GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB[1]

Manual for Version 2.0

Wolfgang Windsteiger Bruno Buchberger
RISC-Linz

—

Research Institute for Symbolic Computation
Johannes Kepler Universität Linz
A-4040 Linz, Austria
Tel: +43 (7236) 3231-21, Fax: +43 (7236) 3231-30
email: wwindste@risc.uni-linz.ac.at
buchberg@risc.uni-linz.ac.at

—

March 16, 1995

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

Abstract

Almost every Computer Algebra System contains some implementation of the Gröbner bases algorithm. The present implementation has the following specific features:

- ◇ The *source code is distributed* and publically available *free of charge*.
- ◇ The library is *written in C*.
- ◇ A simple but efficient mechanism of *polymorphism* is implemented that enables the user to adjust the library to a wide variety of coefficient domains, power product and polynomial representations, admissible orderings, selection strategies for pairs etc.

Thus, GRÖBNER should be a useful tool

- ◇ for those who want to do *research in Gröbner bases theory and applications* and, hence, need access to all details of the implementation
- ◇ and also for those who want to *apply the algorithm as a black box*, possibly as a subalgorithm in a larger implementation, and need high efficiency.

Contents

List of Figures

List of Tables

Introduction

This document is a manual for the library GRÖBNER, which is a C implementation, written by the first author, of the second author's Gröbner bases algorithm. The implementation is part of the first author's diploma thesis, [Windsteiger, 1992] and is modeled after earlier implementations by the second author in various symbolic computation languages, notably in *Mathematica*, see [Buchberger, 1990]. The design reasons for using C in the present implementation are described in [Buchberger, 1991]. Summarizing, a good implementation of an algebraic algorithm should combine both speed and "polymorphism". By using C and a systematic renaming mechanism, which is based on a proposal by the second author and is described in the present document, we believe that, in fact, we can achieve C speed and reasonable polymorphic generality.

The Use of C in GRÖBNER

GRÖBNER is based on SACLIB, a library of C routines derived from George Collins' well-known SAC-2 Computer Algebra system by translating it into C and systematizing it in various respects, see [Hong *et al.*, 1993]. Various authors, under the direction of Hoon Hong, cooperated for implementing SACLIB. Before GRÖBNER can be used, SACLIB must be obtained and installed, see Section 1 for details. SACLIB provides a list processing system together with an appropriate storage management, and various sublibraries for elementary and advanced problem areas in multivariate polynomial algebra, for example cylindrical algebraic decomposition. In fact, for GRÖBNER, we need only the list processing system, long integer arithmetic, rational number arithmetic, finite field arithmetic, and polynomial arithmetic of SACLIB. Also, since GRÖBNER should be easily portable to different hardware architectures, we decided to use SACLIB's `Word` as the only basic type in GRÖBNER: `Word` is either a small integer or a pointer to a list; SACLIB provides the storage management for `Word`.

Polymorphism in GRÖBNER

GRÖBNER is designed "polymorphically", i.e. in such a way that the implementation of algorithms for a domain is totally independent of the subdomains involved. For example, the domain of *monomials* naturally involves *two subdomains*, the domain of *coefficients* and the domain of *power products*. In the spirit of polymorphism, the code for monomial multiplication ("componentwise multiplication") should be independent of the names for coefficient multiplication and power product multiplication. It should be avoided to have an extra implementation for monomial multiplication for each combination of a coefficient domain and a power product domain.

In GRÖBNER, polymorphism is accomplished by introducing a *virtual domain* as an additional layer between a domain and any of its subdomains. Instead of calling a routine of a subdomain directly, a routine of the corresponding virtual domain is called. For each possible concrete subdomain, we supply a *coercion unit*, which replaces each call to the virtual domain by a call to the concrete subdomain. The virtual domains together with their coercion units describe the interface between a domain and its subdomains. We will explain this mechanism in more detail in the subsequent sections.

Structure of the Document

The first part of this manual is a User's Guide that explains the design objectives of GRÖBNER and their realization:

- ◇ a guide for installing and using GRÖBNER (Section 1),
- ◇ the concept of a virtual domain consisting of a virtual type and virtual operations (Section 2),
- ◇ all available domains (data types) in GRÖBNER (Section 3) and their interplay via coercion units (Section 4),
- ◇ additional possibilities of adjusting the library (Section 5),
- ◇ all global variables used in GRÖBNER (Section 6), and
- ◇ a survey on our naming conventions (Section 7).

The second part contains a reference manual with a detailed description of each function contained in GRÖBNER.

Part I
User's Guide

1 How to Use GRÖBNER

This section describes, how to install and use GRÖBNER on a UNIX system.

1.1 Availability of GRÖBNER and SACLIB

GRÖBNER and SACLIB are available via anonymous `ftp` from

`melmac.risc.uni-linz.ac.at`

in the directories `pub/GB` and `pub/saclib` respectively. A `dvi`-file of this manual can also be found in `pub/GB`.

For further information concerning GRÖBNER contact:

Prof. Dr. Bruno Buchberger (`buchberg@risc.uni-linz.ac.at`) or
Dipl.-Ing. Wolfgang Windsteiger (`wwindste@risc.uni-linz.ac.at`)

For further information concerning SACLIB contact:

Prof. Dr. George Collins (`gcollins@risc.uni-linz.ac.at`) or
Dr. Hoon Hong (`hhong@risc.uni-linz.ac.at`) or
`saclib@risc.uni-linz.ac.at`

1.2 Installation Procedure

1. *Retrieve* SACLIB and install it on your machine (see Section 1.1).

2. *Retrieve* GRÖBNER. The file available in the directory stated in Section 1.1 is a compressed `tar`-file, `GB.tar.Z`. (When using `ftp`, make sure to transfer in binary mode!).

3. *Move* the file into the directory that should be the parent directory of the "Gröbner-directory", see also Section 1.3.

4. *Uncompress* the file with the command

```
unix% uncompress GB.tar.Z
```

5. *Extract* the directory structure from the `tar`-file with the command

```
unix% tar -xvf GB.tar
```

A directory called `Groebner` will be created, which contains the directories described in Section 1.4.

6. *Modify* the installation script so that it fits to your environment. Use an available text editor (e.g. `vi`). (See the description of `install_groebner` in Section 1.4 and the comments in the script).

```
unix% cd Groebner/bin
unix% vi install_groebner
```

7. *Run* the installation script.

```
unix% install_groebner
```

8.
   - ◇ *Make sure* that the directory, which contains the scripts `groebner` and `xgroebner`, is contained in your search path or
   - ◇ *Move/Copy* those scripts into any of the directories contained in the search path or
   - ◇ *Provide* symbolic links from a directory in the search path to those scripts.

9. GRÖBNER is now installed. In order to change e.g. the coefficient domain of polynomials or the ordering of monomials you have to

   - ◇ *Edit* the respective setup file (see page 24) in the directory `GROEBNER/include`.
   - ◇ *Recompile* GRÖBNER, i.e. issue the command `make` in the directory `GROEBNER/src`,
   - ◇ *Eventually recompile* the I/O library, i.e. issue the command `make` in the directory `GROEBNER-IO/src`. This step is of course only necessary when the changes have an impact on in- or output. For instance, changing the coefficient domain influences the IO-library, whereas changing the ordering of monomials does not.
   - ◇ *Recompile* the application program, i.e. issue the command `make` in the directory `APP`.

