length is measured in bits. **)
      gcdCount-: LONGINT; (** The number of gcd calls. **)

   PROCEDURE InitProfiling*;
   (**
      Initializes maxLen & gcdCount with 0.
   **)
   END InitProfiling;

   PROCEDURE Len* (a: Integer): Length;
   (**
      Returns the number of bits neccesary to represent a.
   **)
   END Len;

   PROCEDURE (a: Integer) IsZero* (): BOOLEAN;
   PROCEDURE (a: Integer) IsOne* (): BOOLEAN;
   PROCEDURE (a: Integer) IsNegOne* (): BOOLEAN;
   PROCEDURE (a: Integer) IsAbsOne* (): BOOLEAN;
   PROCEDURE (a: Integer) IsNeg* (): BOOLEAN;
   PROCEDURE (a: Integer) Zero* (): A.Object;
   PROCEDURE (a: Integer) One* (): A.Object;
   PROCEDURE (a: Integer) NegOne* (): A.Object;
   PROCEDURE (a: Integer) Neg* (): A.Object;
   PROCEDURE (a: Integer) Abs* (): A.Coefficient
   PROCEDURE (a: Integer) Eql* (b: A.Object): BOOLEAN;
   PROCEDURE (a: Integer) Lss* (b: Integer): BOOLEAN;
   (** result = (a < b). **)
   END Lss;
   PROCEDURE (a: Integer) Add* (b: A.Object): A.Object;
   PROCEDURE (a: Integer) Mul* (b: A.Object): A.Object;
   PROCEDURE (a: Integer) QuoRem* (b: IQ.Integral; VAR q(*out*), r(*out*): IQ.Integral);
   PROCEDURE (a: Integer) Quo* (b: IQ.Integral): IQ.Integral;
   PROCEDURE (a: Integer) Rem* (b: IQ.Integral): IQ.Integral;
   PROCEDURE (a: Integer) IDiv* (b: IQ.Integral): IQ.Integral;
   PROCEDURE (a: Integer) Gcd* (b: IQ.Integral): IQ.Integral;
   PROCEDURE (a: Integer) Write* (VAR f(*inout*): F.Formatter);
   PROCEDURE ConvertFromString (c: Integer; VAR str(*in*): ARRAY OF CHAR; len: LONGINT);
   (*
      Converts str into an integer (basis 10).
      number = digit {digit}.
      digi = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
      str is declared as VAR parameter for performance reasons; it is never changed.

      Pre
      c # NIL 20
      (len < LEN(str)) & (str[len] = 0X 21
      ("0" <= str[i]) & (str[i] <= "9") for 0 <= i < le 22, 23
   *)
   END ConvertFromString;

   PROCEDURE FromString* (c: Integer; str: ARRAY OF CHAR);
   (*
      Converts str into an integer (basis 10).
      number = digit {digit}.
      digi = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

      Pre
      c # NIL
      there is a len s.t. (len < LEN(str)) & (str[len] = 0X)
      ("0" <= str[i]) & (str[i] <= "9") for 0 <= i < len
   *)
   END FromString;

   PROCEDURE (a: Integer) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): A.Object);

   PROCEDURE Long* (c: Integer; a: LONGINT);
   (*
```

```
      Convert the LONGINT a into an Integer.
      c = a.
   *)
   END Long;

   PROCEDURE Short* (a: Integer): LONGINT;
   (*
      Convert the Integer a into a LONGINT.
      result = a.

      Pre
      MIN(LONGINT) <= a <= MAX(LONGINT) 20, 21, 22
   *)
   END Short;

END Integers.
```

```
MODULE IntegerModuli;
(**
   Arithmetic for elements of Galois Fields of type GF(p) with p integer.
   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic, I := Integers;

   TYPE
      IntegerModulus* = POINTER TO IntegerModulusDesc;
      IntegerModulusDesc* = RECORD (A.ModulusDesc)
      (**
         Abtsract, Extension.
         Abtsract class for elements of Galois fields of type GF(p).
      **)
      END;

   VAR
      (**
         If primeNumSigned is true IntegerModuli are centered around 0; affects IsNeg, Demod and Write.

         Let a = x mod p.
         If primeNumSigned then -p/2 <= x <= p/2 else 0 <= x < p.

         Intial value is FALSE.
      **)
      primeNumSigned*: BOOLEAN;

   PROCEDURE (a: IntegerModulus) Mod* (b: A.Modulus): A.Object;
   (**
      Extension.
      Overwrites Algebraic.Object.Mod.

      Since a is already "x mod p" you should not call Mod in the first place.
      HALT(domainMismatch).
   **)
   END Mod;

   PROCEDURE (a: IntegerModulus) Prime* (): I.Integer;
   (**
      Abstract method. Must be overwritten.

      Let a = x mod p.
      result = p.
   **)
   END Prime;

END IntegerModuli.
```

```
MODULE IntegersModSmallPrime;
(**
   Arithmetic for elements of Galois Fields of type GF(p) with p a small integer (fitting in a computer word).
   Author: Gerald Slacik (gs)
*)

   IMPORT
      A := Algebraic, IQ := IntegralsAndQuotients, I := Integers, IM := IntegerModuli, S := Scanners, F := Formatters;

   CONST
      maxPrime* = 46337; (** greatest prime for IntModPrimes **)

   TYPE
      IntModPrime* = POINTER TO IntModPrimeDesc;
      IntModPrimeDesc* = RECORD (IM.IntegerModulusDesc)
      (**
         Extension.
         Class for elements of Galois fields of type GF(p) with p <= maxPrime.
      **)
      END;

   VAR
      imaxPrime-: I.Integer;

   PROCEDURE (a: IntModPrime) IsZero* (): BOOLEAN;
   PROCEDURE (a: IntModPrime) IsOne* (): BOOLEAN;
   PROCEDURE (a: IntModPrime) IsNegOne* (): BOOLEAN;
   PROCEDURE (a: IntModPrime) IsAbsOne* (): BOOLEAN;
   PROCEDURE (a: IntModPrime) IsNeg* (): BOOLEAN;
   PROCEDURE (a: IntModPrime) Eql* (b: A.Object): BOOLEAN;
   (**
      Extension.
      Overwrites Algebraic.Object.Eql.

      IF b IS IntModPrime THEN
         IF a.Prime() = b.Prime() THEN
               result = (x = y)
         ELSE
               Trap(domainMismatch)
         END
      ELSIF b IS Integer THEN
         Trap(domainMismatch)
      ELSE
         result = b.Eql(a)
      END

      Pre
      ~(b IS Integer OR b IS IntModPrime & a.Prime() # b.Prime() domainMismatch
   **)
   END Eql;

   PROCEDURE (a: IntModPrime) Zero* (): A.Object;
   PROCEDURE (a: IntModPrime) One* (): A.Object
   PROCEDURE (a: IntModPrime) NegOne* (): A.Object;
   PROCEDURE (a: IntModPrime) Neg* (): A.Object;
   PROCEDURE (a: IntModPrime) Inv* (): A.Object;
   PROCEDURE (a: IntModPrime) Demod* (): A.Object;
   (**
      Extension.

      Let a = x mod p.
      result = x.

      Post
      primeNumSigned
         -p/2 <= x <= p/2
      ~primeNumSigned
         0 <= x < p.
   **)
   END Demod;

   PROCEDURE (a: IntModPrime) Prime* (): I.Integer;
   (**
      Extension.

      Let a = x mod p.
      result = p.
   **)
   END Prime;

   PROCEDURE (a: IntModPrime) Add* (b: A.Object): A.Object;
   PROCEDURE (a: IntModPrime) Sub* (b: A.Object): A.Object;
   PROCEDURE (a: IntModPrime) Mul* (b: A.Object): A.Object;
   PROCEDURE (a: IntModPrime) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): A.Object);
   PROCEDURE (a: IntModPrime) Write* (VAR f(*inout*): F.Formatter);
```

```
  PROCEDURE IsPrime* (num: LONGINT): BOOLEAN;
  (**
     Checks if num is a prime < maxPrime.

     result = (num is prime) & (prime < maxPrime)
  **)
  END IsPrime;

  PROCEDURE NextPrime* (num: LONGINT): LONGINT;
  (**
     Returns a prime number > num s.t.

     Post
     IsPrime(result)
     (result > num) OR (result = maxPrime)
  **)
  END NextPrime;

  PROCEDURE PrevPrime* (num: LONGINT): LONGINT;
  (**
     Returns a prime number < num s.t.

     Post
     IsPrime(result)
     (result < num) OR (result = 2)
  **)
  END PrevPrime;

  PROCEDURE InitIntModPrime* (p: IntModPrime; num, prime: I.Integer);
  (**
     Initializes the IntModPrime p with num and prime.

     Pre
     (1 < prime ) & (prime <= maxPrime) 20
     IsPrime(prime) 21
  **)
  END InitIntModPrime;

END IntegersModSmallPrime.
```

```
MODULE Matrices;
(**
   Matrices over Coefficients.

   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic, F := Formatters;

   TYPE
      Dimension* = LONGINT;
      Matrix* = POINTER TO MatrixDesc;
      MatrixDesc* = RECORD (A.ObjectDesc)
      (** Matrices over Coefficients *)
      END;

   PROCEDURE (a: Matrix) Fill* (elem: A.Coefficient);
   (**
      Fill matrix a with elem Elements.

      Pre
      elem # NIL 20

      Post
      a.Get(i, j) = elem for all 0 <= i < a.Rows(), 0 <= j < a.Columns().
   **)
   END Fill;

   PROCEDURE (a: Matrix) IsZero* (): BOOLEAN;
   PROCEDURE (a: Matrix) IsOne* (): BOOLEAN;
   PROCEDURE (a: Matrix) IsNegOne* (): BOOLEAN;
   PROCEDURE (a: Matrix) Zero* (): A.Object;
   PROCEDURE (a: Matrix) One* (): A.Object;
   PROCEDURE (a: Matrix) NegOne* (): A.Object;
   PROCEDURE (a: Matrix) Neg* ():A.Object;
   PROCEDURE (a: Matrix) Inv* (): A.Object;
   PROCEDURE (a: Matrix) Eql* (b: A.Object): BOOLEAN;
   PROCEDURE (a: Matrix) Add* (b: A.Object): A.Object;
   PROCEDURE (a: Matrix) Sub* (b: A.Object): A.Object;
   PROCEDURE (a: Matrix) Mul* (b: A.Object): A.Object;
   PROCEDURE (a: Matrix) Mod* (b: A.Modulus): A.Object;
   PROCEDURE (a: Matrix) Demod* (): A.Object;
   PROCEDURE (a: Matrix) Write* (VAR f(*inout*): F.Formatter);

   PROCEDURE (a: Matrix) Map* (f: A.MapFunc; VAR param(*inout*): A.Param): Matrix;
   (**
      Applies the function f to every element of matrix a.

      result = f(a).

      Post
      c.Get(i, j) = f(a.Get(i, j)) for all 0 <= i < a.Rows() & 0 <= j < a.Columns()
   **)
   END Map;

   PROCEDURE (a: Matrix) Rows* (): Dimension;
   (** Returns the number of rows of matrix a. **)
   END Rows;

   PROCEDURE (a: Matrix) Columns* (): Dimension;
   (** Returns the number of columns of matrix a. **)
   END Columns;

   PROCEDURE (a: Matrix) Get* (row, col: Dimension): A.Coefficient;
   (**
      Returns the element in the row-th row and col-th column of matrix a.

      result = a[row, col].

      Pre
      (col >= 0) & (col < a.Columns()) & (row >= 0) & (row < a.Rows()) 20
   **)
   END Get;

   PROCEDURE (a: Matrix) Put* (row, col: Dimension; elem: A.Coefficient);
   (**
      Set the element in the row-th row and col-th column of matrix a to elem.

      a[row, col] := elem.

      Pre
      (col >= 0) & (col < a.Columns()) & (row >= 0) & (row < a.Rows()) 20
      elem # NIL 21

      Post
```

```
    a.Get(row, col) = elem
    a.Get(i, j) = a'.Get(i, j) for all 0 <= i # row < a.Rows() & 0 <= j # col < a.Columns()
  **)
  END Put;

  PROCEDURE (a: Matrix) JoinRows* (b: Matrix): Matrix;
  (**
    Joins the rows of matrix a and matrix b.

    Pre
    a.Columns() = b.Columns() 20

    Post
    result.Get(i, j) = a.Get(i, j) for all 0 <= i < a.Rows() & 0 <= j < a.Columns()
    result.Get(i + a.Rows(), j) = b.Get(i, j) for all 0 <= i < b.Rows() & 0 <= j < b.Columns()
  **)
  END JoinRows;

  PROCEDURE (a: Matrix) JoinColumns* (b: Matrix): Matrix;
  (**
    Joins the columns of matrix a and matrix b.

    Pre
    a.Rows() = b.Rows() 20

    Post
    result.Get(i, j) = a.Get(i, j) for all 0 <= i < a.Rows() & 0 <= j < a.Columns()
    result.Get(i, j + b.Columns()) = b.Get(i, j) for all 0 <= i < b.Rows() & 0 <= j < b.Columns()
  **)
  END JoinColumns;

  PROCEDURE Init* (a: Matrix; rows, cols: Dimension);
  (**
    Initalize a as rows x cols matrix.

    Pre
    (cols > 0) & (rows > 0) 20

    Post
    a.Rows() = rows
    a.Columns = cols
  **)
  END Init;

END Matrices.
```

```
MODULE Polynomials;
(**
   Polynomials over Coefficients.
   Polynomials are polynomials with coefficients of type Coefficient.
   Since coefficients of polynomials can be polynomials too, we differentiate between them with the extension level extlvl.
   E.g.: let p1 be a polynomial with extension level l1 and p2 a polynomial with extensionlevel l2.
   If l1 < l2 then p1 is in the coefficient ring of p2.
   Coefficients which are not polynomials have an imaginary extension level of 0.
   We demand that results of operation with polynomials are of type Polynomial (e.g. 0), so operations that are only appplicable to
   polynomials need not be defined for coefficients (eg. LeadingPowerProduct(1) ).

   Author: Gerald Slacik (gs)
*)

   IMPORT IQ := IntegralsAndQuotients, A := Algebraic;

   TYPE
      Length* = INTEGER;
      ExtLvl* = INTEGER; (** Extension level > 0**)
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (A.CoefficientDesc)
      (**
         Abstract, Extension.
         Polynomials over Coefficients.
      **)
      END;

   PROCEDURE (a: Polynomial) Length* (): Length;
   (**
      Abstract method. Must be overwritten.

      The number of coefficients # 0. Length(0) = 0.
   **)
   END Length;

   PROCEDURE (a: Polynomial) Coeff* (n: Length): A.Coefficient;
   (**
      Abstract method. Must be overwritten.

      Returns the nth coefficient.
      a.Coeff(0) returns the "leading" coefficient.
      The ordering of the coefficients is determined by the extension.

      Pre
         a.IsZero() OR 0 <= n < a.Length() 20

      Post
      a.IsZero()
         a.Coeff(0).IsZero()
      ~a.IsZero()
         ~a.Coeff(n).IsZero() for 0 <= n < a.Length()
   **)
   END Coeff;

   PROCEDURE (a: Polynomial) Map* (f: A.MapFunc; VAR param(*inout*): A.Param): Polynomial;
   (**
      Abstract method. Must be overwritten.

      Function f is applied to all coefficients of a.

      Let a = Sum ci*pi.
      result = Sum f(ci, param)*pi.
   **)
   END Map;

   PROCEDURE (a: Polynomial) Mod* (b: A.Modulus): A.Object;
   (**
      Extension. Default method.
      Overwrites Coefficient.Mod.

      Let b = x mod p.
      result = a mod p.

      Except for performace, equivalent to:
         RETURN a.Map(Mod, b)
   **)
   END Mod;

   PROCEDURE (a: Polynomial) Demod* (): A.Object;
   (**
      Extension. Default method.
      Overwrites Coefficient.Demod.

      Let b = x mod p.
      result = x.
```

```
    Except for performace, equivalent to:
        RETURN a.Map(Demod, emptyParam)
END Demod;

PROCEDURE (a: Polynomial) Content* (): A.Coefficient;
(**
    Default method.

    For integtal coefficients (i.e. GbAlgebraic.Integral) content is the greatest common divisor of the coefficients.
    For rational coefficients (i.e. GbAlgebraic.Quotient) content is the greatest common divisor of the numerators
    of the coefficients over the least common multiple of the denominators of all fractions.
    Content of 0 is 0.
    See also: ContentAndPrimPart, PrimPart

    Pre
    (Coeff(n) IS Integral) OR (Coeff(n) IS Quotient) for all 0 <= n < a.Length() domainMismatch

    Post
    (result IS Integral) OR (result IS Quotient) 60
    a.IsZero()
        result.IsZero()
**)
END Content;

PROCEDURE (a: Polynomial) ContentAndPrimPart* (VAR content(*out*): A.Coefficient; VAR primPart(*out*): Polynomial);
(**
    Default method.

    Content and Primitive part. Primitive part of 0 is 0. Content of 0 is 0.
    See also: Content, PrimPart

    Pre
    (Coeff(n) IS Integral) OR (Coeff(n) IS Quotient) for all 0 <= n < a.Length() domainMismatch

    Post
    a.IsZero()
        result.IsZero() 60
    ~a.IsZero()
        result.Content().IsOne() 61

    Except for performace, equivalent to:
        content := a.Content();
        IF ~content.IsZero() THEN primPart := primPart.Div(Content()) END

**)
END ContentAndPrimPart;

PROCEDURE (a: Polynomial) PrimPart* (): Polynomial;
(**
    Default method.

    Primitive part. Primitive part of 0 is 0.
    See also: ContentAndPrimPart, Content

    Pre
    (Coeff(n) IS Integral) OR (Coeff(n) IS Quotient) for all 0 <= n < a.Length() domainMismatch

    Post
    a.IsZero()
        result.IsZero() 60
    ~a.IsZero()
        result.Content().IsOne( 61

    Except for performace, equivalent to:
        a.ContentAndPrimPart(content, primPart);
        RETURN primPart
**)
END PrimPart;

PROCEDURE (a: Polynomial) ExtLvl* (): ExtLvl;
(**
    Abstract method. Must be overwritten.

    Extension level.

    Since coefficients of polynomials can be polynomials too, we differentiate between them with the extension level extlvl.
    E.g.: let p1 be a polynomial with extension level l1 and p2 a polynomial with extensionlevel l2.
    If l1 < l2 then p1 is in the coefficient ring of p2.
    Coefficients which are not polynomials have an imaginary extension level of 0.

    Post
    result > 0
**)
END ExtLvl;
```

```
END Polynomials.
```

```
MODULE Dictionary;
(**
   Dictionary for algebraic objects with names and keys. It associates names with objects and integer keys with names.
   It is primarily used to read algebraic objects. Every scanned named is matched with the object stored under that name.
   Keys are used for writing names. Other uses would be a dictionary for variables (i.e. the stored value is the variables contents).
   You can:
      find an object by its name,
      insert or remove an onbject with its name and key,
      write a name by its key,
      reset all names, keys and objects.

   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic, F := Formatters;

   CONST
      maxNameLen* = 32; (** maximum length of a name including terminating 0X. **)
      maxKey* = 127; (** maximum value for keys. **)

   TYPE
      Name* = ARRAY maxNameLen OF CHAR; (** Name of variables. **)

   PROCEDURE Find* (name: ARRAY OF CHAR): A.Object;
   (**
      Searches the name in the dictionary and returns its value. If the name is not found NIL is returned.

      result = object with name or NIL if no object with name exists.

