Gröbner Bases: A Characterization by Syzygy Completeness and an Implementation [1]

Wolfgang Windsteiger
RISC-Linz

—

Research Institute for Symbolic Computation
Johannes Kepler Universität Linz
A-4040 Linz, Austria
Tel: +43 (7236) 3231-27, Fax: +43 (7236) 3231-30
email: wwindste@risc.uni-linz.ac.at

—

December 24, 1991

[1] This work was sponsored by the Gröbner Bases project of the *Bundesministerium für Wissenschaft und Forschung, Austria.*

Contents

Acknowledgment

I deeply want to thank Prof. Dr. Bruno Buchberger for all his help and advice during my work and especially for the enormous effort he has put in creating and maintaining the RISC institute, which is a both fascinating and inspiring environment for doing research.

Furthermore, I want to thank all other colleagues at RISC for all the valuable discussions, in particular all members of the Computer Algebra Lab and the Softautomation Lab, and all people at the institute who spend lot of their time to keep the whole environment in order.

This work was sponsored by the Gröbner Bases project of the *Bundesministerium für Wissenschaft und Forschung, Austria.*

Abstract

Gröbner bases are well-known in polynomial ideal theory. Many problems for ideals can easily be solved after computing a Gröbner basis for the ideal. In this paper we treat a characterization of Gröbner bases by a certain completeness property of the syzygies of the leading monomials of the polynomials in the ideal. We particularly emphasize the one-to-one correspondence between this characterization of Gröbner bases and the traditional characterization by reducibility of polynomials in the ideal.

Furthermore, we report on an implementation of some variants of the Buchberger algorithm for computing Gröbner bases for polynomials over various coefficient domains with respect to several different orderings of monomials.

Chapter 1

Theory of Gröbner Bases

1.1 Introduction

The notion of Gröbner bases and an algorithm for actually computing Gröbner bases was originally introduced in [Buchberger, 1965]. Since then the method was improved, extended, modified and applied by himself (see [Buchberger, 1970], [Buchberger, 1976a], [Buchberger, 1976b], [Buchberger, 1979], [Buchberger, 1983] and [Buchberger, 1985], [Buchberger, 1987a] for survey articles on Gröbner bases) and many other authors. See for example [Mora and Möller, 1986] for a list of eleven equivalent definitions of Gröbner bases.

In this chapter, we present a characterization of Gröbner bases by a property of the syzygies of the leading terms of the polynomials, which first appeared in the literature in [Möller, 1985].

Given a vector of polynomials G, this characterization states that G is a Gröbner basis if and only if *all* syzygies of the leading monomials of the polynomials in G are *extensible* (see Definition 2.19). Roughly this means that they can be transformed to syzygies of G by *adjoining only lower terms.* This property can be reduced to the equivalent statement that *all elements of a basis* of the module of syzygies of the leading monomials are extensible (see Lemma 2.24). In fact, we can always construct such a basis, which has a very simple structure, namely each element in the basis contains exactly two non-vanishing components (see Lemma 2.13).

In addition, we point out the connection between the concept of extensibility of the syzygies of the leading monomials and the original concept for characterizing Gröbner bases by reducibility of polynomials in the generated ideal. Since the concepts are in a very close relationship, we discuss some connections in Section 1.4. In this context, we give an *algorithm* for constructing the extension of a syzygy by reducing a polynomial with respect to a vector of polynomials.

However, *all proofs* concerning extensibility of syzygies are done *without utilizing the concept of reducibility of polynomials.*

In the last section of this chapter we discuss *orderings on power products* that can be used for computing a Gröbner basis for a given ideal. Since the computing time depends heavily on the ordering on monomials within a polynomial and on the sequence of reductions (which is also determined by an ordering on power products) – see the time statistics in Chapter 2 – we describe a general setting for admissible orderings on power products, which was stated and proved in the literature in [Robbiano, 1985].

1.2 Characterization of Gröbner Bases

Throughout this paper we denote by $T = \{x_1^{i_1} \cdot \ldots \cdot x_n^{i_n} | (i_1, \ldots, i_n) \in \mathbb{N}_0^n\}$ the set of terms (or power products), by $M = \{c \cdot t | c \in K, t \in T\}$ the set of monomials and by $P = K[x_1, \ldots, x_n]$, the set of polynomials in n indeterminates $x_1, \ldots, x_n$ over an arbitrary field K. We will use the typed variables $M_1, \ldots, M_r$ for monomials in M, $G_1 \ldots, G_r$ for polynomials in P and $G = (G_1, \ldots, G_r)$ for a vector in P^r. Without loss of generality we assume that all G_i's are normalized, i.e. their leading coefficients are 1. This is no restriction since we want to study the ideal generated by these polynomials, which is the same when all coefficients of a polynomial are divided by its leading coefficient.

1.2.1 The Notion of Gröbner Bases

When dealing with Gröbner bases, one has to fix a total ordering $<_T$ on the set T that satisfies the following conditions for all terms $r, s, t \in T$:

$1 <_T t$
if $r <_T s$ then $r \cdot t <_T s \cdot t$.

We call these orderings *admissible orderings*. Examples for such orderings are the *total degree lexical ordering*, the *total degree inverse lexical ordering* or the *purely lexical ordering*. See Section 1.6 for detailed descriptions of these (and other) admissible orderings.

Definition 2.1 (Basic Notions for Monomials)
For a monomial $m = c \cdot t \in M$ we define

$$\text{coef}(m) = c \text{ and}$$

$$\text{pp}(m) = t.$$

Given two monomials $m_1 = c_1 \cdot t_1, m_2 = c_2 \cdot t_2 \in M$, we define the *least common multiple* of m_1 and m_2 as follows:

$$\text{lcm}(m_1, m_2) = c_1 \cdot c_2 \cdot \text{lcm}(t_1, t_2).$$

Definition 2.2 (Basic Notions for Polynomials)
Given a polynomial $f = \sum_{i=0}^{m} c_i \cdot t_i$ such that $t_0 <_T t_1 <_T \ldots <_T t_m$, $c_i \in K$,

$c_i \neq 0$, $t_i \in T$ for all $0 \leq i \leq m$. We use the convention that, for any term τ_i with free variable i, $\sum_{i=l}^{h} \tau_i = 0$ for $h < l$. For $f \neq 0$, we call

t_m the *leading term* of f w.r.t. $<_T$ and denote it by $\mathrm{lt}_{<_T}(f)$ or
the *leading power product* of f w.r.t. $<_T$,
c_m the *leading coefficient* of f w.r.t. $<_T$ ($\mathrm{lc}_{<_T}(f)$),
$c_m \cdot t_m$ the *leading monomial* of f w.r.t. $<_T$ ($\mathrm{lm}_{<_T}(f)$),
$\sum_{i=0}^{m-1} c_i \cdot t_i$ the *remaining polynomial* of f w.r.t. $<_T$ ($\mathrm{rp}_{<_T}(f)$), and
$T_f = \{t_0, \ldots, t_m\}$ the *set of terms occurring in f*.

In addition, we define $\mathrm{lm}_{<_T}(0) = \mathrm{lc}_{<_T}(0) = \mathrm{rp}_{<_T}(0) = 0$, $T_0 = \{\}$.

If no confusions arise, we will omit the index $<_T$ and denote the leading term, the leading power product, leading coefficient, the leading monomial and the remaining polynomial of f by $\mathrm{lt}(f)$, $\mathrm{lpp}(f)$, $\mathrm{lc}(f)$, $\mathrm{lm}(f)$ and $\mathrm{rp}(f)$ respectively.

A term t *occurs in f* iff $t \in T_f$ otherwise we say t *vanishes in f*.
Furthermore, we define

$$\mathrm{coef}(t, f) = \begin{cases} c_i & \text{if } t = t_i \in T_f \\ 0 & \text{if } t \notin T_f. \end{cases}$$

A monomial m *occurs in f* iff
$\mathrm{pp}(m)$ occurs in f and
$\mathrm{coef}(m) = \mathrm{coef}(\mathrm{pp}(m), f)$.

Given two vectors of polynomials $g = (g_1, \ldots, g_r)$ and $h = (h_1, \ldots, h_r)$. By
$\langle g, h\rangle$ we denote the usual scalar product in the r-dimensional vectorspace P^r, i.e.
$\langle g, h\rangle = \sum_{i=1}^{r} g_i \cdot h_i$.

Finally, for every unary function F on P and every $(g_1, \ldots, g_r) \in P^r$, we define the vector-extension of F, i.e.
$F((g_1, \ldots, g_r)) := (F(g_1), \ldots, F(g_r))$,
(e.g. $\mathrm{lm}(G) = (\mathrm{lm}(G_1), \ldots, \mathrm{lm}(G_r))$).

Definition 2.3 (Ideal Generated)
By $\mathrm{Ideal}(G)$ we denote the *ideal generated* by G, i.e.

$$\mathrm{Ideal}(G) = \{\sum_{i=1}^{r} p_i \cdot G_i | p_i \in P\}.$$

After these basic notions for polynomials we give a first definition of Gröbner bases in terms of Gröbner representations.

Definition 2.4 (Gröbner Representation)
$\sum_{i=1}^{r} g_i \cdot G_i$ is called a *Gröbner representation of* f iff
$f = \sum_{i=1}^{r} g_i \cdot G_i$ and
for all $1 \leq j \leq r$
$g_j \in P$ and
if $g_j \neq 0$ then $\mathrm{lt}(g_j \cdot G_j) \leq_T \mathrm{lt}(f)$.

Definition 2.5 (Characterization of Gröbner Bases)
G is a *Gröbner basis* iff (GB)
All $f \in \mathrm{Ideal}(G)$ have a Gröbner representation. (GR)

In order to give the desired characterization of Gröbner bases we have to define syzygies of polynomial vectors and introduce the notion of homogeneity of monomial vectors.

1.2.2 Syzygies and Homogeneous Vectors of Monomials

Definition 2.6 (Syzygies)
The elements of the set

$$S_G = \{(g_1, \ldots, g_r) \in P^r \,|\, \sum_{i=1}^{r} g_i \cdot G_i = 0\}$$

are called the *syzygies of* G.

We omit the easy proof of the following lemma.

Lemma 2.7
For every vector of polynomials G, the syzygies of G form a module over P (a P-module).

Definition 2.8 (Linear Combination, Linear Closure, Module Basis)
Given a ring $\mathcal{R}$ and a module $\mathcal{M}$ over $\mathcal{R}$. Let $m_1, \ldots, m_l \in \mathcal{M}$.
m is a *linear combination* of $m_1, \ldots, m_l$ iff $m = \sum_{i=1}^{l} r_i \cdot m_i$, $r_i \in \mathcal{R}$.

The set

$$\mathrm{L}(\mathcal{B}) = \{\sum_{b_i \in \mathcal{B}} r_i \cdot b_i | r_i \in \mathcal{R}, r_i \neq 0 \text{ for only finitely many } i\}$$

is called *linear closure* of $\mathcal{B}$.
$\mathcal{B}$ is a *basis* of $\mathcal{M}$ iff $\mathcal{B} \subseteq \mathcal{M}$ and $\mathrm{L}(\mathcal{B}) = \mathcal{M}$.

Definition 2.9 (S-vectors)
Given monomials $M_1, \ldots, M_r$. Then we define for each pair M_i, M_j the *S-vector* of M_i and M_j, i.e.

$$\text{S-vector}(M_i, M_j) = \frac{\mathrm{lcm}(M_i, M_j)}{M_i} \cdot e_i - \frac{\mathrm{lcm}(M_i, M_j)}{M_j} \cdot e_j,$$

where e_k denotes the k-th canonical unit vector of M^r.

Definition 2.10 (Homogeneous Vectors of Monomials)
Given terms $T_1, \ldots, T_r, t \in T$. An element $(m_1, \ldots, m_r) \in M^r$ is called *T-homogeneous of degree t* (w.r.t. $T_1, \ldots, T_r$) iff

for every $1 \leq i \leq r$ with $m_i \neq 0$ there exists a $c_i \in K$ s.t.
$m_i \cdot T_i = c_i \cdot t$.

$m = (m_1, \ldots, m_r) \in M^r$ is *T-homogeneous* (w.r.t. $T_1, \ldots, T_r$) iff there exists a $t \in T$ s.t. m is T-homogeneous of degree t (w.r.t. $T_1, \ldots, T_r$).
A set of vectors of polynomials S is called *T-homogeneous* (w.r.t. $T_1, \ldots, T_r$) iff every element $g \in S$ can be expressed as a sum of elements of S that are T-homogeneous (w.r.t. $T_1, \ldots, T_r$).

Notation: By $\mathrm{NZ}(m)$ we denote the number of non-zero coordinates of a T-homogeneous vector m.

Remark 2.11
Given monomials $M_1, \ldots, M_r$. For all M_i, M_j, S-vector(M_i, M_j) is T-homogeneous of degree $\mathrm{lcm}(\mathrm{pp}(M_i), \mathrm{pp}(M_j))$ w.r.t. $M_1, \ldots, M_r$.

Remark 2.12
If a T-homogeneous set B constitutes a basis for a module in P^r, it is still a basis after canceling all non-homogeneous elements, because, by definition, every non-homogeneous element in B can again be expressed as a sum of homogeneous basis elements. Hence, w.l.o.g. a T-homogeneous basis is always taken to be reduced in this sense, i.e. it contains only T-homogeneous

elements.

For the syzygies of the leading monomials $S_{\mathrm{lm}(G)}$ we know a basis due to the following lemma.

Lemma 2.13
The set

$$B_M = \{\text{S-vector}(M_i, M_j) | 1 \le i < j \le r\}$$

constitutes a (finite) basis of the module $S_{(M_1,\ldots,M_r)}$.

For the proof of this lemma we need the following auxiliary lemma.

Lemma 2.14
Every T-homogeneous $m = (m_1, \ldots, m_r) \in M^r$ (w.r.t. $\mathrm{pp}(M_1), \ldots, \mathrm{pp}(M_r)$) with $\sum_{i=1}^{r} m_i \cdot M_i = 0$ can be represented as a linear combination of elements in B_M.

Proof: We fix an arbitrary $t \in T$ and monomials $M_1, \ldots, M_r \in M$. By $M^{(l)}$ we denote the set of all $m = (m_1, \ldots, m_r) \in M^r$ such that

m is T-homogeneous of degree t w.r.t. $\mathrm{pp}(M_1), \ldots, \mathrm{pp}(M_r)$,
$\sum_{i=1}^{r} m_i \cdot M_i = 0$, and
$\mathrm{NZ}(m) = l$.

For all $l \ge 2$ and $m \in M^{(l)}$ we denote by $i_1, \ldots, i_l$ the indices of all non-zero components of m.

Since m is T-homogeneous of degree t w.r.t. $\mathrm{pp}(M_1), \ldots, \mathrm{pp}(M_r)$, there exist $c_{i_1}, \ldots, c_{i_l} \in K$ such that

$$m_{i_j} \cdot M_{i_j} = c_{i_j} \cdot t, \text{ for all } 1 \le j \le l. \tag{1.1}$$

Furthermore, t must be a multiple of $\mathrm{lcm}(M_{i_v}, M_{i_w})$, for all $1 \le v < w \le l$. Thus, for arbitrary $c_{v,w} \in K$, there exist monomials $m_{v,w} \in M$ with

$$c_{v,w} \cdot t = m_{v,w} \cdot \mathrm{lcm}(M_{i_v}, M_{i_w}). \tag{1.2}$$

We show by induction over l, that

for every $2 \le l \le r$
every $m \in M^{(l)}$ can be represented as a linear combination of S-vectors in B_M.

First assume $l = 2$: Let m be an arbitrary element in $M^{(2)}$. We have

$$\underbrace{m_{i_1} \cdot M_{i_1}}_{\substack{= c_{i_1} \cdot t \\ \text{by (1.1)}}} + \underbrace{m_{i_2} \cdot M_{i_2}}_{\substack{= c_{i_2} \cdot t \\ \text{by (1.1)}}} = 0.$$

Hence,

$$c_{i_1} = -c_{i_2}. \tag{1.3}$$

By (1.2) we can choose $m_{1,2}$ such that

$$\begin{aligned} & m - m_{1,2} \cdot \text{S-vector}(M_{i_1}, M_{i_2}) = \\ & \quad (\underbrace{m_{i_1}}_{\substack{\frac{c_{i_1} \cdot t}{M_{i_1}} \\ \text{by (1.1)}}} - \underbrace{\frac{m_{1,2} \cdot \operatorname{lcm}(M_{i_1}, M_{i_2})}{M_{i_1}}}_{\frac{c_{i_1} \cdot t}{M_{i_1}} \text{ by (1.2)}}) \cdot e_{i_1} + (\underbrace{m_{i_2}}_{\substack{\frac{c_{i_2} \cdot t}{M_{i_2}} \\ \text{by (1.1)}}} + \underbrace{\frac{m_{1,2} \cdot \operatorname{lcm}(M_{i_1}, M_{i_2})}{M_{i_2}}}_{-\frac{c_{i_2} \cdot t}{M_{i_2}} \text{ by (1.2) and (1.3)}}) \cdot e_{i_2} \\ & \quad = 0 \end{aligned}$$

Hence, $m = m_{1,2} \cdot \text{S-vector}(M_{i_1}, M_{i_2})$ and therefore m is a (trivial) linear combination of S-vectors.

Let now $2 < l \leq r$ and assume that every $\bar{m} \in M^{(\bar{l})}$, $\bar{l} < l$, can be represented as a linear combination of S-vectors. Let $m \in M^{(l)}$ arbitrary. By (1.2), we can choose $m_{1,2}$ such that

$$\begin{aligned} & \bar{m} := m - m_{1,2} \cdot \text{S-vector}(M_{i_1}, M_{i_2}) = \\ & \quad \overbrace{(\underbrace{m_{i_1}}_{\substack{\frac{c_{i_1} \cdot t}{M_{i_1}} \\ \text{by (1.1)}}} - \underbrace{\frac{m_{1,2} \cdot \operatorname{lcm}(M_{i_1}, M_{i_2})}{M_{i_1}}}_{\frac{c_{i_1} \cdot t}{M_{i_1}} \text{ by (1.2)}})}^{=0} \cdot e_{i_1} + \\ & \quad + (m_{i_2} + \frac{m_{1,2} \cdot \operatorname{lcm}(M_{i_1}, M_{i_2})}{M_{i_2}}) \cdot e_{i_2} + \sum_{j=3}^{l} m_{i_j} \cdot e_{i_j}. \end{aligned}$$

From this representation we see that $\text{NZ}(\bar{m}) = \bar{l}$, with $\bar{l} < l$. Since both m and $m_{1,2} \cdot \text{S-vector}(M_{i_1}, M_{i_2})$ are elements of $S_{(M_1,\ldots,M_r)}$ and are T-homogeneous of degree t w.r.t. $\text{pp}(M_1), \ldots \text{pp}(M_r)$, also $\bar{m}$ is in $S_{(M_1,\ldots,M_r)}$, i.e. $\sum_{i=1}^{r} \bar{m}_i \cdot M_i = 0$, and is T-homogeneous of degree t w.r.t. $\text{pp}(M_1), \ldots \text{pp}(M_r)$. Hence,

$$\bar{m} \in M^{(\bar{l})} \text{ with } \bar{l} < l.$$

By the induction hypothesis, $\bar{m}$ can be represented as a linear combination of S-vectors, and therefore also

$$m = \bar{m} + m_{1,2} \cdot \text{S-vector}(M_{i_1}, M_{i_2})$$

is a linear combination of S-vectors. □

PROOF (of Lemma 2.13): We have to show that

$$B_M \subseteq S_{(M_1,\ldots,M_r)} \text{ and}$$

$$\mathrm{L}(B_M) = S_{(M_1,\ldots,M_r)}.$$

$B_M \subseteq S_{(M_1,\ldots,M_r)}$ is trivial and, since $S_{(M_1,\ldots,M_r)}$ is a module, every linear combination of elements in B_M is again in $S_{(M_1,\ldots,M_r)}$, therefore

$$\mathrm{L}(B_M) \subseteq S_{(M_1,\ldots,M_r)}.$$

It remains to show that $\mathrm{L}(B_M) \supseteq S_{(M_1,\ldots,M_r)}$.

For $g = (g_1, \ldots, g_r) \in S_{(M_1,\ldots,M_r)}$ we define for $g \neq (0, \ldots, 0)$

$$t_g := \max\{t \in T \mid \text{there exists an } 1 \leq i \leq r \text{ s.t. } t \text{ occurs in } g_i \cdot M_i\} \text{ and}$$

$$g_{\max} := (g_{\max}^{(1)}, \ldots, g_{\max}^{(r)}) \in M^r$$

with

$$g_{\max}^{(i)} = \begin{cases} \mathrm{lm}(g_i) & \text{if } g_i \neq 0 \text{ and } \mathrm{lt}(g_i \cdot M_i) = t_g \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, $t_{(0,\ldots,0)} = 1$ and $(0, \ldots, 0)_{\max} = (0, \ldots, 0)$.

We show by induction over t, that

for all $g \in S_{(M_1,\ldots,M_r)}$ and all $t \in T$,
if $t_g = t$ then $g \in \mathrm{L}(B_M)$.

First assume $t = 1$. In this case $g = (0, \ldots, 0) \in \mathrm{L}(B_M)$.
Let now $t >_T 1$ and assume that for all $\bar{g} \in S_{(M_1,\ldots,M_r)}$ and for every $\bar{t} <_T t$

if $t_{\bar{g}} = \bar{t}$ then $\bar{g} \in \mathrm{L}(B_M)$.

Let $g \in S_{(M_1,\ldots,M_r)}$ be such that $t_g = t$. Then we define

$$\bar{g} := g - g_{\max}.$$

By definition of $g_{\max}$, $t_{\bar{g}} <_T t$ and by the induction hypothesis $\bar{g} \in \mathrm{L}(B_M)$. Since $\sum_{i=1}^r g_i \cdot M_i = 0$, t_g must vanish in $\sum_{i=1}^r g_i \cdot M_i$. Hence,

$$\sum_{i=1}^{r} g_{\max}^{(i)} \cdot M_i = 0,$$

and, by definition, $g_{\max}$ is T-homogeneous of degree t_g w.r.t. $\mathrm{pp}(M_1), \ldots, \mathrm{pp}(M_r)$.

By Lemma 2.14, $g_{\max}$ can be represented as a linear combination of S-vectors in B_M, i.e. $g_{\max} \in \mathrm{L}(B_M)$, and therefore

$$g = \underbrace{\bar{g}}_{\in \mathrm{L}(B_M)} + \underbrace{g_{\max}}_{\in \mathrm{L}(B_M)} \in \mathrm{L}(B_M).$$

□

Notation: We will call $B_{\mathrm{lm}(G)}$ the *S-basis* of $S_{\mathrm{lm}(G)}$ due to the tight connection to S-polynomials (see Definition 2.21), which were invented in [Buchberger, 1965].

Remark 2.15
Obviously, $B_{\mathrm{lm}(G)}$ is T-homogeneous w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$, since every element in $B_{\mathrm{lm}(G)}$ is T-homogeneous w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$.

Lemma 2.16
$S_{\mathrm{lm}(G)}$ is T-homogeneous w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$.

PROOF: Every syzygy $s \in S_{\mathrm{lm}(G)}$ can be expressed as $s = \sum_{i=1}^{l} p_i \cdot b_i$ with $l \in \mathbb{N}_0$ and $p_i \in P, b_i \in B_{\mathrm{lm}(G)}$ for all $1 \leq i \leq l$. By $m_0^{(i)}, \ldots, m_{m_i}^{(i)}$ we denote the monomials occurring in p_i. Then

$$s = \sum_{i=1}^{l} \sum_{j=0}^{m_i} m_j^{(i)} \cdot b_i$$

is a suitable representation of s as a sum of elements in $S_{\mathrm{lm}(G)}$, that are T-homogeneous w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$. □

The syzygies $S_{\mathrm{lm}(G)}$ and the particular basis $B_{\mathrm{lm}(G)}$ will play an important role in the characterization of Gröbner bases and the proof of the equivalence of this new characterization and the definition given in Section 1.2.1.

1.2.3 Another Characterization of Gröbner Bases

In the previous section we introduced the syzygies of a vector of polynomials S_G. By considering only the leading monomials of the polynomials in the vector G and again constructing the syzygies we arrived at the set $S_{\mathrm{lm}(G)}$ and could immediately come up with a basis for this module of syzygies.

What about the opposite direction? Is it possible to construct a syzygy of G by just extending a syzygy of the leading monomials of G with appropriate lower terms? A simple example shows, that this is not always possible for arbitrary vectors of polynomials:

Example 2.17
Consider the vector of polynomials $G = (G_1, G_2, G_3) \in \mathbb{Q}[x, y, z]^3$ with

$$\begin{aligned} G_1 &= xy + y \\ G_2 &= xz + x \\ G_3 &= y^2 + y. \end{aligned}$$

The monomials are ordered w.r.t. the total degree lexical ordering induced by $x > y > z$ (see Section 1.6 for a definition of this ordering). Obviously, $s = (\mathbf{z}, -\mathbf{y}, \mathbf{0})$ is a syzygy of the leading monomials of G, but not a syzygy of G, because

$$\mathbf{z} \cdot g_1 - \mathbf{y} \cdot g_2 + \mathbf{0} \cdot g_3 = -xy + yz \neq 0.$$

We now try to extend s with lower terms to s' such that s' is a syzygy of G. This extension must have the following form:

$$s' = (\mathbf{z} + a, -\mathbf{y} + bz + c, dy + ez + f).$$

We must choose the parameters a, b, c, d, e and $f \in \mathbb{Q}$ in s' such that

$$(\mathbf{z} + a) \cdot G_1 + (-\mathbf{y} + bz + c) \cdot G_2 + (dy + ez + f) \cdot G_3 = 0,$$

to guarantee that s' is a syzygy of G, i.e. we must solve the equation

$$\begin{aligned} &b \cdot xz^2 + d \cdot y^3 + \boxed{e \cdot y^2 z} + (a-1) \cdot xy + (b+c) \cdot xz + \\ &(d+f) \cdot y^2 + \boxed{(e+1) \cdot yz} + c \cdot x + (a+f) \cdot y = 0 \end{aligned}$$

for a, b, c, d, e and f. This equation is unsolvable and therefore such an extension cannot exist. □

However, there are vectors, for which the extension process shown in the above example can be carried out *for every syzygy* of the leading monomials. Again, we give a simple example (in fact, for reasons of simplicity, we just show the extension of *one* syzygy).

Example 2.18
Suppose we adjoin the polynomial $G_4 = yz + y$ as an additional component to the vector G of the above example. G is then a Gröbner basis. $s = (\mathbf{z}, -\mathbf{y}, \mathbf{0}, \mathbf{0})$ is a syzygy of the leading monomials of G. Analogously, the extension s' must have the form

$$s' = (\mathbf{z} + a, -\mathbf{y} + bz + c, d \cdot y + e \cdot z + f, g \cdot y + h \cdot z + i).$$

(See Definition 2.19 for the detailed description of the terms, which may occur in an extension). Now we must solve the equation

$$\begin{aligned}&b \cdot xz^2 + d \cdot y^3 + (e+g) \cdot y^2 z + h \cdot yz^2 + (a-1) \cdot xy +\\&(b+c) \cdot xz + (d+f+g) \cdot y^2 + (e+h+i+1) \cdot yz +\\&c \cdot x + (a+f+i) \cdot y = 0\end{aligned}$$

for a, b, c, d, e, f, g, h and $i \in \mathbb{Q}$. One can easily verify that

$$\begin{aligned}&a = g = 1\\&b = c = d = h = i = 0\\&e = f = -1\end{aligned}$$

is a solution of the equation, i.e.

$$s' = (\mathbf{z} + 1, -\mathbf{y}, -z - 1, y)$$

is a syzygy of G. □

It turns out, that this extension procedure is possible exactly when G is a Gröbner basis. We will now give the details.