   You do not have to reinstall the library!

1.3 Where to Locate the "Gröbner-Directory"

During the installation procedure described in Section 1.2 the directory `Groebner`, the "Gröbner-directory", is created.

We recommend that each user has a "private copy" of GRÖBNER, because each change in the setup (e.g. coefficient domain, ordering of monomials, etc.) makes re-compilation of the library necessary (see Section 2). Therefore, each user should locate the "Gröbner-directory" in one of the subdirectories of the home-directory, where compilation can be done without regard of other users.

A considerable compromise might be to locate versions of the library for some of the most frequently used setups in a publically available directory. Those versions could be utilized for most of the applications then. Only those who need special setups would need their private version then.

Example: Compile the library for polynomials over rational numbers and purely lexical ordering of monomials and put the resulting library into the directory `/usr/local/lib/groebner`, where it is usually available for all users. This version of the library might be sufficient for many users.

Those who want to experiment with different orderings, different coefficient domains, various levels of tracing of the algorithm, etc. have to keep separate copies of GRÖBNER in their own directories. Of course, they need the complete contents of the "Gröbner-directory" (see Section 1.4) for compilation in their private directories.

1.4 Files and Directories

In this section we briefly sketch the structure of the "Gröbner-directory" and give a short description of the contents of the various subdirectories.

`GROEBNER` Main directory for GRÖBNER. It contains the following subdirectories:

- `bin` Directory containing executable programs that are used by GRÖBNER. In the present state, it contains the shell scripts `activate-trace` and `compute-mask` and the C executable program `filter-trace` together with its source file `filter-trace.c`, which are described in Section 1.6. It must be guaranteed, that this directory is contained in the search path of the shell, otherwise the programs cannot be executed.
- `include` Directory containing all files that are included by an `#include` statement in the preprocessing phase. These are the header files for the domains, the coercion units, and all setup files. In addition it also contains the file `trace.setup`, see Section 1.6.

`lib` Directory containing the library file `groebner.a`.

`src` Directory containing the source files together with the makefile and the object files.

`GROEBNER-IO` Main directory for the I/O library provided for GRÖBNER. We do not go into details here, see Section 1.6.

`APP` Directory containing a sample application, which should show, which steps are necessary to use GRÖBNER.

`bin` Directory containing an installation program and two shell-scripts to call the application program:

`install_groebner` This program installs GRÖBNER, the I/O library, and the application program in your environment. The first part of this program contains settings of some variables concerning pathnames in your environment. The comments explain which modifications are necessary for a correct installation. In the second part the library is *automatically* compiled and installed according to the setting of those variables. This step involves adjusting of the variables in the Makefiles, editing the shell-scripts `activate-trace` and `compute-mask` (see Section 1.6), and compilation of the libraries and the sample application. This part needs no modification by the user.

When installation is finished, mail containing a trace of the installation procedure is automatically sent to the authors. This mail serves for two purposes:

- ◇ Information about new users. Your mail address will be put on a list so that you can be informed about new versions, bug fixes, etc.
- ◇ Information about different platforms. Difficulties during installation on platforms different from the ones we know become transparent and can be overcome.

As a courtesy to the authors, do not delete this line from the installation script.

`groebner` to start the application program and set up the environment properly. Note that the command line argument "h" supplies help on the optional parameters that can be passed to the program.

`xgroebner` to start the application program in a separate X-term and also set up the environment.

`lib` Directory containing a bitmap file, which can be used as icon for GRÖBNER. In order to use this, one must configure the window manager appropriately.

`input` Directory containing test examples.

`man` Directory containing help files to be displayed for on-line help provided by the I/O library.

`groebner.Xdefaults` Contains settings of some variables when the application program is used under the X-windows system. One can either include the contents of this file into ones own `.Xdefaults` file or merge the contents into the X resource database when the program is started.

1.5 The Makefile

Since re-compilation of the system is necessary after every change in a setup-file, e.g. after changing the coefficient domain or the ordering of monomials, it is crucial to have a carefully maintained *makefile*. This prevents unnecessary compilation provided that the dependencies between the files are set up correctly. Particularly on slower machines this yields an enormous speed up of the process of re-compilation.

In the following, we will describe the variables defined in the makefile.

`CC` The C compiler that should be used for compilation. This variable is used by `make` in the default rule, how to produce an object file from a source file.

`CFLAGS` Options given to the compiler. Like CC, this variable is used in the default rule.

`SACINCLUDE` Contains the directory, where the header file for SACLIB can be found.

`GBINCLUDE` Contains the directory, where the header files for GRÖBNER can be found.

`GBIOINCLUDE` When our I/O library is used, this variable contains the directory, where the header file for the library can be found.

`ALLDEPEND` Contains the files, on which all other files depend. In the present stage, these files are `groebner.h` and `groebner_globals.h`.

`LIB` Contains the name of the library that is produced.

`FILTER` Contains the program that should be used for filtering trace statements as described in Section 1.6.

`OBJECTS` Contains all object files that should be contained in the library.

1.6 Input and Output

Input and output of data is *not* part of GRÖBNER, but we will provide an additional I/O library for reading and printing of all data types available in GRÖBNER, see also Section 1.4. We will report on this in a subsequent paper, see [Windsteiger, 1993].

However, the code of GRÖBNER contains commands for a trace of the computation of a Gröbner basis, which is on the one hand desirable, when one intends to use GRÖBNER for studying the Gröbner bases algorithm, but on the other hand annoying, when the computation of a Gröbner basis is only a subtask within another program. Since GRÖBNER is intended to serve for both purposes, it supplies several *groups of trace statements*, which can be *activated* or *deactivated* independently from each other.

The suffix `.trace` in a filename indicates that this file contains trace statements. Before compilation, these files are piped through a filter that removes deactivated trace statements. In order to decide about activation of trace statements we provide a *trace code* for each group of trace statements and a *mask*. The trace code must be an integer of the form 2^i and the mask is an arbitrary non-negative integer. A group of trace statements is defined to be active if and only if the *bitwise conjunction* of its trace code with the mask is non-zero. We will illustrate this in the following example.

Example: Suppose the mask is 6 (binary representation: 0110). Then the group with code 4 (0100) will be activated, whereas the group with code 8 (1000) will be deactivated.

mask =	0	1	1	0
code =	0	1	0	0
conjunction =	0	1	0	0

mask =	0	1	1	0
code =	1	0	0	0
conjunction =	0	0	0	0

Trace statements have to match the following pattern so that they can be recognized and processed correctly. They have to be one-line comments starting in column one, beginning with the marker "TRACE" followed by the trace code (separated by at least one blank) followed by the actual command.