      Pre
      0 < Length(name) < maxNameLen 20
   **)
   END Find;

   PROCEDURE Insert* (name: ARRAY OF CHAR; key: INTEGER; object: A.Object);
   (**
      Inserts the name and corresponding key and object in the dictionary.
      You can remove a name and corresponding key and object from the dictionary by setting object NIL .

      Pre
      0 < Length(name) < maxNameLen 20
      0 <= key < maxKey 21

      Post
      Find(name) = object 60
   **)
   END Insert;

   PROCEDURE WriteName* (VAR f(*inout*): F.Formatter; key: INTEGER);
   (**
      Writes the the corresponding name of key.
      The key must be valid i.e. the corresponding name must not be empty.

      Pre
      0 <= key < maxKey 21
      there exists an object with key key 23
   **)
   END WriteName;


   PROCEDURE Reset* ;
   (** Clears all entries in the dictionary. **)
   END Reset;

END Dictionary.
```

```
MODULE Linear;
(**
   Linear polynomials over Coefficients.
   Linear defines linear polynomials. Linear polynomials are an optimized version of the more general mutlivariate polynomials.

   Author: Gerald Slacik (gs)
*)

   IMPORT P := Polynomials, D := Dictionary, A := Algebraic, S := Scanners, F := Formatters;

   TYPE
      ExtLvl* = P.ExtLvl; (** Extension level > 0 **)
      Length* = P.Length;
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (P.PolynomialDesc)
      (**
         Extension.
         Linear polynomials over Coefficients.
         They have the external representation Sum ci*varName_varIndex with varIndex ranging from 0 to MAX(Length).
         The coefficients are ordered with decreasing varindex: n < m ⇒ a.VarIndex(n) > a.VarIndex(m).
         varIndex = -1 represents 1 (i.e. no variable).
      **)
      END;

      (* Polynomials *)

   PROCEDURE Init* (p: Polynomial; co: A.Coefficient; varIndex: Length; extlvl: ExtLvl);
   (**
      Initializes the Polynomial p with coefficient co and variable varIndex.
      varIndex = -1 represents 1 (i.e. no variable).

      Pre
      ~(co IS P.Polynomial) OR (co(P.Polynomial).ExtLvl() < extlvl) 20
      extlvl > 0 21
      varIndex >= -1 22

      Post
      result.ExtLvl() = extlvl
   **)
   END Init;

   PROCEDURE (a: Polynomial) New* (co: A.Coefficient; varIndex: Length): Polynomial;
   (**
      Creates a new Polynomial with coefficient co and variable varIndex and same extensionlevel as a.
      varIndex = -1 represents 1 (i.e. no variable).

      Pre
      ~(co IS P.Polynomial) OR (co(P.Polynomial).ExtLvl() < extlvl) 20
      varIndex >= -1 22

      Post
      result.ExtLvl() = a.ExtLvl()

      Except for performace, equivalent to:
         VAR c: Polynomial;
         NEW(c); InitPolynomial(c, co, varIndex, a.ExtLvl());
         RETURN c

      See also: InitPolynomial.
   **)
   END New;

   PROCEDURE (a: Polynomial) NewVar* (varIndex: Length): Polynomial;
   (**
      Creates a new Polynomial with coefficient 1 and variable varIndex and same extensionlevel as a.
      varIndex = -1 represents 1 (i.e. no variable).

      Pre
      varIndex >= -1 22

      Post
      result.ExtLvl() = a.ExtLvl()

      Except for performace, equivalent to:
         VAR c: Polynomial;
         NEW(c); InitPolynomial(c, a.Coeff(0).One(), varIndex, a.ExtLvl());
         RETURN c

      See also: InitPolynomial.
   **)
   END NewVar;

   PROCEDURE (a: Polynomial) IsZero* (): BOOLEAN;
   PROCEDURE (a: Polynomial) IsOne* (): BOOLEAN;
   PROCEDURE (a: Polynomial) IsNegOne* (): BOOLEAN;
```

```
PROCEDURE (a: Polynomial) IsAbsOne* (): BOOLEAN;
PROCEDURE (a: Polynomial) IsNeg* (): BOOLEAN;
PROCEDURE (a: Polynomial) Eql* (b: A.Object): BOOLEAN;
PROCEDURE (a: Polynomial) Zero* (): A.Object;
PROCEDURE (a: Polynomial) One* (): A.Object;
PROCEDURE (a: Polynomial) NegOne* (): A.Object;
PROCEDURE (a: Polynomial) Neg* (): A.Object;
PROCEDURE (a: Polynomial) Inv* (): A.Object;
PROCEDURE (a: Polynomial) Add* (b: A.Object): A.Object;
PROCEDURE (a: Polynomial) Mul* (b: A.Object): A.Object;
PROCEDURE (a: Polynomial) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): A.Object);
PROCEDURE (a: Polynomial) Write* (VAR f(*inout*): F.Formatter);
PROCEDURE (a: Polynomial) Map* (f: A.MapFunc; VAR param(*inout*): A.Param): P.Polynomial;
PROCEDURE (a: Polynomial) ExtLvl* (): ExtLvl;
PROCEDURE (a: Polynomial) Length* (): Length;

PROCEDURE (a: Polynomial) Coeff* (n: Length): A.Coefficient;
(**
   Extension.

   Returns the nth coefficient.
   The coefficients are ordered with decreasing varindex: n < m ⇒ a.VarIndex(n) > a.VarIndex(m).
   i.e.: the leading coefficient a.Coeff(0) returns the coefficient with the highest varindex.

   See also: VarIndex.

   Pre
   a.IsZero() OR 0 <= n < a.Length() 20

   Post
   a.IsZero()
      a.Coeff(0).IsZero()
   ~a.IsZero()
      ~a.Coeff(n).IsZero() for 0 <= n < a.Length()
**)
END Coeff;

PROCEDURE (a: Polynomial) VarIndex* (n: Length): Length;
(**
   Returns the nth varindex.
   The coefficients are ordered with decreasing varindex: n < m ⇒ a.VarIndex(n) > a.VarIndex(m).
   i.e.: the leading varindex a.VarIndex(0) returns the coefficient with the highest varindex.
   varIndex = -1 represents 1 (i.e. no variable).

   See also: Coeff.

   Pre
   a.IsZero() OR 0 <= n < a.Length() 20

   Post
   a.IsZero()
      a.VarIndex(0) = -1
**)
END VarIndex;

PROCEDURE (a: Polynomial) VarPos* (n: Length): Length;
(**
   Returns the position of the term with varindex n.
   If the variable with index n is not in a (i.e. its coefficient is zero) then result < 0.

   Post
   (result < 0) OR (a.VarIndex(result) = n) 60
**)
END VarPos;

PROCEDURE (a: Polynomial) Eval* (VAR values(*in*): ARRAY OF A.Coefficient): A.Coefficient;
(**
   Evaluates polynomial a at values.
   VAR of values just for performance.

   result = Sum(0 <= i < a.Length(); a.Coeff(i).Mul(values[a.VarIndex(i)])).
**)
END Eval;

PROCEDURE Red* (a: Polynomial; ac: A.Coefficient; n: Length; b: Polynomial; bc: A.Coefficient): Polynomial;
(**
   result = a*ac - b*bc.

   Pre
   ~a.IsZero() 20
   ~ac.IsZero() 21
   ~b.IsZero() 22
   ~bc.IsZero() 23
   0 <= n < a.pol.Length() 24
   a.VarIndx(n) = b.VarIndx(0) 25
```

```
    a.Coeff(n).Mul(ac).Eql(b.Coeff(0).Mul(bc)) 26

    Post
    result.Eql(a.Mul(ac).Sub(b.Mul(bc))) 60
  **)
  END Red;

END Linear.
```

```
MODULE PowerProducts;
(**
   Powerproducts for multivariate polynomials and Groebner Bases.

   Author: Gerald Slacik (gs)
*)

   IMPORT P := Polynomials, A := Algebraic, F := Formatters;

   TYPE
      ExtLvl* = P.ExtLvl; (** Extension level > 0 **)
      Degree* = INTEGER;
      PowerProduct* = POINTER TO PowerProductDesc;
      PowerProductDesc* = RECORD
      (**
         Abstract.
         Powerproducts for multivariate polynomials and Groebner Bases.
         They have an admissible ordering (for Groebner Bases) expressed by Lss.
         All binary operations (i.e Eql, Lss, IsMultipleOf, ProdCrit, Mul, IDiv, Gcd, Lcm) are covariant.
         They have the following preconditions:
         DynamicType(a) = DynamicType(b) 20
         a.ExtLvl() = b.ExtLvl() 21
      **)
      END;

   VAR
      (** modifieable instead of read only for derived classes **)
      maxDeg*: LONGINT; (** maximum degree of a powerproduct **)

   PROCEDURE InitProfiling*;
   (** Set maxDeg to 0 **)
   END InitProfiling;

   PROCEDURE (a: PowerProduct) IsOne* (): BOOLEAN;
   (**
      Abstract method. Must be overwritten.

      result = (a = 1).
   **)
   END IsOne;

   PROCEDURE (a: PowerProduct) One* (): PowerProduct;
   (**
      Abstract method. Must be overwritten.

      result = 1.
   **)
   END One;

   PROCEDURE (a: PowerProduct) Eql* (b: PowerProduct): BOOLEAN;
   (**
      Abstract method. Must be overwritten.
      Eql is covariant.

      result = (a = b).

      Pre
      DynamicType(a) = DynamicType(b) 20
      a.ExtLvl() = b.ExtLvl() 21
   **)
   END Eql;

   PROCEDURE (a: PowerProduct) Lss* (b: PowerProduct): BOOLEAN;
   (**
      Abstract method. Must be overwritten.
      Lss is an admissible ordering for Groebner Bases.
      Lss is covariant.

      result = (a < b).

      Pre
      DynamicType(a) = DynamicType(b) 20
      a.ExtLvl() = b.ExtLvl() 21
   **)
   END Lss;

   PROCEDURE (a: PowerProduct) TotDeg* (): Degree;
   (**
      Abstract method. Must be overwritten.
      It returns the total degree.
   **)
   END TotDeg;

   PROCEDURE (a: PowerProduct) IsMultipleOf* (b: PowerProduct): BOOLEAN;
   (**
```

```
    Abstract method. Must be overwritten.
    IsMultipleOf is covariant.

    result = (there exits c s.t. a = b * c).

    Pre
    DynamicType(a) = DynamicType(b) 20
    a.ExtLvl() = b.ExtLvl() 21
**)
END IsMultipleOf;

PROCEDURE (a: PowerProduct) ProdCrit* (b: PowerProduct): BOOLEAN;
(**
    Abstract method. Must be overwritten.
    Product criterion for Groebner basis computation.
    ProdCrit is covariant.

    result = (a * b = a.Lcm(b)).

    Pre
    DynamicType(a) = DynamicType(b) 20
    a.ExtLvl() = b.ExtLvl() 21
**)
END ProdCrit;

PROCEDURE (a: PowerProduct) Mul* (b: PowerProduct): PowerProduct;
(**
    Abstract method. Must be overwritten.
    Mul is covariant.

    result = a * b.

    Pre
    DynamicType(a) = DynamicType(b) 20
    a.ExtLvl() = b.ExtLvl() 21
**)
END Mul;

PROCEDURE (a: PowerProduct) Pow* (exp: INTEGER): PowerProduct;
(**
    Abstract method. Must be overwritten.

    result = a ^ exp.

    Pre
    exp >= 0 20
**)
END Pow;

PROCEDURE (a: PowerProduct) IDiv* (b: PowerProduct): PowerProduct;
(**
    Abstract method. Must be overwritten.
    IDiv is covariant.

    result = a / b.

    Pre
    DynamicType(a) = DynamicType(b 20
    a.ExtLvl() = b.ExtLvl() 21
    a.IsMultipleOf(b) 22
**)
END IDiv;

PROCEDURE (a: PowerProduct) Gcd* (b: PowerProduct): PowerProduct;
(**
    Abstract method. Must be overwritten.
    Gcd is covariant.

    result = greatest common divisor of a and b.

    Pre
    DynamicType(a) = DynamicType(b) 20
    a.ExtLvl() = b.ExtLvl() 21
**)
END Gcd;

PROCEDURE (a: PowerProduct) Lcm* (b: PowerProduct): PowerProduct;
(**
    Abstract method. Must be overwritten.
    Lcm is covariant.

    result = least common multiple of a and b.

    Pre
    DynamicType(a) = DynamicType(b) 20
```

```
    a.ExtLvl() = b.ExtLvl() 21
  **)
  END Lcm;

  PROCEDURE (a: PowerProduct) Write* (VAR f(*inout*): F.Formatter);
  (**
    Abstract method. Must be overwritten.
    Writes a textual representation of a with f.
    The name of the ith variable in the extension hierearchy can be written with Ditionary.WriteName(f, a.ExtLvl() + i)
    for 0 <= i < a.ExtLvlInc().

    Pre
    The name must have been registered in the dictionary.

    See also: Var.
  **)
  END Write;

  PROCEDURE (p: PowerProduct) WriteF* (VAR f(*inout*): F.Formatter; str: ARRAY OF CHAR);
  (**
    Write formatted power product p.
    Writes the comment str with the following transformations:
      \! ⇒ flush (see Flush)
      \n ⇒ newline (see WriteLn)
      \r ⇒ carriage return (see WriteCr)
      \t ⇒ tab (see WriteTab)
      \@ ⇒ p
      \x ⇒ x where x is in CHAR \ n, t, @, OX, especially \\ ⇒ \.

    Pre
      str does not end with '\' 20
  **)
  END WriteF;

  PROCEDURE (a: PowerProduct) ExtLvl* (): ExtLvl;
  (**
    Abstract method. Must be overwritten.
    Extension level.
    Since powerproducts typically represent more than one variable they provide more than one level of extension.
    ExtLvl return the minimal extension level and ExtLvlInc returns the number of extensions.

    Post
    result > 0

    See also: ExtLvlInc, Polynomials.ExtLvl.
  **)
  END ExtLvl;

  PROCEDURE (a: PowerProduct) ExtLvlInc* (): ExtLvl;
  (**
    Abstract method. Must be overwritten.
    Extension level increment.
    Since powerproducts typically represent more than one variable they provide more than one level of extension.
    ExtLvl return the minimal extension level and ExtLvlInc returns the number of extensions (eg. for Q[x,y,z] ExtLvlInc rightarrow 3).

    Post
    result > 0

    See also: ExtLvl.
  **)
  END ExtLvlInc;

  PROCEDURE (a: PowerProduct) Var* (i: ExtLvl): PowerProduct;
  (**
    Abstract method. Must be overwritten.
    Returns a powerproduct representing the ith variable in the extension hierearchy
    (e.g. p.Var(2) → z for p element of Q[x,y,z] with x < y < z).
    This is used by multivariate polynomials to register name in the dictionary.

    Pre
    0 <= i < a.ExtLvlInc() 20

    See also: Write.
  **)
  END Var;

END PowerProducts.
```

```
MODULE SmallPowerProducts;
(**
   Powerproducts for multivariate polynomials and Groebner Bases
   with small degrees (<=MAX(SHORTINT)) and small # of vars (<= maxDim).

   Author: Gerald Slacik (gs)
*)

   IMPORT PP := PowerProducts, P := Polynomials, A := Algebraic, D := Dictionary, F := Formatters;

   CONST
      maxDim* = 7;

   TYPE
      ExtLvl* = PP.ExtLvl; (** Extension level > 0 **)
      PowerProduct* = POINTER TO PowerProductDesc;
      PowerProductDesc* = RECORD (PP.PowerProductDesc)
      (**
         Abstract.
         Common base for powerproducts with specific ordering.

         See also: TotDeg, PureLex.
      **)
      END;
      TotDeg* = POINTER TO TotDegDesc;
      TotDegDesc* = RECORD (PowerProductDesc)
      (** Powerproducts with total degree ordering. **)
      END;
      PureLex* = POINTER TO PureLexDesc;
      PureLexDesc* = RECORD (PowerProductDesc)
      (** Powerproducts with pure lexicographic ordering. **)
      END;
      RevLex* = POINTER TO RevLexDesc;
      RevLexDesc* = RECORD (PowerProductDesc)
      (** Powerproducts with reverse lexicographic ordering. (as in e.g Maple) **)
      END;

   (* Powerproducts *)

   PROCEDURE (a: PowerProduct) Clone* (): PowerProduct;
   (**
      Abstract method. Must be overwritten.

      Returns a shallow copy of a.
      The result has the same dynamic type as a. The contents of the new object are copied!
      Clone needs to be extended in extensions of Polynomial.
      Extensions should never make a super call.

      Used internally.
   **)
   END Clone;

   PROCEDURE (a: PowerProduct) IsOne* (): BOOLEAN;
   PROCEDURE (a: PowerProduct) One* (): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) Eql* (b: PP.PowerProduct): BOOLEAN;
   PROCEDURE (a: PowerProduct) TotDeg* (): PP.Degree;
   PROCEDURE (a: PowerProduct) IsMultipleOf* (b: PP.PowerProduct): BOOLEAN;
   PROCEDURE (a: PowerProduct) ProdCrit* (b: PP.PowerProduct): BOOLEAN;
   PROCEDURE (a: PowerProduct) Mul* (b: PP.PowerProduct): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) Pow* (exp: INTEGER): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) IDiv* (b: PP.PowerProduct): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) Gcd* (b: PP.PowerProduct): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) Lcm* (b: PP.PowerProduct): PP.PowerProduct;
   PROCEDURE (a: PowerProduct) Write* (VAR f(*inout*): F.Formatter);
   PROCEDURE (a: PowerProduct) ExtLvl* (): ExtLvl;
   PROCEDURE (a: PowerProduct) ExtLvlInc* (): ExtLvl;
   PROCEDURE (a: PowerProduct) Var* (i: ExtLvl): PP.PowerProduct;

   PROCEDURE (a: TotDeg) Clone* (): PowerProduct;
   PROCEDURE (a: TotDeg) Lss* (b: PP.PowerProduct): BOOLEAN;

   PROCEDURE (a: PureLex) Clone* (): PowerProduct;
   PROCEDURE (a: PureLex) Lss* (b: PP.PowerProduct): BOOLEAN;

   PROCEDURE (a: RevLex) Clone* (): PowerProduct;
   PROCEDURE (a: RevLex) Lss* (b: PP.PowerProduct): BOOLEAN;

   PROCEDURE Init* (p: PowerProduct; extlvl: ExtLvl; dim: Dimension);
   (**
      Initializes the PowerProduct p with extension level extlvl and dimesion dim.

      Pre
      0 < extlvl 20
      (dim > 0) & (dim <= maxDim) 21
      extlvl + dim - 1 < D.maxKey 22
```

```
      Post
      result.ExtLvl() = extlvl
      result.ExtLvlInc() = dim
      result.IsOne() 60
   **)
   END Init;

END SmallPowerProducts.
```

```
MODULE Multivariate;
(**
  Multivariate defines multivariate polynomials over coefficients and powerproducts.

  Author: Gerald Slacik (gs)
*)

  IMPORT
    PP := PowerProducts, P := Polynomials, D := Dictionary, A := Algebraic,
    S := Scanners, F := Formatters;

  TYPE
    ExtLvl* = P.ExtLvl; (** Extension level > 0 **)
    Length* = P.Length;
    Polynomial* = POINTER TO PolynomialDesc;
    PolynomialDesc* = RECORD (P.PolynomialDesc)
    (**
      Extension.
      Multivariate polynomials over Coefficients and Powerproducts.
      Polynomials have the external representation Sum ci*ppi.
      The coefficients are ordered with decreasing pp: n < m ⇒ a.PP(n).Lss(a.PP(m)).
    **)
    END;

  (* Polynomials *)

  PROCEDURE Init* (p: Polynomial; co: A.Coefficient; pp: PP.PowerProduct);
  (**
    Initializes the Polynomial p with coefficient co and powerproduct pp.