Definition 2.19 (Extensible Syzygies)
We call an element $g = (g_1, \ldots, g_r) \in S_{\mathrm{lm}(G)}$ *extensible to* $h = (h_1, \ldots, h_r)$ (or *g can be extended to h* or *h is an extension of g*) iff

there exists a t, s.t.
g is T-homogeneous of degree t w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$,
h is a syzygy of G, and
for every $1 \le i \le r$ either $h_i = 0$ or
$\mathrm{lt}(h_i G_i) \le_T t$ and (if $\mathrm{lt}(h_i G_i) = t$ then $\mathrm{lm}(h_i) = g_i$).

We can now characterize Gröbner bases by extensible syzygies.

Theorem 2.20 (Characterization of Gröbner Bases by Syzygies)
The following conditions are equivalent:

G is a Gröbner basis. (GB)
All T-homogeneous elements of $S_{\mathrm{lm}(G)}$ are extensible. (Extensibility)

For the proof of this equivalence we will need the notion of *S-polynomials*, which goes back to [Buchberger, 1965] and plays a central role in the Buchberger-Algorithm for actually *constructing* Gröbner bases.

Definition 2.21 (S-polynomials)
Given two polynomials $f, g \in P - \{0\}$. The polynomial

$$\text{S-poly}(f,g) = \frac{\mathrm{lcm}(\mathrm{lm}(f), \mathrm{lm}(g))}{\mathrm{lm}(f)} \cdot f - \frac{\mathrm{lcm}(\mathrm{lm}(f), \mathrm{lm}(g))}{\mathrm{lm}(g)} \cdot g$$

is called the *S-polynomial* of f and g.

Remark 2.22
Let $\langle .,. \rangle$ denote the usual scalar product in P^r. Then

$$\text{S-poly}(G_i, G_j) = \langle \text{S-vector}(\mathrm{lm}(G_i), \mathrm{lm}(G_j)), G \rangle.$$

The proof of Theorem 2.20 is subdivided into three parts as shown in the following figure.

$$\begin{array}{c}
\text{(GB)} \\
\Updownarrow \ \text{Main Lemma on Gröbner Representations} \\
\text{(SP-Representation)} \\
\Updownarrow \ \text{Connection between Gröbner Representations and Syzygies} \\
\text{(Basis-Extensibility)} \\
\Updownarrow \ \text{Main Lemma on Extensible Syzygies} \\
\text{(Extensibility)}
\end{array}$$

Lemma 2.23 (Main Lemma on Gröbner Representations)
G is a Gröbner basis iff (GB)
All S-polynomials have a Gröbner representation. (SP-Representation)

An easy way of proving Lemma 2.23 would be as follows:

- ◇ Introduce *reduction of polynomials* (see Section 1.3) and show that (GB) is equivalent to the fact that the normal form of all polynomials in the ideal is zero.
- ◇ Use Buchberger's Main Theorem on Gröbner bases (see Theorem 3.6) to reduce this to the equivalent fact that the normal form of all S-polynomials is zero.
- ◇ Finally, show that zero-reducibility of all S-polynomials is equivalent to (SP-Representation).

However, as an alternative, it may be interesting to study a proof that immediately connects Gröbner representations with syzygy extensions *without* the intermediate notion of reducibility. We will explain the analogy between these concepts then in Section 1.4.

PROOF:
(GB) implies (SP-Representation): This part of the proof is trivial.
(SP-Representation) implies (GB): For $f = \sum_{i=1}^{r} f_i \cdot G_i \in \mathrm{Ideal}(G)$ we define

$$t_f := \max\{t \in T | \text{there exists an } 1 \leq i \leq r \text{ s.t. } t \text{ occurs in } f_i \cdot G_i\} \text{ and}$$

$$f_{\max} := (f_{\max}^{(1)}, \ldots, f_{\max}^{(r)}) \in M^r$$

with

$$f_{\max}^{(i)} = \begin{cases} \mathrm{lm}(f_i) & \text{if } f_i \neq 0 \text{ and } \mathrm{lt}(f_i \cdot G_i) = t_f \\ 0 & \text{otherwise.} \end{cases}$$

We show by induction over t, that

for all $f \in \mathrm{Ideal}(G)$ and all $t \in T$,
if $t_f = t$ then f has a Gröbner representation.

First assume $t \leq_T \mathrm{lt}(f)$. This case is trivial, since $\sum_{i=1}^{r} f_i \cdot G_i$ is already a Gröbner representation of f.

Let now $t >_T \mathrm{lt}(f)$ and assume that for every $\bar{t} <_T t$ and for all $\bar{f} \in \mathrm{Ideal}(G)$

if $t_{\bar{f}} = \bar{t}$ then $\bar{f}$ has a Gröbner representation.

Let $f \in \mathrm{Ideal}(G)$ be such that $t_f = t$. We define

$$\bar{f} := f - \langle f_{\max}, \mathrm{lm}(G) \rangle. \tag{1.4}$$

One can easily see that $f_{\max}$ is T-homogeneous of degree t_f w.r.t. $\mathrm{lt}(G_1), \ldots, \mathrm{lt}(G_r)$ and, since $t_f >_T \mathrm{lt}(f)$, t_f must vanish in f, i.e.

$$\langle f_{\max}, \mathrm{lm}(G)\rangle = \sum_{i=1}^{r} f_{\max}^{(i)} \cdot \mathrm{lm}(G_i) = 0.$$

By Lemma 2.14, $f_{\max}$ can be represented as a linear combination of S-vectors, i.e. there exist $m_1, \ldots, m_l \in M$ and for every $1 \leq k \leq l$ there exist $G_1^{(k)}, G_2^{(k)} \in G$ such that

$$f_{\max} = \sum_{k=1}^{l} m_k \cdot \text{S-vector}(\mathrm{lm}(G_1^{(k)}), \mathrm{lm}(G_2^{(k)})).$$

From the proof of Lemma 2.14 one can read off, that the "coefficients" m_k in the linear combination are such that

$$\mathrm{pp}(m_k \cdot \mathrm{lcm}(\mathrm{lm}(G_1^{(k)}), \mathrm{lm}(G_2^{(k)}))) = t_f, \text{ for all } 1 \leq k \leq l. \tag{1.5}$$

Then,

$$\begin{aligned}
&\langle f_{\max}, \mathrm{lm}(G)\rangle = \\
&\quad \langle f_{\max}, G\rangle - \langle f_{\max}, \mathrm{rp}(G)\rangle = \\
&\quad \langle \sum_{k=1}^{l} m_k \cdot \text{S-vector}(\mathrm{lm}(G_1^{(k)}), \mathrm{lm}(G_2^{(k)})), G\rangle - \langle f_{\max}, \mathrm{rp}(G)\rangle = \\
&\quad \sum_{k=1}^{l} m_k \cdot \underbrace{\langle \text{S-vector}(\mathrm{lm}(G_1^{(k)}), \mathrm{lm}(G_2^{(k)})), G\rangle}_{=\text{S-poly}(G_1^{(k)}, G_2^{(k)})} - \langle f_{\max}, \mathrm{rp}(G)\rangle.
\end{aligned}$$

Hence, from (1.4) we get

$$f = \bar{f} + \underbrace{\sum_{k=1}^{l} m_k \cdot \text{S-poly}(G_1^{(k)}, G_2^{(k)})}_{f^*} - \underbrace{\langle f_{\max}, \mathrm{rp}(G)\rangle}_{f'}. \tag{1.6}$$

By definition of $f_{\max}$, $t_{\bar{f}} <_T t$ and $t_{f'} <_T t$. Furthermore, by (SP-Representation) we know a Gröbner representation of S-poly$(G_1^{(k)}, G_2^{(k)})$, for all $1 \leq k \leq l$. Since $\mathrm{lt}(\text{S-poly}(G_1^{(k)}, G_2^{(k)})) <_T \mathrm{lcm}(\mathrm{lt}(G_1^{(k)}), \mathrm{lt}(G_2^{(k)}))$, we have

$$t^* <_T \mathrm{lcm}(\mathrm{lt}(G_1^{(k)}), \mathrm{lt}(G_2^{(k)})),$$

for all $t^* \in T$ occurring in the Gröbner representation of the S-polynomial and therefore, by (1.5), $t_{f^*} <_T t_f = t$. Hence, $t_{\bar{f}+f^*-f'} <_T t$ and, by the induction hypothesis, we get a Gröbner representation of $\bar{f} + f^* - f'$, i.e. by (1.6), a Gröbner representation of f. □

Lemma 2.24 (Main Lemma on Extensible Syzygies)
All T-homogeneous elements of $S_{\mathrm{lm}(G)}$ are extensible iff (Extensibility)
All elements of a T-homogeneous basis of $S_{\mathrm{lm}(G)}$ are extensible. (Basis-Extensibility)

Proof:
(Extensibility) implies (Basis-Extensibility): This part of the proof is obvious.
(Basis-Extensibility) implies (Extensibility): Let $g = (g_1, \ldots, g_r)$ be an arbitrary T-homogeneous element in $S_{\mathrm{lm}(G)}$ and let $B = \{B_1, \ldots, B_m\} = \{(b_1^{(1)}, \ldots, b_r^{(1)}), \ldots, (b_1^{(m)}, \ldots, b_r^{(m)})\}$ be a T-homogeneous basis of $S_{\mathrm{lm}(G)}$. Suppose each B_j can be extended to $\bar{B}_j$ (with the components denoted by $\bar{b}_i^{(j)}$ analogously). Then there exist polynomials p_j such that $g = \sum_{j=1}^m p_j \cdot B_j$. Then we define for every $1 \leq j \leq m$

$$\mathrm{mon}_j = \begin{cases} g_i / b_i^{(j)} & \text{if there exists } 1 \leq i \leq r \text{ with } g_i \neq 0 \neq b_i^{(j)} \\ & \text{and } g_i \text{ is a multiple of } b_i^{(j)}, \\ 0 & \text{otherwise.} \end{cases}$$

and

$$M_j = \begin{cases} \text{the monomial in the polynomial } p_j \text{ that has the same} \\ \text{power-product as } \mathrm{mon}_j, \text{ if such a monomial exists,} \\ 0 \text{ if } \mathrm{pp}(\mathrm{mon}_j) \text{ vanishes in } p_j \text{ or } \mathrm{mon}_j = 0. \end{cases}$$

For every j, M_j is independent of the i in the definition of mon_j because g and B_j are both T-homogeneous. Let $\bar{p_j} = p_j - M_j$. By the definition of M_j and $\bar{p_j}$ we see that, for all $1 \leq i \leq r$, every power product occurring in $p_j \cdot b_i^{(j)}$ must vanish in g_i, i.e.

$$\sum_{j=1}^{m} \bar{p_j} \cdot B_j = (0, \ldots, 0). \tag{1.7}$$

Then

$$\sum_{j=1}^{m} M_j \cdot B_j = \sum_{j=1}^{m} p_j \cdot B_j - \underbrace{\sum_{j=1}^{m} \bar{p_j} \cdot B_j}_{=(0,\ldots,0) \text{ by } (1.7)} = g.$$

Every non-zero $M_j \cdot B_j$ is T-homogeneous of the same degree as g. Since

$$\sum_{i=1}^{r} G_i \cdot \sum_{j=1}^{m} M_j \cdot \bar{b}_i^{(j)} = \sum_{j=1}^{m} M_j \cdot \underbrace{\sum_{i=1}^{r} G_i \cdot \bar{b}_i^{(j)}}_{\substack{=0,\\ \text{because } \bar{B}_j \text{ is an} \\ \text{extension of } B_j}} = 0,$$

$\sum_{j=1}^{m} M_j \cdot \bar{B}_j$ is a syzygy of G, hence, it is an extension of g. □

Lemma 2.25 (Connection Gröbner Representations – Syzygies)
All S-polynomials have a Gröbner representation iff (SP-Representation)
All elements of a T-homogeneous basis of $S_{\mathrm{lm}(G)}$ are extensible. (Basis-Extensibility)

Proof: We will use the S-basis $B_{\mathrm{lm}(G)}$ as a T-homogeneous basis for $S_{\mathrm{lm}(G)}$.
(SP-Representation) implies (Basis-Extensibility):
Let $b_{i,j} = \text{S-vector}(\mathrm{lm}(G_i), \mathrm{lm}(G_j)) \in B_{\mathrm{lm}(G)}$. $b_{i,j}$ is T-homogeneous of degree $\mathrm{lcm}(\mathrm{lt}(G_i), \mathrm{lt}(G_j))$ and corresponds due to Remark 2.22 uniquely to $s_{i,j} = \text{S-poly}(G_i, G_j)$. From (SP-Representation) we know a Gröbner representation

$$s_{i,j} = \sum_{k=1}^{r} g_k \cdot G_k, \text{ s.t.}$$

$$\text{if } g_k \neq 0 \text{ then } \mathrm{lt}(g_k \cdot G_k) \leq_T \mathrm{lt}(s_{i,j}) <_T \mathrm{lcm}(\mathrm{lt}(G_i), \mathrm{lt}(G_j)).$$

Hence, $h_{i,j} = b_{i,j} - \sum_{k=1}^{r} g_k \cdot e_k$ is an extension of $b_{i,j}$.
(Basis-Extensibility) implies (SP-Representation): Use the one to one correspondence between S-polynomials and elements of $B_{\mathrm{lm}(G)}$ as in the first part of the proof:
Let $h_{i,j} = (h_{i,j}^{(1)}, \ldots, h_{i,j}^{(r)})$ be an extension of $\text{S-vector}(\mathrm{lm}(G_i), \mathrm{lm}(G_j)) \in B_{\mathrm{lm}(G)}$. Then we define $\bar{h}_{i,j} = h_{i,j} - \text{S-vector}(\mathrm{lm}(G_i), \mathrm{lm}(G_j))$, and

$$\sum_{k=1}^{r} -\bar{h}_{i,j}^{(k)} \cdot G_k$$

is a Gröbner representation of $\text{S-poly}(G_i, G_j)$. □

1.3 Reduction of Polynomials

In this section we will introduce the notion of "reduction of a polynomial with respect to a set of polynomials" and will then give the classical characterization of Gröbner bases as it was originally given in [Buchberger, 1965] and [Buchberger, 1976a]. We will omit most of the proofs because in the next section we will establish an easy connection between the concept of reduction and the concept of extensibility of syzygies, which will make these proofs superfluous. (One can then easily translate the proofs of section 1.2 to the corresponding proofs in this section). We will also give a definition of what we mean by "computing a Gröbner basis" for a given ideal.

1.3.1 The Concept of Reduction

By $<_T$ we again denote an admissible ordering (see section 1.2.1) on the set of terms and by F a vector of polynomials. (In fact, it would suffice to define F as a set of polynomials but for reasons of compatibility with the other sections we define reduction of polynomials, normal form of polynomials etc. *modulo vectors of polynomials*). We will often use the binary relation $\in$ between polynomials and vectors of polynomials in the following intuitive way: $f \in F :\Longleftrightarrow$ there exists $1 \leq i \leq |F|$ such that $f = F_i$.

Definition 3.1 (Reduction)
Let g and h be polynomials.

g is reducible (modulo F) using f and m (denoted by $g \rightarrow_{f,m}$) iff
  $f \in F$,
  $m \in M$, and
  the monomial $\mathrm{lm}(f) \cdot m$ occurs in g.
g reduces to h (modulo F) using f and m (denoted by $g \rightarrow_{f,m} h$) iff
  g is reducible (modulo F) using f and m and
  $h = g - f \cdot m$.
g reduces to h (modulo F) (denoted by $g \rightarrow_F h$) iff
  there exist f and m s.t.
    g reduces to h (modulo F) using f and m.
By $\rightarrow_F^+$ we denote the *transitive closure* of $\rightarrow_F$.

Definition 3.2 (Normal Form, Reduction Sequence)
Let g and h again be polynomials.

h is a *normal form* of g (modulo F) (denoted by $g \to_F^* h$) iff
there exists a sequence of reductions of length $l \geq 0$ s.t.
$g \to_F^{(1)} \dots \to_F^{(l)} h$ and
h is not reducible (modulo F).
If g is reducible and h is a normal form of g (modulo F)
then there exist polynomials $f_1, \dots, f_l \in F$ and monomials $m_1, \dots, m_l$ s.t.
$g \to_{f_1,m_1} \dots \to_{f_l,m_l} h$.

We will then call $(g, (f_1, m_1), \dots, (f_l, m_l), h)$ a *reduction sequence* of g to h (modulo F).

Remark 3.3
Let l be the length of the sequence of reductions as in the last definition. $l = 0$ if and only if g is irreducible. In this case, $g = h$ and the reduction sequence of g to h modulo F is (g, h).

1.3.2 Known Results

In this section we compile some known results about Gröbner bases in connection with the principle of reduction. First we give the "classical" characterization of Gröbner bases as it was originally invented in [Buchberger, 1965], and the definition of what we mean by a Gröbner bases algorithm.

Lemma 3.4 (Classical Characterization of Gröbner Bases)
G is a Gröbner basis iff for all $f \in \mathrm{Ideal}(G)$ $f \to_G^* 0$.

The proof that the characterization given in Definition 2.5 is equivalent to this characterization is given in Lemma 4.3.

Definition 3.5 (Gröbner Bases Algorithm)
A *Gröbner bases algorithm* is an algorithm with the following input-output specification:

Input: a vector of polynomials F
Output: a vector of polynomials G such that
1) G is a Gröbner basis, and
2) $\mathrm{Ideal}(G) = \mathrm{Ideal}(F)$. □

The main result on Gröbner bases, and also the key to an algorithm for actually constructing Gröbner bases, is contained in the following Theorem, of which Lemma 2.23 is an analogon. The algorithmic importance of the theorem lies in the fact that it reduces the "infinite" quantifier in the characterization of Gröbner bases (the ideal generated by G has infinitely many elements) to a "finite" one (the set of all S-polynomials is finite).

Theorem 3.6 (Main Theorem on Gröbner Bases [Buchberger, 1965])
G is a Gröbner basis iff
For all $G_i, G_j \in G$: S-poly$(G_i, G_j) \rightarrow_G^* 0$.

In the following lemma we list some known properties of Gröbner bases in connection with the reduction relation $\rightarrow$ and normal forms of polynomials.

Lemma 3.7 (Properties of Gröbner Bases)
The following conditions are equivalent:

G is a Gröbner basis, (GB)
$\rightarrow_G$ has the Church-Rosser property, i.e. (Church-Rosser)
if $g \rightarrow_G^+ g_1$ and $g \rightarrow_G^+ g_2$ then there exists a polynomial h, s.t.
$g_1 \rightarrow_G^+ h$ and $g_2 \rightarrow_G^+ h$,
for all polynomials g, h_1, h_2 (Unique Normal Forms)
if h_1 and h_2 are normal forms of g (modulo G) then $h_1 = h_2$.

An algorithm for constructing a Gröbner basis for a given ideal was first given in [Buchberger, 1965], for an improved version see also [Buchberger, 1985]. By Theorem 3.6 we also get an algorithm to check, whether a given set of polynomials constitutes a Gröbner basis: Reduce the S-polynomials modulo the given vector and test whether all normal forms are zero.

1.4 Connection between Syzygies and Reduction

In section 1.2 we introduced Gröbner bases using the notion of "Gröbner representation", which was then shown to be equivalent to a characterization using extensibility of certain syzygies. In section 1.3 we gave the classical definition of Gröbner bases using reducibility of polynomials in the generated ideal.

In both cases, we made use of the fact that we can use S-polynomials in some way. In the first case, in the way that we can reduce the property that *all* polynomials in the ideal have a Gröbner representation to the property that *all S-polynomials* have a Gröbner representation. Furthermore, the S-vectors, which constitute the S-basis $B_{\mathrm{lm}(G)}$ for the module of syzygies of the leading monomials, are tightly connected to S-polynomials, see Remark 2.22. This basis was then heavily used in the proof that for Gröbner bases all elements of a basis of $S_{\mathrm{lm}(G)}$ must be extensible.

In the second case, we use S-polynomials for reducing the condition of reducibility of *all* polynomials in the generated ideal to reducibility of *all S-polynomials*.

For this reason, we will try to show some of the connections between both concepts in this section. We will see, that there is a very natural correspondence between the reducibility of a polynomial in the ideal and the extensibility of a syzygy in $S_{\mathrm{lm}(G)}$, i.e. from a reduction sequence of an S-polynomial one can immediately construct the extension of the syzygy.

Lemma 4.1 (Construction of Extensions)
Let $g = (g_1, \ldots, g_r) \in S_{\mathrm{lm}(G)}$ be T-homogeneous (w.r.t.$\mathrm{lm}(G_1), \ldots, \mathrm{lm}(G_r)$). Then

$$\sum_{i=1}^{r} g_i \cdot G_i \to_G^* 0 \Longrightarrow g \text{ is extensible.}$$

Proof: Assume $\sum_{i=1}^{r} g_i \cdot G_i \to_G^* 0$. If $\sum_{i=1}^{r} g_i \cdot G_i = 0$ then g is already a syzygy of G and g itself is an extension of g. If $\sum_{i=1}^{r} g_i \cdot G_i \neq 0$ there exist monomials $m_1^{(1)}, \ldots, m_{l_1}^{(1)}, \ldots, m_1^{(r)}, \ldots, m_{l_r}^{(r)}$, $l_i \geq 0$ for all $1 \leq i \leq r$ such that

$$(\sum_{i=1}^{r} g_i \cdot G_i, (G_1, m_1^{(1)}), \ldots, (G_1, m_{l_1}^{(1)}), \ldots, (G_r, m_1^{(r)}), \ldots, (G_r, m_{l_r}^{(r)}), 0)$$

is a reduction sequence of $\sum_{i=1}^{r} g_i \cdot G_i$ to zero. Let $t = g_i \cdot \mathrm{lt}(G_i)$ for an arbitrary i such that g_i is non-zero. Since g is a syzygy of the leading monomials of G, i.e. $\sum_{i=1}^{r} g_i \cdot \mathrm{lm}(G_i) = 0$, g is T-homogeneous of degree t w.r.t. the leading monomials of G and $\mathrm{lt}(\sum_{i=1}^{r} g_i \cdot G_i) <_T t$. Since all monomials of the form $m_j^{(i)} \cdot \mathrm{lm}(G_i)$ must occur in $\sum_{i=1}^{r} g_i \cdot G_i$ by definition of reduction, we have

$$\mathrm{pp}(m_j^{(i)} \cdot \mathrm{lm}(G_i)) \leq_T \mathrm{lt}(\sum_{i=1}^{r} g_i \cdot G_i) <_T t.$$

Hence,

$$(g_1 - \sum_{i=1}^{l_1} m_i^{(1)}, \ldots, g_r - \sum_{i=1}^{l_r} m_i^{(r)})$$

is an extension of g. □

The correspondence between extensibility of elements of the S-basis $B_{\mathrm{lm}(G)}$ to zero-reduction of S-polynomials follows as a corollary to the last lemma.

Corollary 4.2 (Construction of Basis Extensions)
If S-poly$(G_i, G_j) \rightarrow_G^* 0$ then S-vector$(G_i, G_j) \in B_{\mathrm{lm}(G)}$ is extensible.

PROOF: For $g =$ S-vector$(G_i, G_j) \in B_{\mathrm{lm}(G)}$ the corresponding $\sum_{i=1}^{r} g_i \cdot G_i$ is just the S-polynomial of G_i and G_j. Therefore, extensibility of g follows from the lemma above. □

We will now establish the connection between the concept of extensibility of syzygies and the concept of reducibility of polynomials.

Lemma 4.3 (Connection between Extensibility and Reducibility)
All T-homogeneous elements in $S_{\mathrm{lm}(G)}$ are extensible iff (Extensibility)
For all $f = \sum_{i=1}^{r} f_i \cdot G_i \in \mathrm{Ideal}(G)$: $f \rightarrow_G^* 0$. (Reducibility)

PROOF:
(Reducibility) implies (Extensibility): Lemma 4.1.

(Extensibility) implies (Reducibility): Similar to the proof of Lemma 2.23 we define

$$t_f := \max\{t \in T \mid \text{there exists an } 1 \le i \le r \text{ s.t. } t \text{ occurs in } f_i \cdot G_i\} \text{ and}$$

$$f_{\max} := (f_{\max}^{(1)}, \ldots, f_{\max}^{(r)}) \in M^r$$

with

$$f_{\max}^{(i)} = \begin{cases} \mathrm{lm}(f_i) & \text{if } f_i \ne 0 \text{ and } \mathrm{lt}(f_i \cdot G_i) = t_f \\ 0 & \text{otherwise.} \end{cases}$$

We show by induction over t, that

$$\text{for all } f = \textstyle\sum_{i=1}^r f_i \cdot G_i \in \mathrm{Ideal}(G) \text{ and all } t \in T, \quad \text{if } t_f = t \text{ then } f \rightarrow_G^* 0.$$

First assume $t = 1$. In this case we have $\mathrm{lt}(G_i) = 1$, for all $1 \le i \le r$. Therefore, there exists $c \in K$ such that $f \rightarrow_{G_1,c} 0$.

Let now $t >_T 1$ and assume that for every $\bar{t} <_T t$ and for all $\bar{f} \in \mathrm{Ideal}(G)$

$$\text{if } t_{\bar{f}} = \bar{t} \text{ then } \bar{f} \rightarrow_G^* 0.$$

Let $f \in \mathrm{Ideal}(G)$ be such that $t_f = t$.

<u>Case I:</u> t_f occurs in f, i.e. $\mathrm{lt}(f) = t_f$. Let j be such that $f_{\max}^{(j)} \ne 0$. Then there exists $c \in K$ such that

$$c \cdot f_{\max}^{(j)} \cdot \mathrm{lm}(G_j) = \mathrm{lm}(f).$$

We define

$$\bar{f} := f - c \cdot f_{\max}^{(j)} \cdot G_j \in \mathrm{Ideal}(G).$$

By definition of $\bar{f}$, $t_{\bar{f}} <_T t$ and, by induction hypothesis, $\bar{f} \rightarrow_G^* 0$. Furthermore, $f \rightarrow_{G_j, c \cdot f_{\max}^{(j)}} \bar{f}$ and, by transitivity of $\rightarrow$, $f \rightarrow_G^* 0$.

<u>Case II:</u> t_f vanishes in f, i.e. $\sum_{i=1}^r f_{\max}^{(i)} \cdot \mathrm{lm}(G) = 0$. Thus, $f_{\max} \in S_{\mathrm{lm}(G)}$ and is T-homogeneous of degree t_f w.r.t. $\mathrm{lm}(G_1), \ldots, \mathrm{lm}(G_r)$. By (Extensibility), $f_{\max}$ can be extended to $f_{\max} + \overline{f_{\max}}$ such that

$$\mathrm{lt}(\overline{f_{\max}}^{(i)}) <_T \mathrm{pp}(f_{\max}^{(i)}) \text{ for all } 1 \le i \le r \text{ and} \tag{1.8}$$

$$\sum_{i=1}^r (f_{\max} + \overline{f_{\max}}) \cdot G_i = 0.$$

Furthermore, we define

$$\bar{f} := f - f_{\max}, \tag{1.9}$$

Then

$$\begin{aligned} f = \sum_{i=1}^{r} f_i \cdot G_i &= \sum_{i=1}^{r} f_{\max}^{(i)} \cdot G_i + \sum_{i=1}^{r} \bar{f}_i \cdot G_i \\ &= \underbrace{\sum_{i=1}^{r} -\overline{f_{\max}}^{(i)} \cdot G_i}_{=f' \in \mathrm{Ideal}(G)} + \underbrace{\sum_{i=1}^{r} \bar{f}_i \cdot G_i}_{=f^* \in \mathrm{Ideal}(G)} \ . \end{aligned} \tag{1.10}$$

By (1.8) we get $t_{f'} <_T t$ and by (1.9) $t_{f^*} <_T t$. Therefore, $t_{f'+f^*} <_T t$ and $f' + f^* \in \mathrm{Ideal}(G)$. Hence, by induction hypothesis, we have: $f' + f^* \rightarrow_G^* 0$ and, by (1.10), $f \rightarrow_G^* 0$. □

1.5 Three Concepts – A Conclusive View

We have now discussed three concepts how to characterize Gröbner bases,

- ◇ polynomials having Gröbner representations,
- ◇ extensibility of syzygies of the leading monomials, and
- ◇ reducibility of polynomials to zero.