Hence, a typical trace statement looks as follows:

```
/* TRACE 4 print_pp( leading_power_product_lp( h));*/
```

The filter `activate-trace` takes a list of files to be filtered. If parameters are omitted, all files ending in `.trace` are filtered. First the program `filter-trace` is called, that deletes all deactivated trace statements. By a simple `sed` program the marker "TRACE", the trace code, and the comment characters are stripped from the remaining activated trace statements.

Furthermore, we provide a program `compute-mask` that computes the desired mask from the entries in the file `trace.setup`, which contains the description of the available groups of trace statements (one per line). If a group should be activated the corresponding line has to contain one of the characters "Y", "y", "X", or "x" at the end of the line, any other character is interpreted as deactivation of the respective group. The group description in line n of the file `trace.setup` corresponds to the trace statements with code 2^{n-1}, i.e. the characters "Y", "y", "X", and "x" at the end of line n correspond to a "1" in the n-th bit in the binary representation of the mask.

1.7 Sample Session

In this section we show how one can actually compute a Gröbner basis using GRÖBNER together with the supplied I/O library and the sample application program. This application repeats the following steps in a loop:

1. Reading in a set of polynomials from a file or from the terminal by means of the routines provided by the I/O library,
2. Computing a Gröbner basis of this set by one of the variants of the Gröbner bases algorithm contained in GRÖBNER,
3. Printing intermediate results to the screen or to a file according to the selected trace level (see Section 1.6), and
4. Printing of the Gröbner basis to the screen or to a file by means of the routines provided by the I/O library.

Suppose we want to compute the Gröbner basis of the set

$$G = \{z^2 - ax^2 - ay^2, xz - 2z + xy, x^2 - y\}$$

over the rational numbers w.r.t. the purely lexical ordering induced by $a < x < y < z$.

```
unix% groebner
Starting SACLIB with 4000000 cells in memory
Path for Input-file: /usr/users/wwindste/Groebner/input

Type input-file name ( <RETURN> = stdin, ? = help):

Path for Output-file: /usr/users/wwindste/Groebner/input

Output-file name ( <RETURN> = stdout, ? = help):

***********************************************************
Input File:  stdin
***********************************************************
***********************************************************
Output File: stdout
```

```
***********************************************************

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Current Compiled Version:

Coefficient Domain: Rational Numbers
Term Ordering: Purely Lexicographic Ordering
Pair Ordering: Purely Lexicographic Ordering
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

=========================
WELCOME TO GROEBNER
  by RISC-Linz Institute
Good Luck ...
=========================

Compute GROEBNER BASIS (Crude Version) ........... 1
Compute GROEBNER BASIS (Criteria) ................ 2
Compute GROEBNER BASIS (Reduce All) .............. 3
Compute GROEBNER BASIS (Small Pair Set) .......... 4
EXIT ............................................. 0

Enter your choice: 3

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Groebner Basis with Reduce All
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Initializing Global Variables for GROEBNER ...

Initializing Global Variables for GROEBNER-IO ...

Enter ring-variables in ascending order ( ? for help): a,x,y,z;

***********************************************************
Number of Variables: 4
Ring Variables: z > y > x > a
***********************************************************

Enter Polynomial Set (? for help):

{z^2 - a*x^2 - a*y^2, x*z - 2*z + x*y, x^2 - y}
Reading of polynomial set complete

{
```

```
       2
- y + x ,


z x - 2 z + y x,


 2    2     2
z  - y a - x a

}


Groebner Basis =

{

 6     6      5       4       3       2
x a - x  - 4 x a + 5 x a - 4 x a + 4 x a,


     2
y - x ,


    1  5    1  5   1  4    1  4   1  3    1  3   1  2
z + - x a - - x  - - x a - - x  + - x a - - x  - - x a
    8       8      4       4      8       2      4

}

User Time: 0.016667
System Time: 0.000000

Compute GROEBNER BASIS (Crude Version) ........... 1
Compute GROEBNER BASIS (Criteria) ................ 2
Compute GROEBNER BASIS (Reduce All) .............. 3
Compute GROEBNER BASIS (Small Pair Set) .......... 4
EXIT ............................................. 0

Enter your choice: 0
```

```
*****************
       Bye
*****************
unix%
```

Suppose we now want to compute the Gröbner basis of G over the rational numbers w.r.t. the total degree lexical ordering induced by $x < y < z$. We want to treat a as a constant (a parameter). This implies that we must use rational functions in a as the coefficient domain. The ordering of pairs (see Section 5.2) should be left unchanged.

We are also interested in some intermediate results, namely we wish to be informed

- ◇ in case of zero reduction,
- ◇ about application of the criteria, and
- ◇ about the leading term of a non-zero normal form.

Thus, we have to adjust the setup in the following respects:

- ◇ Term ordering (see Section 5.1),
- ◇ Coefficient domain (see Section 4.2), and
- ◇ Granularity of tracing (see Section 1.6).

First, we have to edit the relevant lines in the respective setup-files (see Section 2.8, in particular figures 2 to 4). Then we have to re-compile GRÖBNER and, since the change of the coefficient domain affects I/O as well, the I/O library. Finally, we re-compile the application program and start it.

```
unix% cd Groebner/GROEBNER/include
unix% vi GB.to.setup
unix% vi GB.coef.setup
unix% vi trace.setup
unix% cd ../src
unix% make
unix% cd ../../GROEBNER-IO/src
unix% make
unix% cd ../../APP
unix% make
unix% groebner
Starting SACLIB with 4000000 cells in memory

Path for Input-file: /usr/users/wwindste/Groebner/input
```

```
Type input-file name ( <RETURN> = stdin, ? = help):

Path for Output-file: /usr/users/wwindste/Groebner/input

Output-file name ( <RETURN> = stdout, ? = help):

***********************************************************
Input File:  stdin
***********************************************************
***********************************************************
Output File: stdout
***********************************************************

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Current Compiled Version:

Coefficient Domain: Rational Functions
Term Ordering: Total Degree Lexicographic Ordering
Pair Ordering: Purely Lexicographic Ordering
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

=========================
WELCOME TO GROEBNER
  by RISC-Linz Institute
Good Luck ...
=========================


Compute GROEBNER BASIS (Crude Version) ........... 1
Compute GROEBNER BASIS (Criteria) ................ 2
Compute GROEBNER BASIS (Reduce All) .............. 3
Compute GROEBNER BASIS (Small Pair Set) .......... 4
EXIT ............................................. 0

Enter your choice: 5

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Groebner Basis with Small Pair Set
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Initializing Global Variables for GROEBNER ...
Initializing Global Variables for GROEBNER-IO ...

Enter ring-variables in ascending order ( ? for help): x,y,z;
Enter field-variables (terminated by ";"): a;
```

```
*************************************************************
Number of Variables: 3
Ring Variables: z > y > x
Field Variables: a
*************************************************************

Enter Polynomial Set (? for help):

{z^2 - a*x^2 - a*y^2, x*z - 2*z + x*y, x^2 - y}
Reading of polynomial set complete


{

 2
x  - y,



z x + y x - (2) z,


 2        2        2
z  - (a) y  - (a) x

}

Product Criterion
New polynomial:
[1 1 0]
Product Criterion
Zero-reduction
New polynomial:
[0 2 1]
Product Criterion
Chain Criterion
New polynomial:
[0 3 0]
Product Criterion
Product Criterion
Product Criterion
Zero-reduction
Zero-reduction
```

```
Zero-reduction
Zero-reduction
Groebner Basis =

{


 2
x  - y,


z x + y x - (2) z,


       2
z y + y  + (2) y x - (4) z,


 2        2
z  - (a) y  - (a) y,



 2    2 a + 2  2   a - 4         8        2 a
y x - ------- y  + ----- y x + ----- z - ----- y,
       a - 1       a - 1       a - 1    a - 1



        2                                                  2
 3    3 a  + 13 a  2       12 a               32 a        4 a  + 4 a
y  - ------------ y  - ------------ y x + ------------ z - ------------ y
       2                 2                  2                 2
     a  - 2 a + 1      a  - 2 a + 1       a  - 2 a + 1      a  - 2 a + 1

}


User Time: 0.216667
System Time: 0.000000


Compute GROEBNER BASIS (Crude Version) ........... 1
Compute GROEBNER BASIS (Criteria) ................ 2
Compute GROEBNER BASIS (Reduce All) .............. 3
```

```
Compute GROEBNER BASIS (Small Pair Set) .......... 4
EXIT ............................................. 0

Enter your choice: 0

*****************
       Bye
*****************
unix%
```