    Pre
    ~(co IS P.Polynomial) OR (co(P.Polynomial).ExtLvl() < pp.ExtLvl()) 20

    Post
    result.ExtLvl() = pp.ExtLvl()
  **)
  END Init;

  PROCEDURE (a: Polynomial) IsZero* (): BOOLEAN;
  PROCEDURE (a: Polynomial) IsOne* (): BOOLEAN;
  PROCEDURE (a: Polynomial) IsNegOne* (): BOOLEAN;
  PROCEDURE (a: Polynomial) IsAbsOne* (): BOOLEAN;
  PROCEDURE (a: Polynomial) IsNeg* (): BOOLEAN;
  PROCEDURE (a: Polynomial) Eql* (b: A.Object): BOOLEAN;
  PROCEDURE (a: Polynomial) Zero* (): A.Object;
  PROCEDURE (a: Polynomial) One* (): A.Object;
  PROCEDURE (a: Polynomial) NegOne* (): A.Object;
  PROCEDURE (a: Polynomial) Neg* (): A.Object;
  PROCEDURE (a: Polynomial) Inv* (): A.Object;
  PROCEDURE (a: Polynomial) Add* (b: A.Object): A.Object;
  PROCEDURE (a: Polynomial) Mul* (b: A.Object): A.Object;

  PROCEDURE MulPP* (a: Polynomial; pp: PP.PowerProduct): Polynomial;
  (**
    Multiplies a with the powerproduct pp.

    result = (a * pp)
  **)
  END MulPP;

  PROCEDURE MulCo* (a: Polynomial; co: A.Coefficient): Polynomial;
  (**
    Multiplies a with the coefficient co.

    result = (a * co)
  **)
  END MulCo;

  PROCEDURE MulTerm* (a: Polynomial; co: A.Coefficient; pp: PP.PowerProduct): Polynomial;
  (**
    Multiplies a with the term co*pp.

    result = (a * co * pp)
  **)
  END MulTerm;

  PROCEDURE (a: Polynomial) Read* (VAR s(*inout*): S.Scanner; VAR res(*out*): A.Object);
  PROCEDURE (a: Polynomial) Write* (VAR f(*inout*): F.Formatter);

  PROCEDURE (a: Polynomial) Map* (f: A.MapFunc; VAR param(*inout*): A.Param): P.Polynomial;
  (**
    Extension.

    Function f is applied to all coefficients of a.
```

```
    Let a = Sum ci*pi.
    result = Sum f(ci)*pi.
**)
END Map;

PROCEDURE (a: Polynomial) Length* (): Length;
(**
    Extension.

    The number of coefficients # 0. Length(0) = 0.
**)
END Length;

PROCEDURE (a: Polynomial) PP* (n: Length): PP.PowerProduct;
(**
    Returns the nth powerproduct.
    The powerproducts are ordered with decreasing pp: n < m ⇒ a.PP(n).Lss(a.PP(m)).
    a.Coeff(0) returns the leading powerproduct.

    See also: COeff.

    Pre
    a.IsZero() OR 0 <= n < a.Length() 20

    Post
    a.IsZero()
        a.PP(0).IsOne()
**)
END PP;

PROCEDURE (a: Polynomial) Coeff* (n: Length): A.Coefficient;
(**
    Extension.

    Returns the nth coefficient.
    The coefficients are ordered with decreasing pp: n < m ⇒ a.PP(n).Lss(a.PP(m)).
    a.Coeff(0) returns the leading coefficient.

    See also: PP.

    Pre
    a.IsZero() OR 0 <= n < a.Length() 20

    Post
    a.IsZero()
        a.Coeff(0).IsZero()
    ~a.IsZero()
        ~a.Coeff(n).IsZero() for 0 <= n < a.Length()
**)
END Coeff;

PROCEDURE (a: Polynomial) ExtLvl* (): ExtLvl;
PROCEDURE (a: Polynomial) ExtLvlInc* (): ExtLvl;
(**
    Extension level increment.
    Since multivariate polynomials typically represent more than one variable they provide more than one level of extension.
    ExtLvl return the minimal extension level and ExtLvlInc returns the number of extensions (eg. for Q[x,y,z] ExtLvlInc → 3).

    Post
    result > 0

    See also: ExtLvl, PowerProducts.ExtLvlInc.
END ExtLvlInc;

PROCEDURE Red* (a: Polynomial; ac: A.Coefficient; n: Length;
                b: Polynomial; bc: A.Coefficient; bpp: PP.PowerProduct): Polynomial;
(**
    result = a*ac - b*bc*bpp.

    Pre
    ~a.IsZero() 20
    ~ac.IsZero() 21
    ~b.IsZero() 22
    ~bc.IsZero() 23
    0 <= n < a.pol.Length() 24
    a.PP(n).Eql(b.PP(0).Mul(bpp)) 25
    a.Coeff(n).Mul(ac).Eql(b.Coeff(0).Mul(bc)) 26

    Post
    result.Eql(MulCo(a, ac).Sub(MulTerm(b, bc, bpp))) 60
**)
END Red;

PROCEDURE Spol* (a: Polynomial; ac: A.Coefficient; app: PP.PowerProduct; n: Length;
                 b: Polynomial; bc: A.Coefficient; bpp: PP.PowerProduct): Polynomial;
```

```
    (**
        result = a*ac*app - b*bc*bpp.

        Pre
        ~a.IsZero() 20
        ~ac.IsZero() 21
        ~b.IsZero() 22
        ~bc.IsZero() 23
        0 <= n < a.pol.Length() 24
        a.PP(n).Mul(app).Eql(b.PP(0).Mul(bpp)) 25
        a.Coeff(n).Mul(ac).Eql(b.Coeff(0).Mul(bc)) 26

        Post
        result.Eql(MulTerm(a, ac, app).Sub(MulTerm(b, bc, bpp))) 60
    **)
    END Spol;

END Multivariate.
```

```
MODULE Gauss;
(**
   Gauss Elimination for Bases of Linear Polynomials

   Author: Gerald Slacik (gs)
*)

   IMPORT L := Linear, A := Algebraic, S := Scanners, F := Formatters;

   TYPE
      Size* = LONGINT; (** type returned by Basis.Size(). **)
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD
      (**
         Abstract.
         Polynomials for bases (i.e. sytem of linear equations).
      **)
         pol-: L.Polynomial; (** Linear polynomial **)
      END;

      Basis* = POINTER TO BasisDesc;
      BasisDesc* = RECORD
      (**
         Basis of linear polynomials.
         Used to solve systems of linear equations by Gauss elimination.
      **)
      END;

      Iterator* = RECORD
      (** Used to perform some action with every element of a basis. **)
      END;

      Write* = RECORD (F.Write)
      (** Procedure Object for WriteFormat. **)
      END;

      (* Polynomials *)

   PROCEDURE Init* (p: Polynomial; a: L.Polynomial);
   (** Initializes the GaussPolynomial p with the linear polynomial a. **)
   END Init;

   PROCEDURE (p: Polynomial) Clone* (): Polynomial;
   (**
      Abstract method. Must be overwritten.

      Returns a shallow copy of p.
      The result has the same dynamic type as p. The contents of the new object are copied!
      Clone is used before the object's contents is copied.
      Clone needs to be extended in extensions of Polynomial.
      Extensions should never make a super call.
   **)
   END Clone;

   PROCEDURE (p: Polynomial) CopyParts* ;
   (**
      Abstract method. Must be overwritten.

      Copies relevant parts.
      Extensions should always make a super call.
   **)
   END CopyParts;

   PROCEDURE (p: Polynomial) Norm* ;
   (**
      Abstract method. Must be overwritten.

      Makes p somehow normalized (e.g. by making it primitive or by dividing it by the leading coefficient).

      Pre
      ~p.pol.IsZero() 20
   **)
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: L.Length; b: Polynomial);
   (**
      Abstract method. Must be overwritten.

      Reduces a with b by eliminating the variable at position n.
      Red is used to eliminate the variable with index a.VarIndex(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
```

```
    0 <= n < a.Length() 23
    a.pol.VarIndex(n) = b.pol.VarIndex(0) 24
    b is normed 25

    Post
    a.pol.VarPos(b.pol.VarIndex(0)) < 0 60

    Except for performace, equivalent to:
        a.pol := a.pol.Sub(b.pol.Mul(a.pol.Coeff(n).Div(b.pol.Coeff(0))))
**)
END Red;

PROCEDURE (p: Polynomial) Scale* (co: A.Coefficient): A.Coefficient;
(**
    Abstract method. Must be overwritten.

    Scales the coeffcient co by the leading coefficient of p.
    Scale is used to correct the results of back substitution.

    Pre
    ~p.pol.IsZero() 20
    p is normed 21

    Except for performace, equivalent to:
        RETURN co.Div(p.GetCoeff(0).Neg())
**)
END Scale;

(* Bases *)

PROCEDURE (b: Basis) Adjoin* (p: Polynomial);
(** Adds the polynomial p to the basis b. If p = 0 then b is not changed. **)
END Adjoin;

PROCEDURE (b: Basis) Triangularize* (VAR unsolveable: BOOLEAN);
(**
    Does a simple Gauss elimination on the basis b. No fancy numeric behavior involved.
    Usage: call basis.Triangularize first and then basis.Solve or call basis.Triangularize twice.

    The basis b itself is modified! If you need the original basis b afterwards you should copy it first by:
    org := b.Clone; org.CopyParts;
**)
END Triangularize;

PROCEDURE (b: Basis) Solve* (VAR soln(*inout*): ARRAY OF A.Coefficient);
(**
    Does a simple Gauss back substituion on the basis b. No fancy numeric behavior involved.
    Usage: call basis.Triangularize first and then basis.Solve;
    Free vars are initialized with their corresponding value in soln.
**)
END Solve;

PROCEDURE (b: Basis) Clone* (): Basis;
(**
    Returns a shallow copy of b.
    The result has the same dynamic type as b. The contents of the new object are copied!
    Clone is used before the object's contents is copied.
    Clone needs to be extended in extensions of Basis.
    Extensions should never make a super call.
**)
END Clone;

PROCEDURE (b: Basis) CopyParts* ;
(**
    Copies relevant parts.
    Extensions should always make a super call.
**)
END CopyParts;

PROCEDURE (basis: Basis) Read* (VAR s(*inout*): S.Scanner; VAR b(*out*): Basis);
(**
    Reads a basis of the form:
    Basi = "{" [Element { "," Element}] "}".
    Elemen = Algebraic.Expression
    The type of polynomial read is determined by a polynomial in basis.

    Pre
    basis.Size() > 20
**)
END Read;

PROCEDURE (b: Basis) Write* (VAR f(*inout*): F.Formatter);
(** Writes a textual representation of a with f. **)
END Write;
```

```
  PROCEDURE (VAR w(*inout*): Write) Write* (VAR f(*inout*): F.Formatter);
  END Write;

  PROCEDURE (b: Basis) WriteF* (VAR f(*inout*): F.Formatter; str: ARRAY OF CHAR);
  (**
    Write formatted basis b.
    See also: WriteFormat.
  **)
  END WriteF;

  PROCEDURE (b: Basis) Normalize*;
  (** Normalizes the elements of basis b. **)
  END Normalize;

  PROCEDURE (b: Basis) Size* (): Size;
  (** Returns the number of elements in basis b. **)
  END Size;

  PROCEDURE InitBasis* (b: Basis);
  (**
    Initializes basis with {}.

    Post
    b.Size() = 0
  **)
  END InitBasis;

  PROCEDURE (VAR it(*out*): Iterator) SetTo* (b: Basis);
  (**
    Set it to the first element of basis.

    Usage:
      Var it: Iterator; p: Polynomial;
      it.SetTo(basis); p := it.Next();
      WHILE p # NIL DO ... p := it.Next() END;

    See also: Next.
  **)
  END SetTo;

  PROCEDURE (VAR it(*inout*): Iterator) Next* (): Polynomial;
  (**
    Returns next (first after SetTo) element of basis or NIl if there are no (more) elements in basis.

    Usage:
      Var it: Iterator; p: Polynomial;
      it.SetTo(basis); p := it.Next();
      WHILE p # NIL DO ... p := it.Next() END;

    See also: SetTo.
  **)
  END Next;

END Gauss.
```

```
MODULE GaussPolys;
(**
   Gauss Polynomials with rational Coefficients.

   Author: Gerald Slacik (gs)
*)

   IMPORT G := Gauss, L := Linear, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (** Polynomials with rational coefficients for bases (i.e. sytem of linear equations). **)
      END;

      (* Polynomials *)

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites Gauss.Polynomial.Norm.

      Normalizes by dividing it by the leading coefficient.

      Pre
      ~a.pol.IsZero() 20
   **)
   END Norm;

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   PROCEDURE (a: Polynomial) Red* (n: L.Length; b: G.Polynomial);
   PROCEDURE (a: Polynomial) Scale* (co: A.Coefficient): A.Coefficient;

END GaussPolys.
```

```
MODULE IntGaussPolys;
(**
   Gauss Polynomials with integral Coefficients.

   Author: Gerald Slacik (gs)
*)

   IMPORT
      G := Gauss, P := Polynomials, L := Linear, IQ := IntegralsAndQuotients, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (** Polynomials with integral coefficients for bases (i.e. sytem of linear equations). **)
      END;

      (* Polynomials *)

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites Gauss.Polynomial.Norm.

      Normalizes a by making it primitive.

      Pre
      ~a.pol.IsZero() 20
   **)
   END Norm;

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   PROCEDURE (a: Polynomial) Red* (n: L.Length; b: G.Polynomial);
   PROCEDURE (a: Polynomial) Scale* (co: A.Coefficient): A.Coefficient;

END IntGaussPolys.
```

```
MODULE Grobner;
(**
   GroebnerBases of Polynomials over a field.

   Author: Gerald Slacik (gs)
*)

   IMPORT P := Multivariate, PP := PowerProducts, A := Algebraic, S := Scanners, F := Formatters;

   CONST
      SetSize = MAX(SET) + 1;

   TYPE
      Size* = INTEGER; (** Type for the number of elements in a basis. **)
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD
      (**
         Abstract.
         Polynomial for (Groebner)bases.
         Used to solve systems of equations by Groebner bases computation.
      **)
         pol-: P.Polynomial; (** Multivariate polynomial **)
         lpp-: PP.PowerProduct; (** leading power product **)
         nxt: Polynomial;
         num: Size (* id number used for chain criterion *)
      END;

      Set = POINTER TO ARRAY OF SET;
      (* BitSet of arbitray Length *)

      Spoly = POINTER TO SpolyDesc;
      SpolyDesc = RECORD
      (*
         Spolys which have to be considered; ordered by ascending lcm;
         Circular list of Spolys with dummy elem as head.
      *)
         nxt: Spoly;
         lcm: PP.PowerProduct;
         p1, p2: Polynomial;
         n1, n2: Size (* n1 > n2 *)
      END;


      Basis* = POINTER TO BasisDesc;
      BasisDesc* = RECORD
      (**
         Basis of multivariate polynomials.
         Used to solve systems of equations by Groebner bases computation.
      **)
         first, last: Polynomial; (* start and end of doubly linked list *)

         polyNum: Size; (* nonrecycled number of next poly in basis *)
         size: Size; (* number of polys in basis *)
         spoly: Spoly; (* Dummy elem as head of a circular list of Spolys which have to be considered; ordered by ascending lcm *)
         invalid: Set; (* set of of polys no longer valid *)
         done: Set (* set of of Spolys already considered; used to check for chain criterion *)
      END;

      MapFunc* = PROCEDURE (p: Polynomial; VAR param(*inout*): A.Param): Polynomial;
      (**
         Function type used by Map.
         Function f is applied to all polynomials of a basis.
         See also: Map.
      **)

      Iterator* = RECORD (* ForAll is not suited for building pairs needed in syzygy *)
      (** Used to perform some action with every element of a basis. **)
         nxt: Polynomial
      END;

      Write* = RECORD (F.Write)
      (** Procedure Object for WriteFormat. **)
         b: Basis
      END;

   (* Polynomials *)

   PROCEDURE Init* (p: Polynomial; a: P.Polynomial);
   (** Initializes the GrobnerPolynomial p with the multivariate polynomial a. **)
   BEGIN
      p.lpp := a.PP(0); p.pol := a
   END Init;

   PROCEDURE (p: Polynomial) Clone* (): Polynomial;
   (**
```

```
    Abstract method. Must be overwritten.

    Returns a shallow copy of p.
    The result has the same dynamic type as p. The contents of the new object are copied!
    Clone is used before the object's contents is copied.
    Clone needs to be extended in extensions of Polynomial.
    Extensions should never make a super call.
**)
BEGIN
    HALT(A.abstractMethod)
END Clone;

PROCEDURE (p: Polynomial) CopyParts* ;
(**
    Abstract method. Must be overwritten.

    Copies relevant parts.
    Extensions should always make a super call.
**)
BEGIN
END CopyParts;

PROCEDURE (p: Polynomial) Norm* ;
(**
    Abstract method. Must be overwritten.

    Makes p somehow normalized (e.g. by making it primitive or by dividing it by the leading coefficient).

    Pre
    ~p.pol.IsZero() 20
**)
BEGIN
    HALT(A.abstractMethod)
END Norm;

PROCEDURE (a: Polynomial) Red* (n: P.Length; b: Polynomial);
(**
    Abstract method. Must be overwritten.

    Reduces a with b by eliminating the power product at position n.
    Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

    Pre
    b IS SameDynamicTypeAs(a) 20
    ~a.pol.IsZero() 21
    ~b.pol.IsZero() 22
    0 <= n < a.pol.Length( 23
    a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
    b is normed 25

    Except for performace, equivalent to:
        a.pol := a.pol.Sub(b.pol.MulTerm(a.pol.Coeff(n).Div(b.pol.Coeff(0), a.pol.PP(n).IDiv(b.pol.PP(0))))
**)
BEGIN
    HALT(A.abstractMethod)
END Red;

PROCEDURE (a: Polynomial) Spol* (b: Polynomial; lcm: PP.PowerProduct): Polynomial;
(**
    Abstract method. Must be overwritten.

    Returns the Spolynomial of a and b.

    Pre
    b IS SameDynamicTypeAs(a) 20
    ~a.pol.IsZero() 21
    ~b.pol.IsZero() 22
    a is normed 23
    b is normed 24
    lcm.Eql(a.lpp.Lcm(b.lpp)) 26


    Post
    result IS SameDynamicTypeAs(a) 60

    Except for performace, equivalent to:
        a.pol := a.pol.MulTerm(b.pol.Coeff(n), lcm.IDiv(b.lpp)).Sub(b.pol.MulTerm(a.pol.Coeff(0)), lcm.IDiv(b.lpp))
**)
BEGIN
    HALT(A.abstractMethod)
END Spol;

PROCEDURE (p: Polynomial) Reduce* (n: P.Length; b: Basis);
(** reduces p with regard to the basis b starting at the nth term (n = 0 .. leading term) **)
    VAR pp: PP.PowerProduct; q: Polynomial; l: P.Length;
```

```
BEGIN
   l := p.pol.Length();
   IF b.first # NIL THEN
      WHILE n < l DO
            pp := p.pol.PP(n); q := b.first;
            REPEAT
               IF pp.Lss(q.lpp) THEN
                  (* p < q ⇒ ~ p isMultipleOf q; basis is ordered by ascending lpp *)
                  IF q = b.first THEN n := l (* exit while *)END;
                  q := NIL (* exit repeat *)
               ELSIF pp.IsMultipleOf(q.lpp) THEN
                  p.Red(n, q);
                  l := p.pol.Length(); DEC(n);
                  q := NIL (* exit repeat *)
               ELSE
                  q := q.nxt
               END
            UNTIL q = NIL;
            INC(n)
      END
   END;
   IF l > 0 THEN p.Norm END
END Reduce;

(* Pairs *)

PROCEDURE Check (set: Set; i, j: Size);
(*
   Marks Spoly(basis.elem[i], basis.elem[j]) as considered.
   Needed for chain criterion.