All these approaches are shown to result in equivalent notions of Gröbner bases.

Furthermore, Lemma 4.1 (in fact, the proof of Lemma 4.1) shows, how to actually *construct* syzygy extensions. The following algorithm can be read off the proof:

$s_{\text{ext}} \longleftarrow$ **extension**(s, G)
Input: $s \in S_{\mathrm{lm}(G)}$,
  a Gröbner basis G.
Output: s_{ext} such that s_{ext} is an extension of s.

  $s_{\text{ext}} := s$
  $f := \sum_{i=1}^{r} s_i \cdot G_i$
  while$(f \neq 0)$
    Let j, m and f' such that $f \rightarrow_{G_j,m} f'$

$s_{\text{ext}} := s_{\text{ext}} - m \cdot e_j$
$f := f'$
return s_{ext}

We remark that the computation in the while-loop is nothing else than the computation of the normal form of $f = \sum_{i=1}^{r} s_i \cdot G_i$. At each entry of the loop, $f \in \text{Ideal}(G)$ and, if $f \neq 0$, f must be reducible since G is a Gröbner basis. Furthermore $f' \in \text{Ideal}(G)$. Termination of the algorithm follows from the fact that each polynomial in $\text{Ideal}(G)$ must be reducible to zero. Correctness is guaranteed by Lemma 4.1.

In this context we can nicely reformulate a theorem, whose proof can be found for example in [Winkler, 1986]. Extensible syzygies allow a much easier formulation than the one given by reduction.

Theorem 5.1
Let G be a Gröbner basis. Then the set

$$B_G = \{\text{extension}(\text{S-vector}(\text{lm}(G_i), \text{lm}(G_j)), G) | 1 \leq i < j \leq r\}$$

constitutes a basis for S_G. □

Roughly spoken, if G is a Gröbner basis, we get a basis for the module of syzygies of G by extending each element of the S-basis for the module of syzygies of the leading monomials. Using the algorithm "extension", the theorem can be viewed as an algorithm for computing the basis of the module of syzygies of G.

Since the essential theorems in all three concepts contain the notion of S-polynomials in some way, we will compile the corresponding results in the following figure. One can see that the key result, namely the reduction of the "general characterization" to a "basic characterization", as well as the proof of this result in one version has analoga then in the other versions.

For a Gröbner basis G the equivalences in Figure 1.1 are true.

G is a Gröbner basis

General Characterization	*Key Result*	*Basic Characterization*	
All polynomials in Ideal(G) have a Gröbner representation	Lemma 2.23 $\Leftrightarrow$	All S-polynomials have a Gröbner representation	*Concept of Gröbner Representation*
		$\Updownarrow$ Lemma 2.25	
All homogeneous elements of $S_{\mathrm{lm}(G)}$ are extensible	Lemma 2.24 $\Leftrightarrow$	All elements of $B_{\mathrm{lm}(G)}$ are extensible	*Concept of Extensibility*
$\Updownarrow$ Lemma 4.3			
The normal form of all polynomials in Ideal(G) is zero	Theorem 3.6 $\Leftrightarrow$	The normal form of all S-polynomials is zero	*Concept of Reducibility*

Figure 1.1: Equivalences for Gröbner Bases

1.6 Orderings on Power Products

Since in practice it turns out that the computing time for calculating a Gröbner basis heavily depends on the ordering on power products, we devote this section to term orderings. The dependence of the computing time on term orderings is at least twofold:

- ◇ on one hand it depends on the *ordering of power products within the polynomials* (selection of the leading monomial depends on this ordering),
- ◇ on the other hand it depends on the *ordering of pairs of polynomials within a set of pairs*, which is usually done by ordering pairs with respect to ascending least common multiples of the leading terms of the occurring polynomials (selection of the pairs for S-polynomial computation and reduction depends on this ordering).

1.6.1 Some Examples of Admissible Orderings for the Gröbner Bases Algorithm

For correctness of the algorithm, the used term ordering has to be *admissible* (see section 1.2.1). Within these orderings the *purely lexical ordering* (see Definition 6.2) and the *total degree lexical ordering* (see Definition 6.5) are of special interest. The purely lexical ordering for monomials has nice properties for solving systems of algebraic equations, whereas the computing time when using the total degree lexical ordering can be expected to be shorter. However, a drastical speed-up can often be gained by using the total degree lexical ordering for the ordering of pairs when computing the Gröbner basis with respect to the purely lexical ordering. Therefore, we treat admissible orderings in a general setting in this section to have a possibility to use *arbitrary admissible orderings* during the computation of Gröbner bases.

If we want to have an ordering for monomials (monomials are of course ordered with respect to their power products), we have to deal with orderings on $\mathbb{Z}^n$ or $\mathbb{Q}^n$, i.e. we represent each power product $x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$ as its vector of exponents $(e_1, \ldots, e_n) \in \mathbb{N}_0^n$. This representation of terms is an isomorphism from $(T, \cdot)$ to $(\mathbb{N}_0^n, +)$. (Hence, an ordering on $\mathbb{N}_0^n$ suffices for our purposes). We will often use both representations of power products simultaneously without actually mentioning it, because it should always be clear from the context, which representation is used. In addition, we will denote the neutral element by "1" when using term representation, but "0" when using exponent vectors. Furthermore, we tacitly assume $x_1 > \ldots > x_n$.

It suffices to treat orderings on $\mathbb{Z}^n$ because of the following lemma.

Lemma 6.1
Every ordering $<_{\mathbb{Z}}$ on $\mathbb{Z}^n$ uniquely extends to an ordering $<_{\mathbb{Q}}$ on $\mathbb{Q}^n$.

PROOF: Let $q = (q_1, \ldots, q_n) \in \mathbb{Q}^n$. Then by multiplying with the least common multiple of all the denominators appearing in q we get the vector $\bar{q} \in \mathbb{Z}^n$. The ordering $<_{\mathbb{Q}}$ is then determined as follows: $q <_{\mathbb{Q}} 0$ iff $\bar{q} <_{\mathbb{Z}} 0$. □

On the other hand, if we have an ordering on $\mathbb{Q}^n$ we derive an ordering on $\mathbb{Z}^n$ by restricting the ordering on $\mathbb{Q}^n$ to $\mathbb{Z}^n$.

In the following, $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ will always denote exponent vectors.

Definition 6.2 (Purely Lexical Ordering)
A total ordering is called *lexicographic* (generated by $x_1 > \ldots > x_n$) (denoted by $<_{\text{lex}}$) iff it is defined by the following rule:

$u <_{\text{lex}} v$ iff the first non-zero coordinate of $u - v$ is negative.

We will often call this ordering *purely lexical ordering* (generated by $x_1 > \ldots > x_n$).
A total ordering $<$ is called *of lexicographic type* iff there is an ordered isomorphism from $(\mathbb{Z}^n, <)$ to $(\mathbb{Z}^n, <_{\text{lex}})$

Remark 6.3
An ordering $<$ is of lexicographic type if there is an isomorphism

$$I : (\mathbb{Z}^n, <) \longrightarrow (\mathbb{Z}^n, <_{\text{lex}}) \text{ such that } u < v \Longleftrightarrow I(u) <_{\text{lex}} I(v).$$

Definition 6.4 (Total Degree, Graded Orderings)
The natural number $\sum_{i=1}^n v_i$ is called the *(total) degree* of v (denoted by $\deg(v)$).
We call an ordering $<_V$ on exponent vectors *graded* iff $\deg(u) < \deg(v) \Rightarrow u <_V v$

Already in the case of two variables one can see that the purely lexical ordering is *not* graded. $1 = \deg((1,0)) < \deg((0,2)) = 2$ but $(1,0) >_{\text{lex}} (0,2)$. A slight modification makes a graded ordering out of the purely lexical ordering.

Definition 6.5 (Total Degree Lexical Ordering)
The ordering

$$u <_{\text{tdeg}} v :\Leftrightarrow (\deg(u), u_1, \ldots, u_n) <_{\text{lex}} (\deg(v), v_1, \ldots, v_n)$$

is called *total degree lexical ordering* or *graded lexical ordering* (generated by $x_1 > \ldots > x_n$).

Definition 6.6 (Inverse Lexical Ordering)
The ordering

$$u <_{\text{ilex}} v :\Leftrightarrow v <_{\text{lex}} u$$

is called *inverse lexical ordering* (generated by $x_1 > \ldots > x_n$).

However, the inverse lexical ordering is not admissible because $v <_{\text{ilex}} 0$ for every $v \in \mathbb{N}^n$. Again, grading this ordering (as it was done in Definition 6.5 with the purely lexical ordering) changes this ordering into an admissible one.

Definition 6.7 (Total Degree Inverse Lexical Ordering)
The ordering

$$u <_{\text{tdilex}} v :\Leftrightarrow \begin{array}{l} \deg(u) < \deg(v) \text{ or} \\ \deg(u) = \deg(v) \text{ and } (u_1, \ldots, u_n) <_{\text{ilex}} (v_1, \ldots, v_n) \end{array}$$

is called *total degree inverse lexical ordering* or *graded inverse lexical ordering* (generated by $x_1 > \ldots > x_n$).

1.6.2 A General Setting

In the last section we gave some examples of admissible orderings, which are frequently used for computation of Gröbner bases. What about other admissible orderings? How can one "generate" further examples of admissible orderings? Is there a general representation for these orderings?

It turns out, that each admissible ordering can be characterized by a matrix (see [Robbiano, 1985]). We will now give details on this characterization of admissible orderings.

Definition 6.8 (Rational Dimension)
Given a vector $\rho \in \mathbb{R}^n$. We call

$$\dim_{\mathbb{Q}}(\rho) = \text{“the dimension of the subvectorspace over } \mathbb{Q} \text{ spanned by the coordinates of } \rho\text{”}$$

the *rational dimension* of ρ.

Example 6.9
A vector $\rho = (\rho_1, \ldots, \rho_n)$ consisting only of rational numbers has rational dimension 1 because every element of the subvectorspace spanned by the coordinates of ρ is of the form $\sum_{i=1}^n q_i \cdot \rho_i$ with $q_i \in \mathbb{Q}$ for every $i \in \{1, \ldots, n\}$. Since all ρ_i are rational numbers, also $\sum_{i=1}^n q_i \cdot \rho_i$ is rational and can therefore be written as $\overline{q} \cdot 1$ with $\overline{q} \in \mathbb{Q}$. Thus, it suffices to have only one basis element (e.g. 1), hence the rational dimension of ρ is 1.

Example 6.10
Let now $\rho = (5, \sqrt{3}, 2, 6, 2\sqrt{3}, \pi)$. $\dim_{\mathbb{Q}}(\rho) = 3$, because every element of the subvectorspace spanned by the elements of ρ can be represented by $q_1 \cdot 1 + q_2 \cdot \sqrt{3} + q_3 \cdot \pi$ with $q_1, q_2, q_3 \in \mathbb{Q}$. So we need three elements in the basis, hence, the rational dimension is three.

Theorem 6.11
For every ordering $<$ on $\mathbb{Q}^n$ there exits an integer s with $1 \leq s \leq n$ and an $(n \times s)$ -matrix $U_<$ consisting of s orthogonal column-vectors $c_1, \ldots, c_s \in \mathbb{R}^n$ with $\dim_{\mathbb{Q}}(c_1) + \ldots + \dim_{\mathbb{Q}}(c_s) = n$ such that the mapping

$$\begin{array}{cccc} \alpha : & (\mathbb{Q}^n, <) & \longrightarrow & (\mathbb{R}^s, <_{\text{lex}}) \\ & v & \longmapsto & v \cdot U_< \end{array}$$

is an injective ordered homomorphism.

<u>Proof:</u> See [Robbiano, 1985].

This result leads to the following simple procedure for comparing power products represented by the exponent vectors u and v:

1. Find the matrix $U_<$ for the considered ordering.

2. Compute $\bar{u} = u \cdot U_<$ and $\bar{v} = v \cdot U_<$.

3. Then decide: $u < v$ iff $\bar{u} <_{\text{lex}} \bar{v}$ iff the first non-zero coordinate of $\bar{u} - \bar{v}$ is negative.

Remark 6.12
In practice, if one knows the ordering, i.e. one knows a rule how to order power products, one will not use this procedure because in most of the cases this would result in a slow-down of computation. However, one can use this procedure for orderings for which one does *not* know a "natural rule". It suffices to set up a matrix fulfilling the conditions of Theorem 6.11 and then use the procedure shown above. The result is an admissible ordering for the power products, which we will often call *matrix ordering*.

1.6.3 Further Examples

Although we will not use the matrix ordering for known orderings, we give examples how these matrices would have to look like for some of the well-known orderings for power products.

The most trivial example is the purely lexical ordering, for which the matrix is simply the $n \times n$-unit-matrix. In the next subsections we will show two slightly more complicated examples.

Total Degree Lexical Ordering

Let $<_{\text{tdeg}}$ be the total degree lexical ordering as defined in Definition 6.5. This ordering can also be described by the following rule: "First order by total degree and within a given degree order lexically w.r.t. $x_1 > \ldots > x_n$".

Lemma 6.13
The matrix $U_{<_{\text{tdeg}}}$ for this particular ordering has the form:

$$U_{<_{\text{tdeg}}} = \begin{pmatrix} 1 & n-1 & 0 & 0 & \ldots & 0 \\ 1 & -1 & n-2 & 0 & \ldots & 0 \\ \vdots & \vdots & -1 & \vdots & \ldots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ldots & 1 \\ 1 & -1 & -1 & -1 & \ldots & -1 \end{pmatrix}$$

In general: $c_1 = (1, \ldots, 1)^T$, $c_i = (c_{1i}, \ldots, c_{ni})^T$, for $2 \leq i \leq n$, with

$$c_{ji} = \begin{cases} 0 & j < i-1 \\ n-i+1 & j = i-1 \\ -1 & j > i-1 \end{cases}$$

PROOF: In this case $s = n$.
(Orthogonality of the columns): $\langle\, .\, ,\, .\, \rangle$ denotes the scalar product in $\mathbb{R}^n$. For $i \neq 1$ the sum of all elements in a column is 0. Thereby $\langle c_1, c_i \rangle = \sum_{j=1}^n c_{ji} = 0$.
Let now $i, k \neq 1$, $i < k$:

$$\begin{aligned} \langle c_i, c_k \rangle &= \sum_{j=1}^n c_{ji} c_{jk} = \sum_{j=k-1}^n c_{ji} c_{jk} = (-1)(n-k+1) + \sum_{j=k}^n \underbrace{c_{ji}}_{-1} \underbrace{c_{jk}}_{-1} = \\ &= -(n-k+1) + (n-k+1) = 0 \end{aligned}$$

Hence, the columns of $U_{<_{\mathrm{tdeg}}}$ are orthogonal.
(Rational dimension of the columns): As mentioned above a vector of rational numbers has rational dimension 1. Hence $\dim_{\mathbb{Q}}(c_1) + \ldots + \dim_{\mathbb{Q}}(c_n) = n$.
(Correctness): Let u and v be two exponent vectors. We then define

$$\begin{aligned} \bar{u} &= u \cdot U_{<_{\mathrm{tdeg}}}, \quad \hat{u} = (\deg(u), u_1, \ldots, u_n) \\ \bar{v} &= v \cdot U_{<_{\mathrm{tdeg}}}, \quad \hat{v} = (\deg(v), v_1, \ldots, v_n) \end{aligned}$$

We have to show:

$$(u <_{\mathrm{tdeg}} v \Leftrightarrow)\ \hat{u} <_{\mathrm{lex}} \hat{v} \Leftrightarrow \bar{u} <_{\mathrm{lex}} \bar{v}.$$

If $u <_{\mathrm{tdeg}} v$ then there exists $1 \leq l \leq n+1$ such that $\hat{u}_l < \hat{v}_l$ and $\hat{u}_j = \hat{v}_j$ for all $j < l$. $\bar{u}_1$ and $\bar{v}_1$ represent the total degrees of u and v respectively, hence, if $l = 1$ then $\bar{u}_1 < \bar{v}_1$ and therefore $\bar{u} <_{\mathrm{lex}} \bar{v}$.
Suppose $1 < l (\leq n+1)$, i.e. $\deg(u) = \deg(v)$ and $u_j = v_j$ for all $j < l-1$. Thus,

$$\begin{aligned} \bar{u}_l - \bar{v}_l &= \\ &= \sum_{j=1}^n c_{jl} \cdot (u_j - v_j) = \\ &= (n-l+1) \cdot (u_{l-1} - v_{l-1}) \underbrace{- \sum_{j=l}^n u_j + \sum_{j=l}^n v_j}_{u_{l-1} - v_{l-1}} = \end{aligned}$$

$$= \underbrace{(n-l+2)}_{>0} \cdot \underbrace{(u_{l-1} - v_{l-1})}_{<0} < 0.$$

On the other hand, for $k \leq l$

$$\bar{u}_k - \bar{v}_k = (n-k+2) \cdot \underbrace{(u_{k-1} - v_{k-1})}_{=0} = 0.$$

Hence, the first non-zero component of $\bar{u} - \bar{v}$ is at the same index (l) and has the same sign as the first non-zero component of $\hat{u} - \hat{v}$, and therefore $\hat{u} <_{\text{lex}} \hat{v} \Leftrightarrow \bar{u} <_{\text{lex}} \bar{v}$. □

Gödel Ordering

The Gödel ordering $<_{\text{G}}$ is defined by the following rule:

$$u <_{\text{G}} v :\Longleftrightarrow p(1)^{u_1} \cdot \ldots \cdot p(n)^{u_n} < p(1)^{v_1} \cdot \ldots \cdot p(n)^{v_n}$$

where $p(i)$ denotes the i-th prime number starting with $p(1) = 2$.

Lemma 6.14
The matrix $U_{<_{\text{G}}}$ for this particular ordering has the form:

$$U_{<_{\text{G}}} = (\log(p(1)), \ldots, \log(p(n)))^T = c_1.$$

Proof: In this case $s = 1$, the matrix only consists of one column, hence orthogonality of the columns is satisfied trivially.
(Rational dimension of the column): We consider the vectorspace

$$V = \{\sum_{i=1}^{n} q_i \cdot \log(p(i)) | q_i \in \mathbb{Q}\}$$

and show that $\log(p(1)), \ldots, \log(p(n))$ constitute a basis for this vectorspace, i.e.

$$\sum_{i=1}^{n} q_i \cdot \log(p(i)) = 0 \text{ implies } q_i = 0 \text{ for all } 1 \leq i \leq n.$$

Let now q_i be such that $\sum_{i=1}^{n} q_i \cdot \log(p(i)) = 0$. W.l.o.g we can assume $q_i \in \mathbb{Z}$, for all $1 \leq i \leq n$ (otherwise we can multiply the equation by the least common multiple of the denominators). Assume there exists a k, such that $q_k \neq 0$. Then we have

$$q_k \cdot \log(p(k)) = \sum_{i \neq k} -q_i \cdot \log(p(i)).$$

If a is the base of the logarithm, we can derive

$$\begin{aligned} p(k)^{q_k} &= a^{(\sum_{i\neq k} -q_i \log(p(i)))} = \\ &= \prod_{i\neq k} a^{-q_i \log(p(i))} = \\ &= \prod_{i\neq k} p(i)^{-q_i}. \end{aligned}$$

This is a contradiction to the fact that $p(k)$ is a prime number, hence such a k cannot exist, i.e. $q_i = 0$ for all $1 \leq i \leq n$. Hence, $\log(p(1)), \ldots, \log(p(n))$ is a basis for V, i.e. $\dim_{\mathbb{Q}}(c_1) = n$.
(Correctness):

$$\begin{aligned} u <_{\mathrm{G}} v &\iff p(1)^{u_1} \cdot \ldots \cdot p(n)^{u_n} < p(1)^{v_1} \cdot \ldots \cdot p(n)^{v_n} \iff \\ &\iff u_1 \log(p(1)) + \ldots + u_n \log(p(n)) < \\ &\quad\; < v_1 \log(p(1)) + \ldots + v_n \log(p(n)) \iff \\ &\iff u \cdot U_{<_{\mathrm{G}}} < v \cdot U_{<_{\mathrm{G}}} \iff \\ &\iff (u - v) \cdot U_{<_{\mathrm{G}}} < 0 \iff \\ &\iff (u - v) \cdot U_{<_{\mathrm{G}}} <_{\mathrm{lex}} 0. \end{aligned}$$

$\square$

Chapter 2

Implementation of Gröbner Bases

2.1 Introduction

In this chapter, we report on GRÖBNER, a software package for computing Gröbner bases. GRÖBNER is meant to serve as both a *research tool* for doing research on Gröbner bases as well as a *"black-box-Gröbner bases-calculator"* for people who want to use the Gröbner bases algorithm just as a subalgorithm in other programs.

In the first sections, we list some of the programming languages and software systems that can be used for the implementation of such a system and show up the advantages and disadvantages of each with respect to the criteria, which we consider as most important for GRÖBNER. Among these criteria we stress the *readability of the code*, *the speed of the resulting programs*, the *ability of generic programming*, and the *availability of symbolic algebraic algorithms*.

In Section 2.5 and Section 2.6 we discuss two possible approaches for realizing generic programming in C in a bit more detail:

- ◇ A concept in which the appropriate function to call depending on the type of the arguments is chosen at run-time (similar to C++) and
- ◇ a concept in which function-names have to be tagged and can therefore be substituted by appropriate functions at compile time (similar to packages in LISP).

In Section 2.7 we report then in all detail about the actual implementation of the package in C language together with SACLIB, a library of C routines derived from the SAC-2 system. This part serves as a rudimentary manual for using the package, although GRÖBNER is not fully developed yet. It is a part of the design of GRÖBNER that new parts will continuously be added to the package.

We conclude with some timing statistics that should clarify the dependence of the computing time on the ordering of monomials within a polynomial and the sequence of reductions during the computation. Furthermore, we compare several variants of the algorithm, which differ in the manner how the criteria for predicting zero-reductions are applied. We do not go into the details of these variants, interested readers might consult the source code, which contains a lot of documentation and is listed in Chapter 3.

2.2 Restrictions for the Software

In this section we report on the implementation of GRÖBNER, a package for the computation of Gröbner bases. The software is meant to be distributed along with a book about the theory of Gröbner bases ([Buchberger, 1991a]), and must therefore fulfill various restrictions:

- ◇ The code should be *as readable as possible*, since source code is distributed to serve as a research tool, i.e. it should be easy to modify, extend or restrict the programs,
- ◇ the programs should run *as fast as possible*, because in research work it is desirable to run many tests (especially for large inputs), hence, the faster each test runs the more can be tested,
- ◇ the programs should be *generic* in the sense that data structures can be built up hierarchically, such that each data structure does not depend on the actual representation of the underlying data structures (we will emphasize this point later in the description of the programs),
- ◇ in a later stage, *algebraic algorithms* such as factorization of polynomials, rational function arithmetic, long integer arithmetic, etc. should already be available,
- ◇ the programs should be available for *as many platforms as possible*, so that users are not restricted to special hardware, and
- ◇ the code of the programs should be *closely related to the language of the book*.

2.3 Choices for the Programming Language

Of course, neither any programming language nor any software system available now meets all these requirements perfectly. Therefore, a compromise has to be made. We list some of the possibilities together with their advantages and disadvantages. In this context, "readability" is to be interpreted in the sense that also people *not familiar* with the language (or system) can easily read the code, it is definitely not measured appropriately for experts in the specific languages or systems.

2.3.1 Pure Programming Languages

For our purposes, all pure programming languages lack in offering algebraic algorithms such as factorization of polynomials, rational function arithmetic etc. We will not mention this fact as a disadvantage for each of the languages.

LISP

Advantages:

- ◇ Fast,
- ◇ widely available,
- ◇ many people are familiar with it (at least in scientific research),
- ◇ arithmetic for arbitrary long integers and rational numbers available in the system,
- ◇ storage management is done by the system.

Disadvantages:

- ◇ Readability (depends of course on the personal taste of the reader!),
- ◇ no generic facilities,

PCL

PCL (Portable Common Loops) is kind of an object-oriented extension of Common LISP. All LISP constructs are available in PCL, in addition one can define *classes* (with sub- and superclasses), define *methods* for the elements in these classes, *inherit methods*, etc. (for a description of PCL see [Bobrow *et al.*, 1988]).
Advantages:

- ◇ Genericity,
- ◇ storage management is done by the system.

Disadvantages:

- ◇ Not very fast,
- ◇ not very well known,

- ◇ readability similar to LISP (depends again on the reader),
- ◇ no algebraic algorithms available.

We coded GRÖBNER partially in PCL, but the speed of the resulting programs was not satisfactory. However, this experiment showed the amount of work that must be done for re-coding algebraic algorithms if such algorithms are not available in the language (see [Windsteiger, 1990b]).

C

Advantages:

- ◇ Very fast,
- ◇ widely available,
- ◇ well-known,
- ◇ readable code (if one restricts oneself and sticks to some rules).

Disadvantages:

- ◇ No generic features,
- ◇ no storage management,
- ◇ no algebraic algorithms available.

For C^{++} most of this is true as well, except that "generic features" change from a disadvantage to an advantage. (We must confess, that we did not run too many test with respect to the speed of the resulting programs).

2.3.2 Computer Algebra Software Systems

In general, all software systems offer the advantage of complete storage management and the availability of various algebraic algorithms. We will not list this advantages for each system. Speed of the system is measured for user-defined functions, *not* for built-in functions.

AXIOM (Scratchpad II)

The most important disadvantage for our intended use is that AXIOM is *not* available on a variety of hardware platforms, therefore it would cause troubles to distribute AXIOM-software along with the Gröbner bases book. All other requirements would be fulfilled by this system.

SAC-2

SAC-2 was one of the first Computer-Algebra systems. It is written in the programming language ALDES, which is translated into FORTRAN code. Therefore, SAC-2 is available wherever FORTRAN is available.
Advantages:

- ◇ Availability of algebraic algorithms of high quality (this is especially true for the polynomial arithmetic),
- ◇ fast.

Disadvantages:

- ◇ Programs are hard to read, since names for functions and procedures are "cryptic",
- ◇ poor documentation of the system,
- ◇ no possibility for generic programming.

MAPLE

Advantages:

- ◇ Available for almost all popular hardware platforms,
- ◇ readable code.

Disadvantages:

- ◇ No generic facilities,
- ◇ not fast (user-defined programs are not compiled but interpreted).

Mathematica

We fully coded GRÖBNER in *Mathematica*, because on the first view the system seemed to fulfill all requirements perfectly (see [Buchberger, 1990]). However, detailed tests showed that for user-defined programs, the system shows a drastical slow-down for slightly larger input. This fact is documented in all detail in [Buchberger, 1991b].
Advantages:

- ⋄ Available for almost all popular hardware platforms,
- ⋄ extremely readable code,
- ⋄ easy handling of generic programming features.

Disadvantages:

- ⋄ Bad performance of user-defined rules and functions for large inputs.

2.4 Approaches for a Solution

In the last sections we listed the advantages and disadvantages of several programming languages and Computer-Algebra systems with respect to the qualities needed for coding a large piece of software like GRÖBNER. It showed out that none of the existing Computer-Algebra systems could offer a programming language that is sufficiently fast. Paying tribute to this experience, we decided to use the programming language C for implementing GRÖBNER. Having fixed this decision, two possibilities to continue remain:

- ◇ Implement all symbolic and algebraic algorithms and a storage management by ourselves in C, or
- ◇ use a Computer-Algebra library and a storage administration written in C language.

The first solution would make a lot of routine-work necessary right at the beginning, such as implementation of arithmetic for arbitrary long integers, rational numbers, finite fields and in the long run then rational functions, factorization of polynomials etc. and any kind of storage management. However, the advantage of this solution would be that one could use the full strength of C with respect to for instance data types, pointer arithmetic and dynamic storage allocation.