The trace during the calculation is to be interpreted as follows: The first S-polynomial need not be reduced because it reduces to zero by the "Product Criterion", the next S-polynomial reduces to a polynomial whose leading term is zy (it has the exponent list `[1 1 0]`), the fourth S-polynomial reduces to zero, which could not be detected by any of the criteria, the seventh S-polynomial reduces to zero by the "Chain Criterion", and so on.

1.8 Using GRÖBNER in Your Own Application

This section describes how to use GRÖBNER (or parts of it) in ones own application.

First, one has to setup the `main` program for calling routines from SACLIB. This step is described in the SACLIB manual, which is contained in the SACLIB distribution.

In addition, all one needs to do is to include the header file `groebner.h` and to link the library `groebner.a` to the application. Functions needed for initialization of the library are `extract_info4groebner`, `init_groebner_globals`, and `start_groebner`. See Part II of this manual for a description of these functions, and the `main` program of our sample application in the directory `Groebner/APP`.

If the provided I/O library is not used, one probably has to provide conversion routines that convert data into the format needed by GRÖBNER. The data structures used in GRÖBNER are described accurately in Section 3.

2 Polymorphism by Virtual Domains

2.1 The Problem of Code Repetition and Polymorphism

One of the most pressing problems in designing computer algebra systems is the problem of avoiding repetition of structurally similar code for similar algebraic structures. This problem is serious because, in systems with deeply nested hierarchies of algebraic structures, a few repetitions of code at each level of the hierarchy may, in fact, lead to exponentially many repetitions of code in the entire system. Such code is hard to write and even harder to maintain and expand.

"Polymorphism" is a solution for this problem. Two essentially different implementation methods for polymorphism are reported in the literature: Replacement of an "abstract" function call by a "concrete" call at *compile-time* (see, for example, the AXIOM Computer Algebra System, [Jenks and Sutor, 1992]) or, alternatively, modification of the abstract call at *run-time* by analyzing the type of the arguments (see, for example, the experimental Gröbner bases implementation in the *Mathematica* system described in [Buchberger, 1990]). Realization of the first approach needs a complicated static type analysis system but yields fast code. Realization of the second approach is straightforward in languages, like *Mathematica*, that allow user-defined type tags for objects. However, this approach results in drastically slower code. In our variant of polymorphism for C programs (and programs in similar languages), we use "virtual" function calls and explicit replacement of these calls at compile time by an easy preprocessor. This mechanism is not as general as the first approach but both easy to implement and efficient.

2.2 An Example: The Implementation of Monomials

The domain of monomials over a field can be defined as the cartesian product of the field domain and the domain of power products. Multiplication of two monomials can be described as "componentwise multiplication". This definition of the product of two monomials is valid for any field of coefficients and any monoid of power products.

However, if we want to implement the algorithm, we must know, *which particular field* is used as the coefficient domain and *which particular monoid* is used as the power product domain in order to call the appropriate multiplication algorithms. This would result in different implementations of the multiplication algorithm for all combinations of subdomains.

Suppose all algorithm names have suffixes depending on the domain, for which they can be applied, such as `rn` for rational numbers, `ff` for finite fields, `iel` for integer exponent lists, and `ge` for Gödel exponents, which is another simple representation of power products. An Implementation of multiplication of monomials with rational number coefficients and integer exponent lists as representation for power products would look as follows[2]:

[2] We use a self-explanatory pseudo-code rather than referring to a particular programming language

product_mon_rn_iel((c_1, pp_1), (c_2, pp_2))
  c = product_rn(c_1, c_2)
  pp = product_iel(pp_1, pp_2)
  return (c, pp)

In contrast, an implementation for monomials with finite field coefficients and Gödel exponents as representation for power products would look as follows:

product_mon_ff_ge((c_1, pp_1), (c_2, pp_2))
  c = product_ff(c_1, c_2)
  pp = product_ge(pp_1, pp_2)
  return (c, pp)

In order to avoid this duplication of code, we introduce two *virtual subdomains* named `coefficient` (suffix `coef`) and `power_product` (suffix `pp`). In the implementation of the multiplication algorithm for monomials we only use the multiplication of these subdomains resulting in *one implementation*:

product_mon((c_1, pp_1), (c_2, pp_2))
  c = product_coef(c_1, c_2)
  pp = product_pp(pp_1, pp_2)
  return (c, pp)

By an appropriate renaming of these calls in the preprocessing phase we establish the connection to the desired function. In C this can be done for instance by renaming `product_coef` to `product_rn` by `#define` instructions to the C-preprocessor.

The general concept of virtual domains, virtual operations, and virtual data types is presented in more detail in the following sections.

2.3 What is a Virtual Domain?

Basically, a virtual domain consists of a *virtual data type* and operations on this data type, the *virtual operations*. As sketched in the previous example, a virtual domain is introduced whenever a domain has subdomains that can have more than one representation. Each subdomain corresponds to a virtual domain, which can be viewed as a variable ranging over all domains that can actually be used in place of the subdomain. We call these domains "virtual" since there is neither an actual implementation of the operations in a virtual domain nor a concrete structure of the virtual data type. Both virtual operations and virtual type are just mapped to already implemented operations and defined data structures, respectively.

A virtual domain is introduced as an intermediate layer between the domain and its subdomain and its purpose is to keep the operations in the domain independent of the currently chosen representation of the subdomain.

2.4 Virtual Operations and Renaming Units

In order to avoid the dependence of a domain on a subdomain, we split the call to an operation in the subdomain into two steps:

1. Call the (uniquely determined) *virtual operation* in the corresponding virtual domain.

2. Map the virtual operation to the appropriate operation in the subdomain.

The second step is carried out by renaming instructions. All renaming instructions from a virtual domain to an actually implemented domain are collected in the *renaming unit*.