   Pre
   i > j > 0 20
*)
   VAR k: LONGINT;
BEGIN
   k := (LONG(i - 1) * (i - 2) DIV 2) + j - 1;
   INCL(set[k DIV SetSize], k)
END Check ;

PROCEDURE Checked (set: Set; i, j: Size): BOOLEAN;
(*
   true if Spoly(basis.elem[i], basis.elem[j]) already considered.
   Needed for chain criterion.

   Pre
   (i > 0) & (j > 0) 20
*)
   VAR k: LONGINT; h: Size;
BEGIN
   IF i < j THEN h := i; i := j; j := h END;
   k := (LONG(i - 1) * (i - 2) DIV 2) + j - 1;
   RETURN k IN set[k DIV SetSize]
END Checked;

(* Spolys *)

PROCEDURE AddSpoly (b: Basis; p, q: Polynomial; VAR hint(*inout*): Spoly);
(*
   Adds Spoly(p, q) to the list of Spolys (with head minSpoly).
   hint is the most recently inserted elem.

   Pre
   p.num > q.num 20
   q.num > 0 21
*)
   VAR s, t, sp: Spoly; lcm: PP.PowerProduct;
BEGIN
   IF p.lpp.ProdCrit(q.lpp) THEN
      Check(b.done, p.num, q.num)
   ELSE
      NEW(sp); lcm := p.lpp.Lcm(q.lpp); sp.lcm := lcm; sp.p1 := p; sp.p2 := q; sp.n1 := p.num; sp.n2 := q.num;
      (* check if insertion can start after the most recently inserted elem *)
      b.spoly.lcm := lcm; (* stop mark; set it here before we use hint, because hint = spoly at least at the first call *)
      IF lcm.Lss(hint.lcm) THEN s := b.spoly (* No, so start at the beginning *) ELSE s := hint END;
      t := s.nxt;
      WHILE t.lcm.Lss(lcm) DO s := t; t := t.nxt END;
      s.nxt := sp; sp.nxt := t; hint := sp;
      b.spoly.lcm := NIL (* free stop mark *)
   END
END AddSpoly;

PROCEDURE RemoveSpolys (b: Basis);
(*
   Remove all invalid spolys from list of Spolys (with head minSpoly).
```

```
   Invalid spolys are marked in invalid.
*)
   VAR s, t: Spoly;
BEGIN
   s := b.spoly; s.n1 := b.polyNum; s.n2 := b.polyNum; (* stop mark for while loop *)
   REPEAT
      t := s.nxt;
      WHILE (t.n1 IN b.invalid[t.n1 DIV SetSize]) OR (t.n2 IN b.invalid[t.n2 DIV SetSize]) DOt := t.nxt END;
      s.nxt := t; s := t
   UNTIL s = b.spoly
END RemoveSpolys;

PROCEDURE MinSpoly (b: Basis): Polynomial;
(* finds and calculates the spoly f with minimal lcm in list of Spolys (with head minSpoly) *)
   VAR n1, n2: Size; p: Polynomial; s: Spoly;
BEGIN
   s := b.spoly.nxt;
   WHILE s # b.spoly DO
      (* check for chain criterion *)
      n1 := s.n1; n2 := s.n2;
      p := b.first; n1 := s.n1; n2 := s.n2;
      WHILE (p # NIL) &
              ~((n1 # p.num) & (n2 # p.num) &
                 Checked(b.done, n1, p.num) & Checked(b.done, n2, p.num) &
                 s.lcm.IsMultipleOf(p.lpp)) DO
            p := p.nxt
      END;
      Check(b.done, n1, n2);
      IF p = NIL THEN b.spoly.nxt := s.nxt; RETURN s.p1.Spol(s.p2, s.lcm) END;
      s := s.nxt
   END;
   b.spoly.nxt := b.spoly; (* free *)
   RETURN NIL
END MinSpoly;

(* Bases *)

PROCEDURE (b: Basis) Adjoin* (p: Polynomial);
(**
   Adds the polynomial p to the basis b if p # 0.

   Pre
   p has been intialized with InitPolynomial 20
**)
   VAR lpp: PP.PowerProduct; q, r: Polynomial;
BEGIN
   IF ~p.pol.IsZero() THEN
      IF b.first = NIL THEN
            b.first := p; b.last := p; p.nxt := NIL
      ELSE
            lpp := p.lpp;
            IF ~lpp.Lss(b.last.lpp) THEN
               b.last.nxt := p; b.last := p; p.nxt := NIL
            ELSE
               q := b.first;
               IF lpp.Lss(q.lpp) THEN
                  b.first := p
               ELSE
                  REPEAT r := q; q := q.nxt UNTIL lpp.Lss(q.lpp); (* during gb computation lpp=q.lpp can't happen *)
                  r.nxt := p
               END;
               p.nxt := q
            END
      END;
      INC(b.size)
   END
END Adjoin;

PROCEDURE ReduceAll (b: Basis; p: Polynomial; VAR added(*out*): BOOLEAN; VAR removed(*out*): BOOLEAN);
(*
   Reduces all polys in basis b with p.
   p may be a list of polys which will all be adjoined to b if not reducible to 0;

   added is the number of elements added to the basis (some of them may have been removed before).
   removed is the number of elements removed from the basis (some of them may have been added before).
   For all p added to the basis p.num is reset to 0. All spolys 'after' p have to be considered again.
*)
   VAR r: Polynomial; lpp: PP.PowerProduct;
BEGIN
   added := FALSE; removed := FALSE;
   REPEAT
      lpp := p.lpp; p.Reduce(0, b);
      IF (p.num > 0) & (lpp # p.lpp) THEN
            (* if lpp of p changes all spolys with p have to be checked again *)
            INCL(b.invalid[p.num DIV SetSize], p.num); removed := TRUE; p.num := 0
```

```
        END;
        r := p.nxt;
        IF ~p.pol.IsZero() THEN
                b.Adjoin(p); added := TRUE;
                IF p.nxt # NIL THEN
                    (* all polys >= p may be reduced; append them to the beginning of r *)
                    b.last.nxt := r; b.last := p; r := p.nxt; p.nxt := NIL
                END
        END;
        p := r
    UNTIL p = NIL
END ReduceAll;

PROCEDURE InitGB (b: Basis);
(*
    prepare basis for groebner basis computation; create list of spolys.

    Pre
    b # {} 20
*)
    VAR p, q: Polynomial; i: Size; hint: Spoly; added, removed: BOOLEAN;
BEGIN
    NEW(b.spoly); b.spoly.nxt := b.spoly; (* create dummy elem as head *)
    p := b.first; b.first := NIL; b.last := NIL; b.size := 0;
    (* we don't need invalid & done here because all p.num = 0 *)
    ReduceAll(b, p, added, removed);
    p := b.first; i := 1;
    REPEAT p.num := i; INC(i); p := p.nxt UNTIL p = NIL;
    b.size := i-1;
    IF i > 2 THEN
        b.polyNum := SHORT(i);
        (* 127,4,251 255,8,1013 511,16,4073 1023,32,16337 *)
        i := ((i + 127) DIV 128) * (128 DIV SetSize);
        NEW(b.invalid, i);
        i := i * SetSize;
        NEW(b.done, ((i * (i - 1)) DIV 2 + SetSize - 1) DIV SetSize);
        p := b.first.nxt; hint := b.spoly;
        REPEAT
                q := b.first;
                REPEAT AddSpoly(b, p, q, hint); q := q.nxt UNTIL q = p;
                p := p.nxt
        UNTIL p = NIL
    END
END InitGB;

PROCEDURE Add (b: Basis; p: Polynomial);
(*
    Add p to b.
    b is reducedAll by p. All invalid spolys are removed and add all new all spolys are added.

    Pre
    b.first # {} 21
    p.nxt = NIL 22
    p has been intialized with InitPolynomial 20
*)
    VAR q: Polynomial; i: Size; added, removed: BOOLEAN; hint: Spoly;
    PROCEDURE IncSetSizes(min: Size);
        VAR i: LONGINT; set: Set;
    BEGIN
        min := ((min + 127) DIV 128) * (128 DIV SetSize);
        NEW(set, min);
        i := LEN(b.invalid↑);
        REPEAT DEC(i); set[i] := b.invalid[i] UNTIL i = 0;
        b.invalid := set;
        i := min * SetSize; i := ((i * (i - 1)) DIV 2 + SetSize - 1) DIV SetSize;
        NEW(b.done, i);
        i := LEN(b.done↑);
        REPEAT DEC(i); set[i] := b.done[i] UNTIL i = 0;
        b.done := set;
    END IncSetSizes;
BEGIN
    ReduceAll(b, p, added, removed);
    IF added THEN
        IF removed THEN RemoveSpolys(b) END;
        (* add all new spolys *)
        i := b.polyNum; p := b.first; b.size := 0;
        REPEAT INC(b.size); IF p.num = 0 THEN p.num := b.polyNum; INC(b.polyNum) END; p := p.nxt UNTIL p = NIL;
        IF b.polyNum > LEN(b.invalid↑) * SetSize THEN IncSetSizes(b.polyNum) END;
        p := b.first; hint := b.spoly;
        REPEAT
                IF p.num >= i THEN
                    q := b.first;
                    REPEAT
                        IF p.num > q.num THEN AddSpoly(b, p, q, hint) END;
                        q := q.nxt
```

```
            UNTIL q = NIL
          END;
          p := p.nxt
      UNTIL p = NIL
   END
END Add;

PROCEDURE (b: Basis) GrobnerBasis* ;
(**
   Computes the Groebnerbasis of basis b.
   The basis b itself is modified! If you need the original basis b afterwards you should copy it first by:
   org := b.Clone; org.CopyParts;
**)
   VAR p: Polynomial;
BEGIN
   IF b.size > 0 THEN
      InitGB(b);
      LOOP
            p := MinSpoly(b);
            IF p = NIL THEN EXIT END;
            Add(b, p)
      END;
      b.spoly := NIL; b.invalid := NIL; b.done := NIL (* free *)
   END
END GrobnerBasis;

PROCEDURE (b: Basis) Size* (): Size;
(** Returns the number of elements in basis b. **)
BEGIN
   RETURN b.size
END Size;

PROCEDURE InitBasis* (b: Basis);
(**
   Initializes basis with {}.

   Post
   b.Size() = 0 60
**)

BEGIN
   b.first := NIL; b.last := NIL; b.size := 0
END InitBasis;

PROCEDURE (b: Basis) Map* (f: MapFunc; VAR param(*inout*): A.Param): Basis;
(** Function f is applied to all polynomials of basis b. **)
   VAR new: Basis; p: Polynomial;
BEGIN
   NEW(new); p := b.first;
   WHILE p # NIL DOnew.Adjoin(f(p, param)); p := p.nxt END;
   RETURN new
END Map;

PROCEDURE (b: Basis) Clone* (): Basis;
(**
   Returns a shallow copy of b.
   The result has the same dynamic type as b. The contents of the new object are copied!
   Clone is used before the object's contents is copied.
   Clone needs to be extended in extensions of Basis.
   Extensions should never make a super call.
**)
   VAR new: Basis;
BEGIN
   NEW(new); new↑ := b↑;
   RETURN new
END Clone;

PROCEDURE (b: Basis) CopyParts* ;
(**
   Copies relevant parts.
   Extensions should always make a super call.
**)
   VAR p: Polynomial;
BEGIN
   IF b.size > 0 THEN
      p := b.first.Clone(); p.CopyParts; b.first := p;
      WHILE p.nxt # NIL DO p.nxt := p.nxt.Clone(); p := p.nxt; p.CopyParts END;
      b.last := p
   END
END CopyParts;

PROCEDURE (basis: Basis) Read* (VAR s(*inout*): S.Scanner; VAR b(*out*): Basis);
(**
   Reads a basis of the form:
   Basi = "{" [Element { "," Element}] "}".
```

```
    Elemen = Algebraic.Expression
    The type of polynomial read is determined by a polynomial in basis.

    Pre
    basis.Size() > 0 20
**)
    VAR a: Polynomial; h: A.Object;
BEGIN
    NEW(b); InitBasis(b);
    a := basis.first;
    IF s.type = S.lbrace THEN
        s.Scan;
        IF s.type # S.rbrace THEN
                LOOP
                    a.pol.ReadExpression(s, h); a := a.Clone(); Init(a, h(P.Polynomial));
                    b.Adjoin(a);
                    IF s.type # S.comma THEN EXIT END;
                    s.Scan
                END
        END;
        IF s.type = S.rbrace THEN s.Scan ELSE s.Mark(8) (* "}" expected *) END
    ELSE
        s.Mark(7) (* "{" expected *)
    END
END Read;

PROCEDURE (b: Basis) Write* (VAR f(*inout*): F.Formatter);
(** Writes a textual representation of a with f. **)
    VAR p: Polynomial;
BEGIN
    p := b.first;
    IF f.tex THEN f.WriteString("\{") ELSE f.WriteChar("{") END;
    WHILE p # NIL DO
        p.pol.Write(f);
        p := p.nxt;
        IF p # NIL THEN f.WriteString(", "); f.WriteLn END
    END;
    IF f.tex THEN f.WriteString("\}") ELSE f.WriteChar("}") END
END Write;

PROCEDURE (VAR w(*inout*): Write) Write* (VAR f(*inout*): F.Formatter);
BEGIN
    w.b.Write(f)
END Write;

PROCEDURE (b: Basis) WriteF* (VAR f(*inout*): F.Formatter; str: ARRAY OF CHAR);
(**
    Write formatted basis b.
    See also: WriteFormat.
**)
VAR w: Write;
BEGIN
    w.b := b; f.WriteFormat(str, w)
END WriteF;

PROCEDURE (b: Basis) Normalize*;
(** Normalizes the elements of basis b. **)
    VAR p: Polynomial;
BEGIN
    p := b.first;
    WHILE p # NIL DOp.Norm(); p := p.nxt END
END Normalize;

PROCEDURE (b: Basis) IsReduced* (): BOOLEAN;
(**
    Checks if the basis b is reduced.
    To check if the basus b is a groebner basis you should use
    GrobnerTrace.GrobnerBasisWithTrace, GrobnerTrace.TransformationOf andGrobnerTrace.Syzygy.

    Except for performace, equivalent to:
    FOR all poly p in basis b DO
        IF p.Reduce(0, b\{p}) # p THEN RETURN FALSE END
    END
    RETURN TRUE
**)
    VAR prv, cur, red: Polynomial;
BEGIN
    IF b.size < 2 THEN RETURN TRUE END;
    prv := NIL; cur := b.first; DEC(b.size);
    REPEAT
        (* b := b - {cur} *)
        IF prv # NIL THEN
                prv.nxt := cur.nxt;
                IF cur = b.last THEN b.last := prv END
        ELSE
```

```
            b.first := cur.nxt;
        END;
        red := cur.Clone(); red.CopyParts;
        red.Reduce(0, b);
        (* b := b + {cur} *)
        IF prv # NIL THEN
            prv.nxt := cur;
            IF prv = b.last THEN b.last := cur END
        ELSE
            b.first := cur
        END;
        (* check *)
        IF ~red.pol.Eql(cur.pol) THEN
            INC(b.size);
            RETURN FALSE
        END;
        prv := cur; cur := cur.nxt
    UNTIL cur = NIL;
    INC(b.size);
    RETURN TRUE
END IsReduced;

PROCEDURE (a: Basis) Eql* (b: Basis): BOOLEAN;
(**
    Compares the bases a and b. Equality is checked element wise and not in an Ideal theoretic sense.

    result = (a = b).
**)
        VAR p, q: Polynomial;
BEGIN
    p := a.first; q := b.first;
    WHILE (p # NIL) & (q # NIL) & p.pol.Eql(q.pol) DO p := p.nxt; q := q.nxt END;
    RETURN p = q
END Eql;

PROCEDURE (VAR it(*out*): Iterator) SetTo* (b: Basis);
(**
    Set it to the first element of basis.

    Usage:
        Var it: Iterator; p: Polynomial;
        it.SetTo(basis); p := it.Next();
        WHILE p # NIL DO ... p := it.Next() END;

    See also: Next.
**)
BEGIN
    it.nxt := b.first
END SetTo;

PROCEDURE (VAR it(*inout*): Iterator) Next* (): Polynomial;
(**
    Returns next (first after SetTo) element of basis or NIl if there are no (more) elements in basis.

    Usage:
        Var it: Iterator; p: Polynomial;
        it.SetTo(basis); p := it.Next();
        WHILE p # NIL DO ... p := it.Next() END;

    See also: SetTo.
**)
    VAR p: Polynomial;
BEGIN
    p := it.nxt;
    IF p # NIL THEN it.nxt := p.nxt END;
    RETURN p
END Next;

BEGIN
END Grobner.
```

```
MODULE GrobnerPolys;
(**
   Grobner Polynomials with rational Coefficients.

   Author: Gerald Slacik (gs)
*)

   IMPORT G := Grobner, P := Multivariate, PP := PowerProducts, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (**
         Polynomial for (Groebner)bases with rational coefficients.
         Used to solve systems of equations by Groebner bases computation.
      **)
      END;

   (* Polynomials *)

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Clone.

      Returns a shallow copy of p.
   **)
      VAR c: Polynomial;
   BEGIN
      NEW(c); c↑ := a↑;
      RETURN c
   END Clone;

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Norm.

      Normalized a by dividing it by the leading coefficient.

      Pre
      ~a.pol.IsZero() 20
   **)
      VAR lc: A.Coefficient; h: A.Object;
   BEGIN
      lc := a.pol.Coeff(0);
      IF ~lc.IsOne() THEN h := a.pol.Div(lc); G.Init(a, h(P.Polynomial)) END
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: P.Length; b: G.Polynomial);
   (**
      Extension.
      Overwrites Grobner.Polynomial.Red.

      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      0 <= n < a.pol.Length() 23
      a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
      b is normed 25

      Except for performace, equivalent to:
         a.pol := a.pol.Sub(b.pol.MulTerm(a.pol.Coeff(n).Div(b.pol.Coeff(0), a.pol.PP(n).IDiv(b.pol.PP(0))))
   **)
      VAR h: A.Object; pp: PP.PowerProduct;
   BEGIN
      pp := a.pol.PP(n); pp := pp.IDiv(b.lpp);
      G.Init(a, P.Red(a.pol, b.pol.Coeff(0), n, b.pol, a.pol.Coeff(n), pp))
   END Red;

   PROCEDURE (a: Polynomial) Spol* (b: G.Polynomial; lcm: PP.PowerProduct): G.Polynomial;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Spol.

      Returns the Spolynomial of a and b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
```

```
    a is normed 23
    b is normed 24
    lcm.Eql(a.lpp.Lcm(b.lpp)) 25

    Post
    result IS SameDynamicTypeAs(a) 60

    Except for performace, equivalent to:
        a.pol := a.pol.MulTerm(b.pol.Coeff(n), lcm.IDiv(b.lpp)).Sub(b.pol.MulTerm(a.pol.Coeff(0)), lcm.IDiv(b.lpp))
  **)
    VAR h: A.Object; c: Polynomial;
  BEGIN
    NEW(c); G.Init(c, P.Spol(a.pol, b.pol.Coeff(0), lcm.IDiv(a.lpp), 0, b.pol, a.pol.Coeff(0), lcm.IDiv(b.lpp)));
    RETURN c
  END Spol;

END GrobnerPolys.
```

```
MODULE IntGrobnerPolys;
(**
   Grobner Polynomials with integral Coefficients.