For the second solution we could use SACLIB, a library of C programs based on the SAC-2 system. Furthermore, SACLIB contains a garbage collector, so that we need not worry about that anymore. The disadvantage of this solution is, that we cannot utilize the whole power of C, since we must restrict the data types being used to SAC-lists, in order to guarantee that the garbage collection works correctly.

The qualities offered by SACLIB for implementing GRÖBNER can be summarized in the following table:

- ◇ Available for a lot of hardware platforms,
- ◇ availability of high-quality algebraic algorithms,
- ◇ readable code,
- ◇ fast,
- ◇ storage management can be done by the system,

◇ easy to handle.

In both cases, we have to implement a rudimentary form of genericity that suffices for our purposes. Either in a form that a dispatch happens at run-time or a kind of renaming of functions that can be done at compile-time. Before we will report on these two approaches in all details, we will outline which "generic features" we expect GRÖBNER to have and how we intend to make use of them.

2.4.1 Motivation for Generic Programming in Algebraic Computation

Mathematicians use the same names for algorithms over different data domains because by inspecting the arguments it can be decided which algorithm has to be used. This is not possible in conventional programming languages where two different routines must have different names. The desire to overload names has mainly two reasons:

◇ Some algorithms can be implemented *independently of the data domain*, e.g. the multiplication algorithm for polynomials does not depend on the domain of coefficients, and

◇ the algorithms should be *independent of the concrete data representation* that is chosen, e.g. the multiplication algorithm for polynomials should be the same regardless whether monomials are represented as lists, arrays, structures or whatever.

The advantages are obvious:

◇ The amount of code is reduced because algorithms do not have to be implemented separately for each combination of domains (polynomials over the rationals, over finite fields, over rational functions,...), and

◇ if once a hierarchy of domains is established, algorithms are independent of changes in the underlying domains, i.e. the "interface" between the domains can be defined easily.

2.5 Dynamic Dispatch at Run-Time

2.5.1 First Approach

A solution for the problems stated above is to wrap a structure around every element containing two pieces of information as shown in Figure 2.1:

1. A pointer to an array of functions (in fact it is an array of pointers to functions) that are applicable to the element and

2. a pointer to the element itself, which has to be of type `void*` because it may point to various types.

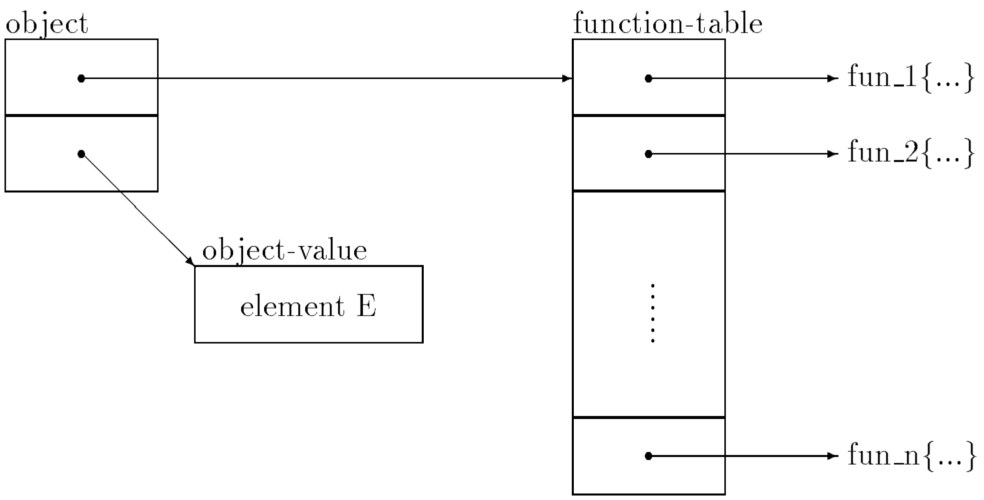

Figure 2.1: Object Structure

The types are declared as follows:

```
#define MAXFUN 20

typedef void* ((*PF)());/* Pointer to function returning (void*)*/

typedef PF Operations[ MAXFUN]; /* Array of pointers to functions
                                   returning (void*)  */
```

```
typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  } object;
```

The functions fun_1 to fun_n in Figure 2.1 are those functions that are applicable to the element E and are called *element-functions*. Note that the array of function-pointers has to be stored only once per class of elements. In the sequel this array is called *function-table*, the element E is called *object-value* and the structure wrapped around is called *object*.
The following conditions must hold:

- ◇ The index of the pointer to an element-function has to be fixed within the function-table, e.g. the index of the pointer to the *addition-function* must be 2 in *every* domain. These indices have to be fixed by the programmer.
- ◇ If in a domain a function does not exist, the pointer at the corresponding index in the function-table must be the null-pointer, e.g. in the domain *exponent array* there is no addition and therefore the second element in the function-table has to be the null-pointer.
- ◇ For every function that should be used generic a *Generic Function* has to be supplied. All these Generic Functions have a common pattern. When getting an object as an argument, the corresponding pointer in the function-table is evaluated and stored in the variable *fun*. In the case where this is not the null-pointer, *fun* is called with the *object-value* as the argument otherwise an error is detected.
- ◇ The element-functions must have `void*` arguments because they are called with the *object-values* as parameters, which are declared as `void*`.

Therefore a typical Generic Function looks as follows:

```
#define SUM 2

object *Sum( object *o1, object *o2)
  {
  PF fun;
  if( fun = (o1->type)[ SUM])
```

```
    return fun( o1->value, o2->value);
  printf("\nNo function Sum available for this object\n");
  }
```

2.5.2 Advanced Solution

When running examples with big input-size, one has the problem when to free memory. We chose a mechanism of *Reference Counting* for solving this problem. All objects as well as all types that are defined by the user (arrays, structures,...) must have an additional slot for a reference counter. The reference counter is an integer containing the number of pointers that have a reference to the particular object, array or structure.
Advanced object declaration:

```
typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  int ref_count;
  } object;
```

In addition, the assignment-operation for objects has to be defined. Each time an object gets a new value,

1. the old value can be freed, and
2. the reference counter of the new value has to be increased by one.

Therefore, each object has to be initialized by zero to indicate that there is no old value to be freed.
Assignment operation for objects:

```
void Set( object **o1, object *o2)
  {
  if( *o1) Free( *o1);
  *o1 = o2;
  o2->ref_count++;
  }
```

Freeing an object is done as follows:

1. Decrease the reference counter by one and

2. if the reference counter is zero then, free the object-value recursively otherwise nothing has to be done.

Moreover, memory is not freed immediately but the pointer to the memory-space to be freed is put into an array of such pointers, the *free-array*. Hence, if new memory has to be allocated, this can then be easily done by just taking a pointer from the free-array, which has to be supplied for each type one defines. We illustrate this allocation-freeing-process for *objects*. The global variable `next_object` stores the index in the free-array where a pointer to available memory-space can be taken. If no pointer is in the free-array, memory must be allocated explicitly by `malloc`.

```
object *allocate_object( void)
  {
  if( next_object >= 0)
    return free_object_array[ next_object--];
  else
    return (object*)malloc( sizeof( object));
  }
```

The size of the free-array is specified by the global variable `MAXFREE`, which is defined in the header file "generic.h". If the free-array for a particular type is already full, space must be freed explicitly by `free`. (Another possibility would be to use the C library function `realloc` to reallocate the free-array with larger size. We do not go into the details of this approach here.)

```
void Free( object *o)
  {
  if( !( -- o->ref_count))
    {
    (o->type)[ FREE]( o->value);
    if( ++next_object < MAXFREE)
      free_object_array[ next_object] = o;
    else free( o);
    }
  }
```

The global structure of this advanced solution is shown in Figure 2.2.

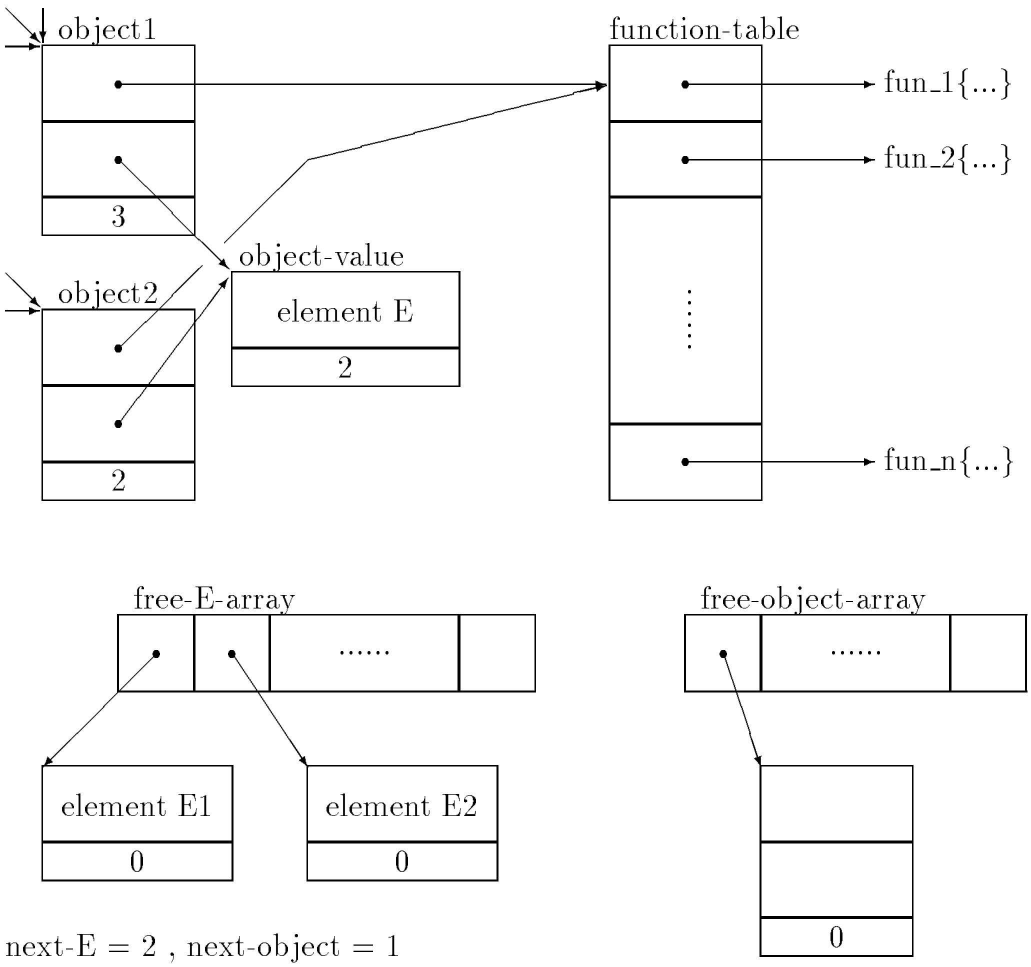

Figure 2.2: Advanced Object Structure

2.5.3 Description for the User

This section describes, which functions the programmer has to make available for self-defined domains in order to guarantee that this domain fits into the whole system, i.e. that reference counting as well as generic function calls are provided for this domain. Moreover, it contains an "algorithm", how to implement self-defined domains. For details on a concrete implementation see [Windsteiger, 1990a].

Type Definitions

The only restriction concerning the type definition is that it has to contain a reference counter. It is the decision of the user to implement the reference counter as a slot in a structure or an additional array element or whatever.

Function Prototypes

Element-functions must return a pointer to an object or `void` if they are used as procedures. The arguments have to be of type `void*` and must afterwards be casted to the respective type.

Global Variables

For each domain one has to provide a global variable of type `Operations` (the function-table for this domain), an array of size `MAXFREE` of pointers to elements of the domain (the free-array for this domain) and the integer variable, which contains the index of the next free element that can be allocated. The free-array and the integer variable are only needed if one decides to use the allocate-free-mechanism described in section 2.5.2.

Assignment Operation

The generic assignment operation `Set` must be used whenever an object is assigned to another object in order to invoke reference counting.

Additional Functions

The following functions must be available for each domain:

1. A *generation function*: This function gets a pointer to a domain element as an argument and returns a pointer to the whole object. In the object the pointer to the function-table and the pointer to the object-value have to be initialized, the reference counter of the produced object must be set to zero and the reference counter of the object-value has to be increased by one. Since this function is local to each domain it need not be put into the function-table. Another possibility would be

to allocate the new object in the generic functions (actually this is the better way).

2. An *initialization function* that initializes the function-table and the global integer variable mentioned in section 2.5.3.

Furthermore, for the allocate-free-mechanism are necessary:

- ◇ An *allocate-function* as described in section 2.5.2.
- ◇ A *free-function* as described in section 2.5.2. The index in the function-table must be `FREE`, a global constant that is defined in the header file "generic.h".

2.5.4 "Algorithm" for Implementing New Domains

In this section we start from the following situation: The element-functions are already available but we still have to adapt them to fit into our system. We consider the example of multiplication of monomials. The implementation we start from is the following:

```
typedef int *iea;

typedef struct
  {
  int coef;
  iea pp;   /* concrete representation of power-products */
  } monomial;

typedef monomial *mon;

mon product_m( mon m1, mon m2)
  {
  mon m0;
  m0 = new_m();

  m0->coef = m1->coef * m2->coef;
  m0->pp   = product_iea( m1->pp, m2->pp);

  return m0;
  }
```

The disadvantages are obvious: The type-definition of `monomial` depends on the type of the coefficients and the concrete representation of power-products. Hence, if one wants to study another domain of coefficients or another representation of power-products one has to change the definition as well as the code of all functions. The multiplication of the coefficients (`*`) and the power-products (`product_iea`) must be altered to the multiplication functions for the new coefficient domain and the new representation of power-products. The following changes are necessary to avoid these disadvantages:

1. Change type definition of monomial into

```
typedef struct
  {
  object *coef;
  object *pp;
  int ref_count;
  } monomial;
```

2. Change the function prototype:

```
object *product_mon( void*, void*);
```

3. Introduce the global variables as stated in section 2.5.3.

4. Substitute every occurrence of the assignment operator '`=`' by the generic procedure `Set` (if '`=`' was used with objects.).

5. Implement the additional functions as stated in section 2.5.3. On implementing the initialization function one has to consult the header file "generic.h". In this file one finds a list of all generic functions that are available. If the considered generic functions already exist nothing more has to be done. Otherwise one has to introduce a new generic function as described in all detail in section 2.5.1 and add this function to these in the file "generic.c". If necessary the global variable `MAXFUN` has to be increased.

2.5.5 Summary on this Solution

The "algorithm" given in the previous section for the multiplication of monomials seems to be rather complicated but all the work has to be done only once per new domain. Implementing the next element-function for monomials needs no more additional routines. In addition, if some domains are implemented most of the generic functions one needs are already available. The generic function calls could be improved because in the actual version it is assumed that all arguments of a function are of the same domain and due to this one only has to call the appropriate element-function stored in the function-table of the *first* argument. In an improved version one could check whether the arguments are of the same type and one could think about a strategy how to find the correct element-function in the case where the arguments do not have the same type. For any detail on the actual implementation see [Windsteiger, 1990a].

However, we did not go on implementing GRÖBNER mainly because of two reasons:

1. The mechanism of storage management may work as shown above, but it may also be very hard to keep this system running (especially when it grows) for users not familiar with C, and

2. we do not want to code all the basic algebraic algorithms in C from the scratch, since in our experiments with PCL we saw the huge amount of work arising for this purpose.

We concentrated our further work on the second solution, using SACLIB together with a renaming of functions happening at compile-time to realize a simple "imitation of generic programming".

2.6 SACLIB and Compile-Time Renaming

In this section, we report on our experiments using the SAC-2 library for the implementation of GRÖBNER. Since this is a library of C routines, it can be used from every C program without loss of speed. The only data type in SACLIB is *Word*, which can be either an *integer* or a *list*. This fact is heavily used by the garbage collector, which only works under the assumption that all data in the storage are either integers ore lists. For reasons of portability it was decided to use only this single data type within SACLIB, since the resulting garbage collection scheme is very simple and therefore easy to adapt to different hardware platforms. (Anyway, lists are sufficient for most of the purposes).

This has the consequence that, if one wants to use the garbage collector of SACLIB, one should restrict ones data types to SAC-lists. It would theoretically be possible to define own types but it would then be required to provide a fitting storage management for this new type. This would end up in *several different* storage administrations working together, which would only cause troubles in the long run, especially when the system is to be ported to different hardware. Therefore, the only data type available in GRÖBNER is the SAC-list, because we get rid of all the troubles concerning storage management.

We now concentrate on the mechanism of genericity used in GRÖBNER. Similar to a package in LISP, all function names within a common domain are taged. By renaming these tags we can then branch to different domains. This mechanism is from now on called *Simulated Package Principle (SPP)* and will also be described in a subsequent paper (see [Buchberger *et al.*, 1992]).

2.6.1 The Simulated Package Principle

The two aspects of generic programming we want to use in GRÖBNER are:

1. The possibility to use *different data domains* without the necessity of duplicating the source code, e.g. using various coefficient domains for monomials without having to change the code of the arithmetic on monomials, and

2. the possibility to use *different data representations* in underlying domains, e.g. using various representations of power products without having to change the code of monomials.

As already mentioned above, SPP works via renaming of functions at compile-time. For this renaming we use the `#define` construct of C. Consider a domain `new_domain` consisting of two subdomains `subdomain_1` and `subdomain_2`. Implementing this domain directly would result in a dependence of the code of `new_domain` on the actual representation of `subdomain_1` and `subdomain_2` similar to the situation in Section 2.5.4.

In SPP we handle this problem by defining interfaces between the new domain and each subdomain. Each of these interfaces actually consists of two parts:

1. A *virtual domain* between the domain and each subdomain, and

2. a *coercion unit* between each virtual domain and the actual representation of the corresponding subdomain.

Every "generic function call" in the new domain must use a *virtual function* in a virtual domain which is mapped to the correct function by the coercion unit as shown in Figure 2.3.

This figure is to be interpreted as follows: Suppose we want to create a new domain called 'New Domain' that is composed of two subdomains. `fun_1` and `fun_2` are generic calls to functions from the first and second subdomain respectively. Domain 1 and domain 2 are two possible representations of the first subdomain whereas domain 3 and domain 4 are actual representations of the second subdomain, i.e. `fun_1_ar1` and `fun_1_ar2` are possible interpretations of the generic `fun_1`, analogously `fun_2_ar3` and `fun_2_ar4`. We create two virtual domains containing virtual functions `fun_1_vd1` and `fun_2_vd2` corresponding to `fun_1` and `fun_2` respectively. Furthermore, we create a coercion unit between each virtual domain and the actual representations of the respective subdomains. At compile-time we decide, which coercion units should to be active and which to be inactive (in the figure, the dashed coercion units denote inactive ones). The active coercion units map the virtual functions to fitting functions in the actual representations of the corresponding domains. Changing from one representation to another is done by making another coercion unit active at compile-time.

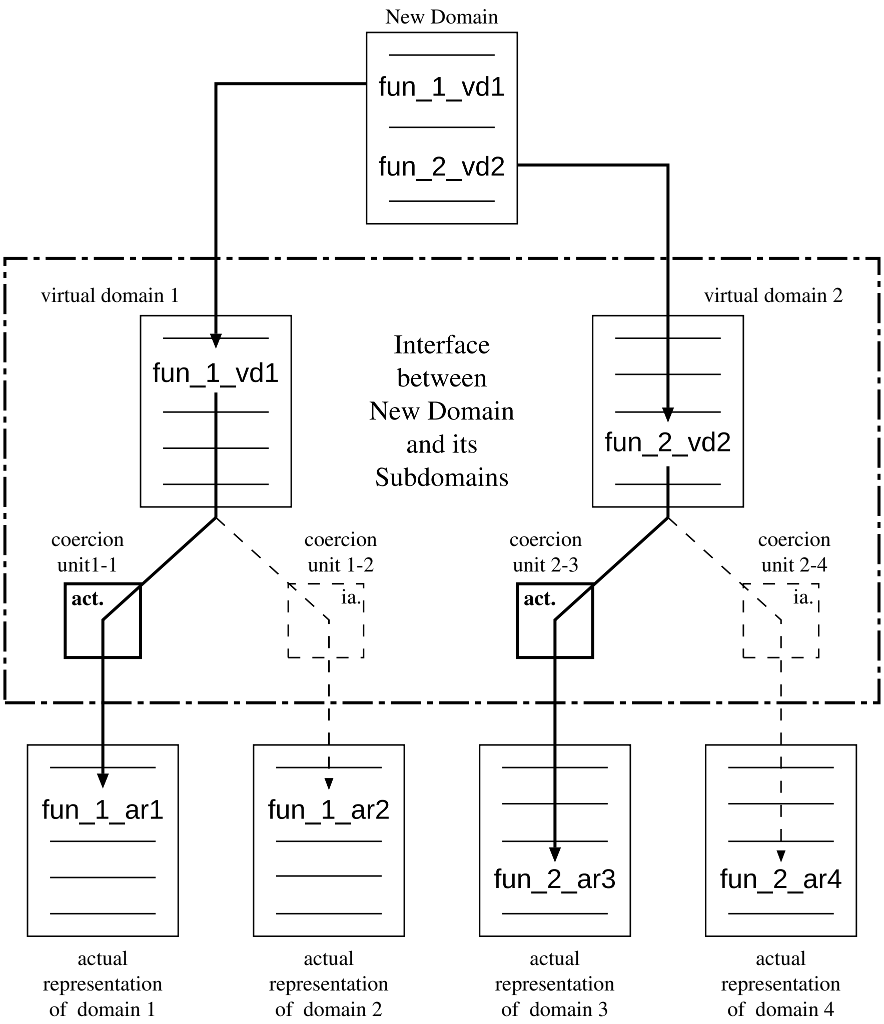

Figure 2.3: Interface between Domains

Whenever we desire to call the generic function `fun_1` (or `fun_2`) in the new domain, we have to call the virtual function `fun_1_vd1` (`fun_2_vd2`) directly, but, since the programmer knows the structure of the domain, it should not cause any problems to call the right virtual function.

There are parallels to the approach discussed in the previous section in following sense:

◇ The direct call of a virtual function corresponds to the call that is done in a Generic Function call at run-time (the slight difference is that the correct virtual function has to be chosen by the programmer, whereas the Generic Function does this "without assistance of the programmer"), and

◇ an active conversion unit of a domain corresponds to the function-table of a domain.

Consider now the case of monomials over different coefficient domains and different representations of power products. The code for the multiplication of monomials looks as follows:

```
typedef Word Mon; /* from SACLIB */

Mon product_mon( Mon, Mon); /* function-prototype */

/*--------- Selectors --------*/

#define coefficient( m) FIRST( m)
#define power_product( m) SECOND( m)

/*--------- Constructor --------*/

#define monomial( c, pp) LIST2( c, pp)

Mon product_mon( Mon m1, Mon m2)
  {
  Coef c1, c2;
  PP pp1, pp2;
  c1  = coefficient( m1);
  c2  = coefficient( m2);
  pp1 = power_product( m1);
```

```
  pp2 = power_product( m2);
  return monomial( product_coef( c1, c2),
                   product_pp( pp1, pp2));
  }
```

The data structure for monomials is a list consisting of two elements, the coefficient and the power product. These parts are selected by the selectors `coefficient` and `power_product`, which use the SACLIB-functions `FIRST` and `SECOND` to get the first and second element of a list respectively. Having a coefficient and a power product, a monomial is constructed out of them by the constructor `monomial`, which calls the SACLIB-function `LIST2` to make list of two elements. `Coef` and `PP` are virtual data types for coefficients and power products, `product_coef` and `product_pp` are virtual functions for multiplication of coefficients and power products. We have to provide coercion units if we want to run the program for concrete representations of the coefficient domain and the domain of power products. A coercion unit consists of three parts:

Renaming Unit In this part, the virtual functions are renamed to existing functions in the actual representation of corresponding domain.

Inclusion Unit The inclusion of the appropriate header files for the concrete implementation of the domain is done in this unit.

Definition Unit The definition of the virtual types and the translation into existing types goes into this part.

In our concrete example, the corresponding coercion units provided for coefficient domains *integer*, *rational numbers* and *rational functions* and domain of power products *exponent list* look as follows:

+++++ Conversion unit `coef.version.int` +++++

```
/* >>>>> Renaming Unit <<<<< */

#define product_coef product_int

/* >>>>> Inclusion Unit <<<<< */

#include "int.h"
```

```
/* >>>>> Definition Unit <<<<< */

typedef Int Coef;
```

+++++ Conversion unit `coef.version.rat` +++++

```
/* >>>>> Renaming Unit <<<<< */

#define product_coef product_rat

/* >>>>> Inclusion Unit <<<<< */

#include "rat.h"

/* >>>>> Definition Unit <<<<< */

typedef Rat Coef;
```

+++++ Conversion unit `coef.version.rf` +++++

```
/* >>>>> Renaming Unit <<<<< */

#define product_coef product_rf

/* >>>>> Inclusion Unit <<<<< */

#include "rf.h"

/* >>>>> Definition Unit <<<<< */

typedef RF Coef;
```

+++++ Conversion unit `pp.version.el` +++++

```
/* >>>>> Renaming Unit <<<<< */

#define product_pp product_el
```

```
/* >>>>> Inclusion Unit <<<<< */

#include "el.h"

/* >>>>> Definition Unit <<<<< */

typedef EL PP;
```

(Of course, in the real implementation of the coercion units, the renaming unit contains lot of other conversions).

Activation of the desired coercion unit is done in the central header file of GRÖBNER, `groebner.h`, by the following mechanism:

```
#define integers 1
#define rational_numbers 2
#define rational_functions 3

#define coefficient_domain integers   /* <<-------- */

#if coefficient_domain == integers
#  include "coef.version.int"
else
#if coefficient_domain == rational_numbers
#  include "coef.version.rat"
else
#if coefficient_domain == rational_functions
#  include "coef.version.rf"
#endif
#endif
#endif
```

If the domain of coefficients should be changed, only the line marked by the arrow has to be changed.

The actual implementations of the virtual functions `product_coef` and `product_pp` could look as follows (under the assumption that the active conversion units are `coef.version.int` and `pp.version.el` respectively).

```
/************** File int.h **************/

typedef Word Int;
Int product_int( Int, Int);

/************** File int.c **************/

Int product_int( Int i1, Int i2)
  {
  return IPROD( i1, i2); /* long integer product from SACLIB */
  }

/************** File el.h **************/

typedef Word EL;
EL product_el( EL, EL);

/************** File el.c **************/

EL product_el( EL el1, EL el2)
  {
  int e1, e2;
  EL el0=EMPTY;
  while( el1 != EMPTY)
    {
    e1 = FIRST( el1);
    e2 = FIRST( el2);
    el1 = REST( el1);
    el2 = REST( el2);
    el0 = COMP( e1 + e2, el0);
    }
  return INV( el0);
  }
```

Finally, we list all virtual domains implemented so far together with the corresponding virtual data types and available actual representations of these virtual domains in Table 2.1 (those domains marked by (*) are not implemented yet, but will be added in the near future, lines marked by (**) represent

Virtual Domain	Virtual Data Type	Concrete Data Type	Concrete Domain
Coefficient	Coef	Int	Integer
		Rat	Rational Number
		RF	Rational Function
		FF	Finite Field (*)
Power Product	PP	EL	Exponent List
		GE	Gödel Exponent (*)
Data Structure of Polynomials	PDS	DPL	Distributive Polynomials as Lists
		SP	SAC-internal Polynomials (**)
Polynomials	Poly	PDS	the respective data structure used
		PC	Polynomials with Cofactors

Table 2.1: Virtual Domains

other possibilities of domains to be used, but are not planned to be implemented).