2.5 Virtual Data Types and Definition Units

For each virtual domain we define a *virtual data type* as well. This is necessary as soon as we need to allocate new objects from a virtual domain. Conversion (i.e. in this case: renaming) from a virtual type to an already existing type is done by the *definition unit*.

Due to the fact that all domains are currently implemented as SACLIB lists, all data types in the present stage coincide with `Word`. However, we introduce a new data type for each domain, since lists representing elements of different domains have different structures. This semantic difference is reflected in the choice of *different names* for *semantically different types* of lists.

For instance, we interpret a list consisting of only small integers as an exponent list, whereas we interpret a list consisting of an arbitrary integer and a list of only small integers as a monomial with integer coefficient, whose power product is represented as exponent list. Although both types are lists we name a list of the first type `EL` and a list of the second type `Mon` instead of just defining both as `Word`.

Thereby, we can increase

- ◇ the *generality* of the concept of a virtual domain, since we do not rely on the existence of one basic data type and

- ◇ the *readability* of the resulting code.

2.6 Coercion Units

For each implementation of a domain that can be used as subdomain of some other domain we provide a renaming unit and a definition unit. The renaming unit together with the definition unit forms the *coercion unit*.

Including a coercion unit during the preprocessing step carries out all the renaming instructions from the renaming unit and defines the appropriate virtual data type. By conditional inclusion of coercion units we make sure that we include *exactly one coercion unit* for each virtual domain. We will call these coercion units *active* and those, which are not included, *passive*, respectively.

Depending on which unit is active, the virtual operations are mapped to operations in that particular implementation of the subdomain as depicted in Figure 1 and the virtual data type is mapped to (possibly) different data structures.

2.7 The Simulated Package Principle (SPP)

The mapping by renaming of functions as explained in the previous sections can only work properly if each function carries a *tag* that identifies, for which domain of arguments the function can be applied. This mechanism is similar to the package principle in LISP, except that tagging does not happen automatically but must be done by the programmer. We call it therefore the *Simulated Package Principle (SPP)*.

The name of the tag corresponds to the data type up to capitalization. (Syntactically, we append an abbreviation of the domain name as tag, separated by the underscore character "_").

By this mechanism we keep the domain independent of all its subdomains. Summarizing, the advantages of this independence are that

- ◇ we avoid unnecessary duplication of code,
- ◇ it is very easy to change from one implementation of a domain to another one, and
- ◇ the system can very easily be extended by new implementations for already existing domains, because the only thing necessary is to supply a coercion unit for the new implementation and make this active.

The last point is particularly important if the changes are supposed to be made on a very low level in a hierarchy of domains (in our case, for instance, the domain of power products), because without SPP every change would have to be propagated until to the top of the hierarchy.

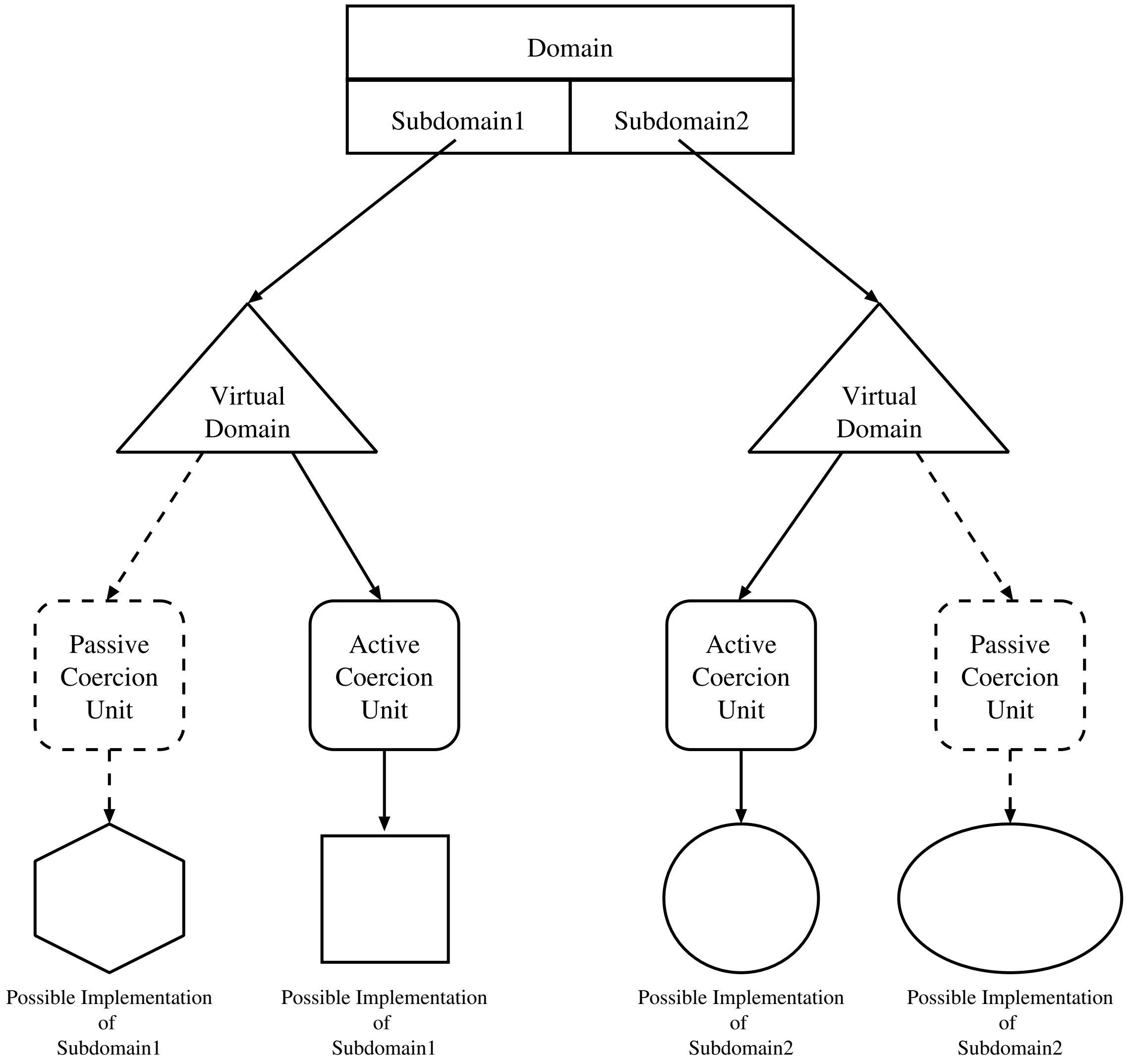

Figure 1: Interface between Domains using Virtual Domains

2.8 The Implementation of Monomials Using SPP

In this section we list the C implementation of the product function for monomials together with all type definitions and renamings that are necessary for SPP. In the Figures 2 to 4 we show how these blocks of code are distributed among source files, header files, and coercion units.

```
#define coefficient( m) FIRST( m)
#define power_product( m) SECOND( m)
#define monomial( c, pp) LIST2( c, pp)

typedef Word Mon;

typedef RN Coef;  /* Coefficient Domain   = Rational Numbers */
typedef EL PP;    /* Power Product Domain = Exponent List    */

#define product_coef product_rn   /* Renaming */
#define product_pp product_el     /* Renaming */

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));
  }

RN product_rn( RN r1, RN r2)
  {
  /* Here goes the implementation of the product
     function for rational numbers */
  }

EL product_el( EL el1, EL el2)
  {
  /* Here goes the implementation of the product
     function for exponent lists */
  }
```

Figure 2 shows how virtual data types and calls to virtual domains are used. The dashed lines indicate, where a function call leads to.