   Author: Gerald Slacik (gs)
*)

   IMPORT
      G := Grobner, P := Multivariate, PP := PowerProducts, Pol := Polynomials, IQ := IntegralsAndQuotients, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (**
         Polynomial for (Groebner)bases with integral coefficients.
         Used to solve systems of equations by Groebner bases computation.
      **)
      END;

   (* Polynomials *)

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Clone.

      Returns a shallow copy of p.
   **)
      VAR c: Polynomial;
   BEGIN
      NEW(c); c↑ := a↑;
      RETURN c
   END Clone;

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Norm.

      Normalizes a by making it primitive.

      Pre
      ~a.pol.IsZero() 20
   **)
      VAR h: Pol.Polynomial;
   BEGIN
      h := a.pol.PrimPart();
      G.Init(a, h(P.Polynomial))
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: P.Length; b: G.Polynomial);
   (**
      Extension.
      Overwrites Grobner.Polynomial.Red.

      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      0 <= n < a.pol.Length() 23
      a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
      b is normed 25

      Except for performace, equivalent to:
         a.pol := a.pol.Sub(b.pol.MulTerm(a.pol.Coeff(n).Div(b.pol.Coeff(0), a.pol.PP(n).IDiv(b.pol.PP(0))))
   **)
      VAR ac, bc, g: IQ.Integral; h: A.Object; pp: PP.PowerProduct;
   BEGIN
      h := a.pol.Coeff(n); ac := h(IQ.Integral);
      h := b.pol.Coeff(0); bc := h(IQ.Integral);
      g := ac.Gcd(bc);
      IF ~g.IsOne() THEN ac := ac.IDiv(g); bc := bc.IDiv(g) END;
      pp := a.pol.PP(n); pp := pp.IDiv(b.lpp);
      G.Init(a, P.Red(a.pol, bc, n, b.pol, ac, pp))
   END Red;

   PROCEDURE (a: Polynomial) Spol* (b: G.Polynomial; lcm: PP.PowerProduct): G.Polynomial
   (**
      Extension.
      Overwrites Grobner.Polynomial.Spol.

      Returns the Spolynomial of a and b.
```

```
        Pre
        b IS SameDynamicTypeAs(a) 20
        ~a.pol.IsZero() 21
        ~b.pol.IsZero() 22
        a is normed 23
        b is normed 24
        lcm.Eql(a.lpp.Lcm(b.lpp)) 25

        Post
        result IS SameDynamicTypeAs(a) 60

        Except for performace, equivalent to:
            a.pol := a.pol.MulTerm(b.pol.Coeff(n), lcm.IDiv(b.lpp)).Sub(b.pol.MulTerm(a.pol.Coeff(0)), lcm.IDiv(b.lpp))
    **)
        VAR ac, bc, g: IQ.Integral; h: A.Object; c: Polynomial;
    BEGIN
        lcm := a.lpp.Lcm(b.lpp);
        h := a.pol.Coeff(0); bc := h(IQ.Integral);
        h := b.pol.Coeff(0); ac := h(IQ.Integral);
        g := ac.Gcd(bc);
        IF ~g.IsOne() THEN ac := ac.IDiv(g); bc := bc.IDiv(g) END;
        NEW(c); G.Init(c, P.Spol(a.pol, ac, lcm.IDiv(a.lpp), 0, b.pol, bc, lcm.IDiv(b.lpp)));
        RETURN c
    END Spol;

END IntGrobnerPolys.
```

```
MODULE Expressions;
(**
   Expressions of algbebraic objects.
   The operations are not executed immediatedly. They are just memorized. Calling Expand on an Expression executes all necessry
   opertions to obtain the result.
   Motivation: During Groebner Basis computation many zero coefficients are calculated. Keeping this calculations in Expressions
   gives us the possibility to do the computation only on the coefficients we need for the final result.

   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic;

   TYPE
      Expression* = POINTER TO ExpressionDesc;
      ExpressionDesc* = RECORD
      (**
         Abstract.
         Expression, with actual calculation done only on demand via Expand
      **)
      res: A.Object (* cache result of Expand; NIl if Expand was not called for this expression *)
      END;

      ConstExpression* = POINTER TO ConstExpressionDesc;
      ConstExpressionDesc* = RECORD (ExpressionDesc)
      (** Constant Expression represents itself; used as starting point for evaluation via Expand **)
      END;

      UnaryExpression* = POINTER TO UnaryExpressionDesc;
      UnaryExpressionDesc* = RECORD (ExpressionDesc)
      (**
         Abstract.
         Unary Expression, with actual calculation done only on demand via Expand
      **)
         op: Expression (* operand of unary expression *)
      END;
      ZeroExpression* = POINTER TO ZeroExpressionDesc;
      ZeroExpressionDesc* = RECORD (UnaryExpressionDesc)
      END;
      OneExpression* = POINTER TO OneExpressionDesc;
      OneExpressionDesc* = RECORD (UnaryExpressionDesc)
      END;
      NegOneExpression* = POINTER TO NegOneExpressionDesc;
      NegOneExpressionDesc* = RECORD (UnaryExpressionDesc)
      END;
      NegExpression* = POINTER TO NegExpressionDesc;
      NegExpressionDesc* = RECORD (UnaryExpressionDesc)
      END;
      InvExpression* = POINTER TO InvExpressionDesc;
      InvExpressionDesc* = RECORD (UnaryExpressionDesc)
      END;

      BinaryExpression* = POINTER TO BinaryExpressionDesc;
      BinaryExpressionDesc* = RECORD (ExpressionDesc)
      (**
         Abstract.
         Binary Expression, with actual calculation done only on demand via Expand
      **)
         lhs, rhs: Expression (* left & right hand side of binary expression *)
      END;
      AddExpression* = POINTER TO AddExpressionDesc;
      AddExpressionDesc* = RECORD (BinaryExpressionDesc)
      END;
      SubExpression* = POINTER TO SubExpressionDesc;
      SubExpressionDesc* = RECORD (BinaryExpressionDesc)
      END;
      MulExpression* = POINTER TO MulExpressionDesc;
      MulExpressionDesc* = RECORD (BinaryExpressionDesc)
      END;
      DivExpression* = POINTER TO DivExpressionDesc;
      DivExpressionDesc* = RECORD (BinaryExpressionDesc)
      END;

   VAR
      addCount-, subCount-, mulCount-, divCount-: LONGINT; (** counters for expressions **)
      addXCount-, subXCount-, mulXCount-, divXCount-: LONGINT; (** counters for expanded expressions **)

   PROCEDURE InitProfiling*;
   BEGIN
      addCount := 0; subCount := 0; mulCount := 0; divCount := 0;
      addXCount := 0; subXCount := 0; mulXCount := 0; divXCount := 0
   END InitProfiling;

   PROCEDURE Init* (exp: ConstExpression; const: A.Object);
   (**
```

```
   Initializes the Expression exp with the constant const.

   Pre
   const # NIL 20

   Post
   exp.Expand() = const
**)
BEGIN
   exp.res := const
END Init;

(** Expression **)

PROCEDURE (a: Expression) Zero* (): ZeroExpression;
(**
   result = 0.
**)
   VAR c: ZeroExpression;
BEGIN
   NEW(c); c.op := a;
   RETURN c
END Zero;

PROCEDURE (a: Expression) One* (): OneExpression;
(**
   result = 1.
**)
   VAR c: OneExpression;
BEGIN
   NEW(c); c.op := a;
   RETURN c
END One;

PROCEDURE (a: Expression) NegOne* (): NegOneExpression;
(**
   result = -1.
**)
   VAR c: NegOneExpression;
BEGIN
   NEW(c); c.op := a;
   RETURN c
END NegOne;

PROCEDURE (a: Expression) Neg* (): NegExpression;
(**
   result = -a.
**)
   VAR c: NegExpression;
BEGIN
   NEW(c); c.op := a;
   RETURN c
END Neg;

PROCEDURE (a: Expression) Inv* (): InvExpression;
(**
   result = 1/a.
**)
   VAR c: InvExpression;
BEGIN
   NEW(c); c.op := a;
   RETURN c
END Inv;

PROCEDURE (a: Expression) Add* (b: Expression): AddExpression;
(**
   result = a+b.
**)
   VAR c: AddExpression;
BEGIN INC(addCount);
   NEW(c); c.lhs := a; c.rhs := b;
   RETURN c
END Add;

PROCEDURE (a: Expression) AddObj* (b: A.Object): AddExpression;
(**
   result = a+b.
**)
   VAR c: AddExpression; const: ConstExpression;
BEGIN INC(addCount);
   NEW(const); Init(const, b);
   NEW(c); c.lhs := a; c.rhs := const;
   RETURN c
END AddObj;
```

```
PROCEDURE (a: Expression) Sub* (b: Expression): SubExpression;
(**
   result = a-b.
**)
   VAR c: SubExpression;
BEGIN INC(subCount);
   NEW(c); c.lhs := a; c.rhs := b;
   RETURN c
END Sub;

PROCEDURE (a: Expression) SubObj* (b: A.Object): SubExpression;
(**
   result = a-b.
**)
   VAR c: SubExpression; const: ConstExpression;
BEGIN INC(subCount);
   NEW(const); Init(const, b);
   NEW(c); c.lhs := a; c.rhs := const;
   RETURN c
END SubObj;

PROCEDURE (a: Expression) Mul* (b: Expression): MulExpression;
(**
   result = a*b.
**)
   VAR c: MulExpression;
BEGIN INC(mulCount);
   NEW(c); c.lhs := a; c.rhs := b;
   RETURN c
END Mul;

PROCEDURE (a: Expression) MulObj* (b: A.Object): MulExpression;
(**
   result = a*b.
**)
   VAR c: MulExpression; const: ConstExpression;
BEGIN INC(mulCount);
   NEW(const); Init(const, b);
   NEW(c); c.lhs := a; c.rhs := const;
   RETURN c
END MulObj;

PROCEDURE (a: Expression) Div* (b: Expression): DivExpression;
(**
   result = a/b.
**)
   VAR c: DivExpression;
BEGIN INC(divCount);
   NEW(c); c.lhs := a; c.rhs := b;
   RETURN c
END Div;

PROCEDURE (a: Expression) DivObj* (b: A.Object): DivExpression;
(**
   result = a/b.
**)
   VAR c: DivExpression; const: ConstExpression;
BEGIN INC(divCount);
   NEW(const); Init(const, b);
   NEW(c); c.lhs := a; c.rhs := const;
   RETURN c
END DivObj;

PROCEDURE (a: Expression) Expand* (): A.Object;
(**
   Abstract method. Must be overwritten.
   Evaluates the expression a. Caches the result. Calling Expand a second time for an expression just
   returns the cached result. The links to the operands of the expression are set to NIL for storage optimization.
**)
BEGIN
   HALT(A.abstractMethod)
END Expand;

(** ConstantExpression **)

PROCEDURE (a: ConstExpression) Expand* (): A.Object;
(**
   Evaluates the expression a.
**)
BEGIN
   RETURN a.res
END Expand;

(** UnaryExpression **)
```

```
PROCEDURE (a: ZeroExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.op.Expand(); a.op := NIL; a.res := h.Zero() END;
   RETURN a.res
END Expand;

PROCEDURE (a: OneExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.op.Expand(); a.op := NIL; a.res := h.One() END;
   RETURN a.res
END Expand;

PROCEDURE (a: NegOneExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.op.Expand(); a.op := NIL; a.res := h.NegOne() END;
   RETURN a.res
END Expand;

PROCEDURE (a: NegExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.op.Expand(); a.op := NIL; a.res := h.Neg() END;
   RETURN a.res
END Expand;

PROCEDURE (a: InvExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.op.Expand(); a.op := NIL; a.res := h.Inv() END;
   RETURN a.res
END Expand;

(** BinaryExpression **)

PROCEDURE (a: AddExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.lhs.Expand(); a.lhs := NIL; a.res := h.Add(a.rhs.Expand()); a.rhs := NIL; INC(addXCount) END;
   RETURN a.res
END Expand;

PROCEDURE (a: SubExpression) Expand* (): A.Object;
(**
   Extension.
   Overwrites Expression.Expand.
   Evaluates the expression a.
**)
   VAR h: A.Object;
BEGIN
   IF a.res = NIL THEN h := a.lhs.Expand(); a.lhs := NIL; a.res := h.Sub(a.rhs.Expand()); a.rhs := NIL; INC(subXCount) END;
   RETURN a.res
END Expand;
```

```
   PROCEDURE (a: MulExpression) Expand* (): A.Object;
   (**
      Extension.
      Overwrites Expression.Expand.
      Evaluates the expression a.
   **)
      VAR h: A.Object;
   BEGIN
      IF a.res = NIL THEN h := a.lhs.Expand(); a.lhs := NIL; a.res := h.Mul(a.rhs.Expand()); a.rhs := NIL; INC(mulXCount) END;
      RETURN a.res
   END Expand;

   PROCEDURE (a: DivExpression) Expand* (): A.Object;
   (**
      Extension.
      Overwrites Expression.Expand.
      Evaluates the expression a.
   **)
      VAR h: A.Object;
   BEGIN
      IF a.res = NIL THEN h := a.lhs.Expand(); a.lhs := NIL; a.res := h.Div(a.rhs.Expand()); a.rhs := NIL; INC(divXCount) END;
      RETURN a.res
   END Expand;

BEGIN
   InitProfiling
END Expressions.
```

```
MODULE GrobnerTrans;
(**
   Groebner Bases with Transformation Matrices.

   Author: Gerald Slacik (gs)
*)

   IMPORT E := Expressions, G := Grobner, M := Matrices, P := Multivariate, PP := PowerProducts, A := Algebraic, F := Formatters;

   CONST
      SetSize = MAX(SET) + 1;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (**
         Abstract.
         Polynomial for (Groebner)bases.
         Memorizes its transformations.
      **)
         transExp-: E.Expression;
      END;

      Basis* = POINTER TO BasisDesc;
      BasisDesc* = RECORD (G.BasisDesc)
      (**
         Used to solve systems of equations by Groebner bases computation.
         Memorizes its transformations.
      **)
      END;

      Poly2TransPoly* = PROCEDURE (p: G.Polynomial; VAR param(*inout*): A.Param): Polynomial;
      (** constructs a GrobnerTrans.Polynomial from p **)

   (* Polynomials *)

   PROCEDURE Init* (p: Polynomial; a: P.Polynomial; transExp: E.Expression);
   (** Initializes the GrobnerPolynomial p with the multivariate polynomial a and transExp **)
   BEGIN
      G.Init(p, a); p.transExp := transExp
   END Init;

   PROCEDURE InitWithObj* (p: Polynomial; a: P.Polynomial; transExp: A.Object);
   (** Initializes the GrobnerPolynomial p with the multivariate polynomial a and transExp **)
      VAR const: E.ConstExpression;
   BEGIN
      NEW(const); E.Init(const, transExp);
      Init(p, a, const)
   END InitWithObj;

   (* Bases *)

   PROCEDURE (b: Basis) Clone* (): G.Basis;
   (**
      Extension.
      Overwrites Grobner.Basis.Clone.
      Returns a shallow copy of b.
      The result has the same dynamic type as b. The contents of the new object are copied!
      Clone is used before the object's contents is copied.
      Clone needs to be extended in extensions of Basis.
      Extensions should never make a super call.
   **)
      VAR new: Basis;
   BEGIN
      NEW(new); new↑ := b↑;
      RETURN new
   END Clone;

   PROCEDURE InitBasis* (b: Basis);
   (**
      Initializes basis b with {}.

      Post
      b.Size() = 0 60
   **)
   BEGIN
      G.InitBasis(b)
   END InitBasis;

   PROCEDURE MatrixFrom* (b: G.Basis): M.Matrix;
   (**
      Extracts the transExps of b into a matrix.

      Post
      b.Size() = 0
```

```
      result = NIL
    b.Size() # 0
      result.Rows() = b.Size()
      result.Columns() = 1
**)
  VAR it: G.Iterator; p: G.Polynomial; n: G.Size; m: M.Matrix;
BEGIN
  n := b.Size();
  IF n = 0 THEN RETURN NIL END;
  NEW(m); M.Init(m, n, 1);
  it.SetTo(b); p := it.Next();
  WHILE p # NIL DO DEC(n); m.Put(n, 0, p.pol); p := it.Next() END;
  RETURN m
END MatrixFrom;

PROCEDURE Basis2TransBasis* (basis: G.Basis; T: Poly2TransPoly; VAR param(*inout*): A.Param): Basis;
(**
  Converts basis b into one that memorizes its transformations.
  Resets the memorized transformations to identity.

  Post
  result.Eql(basis)
**)
  VAR it: G.Iterator; p: G.Polynomial; b: Basis;
BEGIN
  NEW(b); InitBasis(b);
  it.SetTo(basis); p := it.Next();
  WHILE p # NIL DO b.Adjoin(T(p, param)); p := it.Next() END;
  RETURN b
END Basis2TransBasis;

PROCEDURE (b: Basis) GrobnerBasisAndTrans* (VAR transMatrix(*out*): M.Matrix);
(**
  Computes the Groebnerbasis of b.
  The basis b itself is modified! If you need the original basis b afterwards you should copy it first by:
  org := b.Clone; org.CopyParts;
  Post
  b.Size() = 0
    transMatrix = NIL
  b.Size() # 0
    transMatrix.Rows() = b'.Size(), where b' is b after the call of GrobnerBasisWithTrac 61
    transMatrix.Columns() = b.Size( 61
    MatrixFrom(b').Eql(transMatrix.Mul(MatrixFrom(b)) 62
**)
  VAR i, j, n: G.Size; it: G.Iterator; p: G.Polynomial; t, trans: M.Matrix; h: A.Object; one, zero: A.Coefficient; c: E.ConstExpression;
BEGIN
  n := b.Size(); (* size of F; to get dimensions of matrix right *)
  IF n = 0 THEN transMatrix := NIL; RETURN END;

  (* Sets transExp of polynomials of basis b to constant expressions of unity vectors of length m = b.Size() *)
  it.SetTo(b); p := it.Next();
  h := p.pol.One(); one := h(P.Polynomial); h := one.Zero(); zero := h(P.Polynomial);
  i := n;
  REPEAT
    DEC(i);
    NEW(trans); M.Init(trans, 1, n); trans.Fill(zero); trans.Put(0, i, one);
    NEW(c); E.Init(c, trans);
    p(Polynomial).trans := c;
    p := it.Next()
  UNTIL p = NIL;

  b.GrobnerBasis;

  (* Extracts transExps of basis b and puts it into Matrix trans *)
  i := b.Size();
  NEW(trans); M.Init(trans, i, n);
  it.SetTo(b); p := it.Next();
  REPEAT
    DEC(i);
    h := p(Polynomial).transExp(E.Expression).Expand(); p(Polynomial).transExp := NIL;
    t := h(M.Matrix);
    j := n; REPEAT DEC(j); trans.Put(i, j, t.Get(0, j)) UNTIL j = 0;
    p := it.Next()
  UNTIL p = NIL;
  transMatrix := trans
END GrobnerBasisAndTrans;

PROCEDURE (grobnerbasis: Basis) TransformationOf* (basis: G.Basis): M.Matrix;
(**
  Reduces basis with grobnerbasis and returns the transformation matrix of this reduction.