The structure of polynomials needs some explanation: For the computation of Gröbner bases, we might want to adjoin additional information to each polynomial, such as for instance the "cofactors" of a polynomial with respect to the polynomials in the basis (see Section 3), or whatever. However, the implementation of the Gröbner bases algorithm should be independent of this additional information a polynomial carries.

Therefore, we introduce a virtual domain `Poly` which represents the domain of "representations of polynomials that can be used in the Gröbner bases algorithm". This can either be a "pure polynomial" or a polynomial carrying some supplementary information. Whatever the actual instance of `Poly` is, it will contain "the polynomial itself" as a subdomain.

This subdomain should then be independent of the actual data structure we use for a polynomial, since we can choose various ways of internal representations of a polynomial. Distributive representation, recursive representation, sparse representation are only some of the possibilities. Of course, the data types mentioned above (polynomials, polynomials with additional information,...) should not depend on the current representation of a single polynomial. Hence, we have to create an additional layer, the *Data Structure of Polynomials*, which we will use as the virtual domain for a subdomain that represents a “pure polynomial”. We illustrate this structure in Figure 2.4.

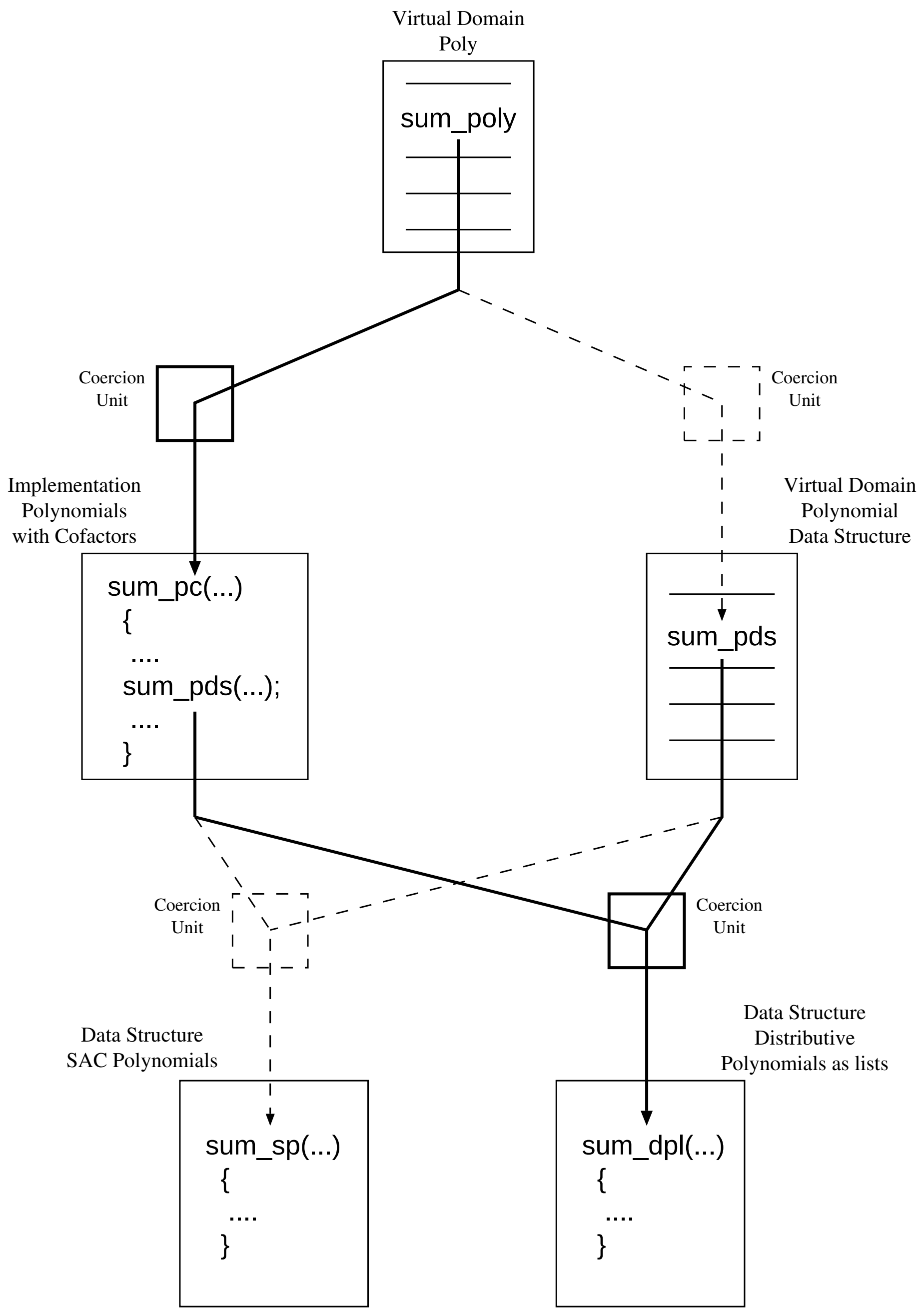

Figure 2.4: Structure of Polynomials

2.7 About GRÖBNER

Before we go into the details we state that it is just an *intermediate version of GRÖBNER* that we report on in this paper. In later versions of the package, strategies may have changed and pieces of the code may be modified or deleted.

2.7.1 Naming Conventions

In this section, we want to discuss the rules, by which functions and variables are given names in GRÖBNER. First of all, *only* lower-case letters are used for naming both functions and variables. Exceptions to this rule are

- ◇ calls to the SAC library, which consist *only* of upper-case letters,
- ◇ symbolic constants defined at compile-time by `#define`, which consist only of upper-case letters, and
- ◇ functions that have a certain name in literature (e.g. S-polynomials are always spelled with uppercase "S", so we do not change this habit in the naming of the function).

For semantical partition of names we use the underscore ("_"), because we think that this improves the readability of the programs. An example for this use of the underscore is the predicate `is_leading_monomial_greater`. Furthermore, all functions are tagged with an abbreviation of the domain of the arguments for which they are applied. A typical example for this is the summation function for dense polynomials in list representation, which is called `sum_dpl`. (See Section 2.6.1 for an explanation, why this tagging is necessary).

We do not stress the semantical aspect of naming (i.e. "What names should we choose for particular functions, predicates and variables?") in this section. Just as a guideline, we choose rather *long names* for global variables, function- and predicate names and rather *short names* for local variables inside functions and predicates. In the end, the code should be in a one-to-one correspondence with the written text of the Gröbner bases-book.

2.7.2 Features of GRÖBNER

GRÖBNER is a package for computing Gröbner bases that offers generic facilities with respect to

- ◇ the *domain of coefficients of the polynomials*,
- ◇ the *representation of power products*,
- ◇ the *representation of polynomials*,
- ◇ the *ordering of power products within a polynomial*,
- ◇ the *sequence, in which S-polynomials are considered for reduction during the Gröbner bases algorithm.*

(By introduction of new virtual domains as shown on Section 2.6.1, it is easily possible to introduce additional generic features).

This has the consequence that all the items listed above can be easily changed or extended by the user. Furthermore, several different versions of the Gröbner bases algorithm and two different normal form algorithms are implemented, which makes it possible to apply different strategies for different applications. These features of GRÖBNER are especially valuable for doing research on Gröbner bases, where one does not want to use a "black-box algorithm" for computing Gröbner bases, but needs the possibility to inspect the impact of different strategies during the computation on the computing time.

We want to treat in- and output in a little bit more detail.

Input

For most of the domains implemented we supply "rudimentary" input routines, in the sense that they *only* serve for getting data into the system quickly. They are very much oriented on the internal structure of the data to be read in and do neither support features like error checking or -correcting nor accept input in a "written-like" form.

We illustrate this kind of input routine in the example of the routine `read_dpl`, the routine for reading in polynomials in distributive list representation, which is a list of monomials. Of course, this function then uses the input routine for monomials and iterates it in a loop until the users tells to stop. Take the polynomial $\frac{2}{3}xy^2z + \frac{1}{4}x^2$. The internal representation as a list of monomials, where each monomial is a list consisting of a coefficient and an exponent list, looks as follows:

$$((\tfrac{2}{3} \ (1 \ \ 2 \ \ 1)) \quad (\tfrac{1}{4} \ (2 \ \ 0 \ \ 0)))$$

The sequence of keystrokes to enter this polynomial using `read_dpl` would be

$$2 \,/\, 3 \boxed{\hookleftarrow} 1 \boxed{\sqcup} 2 \boxed{\sqcup} 1 \boxed{\hookleftarrow} 1 \,/\, 4 \boxed{\hookleftarrow} 2 \boxed{\sqcup} 0 \boxed{\sqcup} 0 \,\$ \boxed{\hookleftarrow} \ ,$$

where the $ sign must be put immediately after the last exponent to mark the end of the input, $\boxed{\hookleftarrow}$ denotes the RETURN-key and $\boxed{\sqcup}$ denotes blank.

This gives the possibility to enter input very quickly but users not familiar with the internal data representation might have troubles to enter a slightly longer polynomial correctly. This problem takes even more effect if one has to enter sets of polynomials.

For this reason, GRÖBNER provides input routines for polynomials and polynomial sets, which accept written-like input and check for syntax errors. The syntax specification for polynomials and polynomial sets is as follows:

- ◇ An element of the coefficient domain is a polynomial,
- ◇ a variable is a polynomial, and
- ◇ if p, p_1 and p_2 are polynomials and n an integer then
  - ○ (p),
  - ○ $p_1 + p_2$,
  - ○ $p_1 - p_2$,
  - ○ $p_1 * p_2$,
  - ○ p_1/p_2 (if p_2 is constant),
  - ○ $-p$,
  - ○ $p ** n$, and
  - ○ p^n

  are again polynomials.
- ◇ A single polynomial is a sequence of polynomials, and
- ◇ if s_p is a sequence of polynomials and p a polynomial then

◦ s_p , p

is again a sequence of polynomials.

⋄ The empty set {} is a polynomial set, and

⋄ if s_p is a sequence of polynomials then

◦ $\{s_p\}$

is a polynomial set.

These routines, called `parse_poly` and `parse_ps` respectively, parse the input according to the syntax as specified above and return a polynomial or polynomial set. To enter the polynomial $\frac{2}{3}xy^2z + \frac{1}{4}x^2$, the input for `parse_poly` could look as follows:

```
2/3*x*y^2*z + 1/4*x**2
```

This form has the advantage that it is much easier to memorize for the user although it needs more symbols to be typed. By the specification of the syntax, it is also possible to parse polynomial expressions like

```
(2/3x*y^2*z + 1/4*x**2)^20 - 30*(x^6 + 1)
```

Furthermore, these input routines recognize syntactical errors in the input and, after detecting an error, try to recover from it to be ready for reentering the input. The detection of an error consists of two parts:

⋄ A specification, *which kind of error* occurred and

⋄ the input that was ignored *after detecting the error*.

When having large inputs, especially the second part of this information is very important for correcting the error, since it is a rather precise hint *where* the error can be found.

Output

Similar to the input routines, GRÖBNER offers several types of output routines. The first type, always named `print_tag`, where `tag` is the abbreviation for the data type of the argument, produces output that corresponds to the internal representation of the printed object. We illustrate this again with `print_dpl`, the print routine for distributive polynomials in list representation. The result of printing the polynomial $\frac{2}{3}xy^2z + \frac{1}{4}x^2$ using `print_dpl` would be

```
---- Begin Poly ----
(2/3 [1 2 1])
(1/4 [2 0 0])
---- End Poly ----
```

The advantage of this representation of polynomials is that it is very easy to *recognize structures* in a polynomial or in a set of polynomials (e.g. a triangular structure in a polynomial set (i.e. the set contains one univariate polynomial, one bivariate polynomial, etc.) is very easy detectable), which might often be valuable when dealing with Gröbner bases.

However, sometimes a written-like output is desirable. For this purpose, GRÖBNER offers for most of the domains print routines, which produce this kind of output. On the one hand, a "one-dimensional" form, which corresponds *exactly* to the input form of the particular data. This means, that the output produced by these routines can directly be read into the system again by the corresponding input routines `parse_tag`.

On the other hand, GRÖBNER offers a "two-dimensional" output format, i.e. exponents are on another level than the basis, numerator and denominator of a fraction are positioned one above the other, etc. like in the example

```
2   2   1 2
- x y z + - x
3       4
```

To achieve this kind of output, we divide the output into single lines, associate a list of characters with each of these lines, and collect these lists in an array. We will again demonstrate some details on the print routine for distributive polynomials, `pretty_print_dpl`. The number of lines needed (the size of the array) depends on the height of the output, in the case of a polynomial on the height of a monomial, which depends on the height of the coefficient and the height of the power product. All these parameters are known at compile-time and are therefore set by a `#define` in dependence of the used domain of coefficients and the representation of power products.

In the example above, using rational number coefficients, the height of a monomial is three, and the separation into an array of lines looks as follows:

line[0]	$\rightarrow$	2					2					1			2
line[1]	$\rightarrow$	-		x		y		z		+		-		x	
line[2]	$\rightarrow$	3										4			

(The elements of the list that are left free in the figure above contain the blank character). These lists printed one after the other in separate lines give the result that we want.

The most important subroutine in the pretty-print-procedure is the formatting step, which fills each of the lists with characters and blanks at the appropriate places. This formatting is programmed in such a way, that it can be used for two purposes: On the one hand in the preprocessing for the print routine and on the other hand as a subroutine in formatting procedures of more complicated domains (e.g. formatting of polynomials is used as a subroutine in formatting of polynomial sets).

Very important for readable output of large data is the choice of linebreaks. We use the following strategy:

- ◇ Elements of domains up to polynomials (i.e. all coefficient domains, power products, and monomials) are printed into *one* line.
- ◇ Environment variables (UNIX) hold the *width* of the window, in which the program runs, and a value that specifies the offset of the output's right margin from the right border of the window. These variables are called GBTERMWIDTH and GBRIGHTMARGIN, and have default values "80" and "15", respectively.
- ◇ Within a polynomial, a linebreak is preferably made *between* monomials, namely at the *first boundary between monomials, that occurs after (GBTERMWIDTH - GBRIGHTMARGIN) symbols in a line.*
- ◇ At the latest, a linebreak is made *after GBTERMWIDTH symbols in a line*, without consideration of monomial boundaries. Nevertheless, at the next monomial boundary a linebreak is made as well.
- ◇ Within a polynomial set, polynomials are treated as above. Each polynomial starts at the beginning of a new line.

We will now illustrate, how this mechanism for choosing appropriate places for linebreaks works in practice. The variables GBTERMWIDTH and

GBRIGHTMARGIN have the values “50” and “10” respectively. Hence, the region where linebreaks are placed within a polynomial is between the 40th and the 50th symbol in a line. For the first polynomial, it is possible to split the polynomial at the boundary between monomials within this region, in the second example we see that within this region there is monomial boundary, therefore it is split after the 50th symbol in the line.

```
          45373    2   14216       515111261
x2 x3 + ------- x3   + ----- x1 + --------- x2 +
        2068243        20643       10341215

   549999      27070792
+ ------- x3 + --------
  2068243      2068243
```

```
 549999         1142332   2   66538052       240364
------- x1 x3 - ------- x3   - -------- x1 - ------
2068243         2068243        2068243         1034

60112
----- x2
1215
```

Caution: If GRÖBNER runs under a windowing system, resizing of a window does *not* change the values of the environment variable GBTERMWIDTH, therefore, the window, in which GRÖBNER runs, should always be of width GBTERMWIDTH, otherwise the output might not look as desired. Anyway, since the format of the output takes care of the width of the window, there should not be a reason to resize the window during the computation.

2.7.3 Structure of GRÖBNER

In this section, we want to give a survey on the structure of the whole package. GRÖBNER has *one* central header file, named `groebner.h`. Only this file has to be included in the remaining source files. This file manages

- ◇ inclusion of all other header files,
- ◇ inclusion of the appropriate coercion units,
- ◇ selection of the desired ordering for the monomials within a polynomial and for pairs of polynomials within a set of such pairs, and
- ◇ selection of the desired normal form algorithm.

Since these fundamental tasks are done in this file, all other files of the package depend on it and therefore, if `groebner.h` is changed, the whole package should be recompiled.

The implementation of a data domain is always split into a *header file* and a *source file*. If the abbreviation of the domain is `dom` then the header file and the source file are named `dom.h` and `dom.c` respectively. The header file contains

- ◇ *type definitions* for `dom`,
- ◇ *prototypes of functions* implemented in the corresponding source file,
- ◇ *declarations of global variables* introduced in `dom` (if any),
- ◇ *definitions of constants* introduced in `dom` by `#define` (if any), and
- ◇ *definitions of macros* for `dom` (if any). (We use macros only in situations where a function or procedure contains only one instruction, e.g. the selection of the leading monomial of a polynomial, which is only the call of the SACLIB function that picks the first element of a list).

The corresponding source file contains

- ◇ the *actual implementation* of functions for `dom` and
- ◇ the *definitions of global variables.*

The functions and procedures constituting GRÖBNER are collected in several files, following the principle, that

- ◇ parts of the code, which contribute to the implementation of the same data domain stay together and are partitioned into the header file and the source file as described above (e.g. `dpl.h` and `dpl.c` contain the definitions and routines for arithmetic of distributive polynomials),
- ◇ each coercion unit (see Section 2.6.1) goes into a separate file, and
- ◇ auxiliary functions for special purposes are collected in own files (e.g. `error.c` contains routines for error handling and printing of the appropriate error- or warning message).

We will discuss the files containing such auxiliary routines briefly in the following sections.

Conversion between Data Types

FILES: `conversions.h`, `conversions.c`.
In these files, some conversion routines between fundamental data types can be found, such as conversion from a string (i.e. an array of characters) to a list of characters and vice versa and from a string of digits to an integer. Furthermore, conversions between coefficients and polynomials (i.e. every single coefficient can be viewed as a polynomial) and between variables and polynomials (i.e. every single variable can be viewed as a polynomial) can be found there (we do not go into the details here). In the course of extending the package, some more conversion routines will be added to this file.

Error Handling

FILES: `error.h`, `error.c`.
In the file `error.c` we define a standard error function, which will be called whenever an error occurs or the program runs into some other special branches of which the user should be informed about (e.g. an undeclared variable is detected in the input). For this purpose, error- and warning codes are declared in `error.h` to distinguish between various classes of errors or warnings. This has the advantage that, in case of detecting an error, this function needs to be called with just two arguments – the error code and the name of the function, where the error occurred – and printing of an appropriate message in a standardized format is done "from alone", instead of having to insert lots of print-statements.

Online Help Facility

FILES: `help.h`, `help.c`.
For each situation where the user is asked for input we supply a brief help page that describes, what kind of input the system is expecting.

Initialization of the System

FILES: `init.h`, `init.c`.
In this part of the package initialization of the system is established. The procedure `initialize_globals` processes the information concerning the desired orderings, normal form algorithms etc. contained in the global header file `groebner.h`. Depending on this information, global variables are initialized and a summary of the settings is printed. Processing of command line arguments given when calling GRÖBNER is done in the procedure `process_command_line`. Start GRÖBNER with the command line argument "h" to get information about possible parameters to be given when starting the program.

Since it is desirable to store large input data in files to prevent oneself from repeatedly entering them into the system and also to store the results in files, GRÖBNER allows input from external files and output to external files. By default, input comes from `stdin` (normally the keyboard) and output goes to `stdout` (normally the screen). In the procedure tt open_IO_files, which is called from `process_command_line`, the initialization of these files for in- and output is done.

Input and Output

FILES: `input.h`, `input.y`, `input.c`, `input.lex`, `lex.c`, `output.h`, `output.c`.
For parsing of polynomials and polynomial sets as described in Section 2.7.2 we use the UNIX-programs *lex* and *yacc*. lex produces a lexical analyzer, which partitions the input stream into strings matching given expressions. By this mechanism, variables and digits can be recognized in the input stream. The lexical analyzer is used by yacc, which produces a parser according to given grammar rules. The syntax specification for polynomials and polynomial sets given in the previous section can easily be transformed to grammar rules fitting for yacc. These grammar rules are contained in the file `input.y`, the resulting C programs are in `input.c`, the description of expressions for

lex can be found in `input.lex`, and the resulting lexical analyzer produced by lex is in the file `lex.c`.

The file `output.c` contains additional routines for two-dimensional output as described in Section 2.7.2. For each domain a formatting routine is supplied, which produces the array of lists of characters and sets marks where to break lines. The procedure `print_lines` sets additional linebreaks (if necessary) and prints the lists to the desired output file afterwards. The strategies, how additional linebreaks are set, are discussed in all detail in Section 2.7.2.

Shell Commands

FILES: `shell.h`, `shell.c`.
With the procedure `exec_command` it is possible to start shell commands from inside GRÖBNER. If one enters *!command* at the place where polynomials or polynomial sets are expected then *command* is executed in a "bourne shell". The C shell command *setenv* is also allowed, and is programmed in such a way that it really influences the environment, *in which GRÖBNER runs* and not just the environment of the shell started from inside GRÖBNER.

Additional List Operations

FILES: `list.h`, `list.c`.
We implemented some useful operations on lists, which are not implemented in SACLIB.

Miscellaneous

FILE: `groebner_globals.h`.
Some global definitions for the whole package are collected in this header file.

2.8 Using GRÖBNER

In this section, we will briefly summarize how to use GRÖBNER. In this context we will list the environment variables that should be set for running the package, the global variables introduced so far, discuss some of the most important SACLIB routines, and close with some timing statistics.

Name	Default	Description
GBINPUTPATH	Home-directory	The name of the directory, where the system searches for input files
GBOUTPUTPATH	Home-directory	The name of the directory, where the system should place output files
GBTERMWIDTH	80	The number of columns of the display, on which the output is displayed
GBRIGHTMARGIN	15	See Section 2.7.2 on output for the description of this variable
GBMANPATH	Home-directory	The name of the directory, where the system searches for manual pages
NODETYPE	None	Just for statistics, the system prints the type and the name of the machine, on which it was started
HOST	None	
TERM	None	If TERM=xterm, then manual pages are displayed in a window by itself

Table 2.2: Environment Variables

2.8.1 Environment Variables

In Table 2.2 we list all environment variables used in GRÖBNER. If they are not set they have default values.

2.8.2 Global Variables

The names of global variables always end with an underscore (_), so that global variables can be easily distinguished from local variables. In Table 2.3 we will list all global variables used so far.

NAME	TYPE	DESCRIPTION
number_of_variables_	int	Number of variables in the polynomial ring
variables_	Word	List of polynomial variables
number_of_field_variables_	int	If coefficient domain are rational functions, the number of variables in the rational functions
field_variables_	Word	List of rational function variables
order_matrix_	Word	If ordering of power products is done by matrices (see Section 1.6), this variable holds the used matrix
variable_permutation_	Word	Permutation of polynomial variables
in_file_	FILE*	File for input
out_file_	FILE*	File for output
environment_array_	char*[]	Array of strings holding the environment variables and their values
error_	Boolean	error_ = TRUE if an error occurred while parsing the input
error_count_	int	Number of errors occurred so far
parsed_poly_	PDS	Parsed polynomial
parsed_poly_set_	PS	Parsed polynomial set
label_	int	Current label for type "labeled polynomials"
number_of_pairs_	int	Number of pairs in the pair set
number_of_reduced_pairs_	int	Number of pairs already considered for reduction

Table 2.3: Global Variables

2.8.3 SACLIB Commands

In this section, we list some of the most important commands from the SAC library, which are used in GRÖBNER.

`FIRST( Word w)`

Returns the first element of the list `w`. (Since `FIRST` is implemented as a macro, it should only be used for non-empty `w`).

`RED( Word w)`

Returns the list consisting of all elements of `w` but the first. In GRÖBNER we renamed `RED` to `REST`. For the same reasons as `FIRST`, `RED` should only be used for non-empty `w`.

`COMP( Word a, Word w)`

Returns the composition of `a` and `w`, i.e. the list `w` prepended by `a`. (`FIRST`, `RED`, and `COMP` are analoga to `car`, `cdr`, and `cons` in LISP).

`ADV( Word w, Word *a, Word *r)`

"Advance on `w`", i.e. `a = FIRST( w)` and `r = RED( w)`.

`BETA`

`BETA` is the representation of the empty list in SACLIB. In GRÖBNER it is renamed to `EMPTY`.

`LIST1( Word w)`

Returns the list containing `w`.

`LIST2( Word w1, Word w2)`

Returns the list containing `w1` and `w2`.

`INV( Word w)`

Returns the inverse of the list `w`.

`CONC( Word w1, Word w2)`

Returns the concatenation of the two lists `w1` and `w2`.

`LENGTH( Word w)`

Returns the length of the list `w`.

2.9 Timing Statistics

In this section, we compile some timings of GRÖBNER focusing on

◇ the impact of the used ordering for monomials on the computing time,

◇ the impact of the used ordering for pairs of polynomials on the computing time (the sequence of reductions of S-polynomials depends on this ordering),

◇ the comparison of the different variants of the Gröbner bases algorithm implemented in GRÖBNER. Descriptions of the variants can be found in the source code documentation in Chapter 3, and

◇ the comparison of GRÖBNER with other implementations of Gröbner bases.

Since it is very difficult to find out a general rule (or just heuristics), which ordering or which variant of the algorithm can be expected to be in general (or at least most of the times) the best choice, we list all timings for some examples. An average over several examples would not show the influence of the several components in such detail.

We use the following abbreviations:

lex ...	purely lexical ordering,
tdeg	total degree lexical ordering,
itdeg	total degree inverse lexical ordering,
Crit	Gröbner bases algorithm variant `gb_criteria`,
RA	Gröbner bases algorithm variant `gb_reduce_all`,
SPS	Gröbner bases algorithm variant `gb_small_pair_set`,
MAP	Built-in MAPLE Gröbner bases algorithm,
Mat	Built-in *Mathematica* Gröbner bases algorithm,
n.i.	not implemented,
O.o.M.	Out of Memory.

All computations are carried out on an Apollo DN3500 (approximately 5 MIPS), All times are measured in seconds. "Out of Memory" is to be interpreted as follows: SACLIB terminated the program since after a garbage collection more than 90% of allocated memory was in use (100% $\hat{=}$ 1.2 MB). ∞ denotes "infinite looping" (no termination after 1 day).

Example 9.1

$$
\begin{aligned}
&-x_1^2/2 - x_2^2/2 + x_3^2\\
&x_1x_2 - 2x_3x_2 + x_1x_3\\
&x_1^2 - x_2
\end{aligned}
$$

$x_1 > x_2 > x_3$						
Ordering on Monomials	Ordering on Polynomial Pairs	Variant of Algorithm			MAP	Mat
		Crit	RA	SPS		
lex	lex	1,55	1,18	1,66	—	—
lex	tdeg	2,96	0,38	2,63	—	—
lex	itdeg	3,08	0,35	15,45	—	—
lex	?	—	—	—	22,03	1,85
tdeg	itdeg	0,23	0,25	0,25	—	—
tdeg	tdeg	0,25	0,23	0,23	—	—
tdeg	?	—	—	—	13,46	n.i.

Example 9.2

$$-x_1^2/2 - x_2^2/2 + 1234567/987654321 x_2 x_3$$
$$x_1 x_2 - 456789/7654321 x_3 + x_1 x_3$$
$$x_1^2 - 333337/555552 x_2$$

$x_1 > x_2 > x_3$						
Ordering on Monomials	Ordering on Polynomial Pairs	Variant of Algorithm			MAP	Mat
		Crit	RA	SPS		
lex	lex	42,83	43,73	53,21	—	—
lex	tdeg	6,30	4,60	5,36	—	—
lex	itdeg	4,80	3,51	5,78	—	—
lex	?	—	—	—	30,10	5,65
tdeg	itdeg	1,10	1,28	1,16	—	—
tdeg	tdeg	1,22	1,48	1,16	—	—
tdeg	?	—	—	—	11,38	n.i.