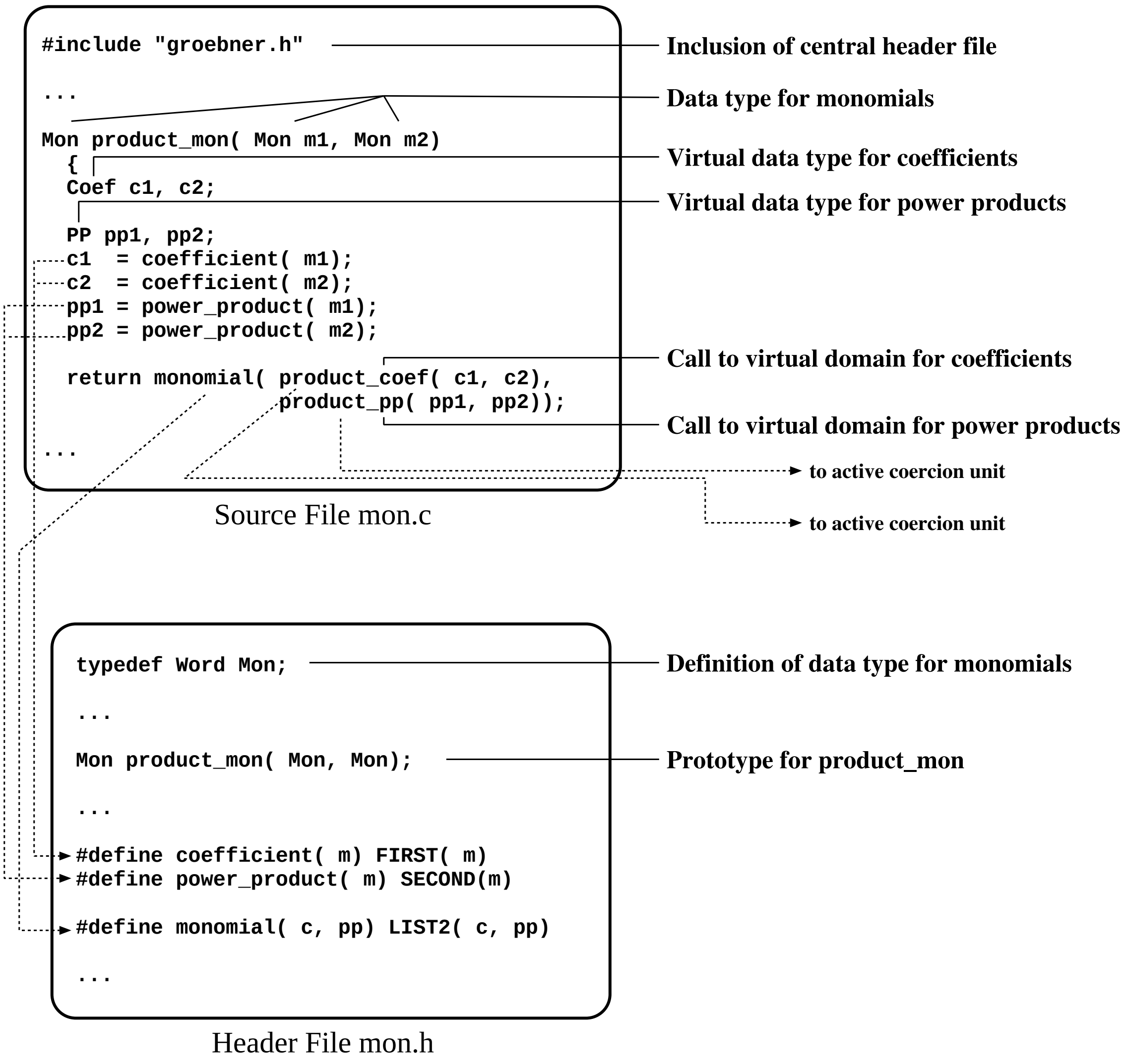

Figure 2: Communication between Domain and its Subdomains

Inclusion of header files

(*) Selection of coefficient domain

Activation of the appopriate coercion unit by conditional inclusion

(*) Selection of power product representation

Activation of the appopriate coercion unit by conditional inclusion

Inclusion of header file

```
...

#include "rn.h"
#include "ff.h"

#define rational_numbers 1
#define finite_field 2

#define coefficient_domain rational_numbers

#if coefficient_domain == rational_numbers
#  include "CU.coef2rn"
#else
#if coefficient_domain == finite_field
#  include "CU.coef2ff"
#endif
#endif

#define exponent_lists 1
#define goedel_exponents 2

#define power_product_rep exponent_lists

#if power_product_rep == exponent_lists
#  include "CU.pp2el"
#else
#if power_product_rep == goedel_exponents
#  include "CU.pp2ge"
#endif
#endif

#include "mon.h"

...
```