  Pre
  all polynomials of basis are reducible to 0 with regard to grobnerbasis 20
  Post
  grobnerbasis.Size() = 0 OR basis.Size() = 0
```

```
        result = NIL
    grobnerbasis.Size() # 0 & basis.Size() # 0
        result.Rows() = basis.Size()
        result.Columns() = grobnerbasis.Size()
        MatrixFrom(basis).Eql(result.Mul(MatrixFrom(grobnerbasis)) 60
**)
    VAR it: G.Iterator; p: G.Polynomial; r: Polynomial; (* r is cloned once from a poly of grobnerbasis and used for all reductions *)
        u, t, trans: M.Matrix; (* u .. unity vector used to keep track of multiplicands of the reduced poly *)
        i, j, m, n: G.Size; c, h: A.Object; one, zero: A.Coefficient;
BEGIN
    n := grobnerbasis.Size();
    IF n = 0 THEN RETURN NIL END;

    (* Sets transExp of polynomials of basis grobnerbasis to unity vectors of length n+1 where n = grobnerbasis.Size() *)
    it.SetTo(grobnerbasis); p := it.Next();
    h := p.pol.One(); one := h(P.Polynomial); h := one.Zero(); zero := h(P.Polynomial);
    i := n;
    REPEAT
        DEC(i);
        NEW(trans); M.Init(trans, 1, n+1); trans.Fill(zero); trans.Put(0, i, one);
        InitWithObj(p(Polynomial), p.pol, trans);
        p := it.Next()
    UNTIL p = NIL;

    (* reduce all polys of b to 0 w.r.t. grobnerbasis; extract the transExp and put it in the result matrix trans *)
    NEW(u); M.Init(u, 1, n+1); u.Fill(zero); u.Put(0, n, one);
    it.SetTo(grobnerbasis); p := it.Next();
    p := p.Clone(); r := p(Polynomial);
    it.SetTo(basis); p := it.Next();
    m := basis.Size(); NEW(trans); M.Init(trans, m, n);
    i := m;
    REPEAT
        DEC(i);
        h := p.pol.Neg(); InitWithObj(r, h(P.Polynomial), u);
        r.Reduce(0, grobnerbasis);
        h := r.transExp.Expand();
        t := h(M.Matrix); c := t.Get(0, n);
        j := n;
        IF c.IsOne() THEN
                REPEAT DEC(j); trans.Put(i, j, t.Get(0, j)) UNTIL j = 0
        ELSE
                c := c.Inv();
                REPEAT DEC(j); h := c.Mul(t.Get(0, j)); trans.Put(i, j, h(A.Coefficient)) UNTIL j = 0
        END;
        p := it.Next()
    UNTIL p = NIL;

    RETURN trans
END TransformationOf;

PROCEDURE (grobnerbasis: Basis) Syzygy* (all: BOOLEAN): M.Matrix;
(**
    Reduces all if all is TRUE and only the necessary if all is FALSE S-polynomials of grobnerbasis
    with grobnerbasis and returns the syzygy matrix.

    Pre
    all S-polynomials are reducible to 0 with regard to grobnerbasis 20
    Post
    grobnerbasis.Size() < 2 OR all S-polynomials are skipped because of Product Criterion
        result = NIL
    else
        result.Columns() = grobnerbasis.Size()
        result.Mul(MatrixFrom(grobnerbasis)).IsZero() 60
**)
    VAR it1, it2, it3: G.Iterator; p1, p2, p3: G.Polynomial; i, j, m, n: G.Size; t, trans: M.Matrix;
        h: A.Object; one, zero: A.Coefficient; lcm: POINTER TO ARRAY OF PP.PowerProduct; l: PP.PowerProduct;
BEGIN
    n := grobnerbasis.Size();
    IF n < 2 THEN RETURN NIL END;

    (* checks unnecessary spolys of basis grobnerbasis (of size n) in skip and returns the number of necessary spolys in m *)
    i := (n*(n-1) DIV 2 + SetSize - 1); NEW(lcm, i);
    (* adjust rows skipped because of Product or Chain Criterion *)
    it1.SetTo(grobnerbasis); p1 := it1.Next(); p1 := it1.Next();
    IF ~all THEN
        m := 0;i := 0;
        REPEAT
                it2.SetTo(grobnerbasis); p2 := it2.Next();
                REPEAT
                    IF ~p1.lpp.ProdCrit(p2.lpp) THEN
                        (* check for chain criterion *)
                        l := p1.lpp.Lcm(p2.lpp);
                        it3.SetTo(grobnerbasis);
                        REPEAT p3 := it3.Next() UNTIL (p3 = NIL) OR ((p3 # p1) & (p3 # p2) & l.IsMultipleOf(p3.lpp));
                        IF p3 = NIL THEN lcm[i] := l; INC(m) END
```

```
                END;
                p2 := it2.Next(); INC(i)
            UNTIL p2 = p1;
            p1 := it1.Next()
        UNTIL p1 = NIL;
        IF m = 0 THEN RETURN NIL END;
    ELSE
        m := 0;
        REPEAT
            it2.SetTo(grobnerbasis); p2 := it2.Next();
            REPEAT lcm[m] := p1.lpp.Lcm(p2.lpp); p2 := it2.Next(); INC(m) UNTIL p2 = p1;
            p1 := it1.Next()
        UNTIL p1 = NIL
    END;

    (* Set transExp of polynomials of basis grobnerbasis to unity vectors of length n = grobnerbasis.Size() *)
    it1.SetTo(grobnerbasis); p1 := it1.Next();
    h := p1.pol.One(); one := h(P.Polynomial); h := one.Zero(); zero := h(P.Polynomial);
    i := n;
    REPEAT
        DEC(i);
        NEW(trans); M.Init(trans, 1, n); trans.Fill(zero); trans.Put(0, i, one);
        InitWithObj(p1(Polynomial), p1.pol, trans);
        p1 := it1.Next()
    UNTIL p1 = NIL;

    (* Compute all necessary spolys and reduces them to 0 w.r.t. grobnerbasis; extract the transExp and put it in the result matrix trans *)
    it1.SetTo(grobnerbasis); p1 := it1.Next(); p1 := it1.Next();
    NEW(trans); M.Init(trans, m, n);
    m := 0; i := 0;
    REPEAT
        it2.SetTo(grobnerbasis); p2 := it2.Next();
        REPEAT
            l := lcm[i];
            IF l # NIL THEN
                p2 := p1.Spol(p2, l); p2.Reduce(0, grobnerbasis);
                h := p2(Polynomial).transExp.Expand();
                t := h(M.Matrix);
                j := n; REPEAT DEC(j); trans.Put(m, j, t.Get(0, j)) UNTIL j = 0;
                INC(m)
            END;
            p2 := it2.Next(); INC(i)
        UNTIL p2 = p1;
        p1 := it1.Next()
    UNTIL p1 = NIL;

    RETURN trans
  END Syzygy;

BEGIN
END GrobnerTrans.
```

```
MODULE GrobnerTransPolys;
(**
   Grobner Polynomials with rational Coefficients keeping track of their transformations to compute transformation matrices.

   Author: Gerald Slacik (gs)
*)

   IMPORT T := GrobnerTrans, G := Grobner, P := Multivariate, PP := PowerProducts, E := Expressions, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (T.PolynomialDesc)
      (**
         Grobner Polynomials with rational Coefficients keeping track of their transformations to compute transformation matrices.
      **)
      END;

   (* Polynomials *)

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Clone.

      Returns a shallow copy of p.
   **)
      VAR c: Polynomial;
   BEGIN
      NEW(c); c↑ := a↑;
      RETURN c
   END Clone;

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Norm.

      Normalizes a by dividing it by the leading coefficient.

      Pre
      ~a.pol.IsZero() 20
   **)
      VAR lc: A.Coefficient; h: A.Object;
   BEGIN
      lc := a.pol.Coeff(0);
      IF ~lc.IsOne() THEN
         h := a.pol.Div(lc);
         T.Init(a, h(P.Polynomial), a.transExp.DivObj(lc))
      END
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: P.Length; b: G.Polynomial);
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Red.

      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      0 <= n < a.pol.Length() 23
      a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
      b is normed 25

      Except for performace, equivalent to:
         a.pol := a.pol.Sub(b.pol.MulTerm(a.pol.Coeff(n).Div(b.pol.Coeff(0), a.pol.PP(n).IDiv(b.pol.PP(0))))
   **)
      VAR h: A.Object; pp: PP.PowerProduct; p: P.Polynomial; t: E.Expression;
   BEGIN
      pp := a.pol.PP(n); pp := pp.IDiv(b.lpp);
      h := a.pol.Coeff(n); h := h.Neg();
      NEW(p); P.Init(p, h(A.Coefficient), pp);
      t := a.transExp.Add(b(Polynomial).transExp.MulObj(p));
      T.Init(a, P.Red(a.pol, b.pol.Coeff(0), n, b.pol, a.pol.Coeff(n), pp), t)
   END Red;

   PROCEDURE (a: Polynomial) Spol* (b: G.Polynomial; lcm: PP.PowerProduct): G.Polynomial;
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Spol.

      Returns the Spolynomial of a and b.
```

```
      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      a is normed 23
      b is normed 24
      lcm.Eql(a.lpp.Lcm(b.lpp)) 25

      Post
      result IS SameDynamicTypeAs(a) 60

      Except for performace, equivalent to:
         a.pol := a.pol.MulTerm(b.pol.Coeff(n), lcm.IDiv(b.lpp)).Sub(b.pol.MulTerm(a.pol.Coeff(0)), lcm.IDiv(b.lpp))
   **)
      VAR h: A.Object; app, bpp: PP.PowerProduct; c: Polynomial; p: P.Polynomial; t: E.Expression;
   BEGIN
      h := a.pol.Coeff(0); h := h.One();
      app := lcm.IDiv(a.lpp);
      NEW(p); P.Init(p, h(A.Coefficient), app);
      t := a.transExp.MulObj(p);
      bpp := lcm.IDiv(b.lpp);
      NEW(p); P.Init(p, h(A.Coefficient), bpp);
      NEW(c); T.Init(c, P.Spol(a.pol, b.pol.Coeff(0), app, 0, b.pol, a.pol.Coeff(0), bpp), t.Sub(b(Polynomial).transExp.MulObj(p)));
      RETURN c
   END Spol;

   PROCEDURE Poly2TransPoly* (p: G.Polynomial; VAR unusedParam(*inout*): A.Param): T.Polynomial;
   (** constructs a rational GrobnerTrans.Polynomial from p **)
      VAR t: Polynomial;
   BEGIN
      NEW(t); T.Init(t, p.pol, NIL);
      RETURN t
   END Poly2TransPoly;

END GrobnerTransPolys.
```

```
MODULE IntGrobnerTransPolys;
(**
   Grobner Polynomials with integral Coefficients keeping track of their transformations to compute transformation matrices.

   Author: Gerald Slacik (gs)
*)

   IMPORT
      T := GrobnerTrans, G := Grobner, P := Multivariate, PP := PowerProducts, E := Expressions,
      Pol := Polynomials, IQ := IntegralsAndQuotients, A := Algebraic;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (T.PolynomialDesc)
      (**
         Grobner Polynomial with integral Coefficients keeping track of their transformations to compute transformation matrices.
      **)
      END;

   (* Polynomials *)

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Clone.

      Returns a shallow copy of p.
   **)
      VAR c: Polynomial;
   BEGIN
      NEW(c); c↑ := a↑;
      RETURN c
   END Clone;

   PROCEDURE (a: Polynomial) Norm* ;
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Norm.

      Normalizes a by making it primitive.

      Pre
      ~a.pol.IsZero() 20
   **)
      VAR cont: A.Coefficient; primPart: Pol.Polynomial;
   BEGIN
      a.pol.ContentAndPrimPart(cont, primPart);
      T.Init(a, primPart(P.Polynomial), a.transExp.DivObj(cont))
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: P.Length; b: G.Polynomial);
   (**
      Extension.
      Overwrites GrobnerTrans.Polynomial.Red.

      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      0 <= n < a.pol.Length( 23
      a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
      b is normed 25

      Except for performace, equivalent to:
         a.pol := a.pol.Sub(b.pol.MulTerm(a.pol.Coeff(n).Div(b.pol.Coeff(0), a.pol.PP(n).IDiv(b.pol.PP(0))))
   **)
      VAR ac, bc, g: IQ.Integral; h: A.Object; pp: PP.PowerProduct; p: P.Polynomial; t: E.Expression;
   BEGIN
      h := a.pol.Coeff(n); bc := h(IQ.Integral);
      h := b.pol.Coeff(0); ac := h(IQ.Integral);
      g := ac.Gcd(bc);
      IF ~g.IsOne() THEN ac := ac.IDiv(g); bc := bc.IDiv(g) END;
      pp := a.pol.PP(n); pp := pp.IDiv(b.lpp);
      h := bc.Neg();
      NEW(p); P.Init(p, h(IQ.Integral), pp);
      t := b(Polynomial).transExp.MulObj(p);
      t := t.Add(a.transExp.MulObj(ac));
      T.Init(a, P.Red(a.pol, ac, n, b.pol, bc, pp), t)
   END Red;

   PROCEDURE (a: Polynomial) Spol* (b: G.Polynomial; lcm: PP.PowerProduct): G.Polynomial
   (**
```

```
      Extension.
      Overwrites GrobnerTrans.Polynomial.Spol.

      Returns the Spolynomial of a and b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      a is normed 23
      b is normed 24
      lcm.Eql(a.lpp.Lcm(b.lpp)) 25

      Post
      result IS SameDynamicTypeAs(a) 60

      Except for performace, equivalent to:
         a.pol := a.pol.MulTerm(b.pol.Coeff(n), lcm.IDiv(b.lpp)).Sub(b.pol.MulTerm(a.pol.Coeff(0)), lcm.IDiv(b.lpp))
   **)
      VAR ac, bc, g: IQ.Integral; h: A.Object; app, bpp: PP.PowerProduct; c: Polynomial; p: P.Polynomial; t: E.Expression;
   BEGIN
      h := a.pol.Coeff(0); bc := h(IQ.Integral);
      h := b.pol.Coeff(0); ac := h(IQ.Integral);
      g := ac.Gcd(bc);
      IF ~g.IsOne() THEN ac := ac.IDiv(g); bc := bc.IDiv(g) END;
      bpp := lcm.IDiv(b.lpp);
      h := bc.Neg();
      NEW(p); P.Init(p, h(IQ.Integral), bpp);
      t := b(Polynomial).transExp.MulObj(p);
      app := lcm.IDiv(a.lpp);
      NEW(p); P.Init(p, ac, app);
      t := t.Add(a.transExp.MulObj(p));
      NEW(c); T.Init(c, P.Spol(a.pol, ac, app, 0, b.pol, bc, bpp), t);
      RETURN c
   END Spol;

   PROCEDURE Poly2TransPoly* (p: G.Polynomial; VAR unusedParam(*inout*): A.Param): T.Polynomial;
   (** constructs a integral GrobnerTrans.Polynomial from p **)
      VAR t: Polynomial;
   BEGIN
      NEW(t); T.Init(t, p.pol, NIL);
      RETURN t
   END Poly2TransPoly;

END IntGrobnerTransPolys.
```

```
MODULE TracedCoefficients;
(**
   Traced expressions of algbebraic coefficients.
   Uses Expresssions to trace calculations. The operations are executed and memorized.
   Motivation: During Groebner Basis computation many zero coefficients are calculated. We are doing the computation
   modulo a luck prime and keep this calculations in Expressions. At the end we do the computation again on the
   integer coefficients we need for the final result.

   Author: Gerald Slacik (gs)
*)

   IMPORT A := Algebraic, E := Expressions, F := Formatters;

   TYPE
      Coefficient* = POINTER TO CoefficientDesc;
      CoefficientDesc* = RECORD (A.CoefficientDesc)
      (**
         Algebraic Coefficient, which memorizes the way it was calculated.
      **)
      coeff: A.Coefficient; (* coeff on wich operations are performed *)
      trace-: E.Expression (** trace of the operations on itself **)
      END;

   (** Coefficient **)

   PROCEDURE Init* (a: Coefficient; coeff: A.Coefficient; trace: E.Expression);
   (**
      Initializes the Coefficient a with the Algebraic.Object coeff and the Expression trace as trace for a.

      Pre
      coeff # NIL 20
      trace # NIL 21

      Post
      a.trace.Expand() = trace
   **)
   BEGIN
      a.coeff := coeff; a.trace := trace
   END Init;

   PROCEDURE (a: Coefficient) IsZero* (): BOOLEAN;
   (**
      Extension.
      Overwrites Algebraic.Object.IsZero.

      result = (a = 0).
   **)
   BEGIN
      RETURN a.coeff.IsZero()
   END IsZero;

   PROCEDURE (a: Coefficient) IsOne* (): BOOLEAN;
   (**
      Extension.
      Overwrites Algebraic.Object.IsOne.

      result = (a = 1).
   **)
   BEGIN
      RETURN a.coeff.IsOne()
   END IsOne;

   PROCEDURE (a: Coefficient) IsNegOne* (): BOOLEAN;
   (**
      Extension.
      Overwrites Algebraic.Object.IsNegOne.

      result = (a = -1).
   **)
   BEGIN
      RETURN a.coeff.IsNegOne()
   END IsNegOne;

   PROCEDURE (a: Coefficient) IsAbsOne* (): BOOLEAN;
   (**
      Extension.
      Overwrites Algebraic.Object.IsAbsOne.

      result = (|a| = 1).
   **)
   BEGIN
      RETURN a.coeff.IsAbsOne()
   END IsAbsOne;

   PROCEDURE (a: Coefficient) IsNeg* (): BOOLEAN;
```

```
(**
   Extension.
   Overwrites Algebraic.Object.IsNeg.

   result = (a < 0).
**)
BEGIN
   RETURN a.coeff.IsNeg()
END IsNeg;

PROCEDURE (a: Coefficient) Zero* (): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Zero.

   result = 0.
**)
   VAR c: Coefficient; co: A.Object;
BEGIN
   co := a.coeff.Zero();
   NEW(c); Init(c, co(A.Coefficient), a.trace.Zero());
   RETURN c
END Zero;

PROCEDURE (a: Coefficient) One* (): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.One.

   result = 1.
**)
   VAR c: Coefficient; co: A.Object;
BEGIN
   co := a.coeff.One();
   NEW(c); Init(c, co(A.Coefficient), a.trace.One());
   RETURN c
END One;

PROCEDURE (a: Coefficient) NegOne* (): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.NegOne.

   result = -1.
**)
   VAR c: Coefficient; co: A.Object;
BEGIN
   co := a.coeff.NegOne();
   NEW(c); Init(c, co(A.Coefficient), a.trace.NegOne());
   RETURN c
END NegOne;

PROCEDURE (a: Coefficient) Neg* (): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Neg.

   result = -a.
**)
   VAR c: Coefficient; co: A.Object;
BEGIN
   co := a.coeff.Neg();
   NEW(c); Init(c, co(A.Coefficient), a.trace.Neg());
   RETURN c
END Neg;

PROCEDURE (a: Coefficient) Inv* (): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Inv.

   result = 1/a.