Example 9.3

$$x_1^2 x_2 - 15 x_1 x_2 x_3 - 17 x_1 x_3 - 13 x_1^2 - 7$$
$$x_1 x_2 x_3 - 19 x_2 x_3 - 5 x_3$$
$$x_1^3 - 2 x_1 x_3 - 17 x_3$$

$x_1 > x_2 > x_3$						
Ordering on Monomials	Ordering on Polynomial Pairs	Variant of Algorithm			MAP	Mat
		Crit	RA	SPS		
lex	lex	33,98	24,08	42,59	—	—
lex	tdeg	O.o.M.	48,05	O.o.M.	—	—
lex	itdeg	O.o.M.	24.447,28	O.o.M.	—	—
lex	?	—	—	—	41,30	∞
tdeg	itdeg	19,25	2,78	10,66	—	—
tdeg	tdeg	7,81	4,45	5,75	—	—
tdeg	?	—	—	—	35,06	n.i.

Example 9.4

$$x_1^2x_2x_3 - 15x_1x_2^2x_3 - 17x_1x_3 - 13x_1^2 - 7$$
$$x_1x_2x_3 - 19x_2x_3 - 5x_3$$
$$x_1^3 - 2x_1x_3 - 17x_3$$

$x_1 > x_2 > x_3$						
Ordering on Monomials	Ordering on Polynomial Pairs	Variant of Algorithm			MAP	Mat
		Crit	RA	SPS		
lex	lex	129,71	3.577,76	137,13	—	—
lex	tdeg	O.o.M.	300,10	O.o.M.	—	—
lex	itdeg	O.o.M.	838,20	O.o.M.	—	—
lex	?	—	—	—	115,2	∞
tdeg	itdeg	5,83	5,76	5,48	—	—
tdeg	tdeg	6,68	5,84	5,38	—	—
tdeg	?	—	—	—	25,23	n.i.

2.9.1 Conclusion

It is well-known that the computing time for the Gröbner bases algorithm depends heavily on the ordering of monomials within a polynomial and that total degree ordering can be expected to be faster than purely lexical ordering in most of the cases. However, the result (the Gröbner basis) also depends on this ordering. Especially for solving systems of equations the purely lexical Gröbner basis has very nice properties, in the sense that it yields kind of a "triangularization" of the system.

When using the purely lexical ordering for monomials, it is also well-known that a permutation of the variables can drastically influence the computing time. There exist heuristics, how to "optimally" permute the variables (see [Böge *et al.*, 1986]). We did not concentrate on this factor in our timings.

From the tables given in the previous section one can see, that *the variant of the Gröbner bases algorithm* and *the ordering on pairs of polynomials* have enormous influence on the computing time. If useful heuristics can be developed to choose the optimal variant of the algorithm combined with a "fitting" ordering on pairs of polynomials, then this could yield a drastical speed up.

From the examples done so far, one can neither read off a favorite variant nor an optimal ordering for pairs. Using total degree ordering for pairs seems to be a good method in many cases (speed-up of factor ≈ 10 in Example 9.2), however, Example 9.3 shows that it can also be a desasterous choice (nearly 7 hours computing time compared to 24 seconds with purely lexical ordering!). Furthermore, variant `gb_reduce_all` shows very good behavior in many examples, although it causes an unexpected slow-down of a factor ≈ 30 in Example 9.4 with purely lexical pair ordering whereas it is the only variant terminating at all using total degree pair ordering.

Compared to MAPLE, GRÖBNER is very fast for total degree ordering on monomials (see all examples). For lexical ordering it is still much faster for small examples (see Example 9.1 and Example 9.2) whereas for huge examples the difference is only little.

Compared to *Mathematica*, computing times lie in the same range for small examples (see Example 9.1 and Example 9.2), but *Mathematica* does very often not terminate for slightly larger examples (see Example 9.3 and Example 9.4).

Chapter 3

Source Code with Documentation

3.1 Introduction

In this section, we list part of the source code of GRÖBNER. Since we want to focus on the interesting parts of the implementation, we do not list the source code of the whole package. Instead we list

- ◇ all header files,
- ◇ all coercion units, and
- ◇ the source code of the variants of the Gröbner bases algorithm together with the important subroutines needed.
- ◇ For the remaining functions we proceed as follows: if the name of the function is not self-explanatory, we list a brief description of the function how it is also given as a documentation in the header of each function. If the name of the function is self-explanatory we omit this description (e.g. `sum_mon` is the summation of two monomials, any further description is superfluous).

3.2 Header Files

3.2.1 System Headers

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  groebner_globals.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GROEBNER_GLOBALS_H_
#define _GROEBNER_GLOBALS_H_

/***********************************
  Definitions for groebner-package
***********************************/

typedef int Boolean;

#define TRUE 1
#define FALSE 0
#define EMPTY BETA
#define REST RED
#define MAX_COMMAND_LENGTH 120

#define maximum(a,b) ((a) > (b) ? (a) : (b))
```

```
#define minimum(a,b) ((a) < (b) ? (a) : (b))

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  groebner.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GROEBNER_H_
#define _GROEBNER_H_

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

#include "saclib.h"
#include "groebner_globals.h"

#include "error.h"
#include "help.h"

#include "list.h"

#include "power_product_domain.setup"
#include "coefficient_domain.setup"

#include "mon.h"

#include "poly.data-structure.setup"
#include "poly.representation.setup"

#include "lp.h"
#include "ps.h"
#include "pa.h"
#include "gb_crude.h"
#include "gb_criteria.h"
#include "gb_reduce_all.h"
#include "gb_small_pair_set.h"
#include "syz.h"

#define total_degree_lexical 1
#define total_degree_inverse_lexical 2
#define lexical 3
#define matrix 4
#define permuted_lexical 5
#define total_degree_permuted_lexical 6

#include "term_ordering.setup"
#include "pair_ordering.setup"
```

```
#include "normal_form.setup"

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  conversions.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _CONVERSIONS_H_
#define _CONVERSIONS_H_

/********************************/
/* Functions to be used outside */
/********************************/

PDS var2poly_rat( Word);
PDS int2poly_rat( Word);
PDS var2poly_rf( Word);
PDS int2poly_rf( Word);

Word string2list( char*);
Word string2integer( char*);
void list2string( Word, char*);
Word integer2list( Word);
Word single_prec2list( Word);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  error.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _ERROR_H_
#define _ERROR_H_

/********************
  Global Variables
*******************/

extern Boolean error_;

/*******************************
  Error Codes
******************************/

#define NOGB 1

/*******************************
  Warning Codes
******************************/
```

```
#define DIVISION 51
#define ZERODIVISION 52

void groebner_error( int, char*);
void print_message( char*, int, char*, char*);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  help.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _HELP_H_
#define _HELP_H_

/********************************
  Functions to be used outside
********************************/

void help_about(char*);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  init.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _INIT_H_
#define _INIT_H_

#define LARGE_MEMORY 1000000
#define SMALL_MEMORY  300000

/********************
  Global Variables
********************/

extern FILE *in_file_;
extern FILE *out_file_;
extern char **environment_array_;

/********************************
  Functions to be used outside
********************************/

void initialize_globals( void);
void process_command_line(int, char*[], char*[]);
void open_IO_files( char*, char*);

#endif
```

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  input.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _INPUT_H_
#define _INPUT_H_

/********************
  Global Variables
********************/

extern PDS parsed_poly;
extern PS parsed_poly_set;

/* Global variables for YACC */

extern Word yylval, yyval, yyv[], *yypv;

#if coefficient_domain == rational_numbers
#  define var2poly_coef var2poly_rat
#  define int2poly_coef int2poly_rat
#else
#if coefficient_domain == rational_functions
#  define var2poly_coef var2poly_rf
#  define int2poly_coef int2poly_rf
#endif
#endif

/********************************
  Functions to be used outside
********************************/

Boolean is_equal_var( Word, Word);

PDS power_of_poly( PDS, PDS);
PDS parse_poly( void);
PS parse_ps( void);

void set_input( void);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  output.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _OUTPUT_H_
#define _OUTPUT_H_

/********************************
```

```
  Functions to be used outside
*******************************/

void print_lines( Word[], Word, int);
Word additional_linebreaks( Word);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  shell.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _SHELL_H_
#define _SHELL_H_

/*******************************
  Functions to be used outside
*******************************/

void shell_escape( Word);
void exec_command( char *);
int split_word( char *, char *);

#endif
```

3.2.2 Data Domain Headers

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  list.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _LIST_H_
#define _LIST_H_

/*******************
 Type Definitions
*******************/

typedef Boolean (*CF)( Word, Word); /* comparison function */
typedef void (*PF)( Word);          /* print function */
typedef Word (*BF)( Word, Word);    /* binary function */
typedef Word (*UF)( Word);          /* unary function */

/*******************************
  Functions to be used outside
*******************************/

Word identity( int);
Word expanded( Word, Word, CF);
```

```
Word thinned( Word, Word, CF);
Word split( Word, Word, CF);
void split2( Word, Word*, Word*, CF);
Word merge( Word, Word, CF);
Word component_wise( BF, Word, Word);
Word apply( UF, Word);
Word step( BF, Word, Word);

#define singleton( a) COMP( a, EMPTY)

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  int.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _INT_H_
#define _INT_H_

/*******************
 Type Definitions
*******************/

typedef Word Int;

/*******************************
  Functions to be used outside
*******************************/

#define absolute_int( i) IABSF( (i))
#define sum_int( i1, i2) ISUM( (i1), (i2))
#define difference_int( i1, i2) IDIF( (i1), (i2))
#define negative_int( i) INEG( (i))
#define product_int( i1, i2) IPROD( (i1), (i2))
#define quotient_int( i1, i2) IQ( (i1),(i2))
#define remainder_int( i1, i2) IREM( (i1),(i2))
#define inverse_int( i) (i)
#define zero_int( i) (0)
#define one_int( i) (1)
#define is_zero_int( i) ((i)==0)
#define is_one_int( i) ((i)==1)

/*----- Order Predicates -----*/

#define is_equal_int( i1, i2) (ICOMP( (i1), (i2)) == 0)
#define is_greater_int( i1, i2) (ICOMP( (i1), (i2)) == 1)
#define is_negative_int( i) (ICOMP( (i), 0) == -1)
#define is_positive_int( i) (ICOMP( (i), 0) == 1)
```

```
#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  rat.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _RAT_H_
#define _RAT_H_

/*******************
 Type Definitions
*******************/

typedef Word Rat;

/*--------- Selectors --------*/

#define numerator_rat( r) RNNUM( (r))
#define denominator_rat( r) RNDEN( (r))

/*******************************
  Functions to be used outside
*******************************/

#define absolute_rat( r) RNABS( (r))
#define sum_rat( r1, r2) RNSUM( (r1), (r2))
#define difference_rat( r1, r2) RNDIF( (r1), (r2))
#define negative_rat( r) RNNEG( (r))
#define product_rat( r1, r2) RNPROD( (r1), (r2))
#define quotient_rat( r1, r2) RNQ( (r1),(r2))
#define inverse_rat( r) RNINV( r)
#define zero_rat( r) RNINT( 0)
#define one_rat( r) RNINT( 1)
#define normalizer_rat( r) RNINV( r)
#define is_zero_rat( r) (RNCOMP( r, RNINT(0))==0)
#define is_one_rat( r) (RNCOMP( r, RNINT(1))==0)
#define lcm_rat( r1, r2) r1

/*----- Order Predicates -----*/

#define is_equal_rat( r1, r2) (RNCOMP( r1, r2)==0)
#define is_greater_rat( r1, r2) (RNCOMP( r1, r2)==1)
#define is_negative_rat( r) (RNCOMP( r, RNINT( 0))==-1)
#define is_positive_rat( r) (RNCOMP( r, RNINT( 0))==1)

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  rf.h
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _RF_H_
#define _RF_H_

/********************
 Type Definitions
********************/

typedef Word RF; /* Rational functions */

/********************
  Global Variables
********************/

extern Vars field_variables_;
extern int number_of_field_variables_;

/********************************
  Functions to be used outside
********************************/

RF normalized_rf( RF);
RF sum_rf( RF, RF);
RF negative_rf( RF);
RF difference_rf( RF, RF);
RF product_rf( RF, RF);
RF inverse_rf( RF);
RF quotient_rf( RF, RF);
RF zero_rf( RF);
RF one_rf( RF);
RF normalizer_rf( RF);
Boolean is_zero_rf( RF);
Boolean is_one_rf( RF);
#define lcm_rf(r1,r2) (r1)

/*--------- Selectors --------*/

#define numerator_rf(rf) FIRST( (rf))
#define denominator_rf(rf) FIRST( REST( (rf)))
#define rational_function(n,d) COMP( (n), COMP( (d), EMPTY))

/*----- Order Predicates -----*/

Boolean is_equal_rf( RF, RF);
Boolean is_negative_rf( RF);
Boolean is_positive_rf( RF);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
  el.h
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _EL_H_
#define _EL_H_

/********************
 Type Definitions
********************/

typedef Word EL; /* exponent-lists */
typedef Word Vars; /* list of variables */

/********************
  Global Variables
********************/

extern int number_of_variables_;
extern Word order_matrix_;
extern Word variable_permutation_;
extern Vars variables_;

/********************************
  Functions to be used outside
********************************/

EL product_el( EL, EL);
EL quotient_el( EL, EL);
EL one_el( EL);
EL lcm_el( EL, EL);
int total_degree_el( EL);
EL permuted_variables( EL, Word);

/*----- Order Predicates -----*/

Boolean is_equal_el( EL, EL);
Boolean is_one_el( EL);
Boolean is_total_degree_greater_el( EL, EL);
Boolean is_total_degree_lexical_greater_el( EL, EL);
Boolean is_total_degree_inverse_lexical_greater_el( EL, EL);
Boolean is_lexical_greater_el( EL, EL);
Boolean is_matrix_greater_el( EL, EL);
Boolean is_permuted_lexical_greater_el( EL, EL);
Boolean is_total_degree_permuted_lexical_greater_el( EL, EL);
Boolean is_multiple_el( EL, EL);
Boolean is_disjoint_el( EL, EL);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
  mon.h
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _MON_H_
#define _MON_H_

/********************
 Type Definitions
********************/

typedef Word Mon; /* from SACLIB */

/********************************
  Functions to be used outside
********************************/

Mon sum_mon( Mon, Mon);
Mon difference_mon( Mon, Mon);
Mon negative_mon( Mon);
Mon product_mon( Mon, Mon);
Mon product_monic_mon( Mon, Mon);
Mon quotient_mon( Mon, Mon);
Mon quotient_monic_mon( Mon, Mon);
Mon one_mon( Mon);
Mon lcm_mon( Mon, Mon);
Mon lcm_monic_mon( Mon, Mon);
int total_degree_mon( Mon);
Boolean is_zero_mon( Mon m);

/*--------- Constructor --------*/

#define monomial( c, pp) LIST2( c, pp)

/*--------- Selectors --------*/

#define coefficient( m) FIRST( m)
#define power_product( m) SECOND( m)

/*----- Order Predicates -----*/

Boolean is_equal_mon( Mon, Mon);
Boolean is_total_degree_greater_mon( Mon, Mon);
Boolean is_total_degree_lexical_greater_mon( Mon, Mon);
Boolean is_total_degree_inverse_lexical_greater_mon( Mon, Mon);
Boolean is_lexical_greater_mon( Mon, Mon);
Boolean is_matrix_greater_mon( Mon, Mon);
Boolean is_permuted_lexical_greater_mon( Mon, Mon);
Boolean is_total_degree_permuted_lexical_greater_mon( Mon, Mon);
Boolean is_multiple_mon( Mon, Mon);
Boolean is_negative_mon( Mon);
```

```
Boolean is_positive_mon( Mon);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  dpl.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _DPL_H_
#define _DPL_H_

/*******************
 Type Definitions
*******************/

typedef Word DPL; /* dense polynomials as lists */

/******************************
  Functions to be used outside
******************************/

DPL sum_dpl( DPL, DPL);
DPL difference_dpl( DPL, DPL);
DPL negative_dpl( DPL);
DPL product_coef_dpl( Coef, DPL);
DPL product_mon_dpl( Mon, DPL);
DPL product_monic_mon_dpl( Mon, DPL);
DPL product_dpl( DPL, DPL);
DPL quotient_mon_dpl( Mon, DPL);
#define zero_dpl( dpl) (EMPTY)
DPL quotient_dpl( DPL, DPL);
DPL one_dpl( DPL);
DPL insert_monomial( Mon, DPL);
DPL normalized_dpl( DPL);
DPL initialized_dpl( DPL, int, int);
#define combine_dpl( dpl1, dpl2) INV( dpl1)

/*--------- Selectors --------*/

#define polynomial_dpl( dpl) dpl
#define leading_monomial_dpl( dpl) FIRST( dpl)
#define remaining_polynomial_dpl( dpl) REST( dpl)
#define leading_coefficient_dpl( dpl) coefficient( FIRST( dpl))
#define leading_power_product_dpl( dpl) power_product( FIRST( dpl))
#define prepend_monomial_dpl( m, dpl) COMP( m, dpl)

/*----- Order Predicates -----*/

Boolean is_equal_dpl( DPL, DPL);
```

```
#define is_zero_dpl( dpl) ((dpl) == EMPTY)
Boolean is_leading_monomial_greater_dpl( DPL, DPL);
Boolean is_leading_monomial_multiple_dpl( DPL, DPL);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  lp.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _LP_H_
#define _LP_H_

/********************
 Type Definitions
********************/

typedef Word LP; /* labeled polynomials */

/********************
  Global Variables
********************/

extern int label_;

/*******************************
  Functions to be used outside
*******************************/

LP labeled_poly( Poly);
Poly unlabeled_lp( LP);

/*--------- Selectors --------*/

#define label( lp) FIRST( (lp))
#define polynomial_lp( lp) SECOND( (lp))
#define leading_monomial_lp( lp) leading_monomial( polynomial_lp( \
  (lp)))
#define remaining_polynomial_lp( lp) remaining_polynomial( \
  polynomial_lp( (lp)))
#define leading_coefficient_lp( lp) leading_coefficient( \
  polynomial_lp( (lp)))
#define leading_power_product_lp( lp) leading_power_product( \
  polynomial_lp( (lp)))

/*----- Order Predicates -----*/

Boolean is_equal_lp( LP, LP);
Boolean is_zero_lp( LP);
Boolean is_leading_monomial_greater_lp( LP, LP);
```

```
Boolean is_leading_monomial_multiple_lp( LP, LP);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  pc.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _PC_H_
#define _PC_H_

/*******************
 Type Definitions
*******************/

typedef Word PC; /* polynomials with cofactors */

/******************************
  Functions to be used outside
******************************/

PC sum_pc( PC, PC);
PC difference_pc( PC, PC);
PC negative_pc( PC);
PC product_coef_pc( Coef, PC);
PC product_mon_pc( Mon, PC);
PC product_monic_mon_pc( Mon, PC);
PC quotient_mon_pc( Mon, PC);
PC zero_pc( PC);
PC normalized_pc( PC);
PC initialized_pc( PDS, int, int);
PC combine_pc( PC, PC);
Word zero_cof( Word);

/*--------- Selectors --------*/

#define polynomial_pc( pc) FIRST( pc)
#define cofactors_pc( pc) SECOND( pc)
#define leading_monomial_pc( pc) leading_monomial_pds( \
  polynomial_pc( pc))
#define remaining_polynomial_pc( pc) LIST2( \
  remaining_polynomial_pds( polynomial_pc( pc)), cofactors_pc( pc))
#define head_pc( pc) LIST2( LIST1( leading_monomial_pds( \
  polynomial_pc( pc))), cofactors_pc( pc))
#define tail_pc( pc) remaining_polynomial_pds( polynomial_pc( pc))
#define leading_coefficient_pc( pc) leading_coefficient_pds( \
  polynomial_pc( pc))
#define leading_power_product_pc( pc) leading_power_product_pds( \
```

```
  polynomial_pc( pc))
#define prepend_monomial_pc( m, pc) LIST2( prepend_monomial_pds( \
  m, polynomial_pc( pc)), cofactors_pc( pc))

/*----- Order Predicates -----*/

Boolean is_equal_pc( PC, PC);
Boolean is_zero_pc( PC);
Boolean is_leading_monomial_greater_pc( PC, PC);
Boolean is_leading_monomial_multiple_pc( PC, PC);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  ps.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _PS_H_
#define _PS_H_

/*******************
 Type Definitions
*******************/

typedef Word PS; /* polynomial sets as lists */

/*******************************
  Functions to be used outside
*******************************/

PS insert_pds_ps( PDS, PS);
PS insert_ps( Poly, PS);
PS insert_lps( LP, PS);
PS normalized_ps( PS);
PS initialized_ps( PS);
PS labeled_ps( PS);
PS unlabeled_ps( PS);
PS poly_extracted_ps( PS);

/*--------- Selectors --------*/

/*----- Order Predicates -----*/

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  pa.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _PA_H_
```

```
#define _PA_H_

/*******************
 Type Definitions
*******************/

typedef Word Pa; /* pairs as lists */
typedef Word PaS; /* pair set */

/*******************
  Global Variables
*******************/

extern int number_of_pairs_;
extern int number_of_reduced_pairs_;

/******************************
  Functions to be used outside
******************************/

PaS new_pairs( Poly, PS);
PaS pair_set( PS);
Pa new_pair( LP, LP);
Pa next_pair( PaS*, PaS*);
PaS initial_pairs( PS);
PaS updated_pairs( PaS, PS, LP);
PaS thinned_pairs( PS, PaS);
void split_pairs( PS, PaS*, PaS*);
Pa pair_of_labels( Pa);
Boolean is_equal_pa( Pa, Pa);
Boolean contains_pa( Pa, PS);
Boolean contains_pas( PaS, Pa);

/*--------- Selectors --------*/

/*----- Order Predicates -----*/

Boolean is_lcm_greater( Pa, Pa);
Boolean is_labeled_lcm_greater( Pa, Pa);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  syz.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _SYZ_H_
#define _SYZ_H_

/*******************
```

```
 Type Definitions
*******************/

typedef Word Syz; /* syzygies*/

/*******************************
  Functions to be used outside
*******************************/

Syz sum_syz( Syz, Syz);
Syz difference_syz( Syz, Syz);
Syz negative_syz( Syz);
Syz zero_syz( Syz);
Syz extension_syz( Syz, PS);
PS basis_module_syz( PS);

/*--------- Selectors --------*/

/*----- Order Predicates -----*/

Boolean is_equal_syz( Syz, Syz);
Boolean is_zero_syz( Syz);

#endif
```

3.2.3 Gröbner Bases Implementation Headers

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  gb_crude.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GB_CRUDE_H_
#define _GB_CRUDE_H_

/*******************************
  Functions to be used outside
*******************************/

typedef PS (*GB_function)( PS);

PS GB_crude( PS);
Poly S_polynomial_poly( Poly, Poly);
Poly partial_normal_form_poly( Poly, PS);
Poly complete_normal_form_poly( Poly, PS);
PS inter_reduced_ps( PS);
PS canceled_multiples_ps( PS);
PS completely_reduced_ps( PS);
Poly head_reduced_poly( Poly, PS);

#endif
```

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  gb_criteria.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GB_CRITERIA_H
#define _GB_CRITERIA_H

/*******************************
  Functions to be used outside
*******************************/

PS GB_criteria( PS);
Poly S_polynomial_lp( LP, LP);
Boolean criteria( Poly, Poly, PS, PaS);
Boolean product_criterion( Poly, Poly);
Boolean explicit_chain_criterion( Poly, Poly, PS, PaS);

PS GB_criteria3( PS);
Boolean criteria3( LP, LP, PS, PaS, PaS);
Boolean explicit_chain_criterion3( LP, LP, PS, PaS, PaS);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  gb_reduce_all.h

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GB_REDUCE_ALL_H_
#define _GB_REDUCE_ALL_H_

/*******************************
  Functions to be used outside
*******************************/

PS GB_reduce_all( PS);
Word all_reduced( PS, PS, PS, PaS);

PS GB_reduce_all2( PS);
void reduce_all( PS, PS, PS*, PaS*);

PS GB_reduce_all3( PS F);
void reduce_all3( PS R, PS C, PS *G, PaS *B, PaS *CP);

Poly normal_form_lp( LP, PS);
LP head_reduced_lp( LP, PS);
PS completely_reduced_lps( PS);

#endif

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
  gb_small_pair_set.h
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

#ifndef _GB_SMALL_PAIR_SET_H_
#define _GB_SMALL_PAIR_SET_H_

/*********************************
  Functions to be used outside
*********************************/

PS GB_small_pair_set( PS);
PaS initial_pairs_with_criteria( PS);
Pa trivial_pair( void);
PaS initial_pairs_with_product_criterion( PS);
PaS updated_pairs_with_product_criterion( PaS, PS, LP);
PaS thinned_by_chain_criterion( PaS, PS, Pa);
Boolean chain_criterion( Pa, PS, Pa);
Boolean triangle_criterion( Pa, LP, Pa);