Header File groebner.h

Figure 3: The Central Header File `groebner.h`

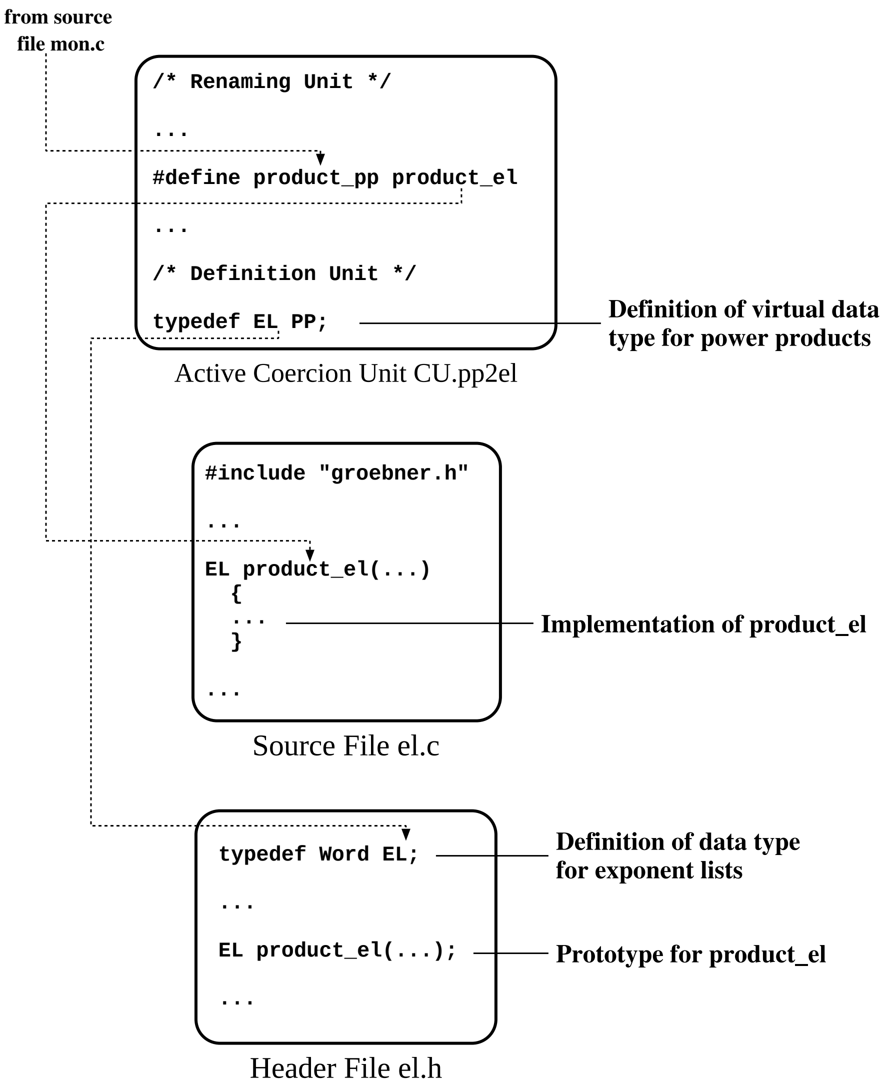

Figure 4: Communication between Coercion Unit and Subdomain

GRÖBNER has one central header file, `groebner.h`, which is included from every source file. In this header file the inclusion of all other header files and the inclusion of the appropriate coercion units is organized. This mechanism is depicted in Figure 3. One can see that for changing the coefficient domain or the power product representation only the lines marked by (*) need to be adjusted.

In fact, in the actual implementation we also distributed the lines marked by (*) together with the following activation of the appropriate coercion unit among several files, the setup-files, in order to enable a more accurate check which part of the setup has actually been changed. When using `make` (or a similar tool) for managing the compilation of the library, we can thus avoid superfluous re-compilation of files that are not influenced by recent changes. (We did not include this into Figure 3 in order not to distract the reader by these details.)

Finally, in Figure 4 the mapping from a virtual domain call to an implemented function and from a virtual type to an existing data type is visualized. We see that both mappings are nothing but renamings done by the coercion unit.

3 Data Types in GRÖBNER

3.1 Basic Data Types

In addition to the built-in C types we use `Word` as the basic data type in GRÖBNER. `Word` is defined in SACLIB, and can either be a small integer or a pointer to a single-linked list. In order to distinguish between small integers and pointers to lists, SACLIB defines a constant `BETA`. A number greater than `BETA` is considered to be a pointer to a list, otherwise it is an integer (an *atom*, following the terminology of SACLIB). For a detailed description of the representation of integers and lists in SACLIB see [Buchberger *et al.*, 1993].

In the present stage of the implementation, all data domains are implemented as lists. Hence, a function taking `Word` as argument, may take an element of *any domain* as argument without resulting in a compile-time error. We can use this to define functions for *arbitrary input* (i.e. any list) as shown in Section 3.3.

On the other hand, the compiler is not able to carry out sophisticated type checking in the parameters given to a function, since all parameters are nothing but lists. It is in the programmer's responsibility to ensure that the lists that are given as parameters to a function have the expected structure.

3.2 Boolean Constants

TYPE NAME: `Boolean`

DEFINITION: `int`
DESCRIPTION: A Boolean constant can have two values, `TRUE` and `FALSE`, where `TRUE` is equivalent to 1 and `FALSE` is equivalent to 0.

3.3 Miscellaneous Functions

3.3.1 Unary Predicate

TYPE NAME: `UP`
DEFINITION: `Boolean (*UP)( Word)`
DESCRIPTION: Pointer to a predicate taking one arbitrary argument.

3.3.2 Binary Predicate

TYPE NAME: `BP`
DEFINITION: `Boolean (*BP)( Word, Word)`
DESCRIPTION: Pointer to a predicate taking two arbitrary arguments.

3.3.3 Unary Function

TYPE NAME: `UF`
DEFINITION: `Word (*UF)( Word)`
DESCRIPTION: Pointer to a function taking one arbitrary argument.

3.3.4 Binary Function

TYPE NAME: `BF`
DEFINITION: `Word (*BF)( Word, Word)`
DESCRIPTION: Pointer to a function taking two arbitrary arguments.

3.4 Lists (iterative implementation)

TYPE NAME: `Word`
DESCRIPTION: See Section 3.1. We implemented some additional operations for SACLIB lists. Elements of the lists are processed iteratively.

3.5 Lists (recursive implementation)

TYPE NAME: `Word`
DESCRIPTION: See Section 3.1. We implemented some additional operations for SACLIB lists. Elements of the lists are processed recursively.

3.6 Integers

Type Name: `Int`
Description: Arbitrarily long integers from SACLIB, see [Buchberger *et al.*, 1993].

3.7 Rational Numbers

Type Name: `RN`
Description: Rational Numbers from SACLIB, see [Buchberger *et al.*, 1993].

3.8 Rational Functions

Type Name: `RF`
Structure: (n, d)
Description: A list consisting of two SACLIB integral polynomials, representing the numerator and the denominator of a rational function. n and d are considered to be relatively prime and the leading coefficient of the denominator is supposed to be positive. We call this representation of a rational function *normalized*. All operations on rational functions are implemented in such a way that the result is normalized again.

Basic polynomial arithmetic and calculation of the greatest common divisor of two polynomials (for cancelling common factors of the numerator and denominator) are used from SACLIB.

3.9 Finite Fields

Type Name: `FF`
Description: The implementation of finite fields is not "complete" in the sense that it does not comprise the general case of finite fields of order p^n but only the case of finite fields of order p, namely Z_p (p prime). We choose $\{0, \ldots, p-1\}$ as representation for Z_p. Thus, elements of a finite field are represented as positive integers and arithmetic is done modulo a prime number, which is stored in the global variable `modulus_`. Modular arithmetic is used from SACLIB.

3.10 Exponent Lists

Type Name: `EL`
Structure: $(e_1, \ldots, e_l)$, $e_i \in \mathbb{N}_0$.

Description: A power product $x_1^{e_1} \cdot \ldots \cdot x_l^{e_l}$ is represented as the list of exponents $(e_1, \ldots, e_l)$. This representation induces an ordering on the variables $x_1 > \ldots > x_l$. Obviously, l is the number of variables.

We do not use arbitrarily long integers for the exponents but only C integers. For usual applications of the Gröbner bases algorithm this should suffice.

3.11 Monomials

Type Name: `Mon`

Structure: (c, pp)

Description: A list consisting of a coefficient c and a power product pp.

3.12 Distributive Polynomials in List Representation

Type Name: `DPL`

Structure: $(m_1, \ldots, m_n)$ for some $n \in \mathbb{N}_0$.

Description: A (possibly empty) list of monomials. The empty list denotes the zero polynomial. The monomials in the list are ordered according to the chosen ordering `is_greater_mon` for monomials. All operations on polynomials preserve this ordering.