   Pre
   ~a.IsZero() divideByZero
**)
   VAR c: Coefficient; co: A.Object;
BEGIN
   co := a.coeff.Inv();
   NEW(c); Init(c, co(A.Coefficient), a.trace.Inv());
   RETURN c
END Inv;

PROCEDURE (a: Coefficient) Eql* (b: A.Object): BOOLEAN;
(**
```

```
   Extension.
   Overwrites Algebraic.Object.Add.

   result = (a = b).
**)
BEGIN
   IF (b IS Coefficient) THEN
      RETURN a.coeff.Eql(b(Coefficient).coeff)
   ELSE
      RETURN a.coeff.Eql(b)
   END
END Eql;

PROCEDURE (a: Coefficient) Add* (b: A.Object): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Add.

   result = a+b.
**)
   VAR c: Coefficient; co: A.Object; tr: E.AddExpression; const: E.ConstExpression;
BEGIN
   IF b IS Coefficient THEN
      co := a.coeff.Add(b(Coefficient).coeff); tr := a.trace.Add(b(Coefficient).trace)
   ELSE
      NEW(const); E.Init(const, b);
      co := a.coeff.Add(b); tr := a.trace.Add(const)
   END;
   NEW(c); Init(c, co(A.Coefficient), tr);
   RETURN c
END Add;

PROCEDURE (a: Coefficient) Sub* (b: A.Object): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Sub.

   result = a-b.
**)
   VAR c: Coefficient; co: A.Object; tr: E.SubExpression; const: E.ConstExpression;
BEGIN
   IF b IS Coefficient THEN
      co := a.coeff.Sub(b(Coefficient).coeff); tr := a.trace.Sub(b(Coefficient).trace)
   ELSE
      NEW(const); E.Init(const, b);
      co := a.coeff.Sub(b); tr := a.trace.Sub(const)
   END;
   NEW(c); Init(c, co(A.Coefficient), tr);
   RETURN c
END Sub;

PROCEDURE (a: Coefficient) Mul* (b: A.Object): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Mul.

   result = a*b.
**)
   VAR c: Coefficient; co: A.Object; tr: E.MulExpression; const: E.ConstExpression;
BEGIN
   IF b IS Coefficient THEN
      co := a.coeff.Mul(b(Coefficient).coeff); tr := a.trace.Mul(b(Coefficient).trace)
   ELSE
      NEW(const); E.Init(const, b);
      co := a.coeff.Mul(b); tr := a.trace.Mul(const)
   END;
   NEW(c); Init(c, co(A.Coefficient), tr);
   RETURN c
END Mul;

PROCEDURE (a: Coefficient) Div* (b: A.Object): A.Object;
(**
   Extension.
   Overwrites Algebraic.Object.Div.

   result = a/b.
**)
   VAR c: Coefficient; co: A.Object; tr: E.DivExpression; const: E.ConstExpression;
BEGIN
   IF b IS Coefficient THEN
      co := a.coeff.Div(b(Coefficient).coeff); tr := a.trace.Div(b(Coefficient).trace)
   ELSE
      NEW(const); E.Init(const, b);
      co := a.coeff.Div(b); tr := a.trace.Div(const)
   END;
```

```
    NEW(c); Init(c, co(A.Coefficient), tr);
    RETURN c
  END Div;

  PROCEDURE (a: Coefficient) Write* (VAR f(*inout*): F.Formatter);
  (**
    Extension.
    Overrites: Algebraic.Object.Write.
    Writes a textual representation of a with f.
  **)
  BEGIN
    a.coeff.Write(f)
  END Write;

END TracedCoefficients.
```

```
MODULE GrobnerPolyExpressions;
(**
   Expressions of algbebraic objects.
   The operations are not executed immediatedly. They are just memorized. Calling Expand on an Expression executes all necessry
   opertions to obtain the result.
   Motivation: During Groebner Basis computation many reductions to zero are calculated. Keeping this calculations in Expressions
   gives us the possibility to do the computation only on the polznomials we need for the final result.

   Author: Gerald Slacik (gs)
*)

   IMPORT G := Grobner, P := Multivariate, PP := PowerProducts, A := Algebraic;

   TYPE
      Expression* = POINTER TO ExpressionDesc;
      ExpressionDesc* = RECORD
      (**
         Abstract.
         Expression, with actual calculation done only on demand via Expand
      **)
      res: G.Polynomial (* cache result of Expand; NIl if Expand was not called for this expression *)
      END;

      ConstExpression* = POINTER TO ConstExpressionDesc;
      ConstExpressionDesc* = RECORD (ExpressionDesc)
      (** Constant Expression represents itself; used as starting point for evaluation via Expand **)
      END;

      NormExpression* = POINTER TO NormExpressionDesc;
      NormExpressionDesc* = RECORD (ExpressionDesc)
         a: Expression
      END;
      RedExpression* = POINTER TO RedExpressionDesc;
      RedExpressionDesc* = RECORD (ExpressionDesc)
         n: P.Length;
         a, b: Expression
      END;
      SpolExpression* = POINTER TO SpolExpressionDesc;
      SpolExpressionDesc* = RECORD (ExpressionDesc)
         lcm: PP.PowerProduct;
         a, b: Expression
      END;

   VAR
      redCount-, spolCount-: LONGINT; (** counters for expressions **)
      redXCount-, spolXCount-: LONGINT; (** counters for expanded expressions **)

   PROCEDURE InitProfiling*;
   BEGIN
      redCount := 0; spolCount := 0;
      redXCount := 0; spolXCount := 0
   END InitProfiling;

   (** Expression **)

   PROCEDURE (a: Expression) Norm* (): NormExpression;
   (**
      Makes a somehow normalized (e.g. by making it primitive or by dividing it by the leading coefficient).
   **)
      VAR c: NormExpression;
   BEGIN
      NEW(c); c.a := a;
      RETURN c
   END Norm;

   PROCEDURE (a: Expression) Red* (n: P.Length; b: Expression): RedExpression;
   (**
      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.
   **)
      VAR c: RedExpression;
   BEGIN INC(redCount);
      NEW(c); c.n := n; c.a := a; c.b := b;
      RETURN c
   END Red;

   PROCEDURE (a: Expression) Spol* (b: Expression; lcm: PP.PowerProduct): SpolExpression;
   (**

      Returns the Spolynomial of a and b.
   **)
      VAR c: SpolExpression;
   BEGIN INC(spolCount);
      NEW(c); c.a := a; c.b := b; c.lcm := lcm;
      RETURN c
```

```
   END Spol;

   PROCEDURE (a: Expression) Expand* (): G.Polynomial;
   (**
      Abstract method. Must be overwritten.
      Evaluates the expression a. Caches the result. Calling Expand a second time for an expression just
      returns the cached result. The links to the operands of the expression are set to NIL for storage optimization.
   **)
   BEGIN
      HALT(A.abstractMethod)
   END Expand;

   (** ConstantExpression **)

   PROCEDURE Init* (exp: ConstExpression; const: G.Polynomial);
   (**
      Initializes the Expression exp with the constant const.

      Pre
      const # NIL 20

      Post
      exp.Expand() = const
   **)
   BEGIN
      exp.res := const
   END Init;

   PROCEDURE (a: ConstExpression) Expand* (): G.Polynomial;
   (**
      Evaluates the expression a.
   **)
   BEGIN
      RETURN a.res
   END Expand;

   (** NormExpression **)

   PROCEDURE (a: NormExpression) Expand* (): G.Polynomial;
   (**
      Extension.
      Overwrites Expression.Expand.
      Evaluates the expression a.
   **)
   BEGIN
      IF a.res = NIL THEN
         a.res := a.a.Expand(); a.a := NIL; a.res := a.res.Clone(); a.res.Norm
      END;
      RETURN a.res
   END Expand;

   (** RedExpression **)

   PROCEDURE (a: RedExpression) Expand* (): G.Polynomial;
   (**
      Extension.
      Overwrites Expression.Expand.
      Evaluates the expression a.
   **)
   BEGIN
      IF a.res = NIL THEN
         a.res := a.a.Expand(); a.a := NIL; a.res := a.res.Clone(); a.res.Red(a.n, a.b.Expand()); a.b := NIL;
         INC(redXCount)
      END;
      RETURN a.res
   END Expand;

   (** SpolExpression **)

   PROCEDURE (a: SpolExpression) Expand* (): G.Polynomial;
   (**
      Extension.
      Overwrites Expression.Expand.
      Evaluates the expression a.
   **)
      VAR h: G.Polynomial;
   BEGIN
      IF a.res = NIL THEN
         h := a.a.Expand(); a.a := NIL; a.res := h.Spol(a.b.Expand(), a.lcm); a.b := NIL;
         INC(spolXCount)
      END;
      RETURN a.res
   END Expand;

BEGIN
```

```
  InitProfiling
END GrobnerPolyExpressions.
```

```
MODULE TracedGrobnerPolys;
(**
   Traced expressions of grobner polynomials..
   Uses Expresssions to trace calculations. The operations are executed and memorized.
   Motivation: During Groebner Basis computation many reductions to zero are calculated. Keeping this calculations in Expressions
   gives us the possibility to do the computation only on the polznomials we need for the final result.

   Author: Gerald Slacik (gs)
*)

   IMPORT G := Grobner, P := Multivariate, PP := PowerProducts, E := GrobnerPolyExpressions;

   TYPE
      Polynomial* = POINTER TO PolynomialDesc;
      PolynomialDesc* = RECORD (G.PolynomialDesc)
      (**
         Grobner Polynomial, which memorizes the way it was calculated.
      **)
         gpol: G.Polynomial; (* pol on wich operations are performed *)
         trace-: E.Expression (** trace of the operations on itself **)
      END;

   (** Polynomial **)

   PROCEDURE Init* (a: Polynomial; pol: G.Polynomial; trace: E.Expression);
   (**
      Initializes the Polynomial a with the Grobner.Polynomial pol and the Expression trace as trace for a.

      Pre
      pol # NIL 20
      trace # NIL 21

      Post
      a.trace.Expand() = trace
   **)
   BEGIN
      G.Init(a, pol.pol);
      a.gpol := pol; a.trace := trace
   END Init;

   PROCEDURE (a: Polynomial) Clone* (): G.Polynomial;
   (**
      Extension.
      Overwrites Grobner.Polynomial.Clone.

      Returns a shallow copy of p.
   **)
      VAR c: Polynomial;
   BEGIN
      NEW(c); c↑ := a↑;
      RETURN c
   END Clone;

   PROCEDURE (a: Polynomial) Norm* ();
   (**
      Extension.
      Overwrites Grobner.Polynomial.Norm.

      Makes a somehow normalized (e.g. by making it primitive or by dividing it by the leading coefficient).

      Pre
      ~a.pol.IsZero() 20
   **)
   BEGIN
      a.gpol.Norm;
      Init(a, a.gpol, a.trace.Norm())
   END Norm;

   PROCEDURE (a: Polynomial) Red* (n: P.Length; b: G.Polynomial);
   (**
      Extension.
      Overwrites Grobner.Polynomial.Red.

      Reduces a with b by eliminating the power product at position n.
      Red is used to eliminate the power product a.pol.PP(n) in a by substracting a suitable multiple of b.

      Pre
      b IS SameDynamicTypeAs(a) 20
      ~a.pol.IsZero() 21
      ~b.pol.IsZero() 22
      0 <= n < a.pol.Length() 23
      a.pol.PP(n).IsMultipleOf(b.pol.PP(0)) 24
      b is normed 25
   **)
   BEGIN
```

```
        a.gpol.Red(n, b(Polynomial).gpol);
        Init(a, a.gpol, a.trace.Red(n, b(Polynomial).trace))
    END Red;

    PROCEDURE (a: Polynomial) Spol* (b: G.Polynomial; lcm: PP.PowerProduct): G.Polynomial;
    (**
        Extension.
        Overwrites Grobner.Polynomial.Spol.

        Returns the Spolynomial of a and b.

        Pre
        b IS SameDynamicTypeAs(a) 20
        ~a.pol.IsZero() 21
        ~b.pol.IsZero() 22
        a is normed 23
        b is normed 24
        lcm.Eql(a.lpp.Lcm(b.lpp)) 25

        Post
        result IS SameDynamicTypeAs(a) 60
    **)
        VAR c: Polynomial;
    BEGIN
        NEW(c); Init(c, a.gpol.Spol(b(Polynomial).gpol, lcm), a.trace.Spol(b(Polynomial).trace, lcm));
        RETURN c
    END Spol;

END TracedGrobnerPolys.
```

```
MODULE ModularGB;
(**
   Collection of Groebner Basis algorithms which try to speed up by calculating the Groebner Basis
   modulo a lucky prime and then compute the original Groebner Basis out of the modular one.

   Author: Gerald Slacik (gs)
*)

   IMPORT
      TC := TracedCoefficients, E := Expressions, SP := IntegersModSmallPrime,
      TP := TracedGrobnerPolys, GPE := GrobnerPolyExpressions, GP := GrobnerPolys, IGP := IntGrobnerPolys,
      T := GrobnerTrans, GTP := GrobnerTransPolys, G := Grobner,
      M := Matrices, L := Linear, P := Multivariate, PP := PowerProducts,
      Ga := Gauss, GaP := GaussPolys, IGaP := IntGaussPolys,
      Pol := Polynomials, I := Integers, IQ := IntegralsAndQuotients, A := Algebraic, Oberon, F := Formatters;

   CONST
      UnluckyPrime = 122;

   TYPE
      ModParam = RECORD (A.Param)
         prime: SP.IntModPrime
      END;
      FuncParam = RECORD (ModParam)
         f: A.MapFunc;
      END;
      IndetParam = RECORD (A.Param)
         indet: L.Polynomial;
         numIndets: L.Length;
         zero, one: A.Coefficient
      END;
      PlugInParam = RECORD (A.Param)
         soln: POINTER TO ARRAY OF A.Coefficient
      END;
      IDivParam = RECORD (A.Param)
         div: I.Integer
      END;
      FareyParam = RECORD (A.Param)
         bigN, pi: I.Integer
      END;

   (** TraceCoeff **)

   PROCEDURE Apply (p: G.Polynomial; VAR param(*in*): A.Param): G.Polynomial;
      VAR r: GP.Polynomial; h: A.Object;
   BEGIN
      h := p.pol.Map(param(FuncParam).f, param);
      NEW(r); G.Init(r, h(P.Polynomial));
      RETURN r
   END Apply;

   PROCEDURE TraceCoeffs (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
      VAR c: TC.Coefficient; e: E.ConstExpression; h: A.Object;
   BEGIN
      h := a.Mod(param(ModParam).prime);
      NEW(e); E.Init(e, a);
      NEW(c); TC.Init(c, h(A.Coefficient), e);
      RETURN c
   END TraceCoeffs;

   PROCEDURE ExpandCoeffs (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
      VAR h: A.Object;
   BEGIN
      h := a(TC.Coefficient).trace.Expand();
      RETURN h(A.Coefficient)
   END ExpandCoeffs;

   PROCEDURE TraceCoeff * (basis: G.Basis; prime: SP.IntModPrime; VAR time: LONGINT): G.Basis;
   (**
      Computes the groebner basis by tracing all operations on the coefficients during
      computation of the groebner basis modulo prime. The necessary operations
      are repeated over the rationals with help of the coefficient traces.
   **)
      VAR mod: FuncParam;
   BEGIN
      IF basis.Size() = 0 THEN RETURN basis END;
      time := Oberon.Time();
      mod.f := TraceCoeffs; mod.prime := prime;
      basis := basis.Map(Apply, mod);
      basis.GrobnerBasis();
      time := Oberon.Time() - time;
      mod.f := ExpandCoeffs;
      RETURN basis.Map(Apply, mod)
   END TraceCoeff;
```

```
(** Trace **)

PROCEDURE TracePolys (p: G.Polynomial; VAR param(*in*): A.Param): G.Polynomial;
   VAR gp: GP.Polynomial; tp: TP.Polynomial; pe: GPE.ConstExpression; h: A.Object;
BEGIN
   h := p.pol.Mod(param(ModParam).prime);
   NEW(gp); G.Init(gp, h(P.Polynomial));
   NEW(pe); GPE.Init(pe, p);
   NEW(tp); TP.Init(tp, gp, pe);
   RETURN tp
END TracePolys;

PROCEDURE ExpandPolys (p: G.Polynomial; VAR param(*in*): A.Param): G.Polynomial;
BEGIN
   RETURN p(TP.Polynomial).trace.Expand()
END ExpandPolys;

PROCEDURE Trace* (basis: G.Basis; prime: SP.IntModPrime; VAR time: LONGINT): G.Basis;
(**
   Computes the groebner basis by tracing all operations on the polynomials during
   computation of the groebner basis modulo prime. The necessary operations
   are repeated over the rationals with help of the polynomial traces.
**)
   VAR mod: ModParam;
BEGIN
   IF basis.Size() = 0 THEN RETURN basis END;
   time := Oberon.Time();
   mod.prime := prime;
   basis := basis.Map(TracePolys, mod);
   basis.GrobnerBasis();
   time := Oberon.Time() - time; PP.InitProfiling;
   RETURN basis.Map(ExpandPolys, mod)
END Trace;

(** Trace **)

PROCEDURE TransModP (p: G.Polynomial; VAR param(*in*): A.Param): T.Polynomial;
   VAR t: GTP.Polynomial; h: A.Object;
BEGIN
   h := p.pol.Mod(param(ModParam).prime);
   NEW(t); T.Init(t, h(P.Polynomial), NIL);
   RETURN t
END TransModP;

PROCEDURE WithIndets (a: A.Coefficient; VAR param(*inout*): A.Param): A.Coefficient;
BEGIN
   IF a.IsZero() THEN
      a := param(IndetParam).zero
   ELSIF a IS P.Polynomial THEN
      a := a(P.Polynomial).Map(WithIndets, param)
   ELSE
      a := param(IndetParam).indet.NewVar(param(IndetParam).numIndets);
      INC(param(IndetParam).numIndets)
   END;
   RETURN a
END WithIndets;

PROCEDURE WithIndetsAndLd1 (a: A.Coefficient; VAR param(*inout*): A.Param): A.Coefficient;
   VAR p: P.Polynomial; h: A.Object;
BEGIN
   IF a.IsZero() THEN
      a := param(IndetParam).zero
   ELSE
      NEW(p); P.Init(p, a(P.Polynomial).Coeff(0), a(P.Polynomial).PP(0));
      h := a.Sub(p);
      h := h(P.Polynomial).Map(WithIndets, param);
      P.Init(p, param(IndetParam).one, a(P.Polynomial).PP(0));
      h := h.Add(p); a := h(P.Polynomial)
   END;
   RETURN a
END WithIndetsAndLd1;

PROCEDURE PlugIn (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
BEGIN
   IF a.IsZero() THEN
      a := I.zero
   ELSIF a IS P.Polynomial THEN
      a := a(P.Polynomial).Map(PlugIn, param)
   ELSIF a IS L.Polynomial THEN
      a := param(PlugInParam).soln[a(L.Polynomial).VarIndex(0)]
   (* ELSE RETURN a *)
   END;
   RETURN a
END PlugIn;
```

```
PROCEDURE Expand (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
BEGIN
   IF a.IsZero() THEN
      a := I.zero
   ELSIF a IS P.Polynomial THEN
      a := a(P.Polynomial).Map(PlugIn, param)
   ELSIF a IS L.Polynomial THEN
      a := a(L.Polynomial).Eval(param(PlugInParam).soln↑)
   (* ELSE RETURN a *)
   END;
   RETURN a
END Expand;