#endif
```

3.3 Coercion Units

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Coefficient Domain INTEGER

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_coef sum_int
#define difference_coef difference_int
#define negative_coef negative_int
#define product_coef product_int
#define quotient_coef quotient_int
#define remainder_coef remainder_int
#define zero_coef zero_int
#define one_coef one_int
#define is_zero_coef is_zero_int
#define is_one_coef is_one_int
#define lcm_coef lcm_int

#define is_equal_coef is_equal_int
#define is_negative_coef is_negative_int
#define is_positive_coef is_positive_int

#define read_coef read_int
#define format_coef format_int
```

```
#define print_coef print_int
#define print_input_form_coef print_int

/* >>>>> Inclusion Unit <<<<< */

#include "int.h"

/* >>>>> Definition  Unit <<<<< */

typedef Int Coef;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Coefficient Domain RATIONAL NUMBERS

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_coef sum_rat
#define difference_coef difference_rat
#define negative_coef negative_rat
#define product_coef product_rat
#define quotient_coef quotient_rat
#define inverse_coef inverse_rat
#define zero_coef zero_rat
#define one_coef one_rat
#define is_zero_coef is_zero_rat
#define is_one_coef is_one_rat
#define lcm_coef lcm_rat
#define normalizer_coef normalizer_rat
#define is_equal_coef is_equal_rat
#define is_negative_coef is_negative_rat
#define is_positive_coef is_positive_rat

#define read_coef read_rat
#define print_coef print_rat
#define print_input_form_coef print_rat
#define format_coef format_rat
#define pretty_print_coef pretty_print_rat

/* >>>>> Inclusion Unit <<<<< */

#include "rat.h"

/* >>>>> Definition Unit <<<<< */

typedef Rat Coef;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Coefficient Domain RATIONAL FUNCTIONS
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_coef sum_rf
#define difference_coef difference_rf
#define negative_coef negative_rf
#define product_coef product_rf
#define quotient_coef quotient_rf
#define inverse_coef inverse_rf
#define zero_coef zero_rf
#define one_coef one_rf
#define is_zero_coef is_zero_rf
#define is_one_coef is_one_rf
#define lcm_coef lcm_rf
#define normalizer_coef(c) normalizer_rf( c)

#define is_equal_coef is_equal_rf
#define is_negative_coef is_negative_rf
#define is_positive_coef is_positive_rf

#define read_coef read_rf
#define print_coef print_rf
#define print_input_form_coef print_input_form_rf
#define format_coef format_rf
#define pretty_print_coef pretty_print_rf

/* >>>>> Inclusion Unit <<<<< */

#include "rf.h"

/* >>>>> Definition Unit <<<<< */

typedef RF Coef;

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Power Product Representation EL

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define product_pp product_el
#define quotient_pp quotient_el
#define one_pp one_el
#define lcm_pp lcm_el
#define total_degree_pp total_degree_el

#define is_equal_pp is_equal_el
#define is_one_pp is_one_el
#define is_total_degree_greater_pp \
```

```
        is_total_degree_greater_el
#define is_total_degree_lexical_greater_pp \
        is_total_degree_lexical_greater_el
#define is_total_degree_inverse_lexical_greater_pp \
        is_total_degree_inverse_lexical_greater_el
#define is_lexical_greater_pp \
        is_lexical_greater_el
#define is_matrix_greater_pp \
        is_matrix_greater_el
#define is_permuted_lexical_greater_pp \
        is_permuted_lexical_greater_el
#define is_total_degree_permuted_lexical_greater_pp \
        is_total_degree_permuted_lexical_greater_el
#define is_multiple_pp is_multiple_el
#define is_disjoint_pp is_disjoint_el

#define read_pp read_el
#define print_pp print_el
#define print_input_form_pp print_input_form_el
#define format_pp format_el
#define pretty_print_pp pretty_print_el

/* >>>>> Inclusion Unit <<<<< */

#include "el.h"

/* >>>>> Definition Unit <<<<< */

typedef EL PP; /* power-products */

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Polynomial Data Structure DPL

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_pds sum_dpl
#define difference_pds difference_dpl
#define negative_pds negative_dpl
#define product_coef_pds product_coef_dpl
#define product_mon_pds product_mon_dpl
#define product_monic_mon_pds product_monic_mon_dpl
#define product_pds product_dpl
#define quotient_mon_pds quotient_mon_dpl
#define quotient_pds quotient_dpl
#define zero_pds zero_dpl
#define one_pds one_dpl
#define normalized_pds normalized_dpl
```

```
#define initialized_pds initialized_dpl
#define combine_pds combine_dpl

/*--------- Selectors --------*/

#define polynomial_pds polynomial_dpl
#define leading_coefficient_pds leading_coefficient_dpl
#define leading_power_product_pds leading_power_product_dpl
#define leading_monomial_pds leading_monomial_dpl
#define remaining_polynomial_pds remaining_polynomial_dpl
#define prepend_monomial_pds prepend_monomial_dpl

/*----- Order Predicates -----*/

#define is_equal_pds is_equal_dpl
#define is_zero_pds is_zero_dpl
#define is_leading_monomial_greater_pds is_leading_monomial_greater_dpl
#define is_leading_monomial_multiple_pds is_leading_monomial_multiple_dpl

/*------ I/O Functions -----*/

#define read_pds read_dpl
#define print_pds print_dpl
#define print_input_form_pds print_input_form_dpl
#define format_pds format_dpl
#define pretty_print_pds pretty_print_dpl

/* >>>>> Inclusion Unit <<<<< */

#include "dpl.h"

/* >>>>> Definition Unit <<<<< */

typedef DPL PDS; /* abstract type PDS */

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Polynomial Representation Polynomial

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_poly sum_pds
#define difference_poly difference_pds
#define negative_poly negative_pds
#define product_coef_poly product_coef_pds
#define product_mon_poly product_mon_pds
#define product_monic_mon_poly product_monic_mon_pds
#define quotient_mon_poly quotient_mon_pds
#define zero_poly zero_pds
#define normalized_poly normalized_pds
```

```
#define initialized_poly initialized_pds
#define combine_poly combine_pds

/*--------- Selectors --------*/

#define polynomial_poly polynomial_pds
#define leading_coefficient leading_coefficient_pds
#define leading_power_product leading_power_product_pds
#define leading_monomial leading_monomial_pds
#define remaining_polynomial remaining_polynomial_pds
#define prepend_monomial prepend_monomial_pds

/*----- Order Predicates -----*/

#define is_equal_poly is_equal_pds
#define is_zero_poly is_zero_pds
#define is_leading_monomial_greater_poly is_leading_monomial_greater_pds
#define is_leading_monomial_multiple_poly is_leading_monomial_multiple_pds

/*------ I/O Functions -----*/

#define print_poly print_pds
#define print_input_form_poly print_input_form_pds
#define format_poly format_pds
#define pretty_print_poly pretty_print_pds

/* >>>>> Inclusion Unit <<<<< */

/* >>>>> Definition Unit <<<<< */

typedef PDS Poly; /* abstract type Poly */

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Coercion Unit for Polynomial Representation
  Polynomial with Cofactors

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

/* >>>>> Renaming Unit <<<<< */

#define sum_poly sum_pc
#define difference_poly difference_pc
#define negative_poly negative_pc
#define product_coef_poly product_coef_pc
#define product_mon_poly product_mon_pc
#define product_monic_mon_poly product_monic_mon_pc
#define quotient_mon_poly quotient_mon_pc
#define zero_poly zero_pc
#define normalized_poly normalized_pc
#define initialized_poly initialized_pc
#define combine_poly combine_pc
```

```
/*--------- Selectors --------*/

#define polynomial_poly polynomial_pc
#define leading_coefficient leading_coefficient_pc
#define leading_power_product leading_power_product_pc
#define leading_monomial leading_monomial_pc
#define remaining_polynomial remaining_polynomial_pc
#define prepend_monomial prepend_monomial_pc

/*----- Order Predicates -----*/

#define is_equal_poly is_equal_pc
#define is_zero_poly is_zero_pc
#define is_leading_monomial_greater_poly is_leading_monomial_greater_pc
#define is_leading_monomial_multiple_poly is_leading_monomial_multiple_pc

/*------ I/O Functions -----*/

#define print_poly print_pc
#define print_input_form_poly print_input_form_pc
#define format_poly format_pc
#define pretty_print_poly pretty_print_pc

/* >>>>> Inclusion Unit <<<<< */

#include "pc.h"

/* >>>>> Definition Unit <<<<< */

typedef PC Poly; /* abstract type Poly */
```

3.4 Source Files

3.4.1 The main Program

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Main program

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"
#include "rf.h"

beginGcGlobals
  &order_matrix_,
  &variable_permutation_, &variables_, &field_variables_,
  &yylval, &yyval, &parsed_poly, &parsed_poly_set
endGcGlobals;

noGcGlobalArrays;
```

```
/*  For global arrays containing sac objects

  beginGcGlobalArrays
  { arrayName, arraySize },
  { arrayName, arraySize },
  ....
  { arrayName, arraySize }
  endGcGlobalArrays;
*/

void main(int argc,char **argv,char **envv)
  {
  GB_function   groebner_basis;
  PS            ps, gb;
  float         ut1, ut2, st1, st2;

  process_command_line( argc, argv, envv);
  beginSacLib( &argc);

  printf("========================\n");
  printf("WELCOME TO GROEBNER\n\n");
  printf("Good Luck ...\n");
  printf("========================\n\n\n");

  initialize_globals();

  ps = parse_ps();
  print_ps( ps, print_pds);

  groebner_basis = GB_reduce_all;

/****************** Alternatives ******************
        groebner_basis = GB_crude;
        groebner_basis = GB_criteria;
        groebner_basis = GB_criteria3;
        groebner_basis = GB_reduce_all3;
        groebner_basis = GB_small_pair_set;
**************************************************/

  ut1 = UserTime();
  st1 = SystemTime();
  gb  = (*groebner_basis)( ps);
  ut2 = UserTime();
  st2 = SystemTime();
  fprintf( out_file_, "Groebner Basis =\n");
  pretty_print_ps( gb);
  fprintf( out_file_, "\n\n");
  fprintf( out_file_, "User Time: %f\n", ut2-ut1);
  fprintf( out_file_, "System Time: %f\n", st2-st1);
```

```
  printf("*****************\n");
  printf("         Bye\n");
  printf("*****************\n");

  endSacLib();

  printf("Closing all files ...\n");
  fclose( in_file_);
  fclose( out_file_);
  return;
  }
```

3.4.2 System Sources

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Conversions between basic data types

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/**************************************************************/
PDS var2poly_rat( Word variable)                              /*

  O u t p u t: variable transformed to polynomial
               over rational numbers.
***************************************************************/

/**************************************************************/
PDS int2poly_rat( Word num)                                   /*

  I n p u t: An integer num.
  O u t p u t: num transformed to polynomial
               over rational numbers.
***************************************************************/

#if coefficient_domain == rational_functions

/**************************************************************/
PDS var2poly_rf( Word variable)                               /*

  O u t p u t: variable transformed to polynomial
               over rational functions.
***************************************************************/

/**************************************************************/
PDS int2poly_rf( Word num)                                    /*

  I n p u t: An integer num.
  O u t p u t: num transformed to polynomial
               over rational functions.
```

```
*******************************************************/
#endif

/******************************************************/
Word string2list( char *a)                          /*

  I n p u t: String a.
  O u t p u t: a converted to list of characters.
*******************************************************/

/******************************************************/
Word string2integer( char *a)                       /*

  I n p u t: An array of digits a.
  O u t p u t: a transformed to integer.
  M e t h o d: Derived from the IREAD-function
               Special: no leading blanks, no sign.
*******************************************************/

/******************************************************/
void list2string( Word l, char *s)                  /*

  I n p u t: l list of characters, s pointer to char
  A c t i o n: list converted to array.
*******************************************************/

/******************************************************/
Word integer2list( Word n)                          /*

  I n p u t: Arbitrary long integer n.
  O u t p u t: n transformed to list of characters
*******************************************************/

/******************************************************/
Word single_prec2list( Word n)                      /*

  I n p u t: Short integer n.
  O u t p u t: n transformed to list of characters
*******************************************************/

/*xxxxxxxxxxxxxxxxxxxxxx

  Error Function

xxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/*****************************************************/
void error( int error_code, char *routine)         /*

  A c t i o n: Print appropriate error message.
******************************************************/
```

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Help function

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/***********************************************/
void help_about(char *s)                       /*

  A c t i o n: Print appropriate help message.
***********************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Initialization of GROEBNER

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/**********************************
  Global Variables               */

FILE *in_file_;
FILE *out_file_;
char **environment_array_;
/*********************************/

/*******************************************************/
void initialize_globals()                              /*

  A c t i o n: Initialize global variables for GROEBNER
               and print a summary on their actual values
*******************************************************/

/*****************************************************************/
void process_command_line(int argc, char *argv[], char *envv[])  /*

  A c t i o n: Process the command line arguments.
               Use default values if no arguments are given.
*****************************************************************/

/************************************************************/
void open_IO_files( char *in_file_arg, char *out_file_arg)  /*

  A c t i o n: Open files for in- and output.
************************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Input for lex to produce lexical analyzer

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"
```

```
%%
"+"                     return( '+');
"-"                     return( '-');
"*"/[^"*"]              return( '*');
"/"                     return( '/');
"^"|"**"                return( '^');
";"                     return( ';');
","                     return( ',');
"{"                     return( '{');
"}"                     return( '}');
"("                     return( '(');
")"                     return( ')');
"?"                     return( HELP);
[ \t\n]                 ;
"!".+   {yylval = string2list( yytext+1); return( SHELL);}
[a-zA-Z][A-Za-z0-9]*    {yylval = string2list( yytext); return( VAR);}
[0-9]+          {yylval = string2integer( yytext); return( INT);}
%%

void set_input()
  {
  yyin = in_file_;
  }

int yywrap()
  {
  return 1;
  }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Input for yacc to produce parser

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
%{
#include "groebner.h"
#include "rf.h"

PDS parsed_poly;
PS parsed_poly_set;
Boolean error_;
static int error_count_ = 0;

#define YYSTYPE Word

%}

%left '+' '-'
%left '*' '/'
```

```
%left '^'
%left PAR
%token VAR INT HELP SHELL ',' '{' '}' ';' '(' ')'
%%
input   :       error '}'
                        { yyerrok; printf("Try again ...\n-> ");}
                input
        |       HELP
                        {help_about( "input_syntax");printf("<---\n");}
                input
        |       SHELL
                        {shell_escape( $1);printf("<---\n");}
                input
        |       '{' '}'
                        {unput('\0'); parsed_poly_set = EMPTY;}
        |       '{' polysequence '}'
                        {unput('\0'); parsed_poly_set = $2;}
        |       poly ';'
                        {unput('\0'); parsed_poly = $1;}
        ;
polysequence    :       poly
                                {$$ = LIST1( $1);}
        |       polysequence ',' poly
                        {$$ = insert_pds_ps( $3, $1);}
        ;
poly    :       VAR
                        {$$ = var2poly_coef( $1);}
        |       INT
                        {$$ = int2poly_coef( $1);}
        |       '(' poly ')' %prec PAR
                        {$$ = $2;}
        |       poly '+' poly
                        {$$ = sum_pds( $1, $3);}
        |       poly '-' poly
                        {$$ = difference_pds( $1, $3);}
        |       poly '*' poly
                        {$$ = product_pds( $1, $3);}
        |       poly '/' poly
                        {$$ = quotient_pds( $1, $3);}
        |       '-' poly %prec  '*'
                        {$$ = negative_pds( $2);}
        |       poly '^' INT
                        {$$ = power_of_poly( $1, $3);}
        ;
```

```
%%
#include "lex.c"

yyerror( char *s)
  {
  error( SYNTAX, "parse_poly");
  if( ++error_count_ > 3)
    exit(1);
  unput('}');
  }

PDS parse_poly()
  {
  set_input();
  parsed_poly = EMPTY;
  do
    {
    error_ = FALSE;
    printf("Enter Polynomial (? for help):\n\n");
    yyparse();
    }
  while( error_);
  printf("Reading of polynomial complete\n");
  return parsed_poly;
  }

PS parse_ps()
  {
  set_input();
  parsed_poly_set = EMPTY;
  do
    {
    error_ = FALSE;
    printf("Enter Polynomial Set (? for help):\n\n");
    yyparse();
    }
  while( error_);
  printf("Reading of polynomial set complete\n");
  return parsed_poly_set;
  }

PDS power_of_poly( Word poly, Word power)
  {
  PDS poly0=poly;

  if( !power)
    return one_pds( poly);
```

```
  while( ICOMP( power, 1) == 1)
    {
    poly0 = product_pds( poly0, poly);
    power = IDIF( power, 1);
    }
  return poly0;
  }

Boolean is_equal_var( Word var1, Word var2)
  {
  Word v1=var1, v2=var2;

  if( LENGTH( v1) != LENGTH( v2))
    return FALSE;
  while( v1 != EMPTY)
    {
    if( FIRST( v1) != FIRST( v2))
      return FALSE;
    v1 = REST( v1);
    v2 = REST( v2);
    }
  return TRUE;
  }
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Escape to shell

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/*********************************************/
void shell_escape( Word s)                   /*

  A c t i o n: Convert s to string, then pass
    the string for execution to the shell.
**********************************************/

/******************************************************/
void exec_command( char *command)                    /*

  A c t i o n: Pass string for execution to the shell.
*******************************************************/
```

3.4.3 Data Domain Sources

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Lists

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
```

```
#include "groebner.h"

/*****************************************************/
Word expanded( Word element, Word list, CF comp)      /*

  O u t p u t:
    list expanded by element.
    list is ordered w.r.t. the comparison function comp.
*****************************************************/

/*****************************************************/
Word thinned( Word element, Word list, CF rel)        /*

  O u t p u t:
    A list derived from list by deleting all elements e
    s.t. rel( e, element) = TRUE.
*****************************************************/

/*****************************************************/
Word split( Word element, Word list, CF equal)        /*

  O u t p u t:
    (A, B) with A, B s.t.
    A contains all elements e of list s.t.
      equal( e, element) = TRUE.
    B = list - A.
*****************************************************/

/************************************************************/
void split2( Word element, Word *equals, Word *list, CF equal) /*

  A c t i o n:
    equals contains all elements e of list s.t.
      equal( e, element) = TRUE.
    list = list - equals.
************************************************************/

/*****************************************************/
Word merge( Word l1, Word l2, CF comp)                /*

  O u t p u t:
    l1 and l2 merged w.r.t. the comparison function comp.
*****************************************************/

/*****************************************************/
Word component_wise( BF bf, Word l1, Word l2)         /*

  I n p u t:
    A binary function bf
    l1 = ( l11, l12, l13,...)
    l2 = ( l21, l22, l23,...).
```

```
  O u t p u t:
    ( bf( l11, l21), bf( l12, l22),...).
*******************************************************/

/*******************************************************/
Word apply( UF uf, Word l)                            /*

  I n p u t:
    A unary function uf
    l = ( l1, l2, l3,...).
  O u t p u t:
    ( uf( l1), uf( l2),...).
*******************************************************/

/*******************************************************/
Word step( BF bf, Word o, Word l)                     /*

  I n p u t:
    A binary function bf
    an element o
    l = ( l1, l2, l3,...).
  O u t p u t:
    ( bf( o, l1), bf( o, l2),...).
*******************************************************/

/*******************************************************************/
void print_list( Word l, PF print_element, char *left_delimiter,
                 char *seperator, char *right_delimiter)          /*

  A c t i o n:
    Print each list element by the function print_element,
    delimited by left_delimiter and right_delimiter,
    seperated by seperator.
*******************************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Integers

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"
#include "int.h"

/*******************************************************/
void print_int( Int A)                                /*

  A c t i o n: Print arbitrary long integer A.
               Derived from the SAC-2 procedure IWRITE
*******************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
  Domain: Rational Numbers

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"
#include "rat.h"
#include "int.h"

/****************************************************/
void print_rat( Rat R)                               /*

  A c t i o n: Print rational number R.
    Derived from the SAC-2 procedure RNWRIT.
*****************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Rational Functions

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Rational Function is a SAC-2 list of the following
form:

    ( n, d)

    where n and d are SAC-2-polynomials.
*/

#include "groebner.h"
#include "rf.h"
#include "int.h"

/*************************************
  Global Variables                 */

Vars field_variables_;
int number_of_field_variables_;
/**********************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Exponent Lists

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

An Exponent List is a SAC-2 list of the following
form:
```

```
    ( a1, ..., al)

    where the ai's are integers.
*/
#include "groebner.h"
#include "int.h"

/************************************
  Global Variables                */

Word    order_matrix_;
Word    variable_permutation_;
int     number_of_variables_;
Vars    variables_;

/************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Monomials

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Monomial is a SAC-2 list of the following
form:

    ( c, pp)

    where c is a coefficient and
    pp is a power product.
*/

#include "groebner.h"

/*******************************************************/
Mon product_monic_mon( Mon m1, Mon m2)               /*

  O u t p u t: Product of a monic monomial m1 and
               a monomial m2.
********************************************************/

/********************************************************/
Mon quotient_monic_mon( Mon m1, Mon m2)               /*

  O u t p u t: Quotient of a monomial m1 and a
               monic monomial m2.
*********************************************************/

/*******************************************************/
void format_mon( Mon mon, Word line[], int *length)    /*
```

```
  A c t i o n: A monomial is formatted in 5 lines,
               however, there are only as many lines
               full with information as necessary.
********************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Dense Polynomial in List representation

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Polynomial is a SAC-2 list of the following
form:

    ( m1, m2, ...)

    where the mi's are monomials.
    The mi's are ordered w.r.t. the ordering
    "is_greater_mon" for monomials.
*/

#include "groebner.h"

/*******************************************************/
DPL product_coef_dpl( Coef c, DPL p)                   /*

  O u t p u t: Product of an element of the domain of
               coefficients c and a polynomial p
********************************************************/

/*******************************************************/
DPL product_mon_dpl( Mon m, DPL p)                     /*

  O u t p u t: Product of a monomial m and a polynomial p
********************************************************/

/*******************************************************/
DPL product_monic_mon_dpl( Mon m, DPL p)               /*

  O u t p u t: Product of a monic monomial m and
               a polynomial p
********************************************************/

/*******************************************************/
DPL quotient_mon_dpl( Mon m, DPL p)                    /*

  O u t p u t: Quotient of a polynomial p and
               a Monomial m
********************************************************/
```

```
/*****************************************************/
DPL normalized_dpl( DPL p)                          /*

  O u t p u t: Leading coefficient of p normalized to 1
******************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Labeled Polynomials

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Polynomial is a SAC-2 list of the following
form:

    ( label, poly)

    where label is an integer and
    poly is a polynomial.
*/

#include "groebner.h"

/*************************
  Global variable       */

int label_= 1;
/************************/

LP labeled_poly( Poly p)
  {
  return COMP( label_++, COMP( p, EMPTY));
  }

Poly unlabeled_lp( LP lp)
  {
  return polynomial_lp( lp);
  }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Polynomial with Cofactors

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Polynomial is a SAC-2 list of the following
form:
```

```
    ( p, c)

    where p is a polynomial of type PDS and
          c is a list of polynomials of type PDS.
*/

#include "groebner.h"
#include "pc.h"

/*********************************************************/
PC product_coef_pc( Coef c, PC pc)                     /*

  O u t p u t: Product of an element of the coefficient
   domain c and a polynomial pc.
*********************************************************/

/*********************************************************/
PC product_mon_pc( Mon m, PC pc)                       /*

  O u t p u t: Product of a monomial m and a
               polynomial pc.
*********************************************************/

/*********************************************************/
PC product_monic_mon_pc( Mon m, PC pc)                 /*

  O u t p u t: Product of a monic monomial m and a
               polynomial pc.
*********************************************************/

/*********************************************************/
PC quotient_mon_pc( Mon m, PC pc)                      /*

  O u t p u t: Quotient of a monomial m and a
               polynomial pc.
*********************************************************/

/*********************************************************/
Word zero_cof( Word cof)                               /*

  O u t p u t: List of zero-cofactors of length
               LENGTH( cof).
*********************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Polynomial Sets

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------
```

```
A Polynomial Set is a SAC-2 list of the following
form:

    ( p1, p2, ...)

    where the pi's are polynomials.
    The mi's are ordered such that the leading
    monomial of mi is less than the leading
    monomial of mi+1 w.r.t the ordering
    "is_greater_mon" for monomials.
*/

#include "groebner.h"

/*******************************************************/
PS insert_pds_ps( PDS p, PS ps)                       /*

  O u t p u t: A polynomial p (data structure) inserted
    into a set of polynomials ps.
*******************************************************/

/*******************************************************/
PS insert_ps( Poly p, PS ps)                          /*

  O u t p u t: An unlabeled polynomial p inserted into
    a set of unlabeled polynomials ps.
*******************************************************/

/*******************************************************/
PS insert_lps( LP lp, PS ps)                          /*

  O u t p u t: A labeled polynomial lp inserted into
    a set of labeled polynomials ps.
*******************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Pairs and Pair Sets

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Pair is a SAC-2 list of the following
form:

    ( p1, p2)

    where the pi's are polynomials.

A Pair Set is a SAC-2 list of the following
form:
```

```
    ( pa1, pa2, ...)

    where the pai's are Pairs.
*/

#include "groebner.h"

/********************************
  Global Variables              */

int number_of_pairs_;
int number_of_reduced_pairs_;
/********************************/

/*****************************************************/
PaS new_pairs( Poly p, PS ps)                         /*

  I n p u t: A polynomial p and a set of polynomials ps.
  O u t p u t: Set of pairs of polynomials consisting
     of p and a polynomial of ps
*****************************************************/

/*****************************************************/
PaS pair_set( PS ps)                                  /*

  I n p u t: A set of polynomials ps.
  O u t p u t: Set of pairs of polynomials in ps.
*****************************************************/

/*****************************************************/
Pa next_pair( PaS *pairs, PaS *reduced_pairs)         /*

  O u t p u t: First element of pairs.
    pairs is changed, first element is adjoined
    to reduced_pairs.
*****************************************************/

/*****************************************************/
PaS initial_pairs( PS ps)                             /*

  I n p u t: A set of polynomials ps.
  O u t p u t: Set of pairs of polynomials in ps.
     The resulting pair set is ordered w.r.t.
     is_lcm_greater.
*****************************************************/

/*****************************************************/
PaS updated_pairs( PaS pairs, PS ps, LP lp)           /*

  O u t p u t: Pairs of polynomials consisting of lp
     and a polynomial of ps are adjoined to pairs.
     The resulting pair set is ordered w.r.t.
```

```
     is_lcm_greater.
*********************************************************/

/*********************************************************/
PaS thinned_pairs( PS ps, PaS pairs)                    /*

  O u t p u t: All pairs in ‘pairs’ which contain one
    of the polynomials of ps are deleted.
*********************************************************/

/**********************************************************/
void split_pairs( PS ps, PaS *pairs, PaS *deleted_pairs)  /*

  A c t i o n: Collect all pairs in ‘pairs’ that contain
    one of the polynomials of ps in ‘deleted_pairs’.
*********************************************************/

/*********************************************************/
Boolean contains_pa( Pa pa, PS ps)                      /*

  S p e c i f i c a t i o n: contains_pa = TRUE iff
    pa contains one of the polynomials of ps.
*********************************************************/

/*********************************************************/
Boolean contains_pas( PaS pas, Pa pa)                   /*

  S p e c i f i c a t i o n: contains_pas = TRUE iff
    pas contains pa.
*********************************************************/

/*********************************************************/
Boolean is_lcm_greater( Pa p1, Pa p2)                   /*

  S p e c i f i c a t i o n:
    p1 = ( poly11, poly12) .... pair of polynomials.
    p2 = ( poly21, poly22) .... pair of polynomials.
    is_lcm_greater( p1, p2) iff
    is_greater_mon_pair_order(
      lcm( lm( poly11), lm( poly12)),
      lcm( lm( poly21), lm( poly22))).
*********************************************************/

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Syzygies of Polynomial Vectors

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------
```

```
A syzygy is a SAC-2 list of the following
form:

    ( p1,..., pn)

    where pi is a polynomial of type PDS.
*/

#include "groebner.h"
#include "pc.h"

/************************************************************/
Syz extension_syz( Syz s, PS gb)                          /*

  I n p u t:
        s ... Syzygy of the leading monomials of gb,
        gb .. A normalized Groebner basis.
  O u t p u t:
        A syzygy of gb.
  A t t e n t i o n:
        "poly_representation" must be set to
        "polynomial_with_cofactors" in the central header file
        "groebner.h".
************************************************************/
```

3.4.4 Gröbner Bases Implementation Sources

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Crude version of the Groebner Bases Algorithm

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/************************************************************/
PS GB_crude( PS F)                                        /*

  I n p u t:
        F ... a set of polynomials.
  O u t p u t:
        G ... a completely reduced Groebner basis.
  M e t h o d:
    GB_crude implements the crude form of the algorithm for
    computing Groebner bases.

    In the present implementation, the algorithm is not tuned in
    any respect.

         The polynomials in the basis are not sorted.

         The polynomial pairs in the pair list are not sorted.
```

```
        No criteria for predicting zero-reductions are applied.
***************************************************************/
  {
  PS G;
  PaS B;
  Pa p;
  Poly f1, f2, h;

fprintf( out_file_, "**** GB ****\n");

  G = normalized_ps( initialized_ps( F));
  B = pair_set( G);
  while( B != EMPTY)
    {
    p  = FIRST( B);
    B  = REST( B);
    f1 = FIRST( p);
    f2 = SECOND( p);
    h = S_polynomial_poly( f1, f2);
    h = normal_form_poly( h, G);
    if( !is_zero_poly( h))
      {
      h = normalized_poly( h);
      fprintf( out_file_, "New Polynomial:\n");
      print_poly( h);
      B = CONC( B, new_pairs( h, G));
      G = insert_ps( h, G);
      }
    else
      fprintf( out_file_, "Zero-reduction\n");
    }
  return inter_reduced_ps( G);
  }
/***************************************************************/
Poly S_polynomial_poly( Poly p1, Poly p2)                    /*

  O u t p u t: The S-polynomial of p1 and p2
***************************************************************/
  {
  Mon m1, m2, u1, u2, lcm;
  Poly r1, r2;

  m1  = leading_monomial( p1);
  m2  = leading_monomial( p2);
  r1  = remaining_polynomial( p1);
  r2  = remaining_polynomial( p2);
```

```
  lcm = lcm_monic_mon( m1, m2);
  u1  = quotient_monic_mon( lcm, m1);
  u2  = quotient_monic_mon( lcm, m2);
  return difference_poly(
           product_monic_mon_poly( u1, r1),
           product_monic_mon_poly( u2, r2));
  }
/**********************************************************/
Poly partial_normal_form_poly( Poly p, PS ps)              /*
  I n p u t: ps ordered w.r.t. ascending leading monomials.
    Polynomials in ps are monic.
  O u t p u t: Iteratively, as many leading monomials of p
    are canceled by subtraction of appropriate multiples of
    polynomials in ps.
**********************************************************/
  {
  Mon lm_p, lm_red, quot;
  Poly p_red;
  PS ps_aux; /* polys that can be used for reduction */
  while( !is_zero_poly( p))
    {
    /* Every monomial is tested whether it can
       be deleted by leading monomials of the
       polynomials in the set */
    lm_p   = leading_monomial( p);
    ps_aux = ps;
    while( ps_aux != EMPTY)
      {
      p_red  = FIRST( ps_aux);
      lm_red = leading_monomial( p_red);
      if( is_multiple_mon( lm_p, lm_red))
        {
        /* reduction possible by using lm_red */
        quot = quotient_monic_mon( lm_p, lm_red);
        p = difference_poly(
              remaining_polynomial( p),
              product_mon_poly(
                quot,
                remaining_polynomial( p_red)));
        break;
        }
      else if( is_greater_mon( lm_red, lm_p))
        /* no reduction of this monomial possible */
```

```
        return p;
      else
        /* no reduction possible by using lm_red but maybe
           by leading monomials of other polynomials */
        ps_aux = REST( ps_aux);
      }
    if( ps_aux == EMPTY)
      return p;
    }
  return p;
  }
/**************************************************************/
Poly complete_normal_form_poly( Poly p, PS ps)              /*
  O u t p u t: Iterative application of partial normal form
***************************************************************/
  {
  Poly cnf = zero_poly( p);
  p = partial_normal_form_poly( p, ps);
  while( !is_zero_poly( p))
    {
    cnf = prepend_monomial( leading_monomial( p), cnf);
    p = partial_normal_form_poly( remaining_polynomial( p), ps);
    }
  return combine_poly( cnf, p);
  }
/**************************************************************/
PS inter_reduced_ps( PS ps)                                 /*
  O u t p u t: Polynomials whoose leading monomial is a
    multiple of one of the others are canceled. Each
    polynomial is reduced w.r.t. the others.
***************************************************************/
  {
  return completely_reduced_ps(
           canceled_multiples_ps( ps));
  }
/**************************************************************/
PS canceled_multiples_ps( PS ps)                            /*
  O u t p u t: Polynomials whoose leading monomial is a
    multiple of one of the others are canceled.
***************************************************************/
  {
  Poly p;
```

```
  PS ps0=EMPTY;
  while( ps != EMPTY)
    {
    p   = FIRST( ps);
    ps  = REST( ps);
    ps0 = COMP( p, ps0);
    ps  = thinned( p, ps,
            is_leading_monomial_multiple_poly);
    }
  return INV( ps0);
  }

/*************************************************************/
PS completely_reduced_ps( PS ps)                            /*

  O u t p u t: Each polynomial is reduced w.r.t. the others.
*************************************************************/
  {
  Poly p, p_red;
  PS ps0=EMPTY, lower_part=EMPTY;
  while( ps != EMPTY)
    {
    p  = FIRST( ps);
    ps = REST( ps);
    p_red = head_reduced_poly( p, lower_part);
    if( !is_zero_poly( p_red))
      ps0 = COMP( p_red, ps0);
    lower_part = CONC( lower_part, singleton( p));
    }
  return INV( ps0);
  }

/*************************************************************/
Poly head_reduced_poly( Poly p, PS ps)                      /*

  O u t p u t: Leading monomial unchanged.
    Remaining polynomial completely reduced.
*************************************************************/
  {
  Mon lm;
  Poly rp, cnf;
  lm  = leading_monomial( p);
  rp  = remaining_polynomial( p);
  cnf = complete_normal_form_poly( rp, ps);
  return prepend_monomial( lm, cnf);
  }
```

```
/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

 Groebner Bases Algorithm
 with application of criteria

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/*********************************************************/
PS GB_criteria( PS F)                                  /*

  I n p u t:
        F ... a set of polynomials.
  O u t p u t:
        G ... a completely reduced Groebner basis.
  M e t h o d:
    This version of the algorithm is also described in [Buchberger, 1987c]
    and was then implemented in MUMATH by Buchberger (see [Buchberger, 1987b]).

    GB_criteria implements the original algorithm for computing
    Groebner bases as introduced in

         B. Buchberger, An Algorithm for Finding tha Basis Elements of
         the Residue Class Ring Modulo a Polynomial Ideal.
         Ph. D. Thesis, Univ. of Innsbruck (Austria), Math. Dept.,
         1965. Published also in:
           Aequationes Mathematicae, Vol. 4/3 (1970), 374 - 383.

    with the refinement of using the "chain criterion" introduced in

         B. Buchberger, A Criterion for Detecting Unnecessary Reductions
         in the Construction of Groebner Bases. Proc. of the EUROSAM 79
         Symposium, Springer LNCS Vol. 72 (1979), 3 - 21.

    A tutorial introduction into the method of Groebner bases with
    detailed explanation of the basic concepts, the algorithm for
    constructing Groebner bases, and the main applications of Groebner
    bases for various fundamental problems in algebraic geometry and,
    in particular, for the solution of multivariate systems of equations
    is given in

          B. Buchberger, Groebner Bases: An Algorithmic Method in
          Polynomial Ideal Theory. In: Multidimensional Systems Theory
          (N. K. Bose ed.), Reidel Publishing Company (1985), Chapter 6,
          184 -232.

     This tutorial is for readers without any special knowledge in
     algebraic geometry or polynomial ideal theory.

    In the present implementation, the algorithm is tuned in several
    ways that crucially improve the computation speed:
```

```
        The polynomials in the basis are kept sorted w.r.t. the
        admissible ordering applied to leading power products.

        The polynomial pairs in the pair list are kept sorted w.r.t.
        to the admissible ordering applied to the "LCM" of the leading
        power products of the two polynomials in the pairs.

   Consequences are:

        The search for an appropriate polynomial in the chain criterion can
        be stopped as soon as a polynomial in the basis is encountered
        whose leading exponent is greater (in the admissible ordering)
        than the "LCM" of the pair considered.

        In the reduction, the search for an appropriate polynomial
        in the basis that reduces the given polynomial can be stopped
        as soon as a polynomial is encountered whose leading exponent
        is greater (in the admissible ordering) than the exponent
        considered for reduction.

        After computation of a partially reduced Groebner basis,
        the formation of a completely reduced Groebner basis may
        proceed by reducing each polynomial in the basis w.r.t. to
        the preceding ones. Polynomials in the Groebner basis
        whose leading exponents are multiples of the leading
        exponents of preceding polynomials in the basis may be canceled
        without actually performing the reduction.

   Other refinements are:

        In the formation of S-polynomials and in the reduction,
        the leading exponents are canceled instead of reduced to
        zero by subtraction.

        Complete reduction is formulated in terms of ``leading exponent
        reduction''. As a consequence, the subtraction in the reduction
        avoids superfluous copying of higher exponents.

***************************************************************/
  {
  PS G;
  PaS B;
  Pa p;
  Poly nf;
  LP h, f1, f2;
  int i=1;

fprintf( out_file_, "**** GB with Criteria ****\n");

  label_ = 1;
```

```
  G = labeled_ps( normalized_ps( initialized_ps( F)));
  B = initial_pairs( G);
  while( B != EMPTY)
    {
    p  = FIRST( B);
    B  = REST( B);
    f1 = FIRST( p);
    f2 = SECOND( p);
    if( !criteria( f1, f2, G, B))
      {
      nf = normal_form_poly( S_polynomial_lp( f1, f2), unlabeled_ps( G));
      if( !is_zero_poly( nf))
        {
        h = labeled_poly( normalized_poly( nf));
        B = updated_pairs( B, G, h);
        G = insert_lps( h, G);
        }
      else
        fprintf( out_file_, "Zero-reduction\n");
      }
    }
  return inter_reduced_ps( unlabeled_ps( G));
  }

Poly S_polynomial_lp( LP lp1, LP lp2)
  {
  return S_polynomial_poly( polynomial_lp( lp1),
                            polynomial_lp( lp2));
  }

Boolean criteria( LP lp1, LP lp2, PS G, PaS B)
  {
  if( product_criterion( lp1, lp2))
    {
    fprintf( out_file_, "Product Criterion\n");
    return TRUE;
    }
  if( explicit_chain_criterion( lp1, lp2, G, B))
    {
    fprintf( out_file_, "Chain Criterion\n");
    return TRUE;
    }
  return FALSE;
  }

Boolean product_criterion( LP lp1, LP lp2)
```

```
  {
  PP pp1, pp2;
  pp1 = leading_power_product_lp( lp1);
  pp2 = leading_power_product_lp( lp2);
  return is_disjoint_pp( pp1, pp2);
  }

Boolean explicit_chain_criterion( LP lp1, LP lp2, PS G, PaS B)
  {
  Mon m1, m2, lm, lcm;
  LP p;
  Pa pa1, pa2;
  m1  = leading_monomial_lp( lp1);
  m2  = leading_monomial_lp( lp2);
  lcm = lcm_monic_mon( m1, m2);
  while( G != EMPTY)
    {
    p = FIRST( G);
    G = REST( G);
    lm = leading_monomial_lp( p);
    if( is_greater_mon( lm, lcm))
      return FALSE;
    if( is_equal_lp( p, lp1) || is_equal_lp( p, lp2))
      continue;
    if( !is_multiple_mon( lcm, lm))
      continue;
    pa1 = new_pair( lp1, p);
    pa2 = new_pair( p, lp2);
    if( !contains_pas( B, pa1) &&
        !contains_pas( B, pa2))
      return TRUE;
    }
  return FALSE;
  }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Groebner Bases Algorithm with Interreduction
  before S-Polynomial Consideration

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/*************************************************************/
PS GB_reduce_all( PS F)                                    /*

  I n p u t:
        F ... a set of polynomials.
```

```
 O u t p u t:
      G ... a completely reduced Groebner basis.
 M e t h o d:
   This version of the algorithm is also described in [Buchberger, 1987c]
   and was then implemented in MUMATH by Buchberger (see [Buchberger, 1987b]).

   GB_reduce_all implements the improved variant of the algorithm
   for computing Groebner bases as introduced in

         B. Buchberger, Groebner Bases: An Algorithmic Method in
         Polynomial Ideal Theory. In: Multidimensional Systems Theory
         (N. K. Bose ed.), Reidel Publishing Company (1985), Chapter 6,
         184 -232.

   (We do not describe the principal strategy of this variant here).

   In the present implementation, the algorithm is tuned in several
   ways that crucially improve the computation speed:

        The polynomials in the basis are kept sorted w.r.t. the
        admissible ordering applied to leading power products.

        The polynomial pairs in the pair list are kept sorted w.r.t.
        to the admissible ordering applied to the "LCM" of the leading
        power products of the two polynomials in the pairs.

   Consequences are:

        The search for an appropriate polynomial in the chain criterion can
        be stopped as soon as a polynomial in the basis is encountered
        whose leading exponent is greater (in the admissible ordering)
        than the "LCM" of the pair considered.

        In the reduction, the search for an appropriate polynomial
        in the basis that reduces the given polynomial can be stopped
        as soon as a polynomial is encountered whose leading exponent
        is greater (in the admissible ordering) than the exponent
        considered for reduction.

   Other refinements are:

        In the formation of S-polynomials and in the reduction,
        the leading exponents are canceled instead of reduced to
        zero by subtraction.

        Complete reduction is formulated in terms of ''leading exponent
        reduction''. As a consequence, the subtraction in the reduction
        avoids superfluous copying of higher exponents.

***************************************************************/
 {
```

```
  PS basis=EMPTY, reducers=EMPTY, reducibles;
  PaS pairs=EMPTY;
  Pa p;
  Poly nf;
  LP nfl, p1, p2;
  Word reducibles_basis, basis_pairs;

fprintf( out_file_, "**** GB All Reduced ****\n");

  label_ = 1;
  reducibles = labeled_ps( normalized_ps( initialized_ps( F)));
  basis_pairs = all_reduced( reducibles, reducers, basis, pairs);
  basis = FIRST( basis_pairs);
  pairs = SECOND( basis_pairs);
  while( pairs != EMPTY)
    {
    p  = FIRST( pairs);
    pairs  = REST( pairs);
    p1 = FIRST( p);
    p2 = SECOND( p);
    if( !criteria( p1, p2, basis, pairs))
      {
      nf = normal_form_poly( S_polynomial_lp( p1, p2),
                             unlabeled_ps( basis));
      if( !is_zero_poly( nf))
        {
        nfl = labeled_poly( normalized_poly( nf));
        reducibles_basis = split( nfl, basis,
                             is_leading_monomial_multiple_lp);
        reducibles = FIRST( reducibles_basis);
        basis = SECOND( reducibles_basis);
        reducers = singleton( nfl);
        pairs  = thinned_pairs( reducibles, pairs);
        basis_pairs = all_reduced( reducibles, reducers, basis, pairs);
        basis = FIRST( basis_pairs);
        pairs = SECOND( basis_pairs);
        }
      else
        fprintf( out_file_, "Zero-reduction\n");
      }
    }
  return unlabeled_ps( basis);
  }

/******************************************************************/
Word all_reduced( PS reducibles, PS reducers, PS basis, PaS pairs)  /*
```

```
  I n p u t: reducibles ... polynomials that can be reduced further
             reducers ..... candidates to get into the basis
             basis ........ current basis
             pairs ........ pair set.
  O u t p u t: (basis, pairs), where
    when reducibles is empty, reducers are adjoined to the basis
    and the pair set is updated.
*********************************************************************/
  {
  Poly nf;
  LP h;
  PS red_basis, red_reducers;
  Word reducible_basis, reducible_reducers;
  while( reducibles != EMPTY)
    {
    h = FIRST( reducibles);
    reducibles = REST( reducibles);
    nf = normal_form_lp( h,
           merge( basis, reducers, is_leading_monomial_greater_lp));
    if( !is_zero_poly( nf))
      {
      nf = labeled_poly( normalized_poly( nf));
      reducible_basis = split( nf, basis,
        is_leading_monomial_multiple_lp);
      reducible_reducers = split( nf, reducers,
        is_leading_monomial_multiple_lp);
      red_basis = FIRST( reducible_basis);
      basis     = SECOND(  reducible_basis);
      red_reducers = FIRST( reducible_reducers);
      reducers     = SECOND( reducible_reducers);
      pairs  = thinned_pairs( red_basis, pairs);
      reducibles = merge( reducibles,
        merge( red_basis, red_reducers, is_leading_monomial_greater_lp),
        is_leading_monomial_greater_lp);
      reducers  = insert_lps( nf, reducers);
      }
    } /* while */
  while( reducers != EMPTY)
    {
    h = FIRST( reducers);
    reducers = REST( reducers);
    pairs = updated_pairs( pairs, basis, h);
    basis = insert_lps( h, basis);
    }
```

```
  basis = completely_reduced_lps( basis);
  return LIST2( basis, pairs);
  }

/*xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Groebner Bases Algorithm with Application of the
  Criteria before Pair Set Update

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/
#include "groebner.h"

/*************************************************************/
PS GB_small_pair_set( PS F)                                /*

  I n p u t:
        F ... a set of polynomials.
  O u t p u t:
        G ... a completely reduced Groebner basis.
  M e t h o d:
    This version of the algorithm is also described in [Buchberger, 1987c]
    and was then implemented in MUMATH by Buchberger (see [Buchberger, 1987b]).

    GB_small_pair_set implements the original algorithm for computing
    Groebner bases as introduced in

         B. Buchberger, An Algorithm for Finding tha Basis Elements of
         the Residue Class Ring Modulo a Polynomial Ideal. Ph. D. Thesis,
         Univ. of Innsbruck (Austria), Math. Dept., 1965. Published also in:
         Aequationes Mathematicae, Vol. 4/3 (1970), 374 - 383.

    with the refinement of using the "chain criterion" introduced in

         B. Buchberger, A Criterion for Detecting Unnecessary Reductions
         in the Construction of Groebner Bases. Proc. of the EUROSAM 79
         Symposium, Springer LNCS Vol. 72 (1979), 3 - 21.

    A tutorial introduction into the method of Groebner bases with
    detailed explanation of the basic concepts, the algorithm for
    constructing Groebner bases, and the main applications of Groebner
    bases for various fundamental problems in algebraic geometry and,
    in particular, for the solution of multivariate systems of equations
    is given in

          B. Buchberger, Groebner Bases: An Algorithmic Method in
          Polynomial Ideal Theory. In: Multidimensional Systems Theory
          (N. K. Bose ed.), Reidel Publishing Company (1985), Chapter 6,
          184 -232.

     This tutorial is for readers without any special knowledge in
     algebraic geometry or polynomial ideal theory.
```

```
In the present implementation, the algorithm is tuned in several
ways that crucially improve the computation speed:

     The polynomials in the basis are kept sorted w.r.t. the
     admissible ordering applied to leading power products.

     The polynomial pairs in the pair list are kept sorted w.r.t.
     to the admissible ordering applied to the "LCM" of the leading
     power products of the two polynomials in the pairs.

This has various consequences for the subalgorithms. The two most
important consequences are:

     The chain criterion for a given pair in the pair set B
     can be checked without any search through B.

     The chain criterion can be applied during formation and updating
     of B. No superfluous pairs are ever added to B.

Other consequences are:

     The search for an appropriate polynomial in the chain criterion can
     be stopped as soon as a polynomial in the basis is encountered
     whose leading exponent is greater (in the admissible ordering)
     than the "LCM" of the pair considered.

     In the reduction, the search for an appropriate polynomial
     in the basis that reduces the given polynomial can be stopped
     as soon as a polynomial is encountered whose leading exponent
     is greater (in the admissible ordering) than the exponent
     considered for reduction.

     After computation of a partially reduced Groebner basis,
     the formation of a completely reduced Groebner basis may
     proceed by reducing each polynomial in the basis w.r.t. to
     the preceding ones. Polynomials in the Groebner basis
     whose leading exponents are multiples of the leading
     exponents of preceding polynomials in the basis may be canceled
     without actually performing the reduction.

Other refinements are:

     The product criterion is also applied already during formation and
     updating of the pair list B.

     In the formation of S-polynomials and in the reduction,
     the leading exponents are canceled instead of reduced to
     zero by subtraction.

     Complete reduction is formulated in terms of ``leading exponent
     reduction''. As a consequence, the subtraction in the reduction
```

```
        avoids superfluous copying of higher exponents.
***************************************************************/
  {
  PS G;
  PaS B;
  Pa p;
  Poly nf;
  LP h, f1, f2;
  int i=1;

fprintf( out_file_, "**** GB Small Pair Set ****\n");

  label_ = 1;
  G = labeled_ps( normalized_ps( initialized_ps(F)));
  B = initial_pairs_with_criteria( G);
  while( B != EMPTY)
    {
    p  = FIRST( B);
    B  = REST( B);
    f1 = FIRST( p);
    f2 = SECOND( p);
    nf = normal_form_poly( S_polynomial_lp( f1, f2), unlabeled_ps( G));
    if( !is_zero_poly( nf))
      {
      h = labeled_poly( normalized_poly( nf));
      B = updated_pairs_with_product_criterion( B, G, h);
      G = insert_lps( h, G);
      B = thinned_by_chain_criterion( B, G, p);
      }
    else
      fprintf( out_file_, "Zero-reduction\n");
    }
  return inter_reduced_ps( unlabeled_ps( G));
  }

/***************************************************************/
PaS initial_pairs_with_criteria( PS ps)                        /*

  O u t p u t: Pairs that can be built of polynomials in ps,
    which can not be deleted by application of the criteria.
***************************************************************/
  {
  return (thinned_by_chain_criterion(
            initial_pairs_with_product_criterion( ps),
            ps,
            trivial_pair()));
```

```
  }

Pa trivial_pair( )
  {
  int l1=1, l2=1;
  return LIST2( LIST1( l1), LIST1( l2));
  }

/*************************************************************/
PaS initial_pairs_with_product_criterion( PS ps)            /*

  O u t p u t: Pairs that can be built of polynomials in ps,
    which can not be deleted by the product criterion.
**************************************************************/
  {
  PaS new_pairs=EMPTY;
  LP lp;

  while( ps != EMPTY)
    {
    lp = FIRST( ps);
    ps = REST( ps);
    new_pairs =
      updated_pairs_with_product_criterion( new_pairs, ps, lp);
    }
  return new_pairs;
  }

/*******************************************************************/
PaS updated_pairs_with_product_criterion( PaS pairs, PS ps, LP lp) /*

  O u t p u t: Union of 'pairs' and pairs that can be built of
    polynomials in ps and lp, which can not be deleted by the
    product criterion.
********************************************************************/
  {
  LP lp1;
  PaS new_pairs=pairs;

  while( ps != EMPTY)
    {
    lp1 = FIRST( ps);
    ps  = REST( ps);
    if( product_criterion( lp, lp1))
      {
      fprintf( out_file_, "Product Criterion\n");
      continue;
      }
```

```
    else
      new_pairs =
          expanded( new_pair( lp, lp1), new_pairs,
                    is_labeled_lcm_greater);
    }
  return new_pairs;
  }

/*******************************************************************/
PaS thinned_by_chain_criterion( PaS pair_set, PS basis, Pa pair)   /*

  O u t p u t: All pairs of pair_set unequal to pair, for which no
    polynomial in basis exists that satisfies the chain criterion.
********************************************************************/
  {
  Pa next_pair;
  PaS thinned_pairs=EMPTY;

  while( pair_set != EMPTY)
    {
    next_pair = FIRST( pair_set);
    pair_set  = REST( pair_set);
    if( chain_criterion( next_pair, basis, pair))
      {
      fprintf( out_file_, "Chain Criterion\n");
      continue;
      }
    else
      thinned_pairs = COMP( next_pair, thinned_pairs);
    }
  return INV( thinned_pairs);
  }

/*******************************************************************/
Boolean chain_criterion( Pa pa, PS basis, Pa pair)                 /*

  S p e c i f i c a t i o n: TRUE iff there exists a polynomial p
    in basis, s.t. triangle criterion holds for pa and p.
********************************************************************/
  {
  Mon lcm;
  LP lp;
  lcm = lcm_monic_mon( leading_monomial_lp( FIRST( pa)),
                       leading_monomial_lp( SECOND( pa)));
  while( basis != EMPTY)
    {
    lp = FIRST( basis);
```

```
    if( triangle_criterion( pa, lp, pair))
      return TRUE;
    if( is_greater_mon_pair_order( leading_monomial_lp( lp), lcm))
      return FALSE;
    else
      basis = REST( basis);
    }
  return FALSE;
  }
/******************************************************************/
Boolean triangle_criterion( Pa pa, LP lp, Pa pair)               /*

  S p e c i f i c a t i o n: TRUE iff pa unequal pair and
    lp is such a polynomial s.t.
      lcm( pa1, pa2) is a multiple of lp,
      (lp, pa1) and (lp, pa2) are less than pa.
******************************************************************/
  {
  Mon lcm;
  LP p1, p2;
  Pa aux1, aux2;
  lcm = lcm_monic_mon( leading_monomial_lp( FIRST( pa)),
                       leading_monomial_lp( SECOND( pa)));
  p1 = FIRST( pa);
  p2 = SECOND( pa);
  if( is_equal_lp( p1, lp) || is_equal_lp( p2, lp))
    return FALSE;
  if( !is_multiple_mon( lcm, leading_monomial_lp( lp)))
    return FALSE;
  aux1 = new_pair( p1, lp);
  aux2 = new_pair( lp, p2);
  if( (is_labeled_lcm_greater( pa, aux1) ||
       product_criterion( p1, lp) ||
       is_equal_pa( aux1, pair))
      &&
      (is_labeled_lcm_greater( pa, aux2) ||
       product_criterion( p2, lp) ||
       is_equal_pa( aux2, pair)))
    return TRUE;
  else
    return FALSE;
  }
```

Bibliography

[Bobrow *et al.*, 1988] D.G. Bobrow, L.G. DeMichiel, R.P Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. COMMON LISP object system specification X3J13 Document 88-002R. *SIGPLAN Notices*, 23(9), 1988.

[Böge *et al.*, 1986] W. Böge, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Gröbner Bases. *J. Symb. Comp.*, 2(1):pages 83–98, 1986.

[Buchberger *et al.*, 1992] B. Buchberger, H. Hong, and W. Windsteiger. The Simulated Package Principle: A Tool for Rudimentary Genericity at Compile-Time. Technical report, Univ. Linz, RISC, in preparation, Linz, Austria, 1992.

[Buchberger, 1965] B. Buchberger. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal (German).* PhD thesis, Univ. Innsbruck, Dept. of Math., Innsbruck, Austria, 1965.

[Buchberger, 1970] B. Buchberger. An algorithmic criterion for the solvability of algebraic sys tems of equations (German). *Aequationes Mathematicae*, 4(3):pages 374–383, 1970.

[Buchberger, 1976a] B. Buchberger. A theoretical basis for the reduction of polynomials to canonical form (part I). *ACM SIGSAM Bull.*, 10(3):pages 19–29, 1976.

[Buchberger, 1976b] B. Buchberger. Some Properties of Gröbner Bases for Polynomial Ideals. *ACM SIGSAM Bull.*, 10(4):pages 19–24, 1976.

[Buchberger, 1979] B. Buchberger. A criterion for detecting unnecessary reductions in the construction of Gröbner bases. In W. Ng, editor, *Proc. EUROSAM'79*, pages 3–21, Marseille, France, June 1979.

[Buchberger, 1983] B. Buchberger. A note on the complexity of constructing Gröbner bases. In H. van Hulzen, editor, *Proc. EUROCAL'83*, pages 137–145, London, England, March 1983.

[Buchberger, 1985] B. Buchberger. Gröbner Bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, Dordrecht-Boston-Lancaster, 1985.

[Buchberger, 1987a] B. Buchberger. Applications of Gröbner Bases in Non-Linear Computational Geometry. In *Proc. Workshop on Scientific Software*, IMA, Minneapolis, USA, 1987. Springer.

[Buchberger, 1987b] B. Buchberger. Implementation of a Gröbner bases package in the muMATH system, 1987.

[Buchberger, 1987c] B. Buchberger. User's manual for GROEBNER, a Gröbner bases package in the muMATH system. Technical Report RISC-Linz Series 87-38, Univ. Linz, RISC, Linz, Austria, 1987.

[Buchberger, 1990] B. Buchberger. An Implementation of Gröbner Bases in Mathematica. Technical Report RISC-Linz Series 90-58, Univ. Linz, RISC, Linz, Austria, 1990.

[Buchberger, 1991a] B. Buchberger. *Gröbner Bases: Elementary Theory and Applications*. Springer Verlag New York - Vienna, in preparation, 1991.

[Buchberger, 1991b] B. Buchberger. Gröbner Bases in Mathematica: Enthusiasm and Frustration. In P.W. Gaffney and E.N. Houstis, editors, *Proc. IFIP TC2/WG2.5 Working Conference on Programming Environments for High-Level Scientific Problem Solving*, Karlsruhe, Germany, 1991.

[Möller, 1985] H.M. Möller. A reduction strategy for the Taylor resolution. In *Proc. EUROCAL'85, Springer Lec. Notes Comp. Sci. 204*, pages 526–534, 1985.

[Mora and Möller, 1986] F. Mora and M. Möller. New constructive methods in classical ideal theory. *J. Algebra*, 100:pages 138–178, 1986.

[Robbiano, 1985] L. Robbiano. Term orderings on the polynomial ring. In *Proc. EUROCAL'85, Springer Lec. Notes Comp. Sci. 204*, pages 513–517, 1985.

[Windsteiger, 1990a] W. Windsteiger. An Approach to Object-Oriented Programming in C. Technical Report RISC-Linz Series 90-57, Univ. Linz, RISC, Linz, Austria, 1990.

[Windsteiger, 1990b] W. Windsteiger. An Implementation of Rational Functions in PCL. Technical Report RISC-Linz Series 90-56, Univ. Linz, RISC, Linz, Austria, 1990.

[Winkler, 1986] F. Winkler. Solution of equations I: Polynomial ideals and Gröbner bases. In *Proc. Conference on Computers in Mathematics*, pages 383–407, Stanford, California, USA, 1986.