PROCEDURE LiftTrans* (basis: G.Basis; prime: SP.IntModPrime; VAR transMatrix(*out*): M.Matrix; VAR time: LONGINT): G.Basis;
(**
   Lifts the groebner basis by computing the transformation matrix during
   computation of the groebner basis modulo prime. The coefficients of the
   polynomials in the matrix and groebner basis are replaced by indeterminants.
   The resulting linear equations are solved over the rationals. The results yield
   the coefficients of the transformation matrix and groebner basis over Q.
   Return NIL if the linear equations are not solvable (because prime is unlucky)
**)
   VAR f, t, g: M.Matrix; h: A.Object; i: M.Dimension; mod: ModParam; indetParam: IndetParam; unsolveable: BOOLEAN;
   PROCEDURE Lift (): G.Polynomial;
      VAR hp: A.Object; gp, ep: P.Polynomial; j: M.Dimension; k: L.Length; p: GP.Polynomial;
            b: Ga.Basis; gap: IGaP.Polynomial; plugInParam: PlugInParam;
   BEGIN
      indetParam.numIndets := 0; hp := WithIndetsAndLd1(g.Get(i, 0), indetParam);
      gp := hp(P.Polynomial);
      j := t.Columns(); hp := hp.Neg();
      REPEAT
            DEC(j);
            h := t.Get(i, j); h := h(P.Polynomial).Map(WithIndets, indetParam); t.Put(i, j, h(P.Polynomial));
            hp := hp.Add(h.Mul(f.Get(j, 0)))
      UNTIL j = 0;
      ep := hp(P.Polynomial);
      NEW(b); Ga.InitBasis(b); k := ep.Length();
      REPEAT DEC(k); h := ep.Coeff(SHORT(k)); NEW(gap); Ga.Init(gap, h(L.Polynomial)); b.Adjoin(gap) UNTIL k = 0;
      b.Triangularize(unsolveable);
      IF unsolveable THEN RETURN NIL END;
      k := indetParam.numIndets; NEW(plugInParam.soln, k);
      REPEAT DEC(k); plugInParam.soln[k] := indetParam.zero UNTIL k = 0;
      b.Solve(plugInParam.soln↑);
      h := gp.Map(PlugIn, plugInParam);
      NEW(p); G.Init(p, h(P.Polynomial));
      j := t.Columns();
      REPEAT
            DEC(j);
            h := t.Get(i, j);
            h := h(P.Polynomial).Map(PlugIn, plugInParam);
            t.Put(i, j, h(P.Polynomial))
      UNTIL j = 0;
      RETURN p
   END Lift;
BEGIN (* LiftTrans *)
   IF basis.Size() = 0 THEN traceMatrix := NIL; RETURN basis END;
   time := Oberon.Time();
   f := T.MatrixFrom(basis);
   mod.prime := prime;
   basis := T.Basis2TransBasis(basis, TransModP, mod);
   basis(T.Basis).GrobnerBasisAndTrans(t);
   time := Oberon.Time() - time;
   g := T.MatrixFrom(basis);
   h := g.Get(0, 0); NEW(indetParam.indet); L.Init(indetParam.indet, I.one, 0, h(P.Polynomial).ExtLvl()-1);
   indetParam.zero := I.zero; indetParam.one := I.one;
   i := g.Rows(); NEW(basis); G.InitBasis(basis);
   REPEAT DEC(i); basis.Adjoin(Lift((*f, t, g, i*))) UNTIL i = 0;
   traceMatrix := t;
   RETURN basis
END LiftTrans;

(** LiftCoeff **)

PROCEDURE Mod (p: G.Polynomial; VAR param(*in*): A.Param): G.Polynomial;
   VAR gp: GP.Polynomial; h: A.Object;
BEGIN
   h := p.pol.Mod(param(ModParam).prime);
   NEW(gp); G.Init(gp, h(P.Polynomial));
   RETURN gp
END Mod;

PROCEDURE WithIndetsWoLt (a: A.Coefficient; VAR param(*inout*): A.Param): A.Coefficient;
(* without leading term *)
   VAR p: P.Polynomial; h: A.Object;
BEGIN
```

```
    IF a.IsZero() THEN
        a := param(IndetParam).zero
    ELSE
        NEW(p); P.Init(p, a(P.Polynomial).Coeff(0), a(P.Polynomial).PP(0));
        h := a.Sub(p);
        h := h(P.Polynomial).Map(WithIndets, param);
        a := h(P.Polynomial)
    END;
    RETURN a
END WithIndetsWoLt;

PROCEDURE IDiv (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
    VAR h: A.Object;
BEGIN
    IF a IS I.Integer THEN
        a := a(I.Integer).IDiv(param(IDivParam).div)
    ELSIF a IS IQ.Quotient THEN
        h := a(IQ.Quotient).num(I.Integer).IDiv(param(IDivParam).div);
        h := h.Div(a(IQ.Quotient).den);
        a := a(IQ.Quotient)
    ELSE
        a := a(P.Polynomial).Map(IDiv, param)
    END;
    RETURN a
END IDiv;

PROCEDURE FareyFraction (a: A.Coefficient; VAR param(*in*): A.Param): A.Coefficient;
    VAR n, d: I.Integer; n1, d1, q, r: IQ.Integral; h: A.Object;
BEGIN
    IF a.IsZero() THEN
    ELSIF a IS P.Polynomial THEN
        a := a(P.Polynomial).Map(FareyFraction, param)
    ELSE
        n := a(I.Integer); d := I.one; n1 := param(FareyParam).pi; d1 := I.zero; h := d;
        IF n.IsNeg() THEN h := n.Neg(); n := h(I.Integer); h := d.Neg(); d := h(I.Integer) END;
        (*IF n.IsNeg() THEN h := param(FareyParam).pi.Add(n); n := h(I.Integer) END;*)
        WHILE param(FareyParam).bigN.Lss(n) OR param(FareyParam).bigN.Lss(h(I.Integer)) (* h = d.Abs() *) DO
                n1.QuoRem(n, q, r);
                n1 := n; n := r(I.Integer);
                h := d1.Sub(q.Mul(d));
                d1 := d; d := h(I.Integer); h := d.Abs()
        END;
        h := n.Div(d); a := h(A.Coefficient)
    END;
    RETURN a
END FareyFraction;

PROCEDURE LiftCoeff* (basis: G.Basis; prime: SP.IntModPrime; coeffBound: I.Integer; VAR time, steps: LONGINT): G.Basis;
(**
    Lifts the groebner basis by computing the groebner basis modulo prime, the
    transformation matrix R and syzygy matrix S (with only the necessary S-pols).
    It makes an ansatz with new indeterminates for the coefficients in G0, R0 and S0.
    The resulting linear equations are solved over Zp. The result yields
    the coefficients of the increment of the transformation matrix, syzygy matrix
    and groebner basis in a lifting step. coeffBound is bound for the coefficients
    in the result basis G and determines how many lifting steps are needed.
    Return NIL if the linear equations are not solvable (because prime is unlucky)
**)
    VAR h, gh, rh: A.Object; fi, g, r, gi, ri, gi1, e: M.Matrix; i: L.Length; pi: I.Integer;
        bf, bg (* mod prime*): G.Basis; eqns: Ga.Basis;
        plugInParam: PlugInParam; idivParam: IDivParam; fareyParam: FareyParam; modParam: ModParam; indetParam: IndetParam;
        unsolveable: BOOLEAN;
    PROCEDURE Join(a, b: M.Matrix): M.Matrix;
        VAR c: M.Matrix; ra, rb, ca, i: M.Dimension;
    BEGIN
        ra := a.Rows(); rb := b.Rows(); ca := a.Columns();
        NEW(c); M.Init(c, ra+rb, ca);
        REPEAT
                DEC(rb); i := ca;
                REPEAT DEC(i); c.Put(ra+rb, i, b.Get(rb, i)) UNTIL i = 0
        UNTIL rb = 0;
        REPEAT
                DEC(ra); i := ca;
                REPEAT DEC(i); c.Put(ra, i, a.Get(ra, i)) UNTIL i = 0
        UNTIL ra = 0;
        RETURN c
    END Join;

    PROCEDURE AddZeros(a: M.Matrix; n: M.Dimension): M.Matrix;
        VAR c: M.Matrix; i: M.Dimension;
    BEGIN
        i := a.Rows();
        NEW(c); M.Init(c, i+n, a.Columns());
        REPEAT DEC(n); c.Put(i+n, 0, I.zero) UNTIL n = 0;
        REPEAT DEC(i); c.Put(i, 0, a.Get(i, 0)) UNTIL i = 0;
```

```
      RETURN c
   END AddZeros;

   PROCEDURE SetupEqns (a, b: M.Matrix; p: SP.IntModPrime): Ga.Basis;
      VAR pa, pb: P.Polynomial; ha, hb: A.Object; ja, jb: L.Length; ppa: PP.PowerProduct;
            i: M.Dimension; gap: GaP.Polynomial; basis: Ga.Basis;
   BEGIN
      NEW(basis); Ga.InitBasis(basis);
      i := a.Rows();
      REPEAT
            DEC(i);
            ha := a.Get(i, 0);
            pa := ha(P.Polynomial); ja := pa.Length();
            hb := b.Get(i, 0);
            IF hb.IsZero() THEN
               REPEAT
                  DEC(ja); ha := pa.Coeff(ja); ppa := pa.PP(ja);
                  NEW(gap); Ga.Init(gap, ha(L.Polynomial));
                  basis.Adjoin(gap)
               UNTIL ja = 0
            ELSE
               pb := hb(P.Polynomial); jb := pb.Length()-1;
               REPEAT
                  DEC(ja); ha := pa.Coeff(ja); ppa := pa.PP(ja);
                  IF (jb >= 0) & ppa.Eql(pb.PP(jb)) THEN
                     hb := pb.Coeff(jb); DEC(jb);
                     ha := ha.Sub(hb.Mod(p))
                  END;
                  NEW(gap); Ga.Init(gap, ha(L.Polynomial));
                  basis.Adjoin(gap)
               UNTIL ja = 0
            END
      UNTIL i = 0;
      RETURN basis
   END SetupEqns;

   PROCEDURE Matrix2Basis(a: M.Matrix): G.Basis;
      VAR h: A.Coefficient; p: GP.Polynomial; n: M.Dimension; basis: G.Basis;
   BEGIN
      n := a.Rows();
      NEW(basis); G.InitBasis(basis);
      REPEAT
            DEC(n); h := a.Get(n, 0);
            NEW(p); G.Init(p, h(P.Polynomial));
            basis.Adjoin(p)
      UNTIL n = 0;
      RETURN basis
   END Matrix2Basis;

BEGIN (* LiftCoeff *)
   IF basis.Size() = 0 THEN RETURN basis END;
   time := Oberon.Time();
   modParam.prime := prime; bg := basis.Map(Mod, modParam); bf := bg.Clone(); bg.CopyParts;
   bg.GrobnerBasis();
   time := Oberon.Time() - time;
   gi := T.MatrixFrom(bg);
   fareyParam.bigN := coeffBound; fareyParam.pi := prime.Prime(); idivParam.div := fareyParam.pi; steps := 0;
   (* coeffBound := 2*coeffBound^2+1 *)
   h := coeffBound.Mul(coeffBound); h := h.Add(h); h := h.Add(I.one); coeffBound := h(I.Integer);
   IF fareyParam.pi.Lss(coeffBound) THEN
      fi := T.MatrixFrom(basis);
      bg := T.Basis2TransBasis(bg, GTP.Poly2TransPoly, modParam);
      ri := bg(T.Basis).TransformationOf(bf); bf := NIL;
      r := bg(T.Basis).Syzygy(FALSE); bg := NIL;
      IF r # NIL THEN ri := Join(ri, r); fi := AddZeros(fi, r.Rows()) END;

      indetParam.numIndets := 0;
      h := prime.One(); indetParam.one := h(SP.IntModPrime);
      h := prime.Zero(); indetParam.zero := h(SP.IntModPrime);
      h := fi.Get(0, 0); i := h(P.Polynomial).ExtLvl()-1;
      NEW(indetParam.indet); L.Init(indetParam.indet, indetParam.one, 0, i);
      g := gi.Map(WithIndetsWoLt, indetParam); r := ri.Map(WithIndets, indetParam);

      h := r.Mul(gi); h := h.Add(ri.Mul(g)); e := h(M.Matrix);
      h := gi.Demod(); gi := h(M.Matrix); h := ri.Demod(); ri := h(M.Matrix);
      h := fi.Sub(ri.Mul(gi)); fi := h(M.Matrix);

      NEW(plugInParam.soln, indetParam.numIndets);
      LOOP
            INC(steps);
            fi := fi.Map(IDiv, idivParam);
            eqns := SetupEqns(e, fi, prime);
            eqns.Triangularize(unsolveable);
            IF unsolveable THEN RETURN NIL END;
            i := indetParam.numIndets; REPEAT DEC(i); plugInParam.soln[i] := indetParam.zero UNTIL i = 0;
```

```
            eqns.Solve(plugInParam.soln↑);eqns := NIL;
            i := indetParam.numIndets;
            REPEAT DEC(i); h := plugInParam.soln[i].Demod(); plugInParam.soln[i] := h(A.Coefficient) UNTIL i = 0;
            pi := fareyParam.pi; h := fareyParam.pi.Mul(idivParam.div); fareyParam.pi := h(I.Integer);
            (* gi1 := g^(i-1) + p^i*g' *)
            gh := g.Map(PlugIn, plugInParam);
            h := gi.Add(gh.Mul(pi));
            gi1 := h(M.Matrix);
            IF coeffBound.Lss(fareyParam.pi) THEN EXIT END;
            rh := r.Map(PlugIn, plugInParam);
            (* fi := f^(i-1) - (r'*g^i + r^(i-1)*g' *)
            h := rh.Mul(gi1); h := h.Add(ri.Mul(gh)); h := fi.Sub(h); fi := h(M.Matrix);
            gi := gi1; gh := NIL;
            (* ri := r^(i-1) + p^i*r' *)
            rh := ri.Add(rh.Mul(pi)); ri := rh(M.Matrix);
            pi := NIL
        END
    ELSE
        h := gi.Demod(); gi1 := h(M.Matrix)
    END;
    gi := gi1.Map(FareyFraction, fareyParam);
    RETURN Matrix2Basis(gi);
  END LiftCoeff;

END ModularGB.
```

Bibliography

[Boege et al., 1986] Boege, W., Gebauer, R., and Kredel, H. (1986). Some examples for solving systems of algebraic equations by calculating Gröbner Bases. *Journal of Symbolic Computation*, 2:83–98.

[Brown, 1971] Brown, W. S. (1971). On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. *JACM*, 18(4):478–504.

[Buchberger, 1965] Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, Universität Innsbruck (Austria).

[Buchberger, 1970] Buchberger, B. (1970). An algorithm for finding a basis for solvability of algebraic systems of equations. *Aequationes mathematicae*, 4(3):374–383.

[Buchberger, 1976] Buchberger, B. (1976). A Theoretical Basis for the Reduction of Polynomials to Canonical Form. *ACM SIGSAM Bull.*, 10(4):19–29.

[Buchberger, 1985] Buchberger, B. (1985). Gröbner bases: An algorithmic method in polynomial ideal theroy. In Bose, N. K., editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publ. Comp.

[Ebert, 1983] Ebert, G. L. (1983). Some comments on the modular approach to Gröbner Bases. *ACM SIGSAM Bulletin*, 17(2):28–32.

[Faugere et al., 1986] Faugere, J. C., p. Gianni, Lazard, D., and Mora, T. (1986). Efficient computation of zerodimensional Gröbner Bases by change of ordering. Preprint.

[Furukawa et al., 1986] Furukawa, A., Sasaki, T., and Kobayashi, H. (1986). Gröbner basis of a module over $K[x_1, \ldots, x_n]$ and polynomial solutions of a system of linear equations. In Char, B. W., editor, *Proc. SYMSAC'86. ACM*, pages 222–224.

[Gräbe, 1993] Gräbe, H. G. (1993). On Lucky Primes. *Journal of Symbolic Computation*, 15(2):199–209.

[Gröbner, 1949] Gröbner, W. (1949). *Moderne algebraische Geometrie.* Springer-Verlag, Wien-Innsbruck.

[Hilbert, 1890] Hilbert, D. (1890). Über die Theorie der algebraischen Formen. *Math. Annalen*, 36:473–534.

[Kalkbrener, 1987] Kalkbrener, M. (1987). Applications of Gröbner Bases: Solution of Algebraic Equations and Decomposition of Radicals. Diploma thesis, RISC-Linz, J. Kepler University Linz, Austria, Europe.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming*, volume 1/Fundamental Algorithms. Addison-Weseley, 2nd edition.

[Kornerup and Gregory, 1983] Kornerup, P. and Gregory, R. T. (1983). Mapping integers and Hensel codes onto Farey fractions. *BIT*, 23:9–20.

[Lauer, 1983] Lauer, M. (1983). Computing by homomorphic imgaes. In Buchberger, B., Collins, G. E., and Loos, R., editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 139–169. Springer-Verlag, Heidelberg.

[Miola and Yun, 1974] Miola, A. and Yun, D. Y. Y. (1974). The computational aspects of Hensel-type univariate greatest common divisor algorithms. In Jenks, R. D., editor, *Proc. EUROSAM'74. SIGSAM Bulletin*, volume 8/3, pages 46–56.

[Möller and Mora, 1986] Möller, H. M. and Mora, F. (1986). New constructive methos in classical ideal theroy. *J. of Algebra*, 100:138–178.

[Moses and Yun, 1973] Moses, J. and Yun, D. Y. Y. (1973). The EZGCD algorithm. *Proc. ACM Annual Conference*, pages 156–166.

[Mössenböck, 1993] Mössenböck, H. (1993). *Objektorientierte Programmierung in Oberon-2*. Springer-Verlag, Heidelberg.

[Musser, 1975] Musser, D. R. (1975). Multivariate polynomial factorization. *JACM*, 22:291–308.

[Reiser, 1991] Reiser, M. (1991). *The Oberon System - User Guide and Programmer's Manual*. Addison-Wesley.

[Reiser and Wirth, 1992] Reiser, M. and Wirth, N. (1992). *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley.

[Traverso, 1988] Traverso, C. (1988). Gröbner trace algorithms. In *Proceedings ISSAC'88, Lecture Notes in Comp. Sci.*, volume 358, pages 125–138.

[Trinks, 1984] Trinks, W. (1984). On improving approximate results of Buchberger's algorithm by Newton's method. *SIGSAM Bull.*, 18(3):7–11.

[van der Waerden, 1967] van der Waerden (1967). *Algebra II*. Springer-Verlag, Heidelberg.

[Wang, 1978] Wang, P. S. H. (1978). An improved multivariate polynomial factorization algorithm. *Math. Comp.*, 32:1215–1231.

[Wang and Rothschild, 1975] Wang, P. S. H. and Rothschild, L. P. (1975). Factoring multivariate polynomials over the integers. *Math. Comp.*, 29:935–950.

[Winkler, 1987a] Winkler, F. (1987a). A recursive method for computing a Gröbner basis of a module in $K[x_1, \ldots, x_\nu]^r$. Technical Report 87-9, RISC-Linz, J. Kepler University Linz, Austria, Europe.

[Winkler, 1987b] Winkler, F. (1987b). p-adic methods for to the Computation of Gröbner Bases. In *EUROCAL'87*, Leipzig, GDR.

[Winkler, 1988] Winkler, F. (1988). A p-adic Approach to the Computation of Gröbner Bases. *Journal of Symbolic Computation*, 6:287–304.

[Winkler, 1992] Winkler, F. (1992). *Computer-Algebra für Fortgeschrittene.* RISC-Linz, J. Kepler University Linz, Austria, Europe.

[Winkler, 1996] Winkler, F. (1996). *Polynomial Algorithms in Computer Algebra.* Springer Verlag.

[Winkler and Buchberger, 1979] Winkler, F. and Buchberger, B. (1979). Miscellaneous results on the construction of Gröbner Bases for polynomial ideals. Technical report, RISC-Linz, J. Kepler University Linz, Austria, Europe.

[Winkler et al., 1985] Winkler, F., Buchberger, B., Lichtenberger, F., and Rolletschek, H. (1985). "Algorithm 628" - An algoritm for constructing canonical bases of polynomial ideals. *ACM Trans. on Math. Software*, 11:66–78.

[Wirth and Gutknecht, 1992] Wirth, N. and Gutknecht, J. (1992). *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley.

[Zassenhaus, 1969] Zassenhaus, H. (1969). On Hensel factorization. In Jenks, R. D., editor, *Proc. EUROSAM'74. SIGSAM Bulletin*, volume 1, pages 46–56.