The monomials are kept ordered since, in this case, the decomposition of a polynomial into the *leading monomial* and its *remaining polynomial*, which is frequently needed during the Gröbner bases algorithm, can be implemented more efficiently.

We also implemented the quotient of two polynomials, which is to be understood as "exact division without remainder". This is of course only possible if the divisor is invertible. In the case of polynomials over a field, which is the case we concentrated on, the invertible elements are the elements of the ground field (i.e. the constant polynomials). When attempting to divide by a non-constant polynomial, GRÖBNER will detect this and handle the error properly. In the case of polynomials over a domain that is no field, an error might be detected when trying to invert the divisor in the case where it is a constant that is not invertible. Implementing exact division in this fashion considerably simplifies the parser for input of polynomials.

3.13 Polynomials with Cofactors

Type Name: `PC`

Structure: $(p, (c_1, \ldots, c_n))$

Description: "Polynomials with cofactors w.r.t. an ordered set of polynomials". $p, c_1, \ldots, c_n$ are polynomials such that

$$p = \sum_{i=1}^{n} c_i \cdot G_i \tag{1}$$

holds for given (ordered) polynomials $G_1 < \ldots < G_n$.

An operation $\circ$ for polynomials is generalized for polynomials with cofactors in the following way:

$$\begin{aligned}(\bar{p},(\bar{c_1},\ldots,\bar{c_n}))\circ(\hat{p},(\hat{c_1},\ldots,\hat{c_n})) := \ & (p,(c_1,\ldots,c_n)) \text{ s.t.}\\ & p=\bar{p}\circ\hat{p} \text{ and}\\ & (1) \text{ holds for } p, c_1,\ldots,c_n.\end{aligned}$$

Hence, whenever a new polynomial with cofactors is the result of a sequence of operations, Equation (1) still holds for the result. For instance, the result of *the calculation of an S-polynomial and reducing it w.r.t. an ordered set of polynomials* is the normal form of the S-polynomial plus a representation in terms of the initial polynomials.

3.14 Labeled Polynomials

TYPE NAME: `LP`
STRUCTURE: (l, p)
DESCRIPTION: A list consisting of an integer l ("a label") and a polynomial p.

Labeled polynomials are used in advanced versions of the Gröbner bases algorithm with application of the criteria to detect zero-reductions, where one frequently has to check for equality of polynomials in the basis. Each polynomial gets a (different) label *before* it is adjoined to the basis. This allows the equality-check between basis polynomials to be performed by *just checking equality of the integer labels.*

3.15 Lists of Polynomials

TYPE NAME: `PL`
STRUCTURE: $(p_1,\ldots,p_l)$ for some $l \in \mathbb{N}_0$.
DESCRIPTION: A (possibly empty) list of polynomials p_i. A "polynomial" in this case can either be an element of the virtual domain of polynomials (in the sense of the description given in Section 4.4, independent from its structure and representation of "pure polynomials", see Section 4.4.1 and Section 4.4.2) or a labeled polynomial (see Section 3.14). Functions for labeled polynomial lists usually carry the tag `lpl`.

Lists of polynomials are always ordered w.r.t. ascending leading monomials.

3.16 Polynomial Pairs

TYPE NAME: `Pa`
STRUCTURE: (p_1, p_2)
DESCRIPTION: A list consisting of two polynomials p_1 and p_2.

3.17 Sets of Polynomial Pairs

TYPE NAME: `PaS`

STRUCTURE: $(pa_1, \ldots, pa_s)$ for some $s \in \mathbb{N}_0$.

DESCRIPTION: A (possibly empty) list of polynomial pairs pa_i. For the advanced versions of the Gröbner bases algorithm the pair set is kept in ascending order w.r.t. the chosen ordering `is_pa_greater`. The first element in the pair set is always the next pair that is considered for S-polynomial computation, hence, the ordering on the pair set determines the strategy for choosing S-polynomials .

Up to now, the only strategy being tested is "take the pair with smallest *least common multiple* of the leading monomials", however, the interface for choosing a different strategy is implemented. A possible (i.e. reasonable) strategy to be tested is "take the pair with smallest *leading monomial* of the S-polynomial".

For the variant of the Gröbner bases algorithm, in which pairs for which a criterion can be applied are deleted immediately, we also have to fix an ordering on the pairs with equal "lcm". This ordering is called `is_labeled_pa_greater` and is a refinement of `is_pa_greater` in the way that pairs with equal "lcm" are ordered by a lexical ordering on the pairs of labels (of the polynomials in the pair).

3.18 Syzygies

TYPE NAME: `Syz`

STRUCTURE: $(g_1, \ldots, g_n)$ for given $n \in \mathbb{N}$, each g_i is of type `PPoly`.

DESCRIPTION: For given $G_1, \ldots, G_n$ the g_i's are such that $\sum_{i=1}^{n} g_i \cdot G_i = 0$, they form a *syzygy* w.r.t. $G_1, \ldots, G_n$.

An interesting algorithm implemented for syzygies is the *extension procedure*, which extends a syzygy w.r.t. the leading monomials of the G_i's to a syzygy w.r.t. the G_i's. See [Möller, 1985] and [Windsteiger, 1992] for details about this extension of syzygies.

4 Virtual Domains and Their Implementations

In this section, we list all virtual domains available in GRÖBNER together with the coercion units provided for each virtual domain. In order to extend the library by a new implementation of an already existing virtual domain, inspect an existing coercion unit to figure out, which operations are expected to exist for the respective domain and provide a new coercion unit.

4.1 Lists

TYPE NAME: `Word`
SETUP IN: `GB.list.setup`
DESCRIPTION: Additional list operations. There is the possibility to choose between an iterative and a recursive implementation. The first approach often suffers from the inevitable inversion of the resulting list after its iterative construction, while on the contrary the latter bears the danger of an intermediate stack overflow. According to our observations the recursive implementation seems to be more efficient in most cases.

COERCION UNIT	LIST OPERATIONS
`CU.list2iter`	Iterative (Section 3.4)
`CU.list2rec`	Recursive (Section 3.5)

Table 1: Available Coercion Units for Lists

4.2 Coefficients

TYPE NAME: `Coef`
SETUP IN: `GB.coef.setup`
DESCRIPTION: Domain of coefficients in a monomial.

COERCION UNIT	COEFFICIENT DOMAIN	DATA TYPE
`CU.coef2rat`	Rational Numbers (Section 3.7)	`RN`
`CU.coef2rf`	Rational Functions (Section 3.8)	`RF`
`CU.coef2ff`	Finite Fields (Section 3.9)	`FF`

Table 2: Available Coercion Units for Coefficients

4.3 Power Products

TYPE NAME: `PP`
SETUP IN: `GB.pp.setup`
DESCRIPTION: Representation of power products in a monomial.

4.4 The Domain of Polynomials

Polynomials receive our special attention. We formulate the Gröbner bases algorithm for "sets of polynomials", hence, the term "polynomial" must be kept rather general: it may be a polynomial in the strict mathematical sense (a "pure polynomial") but it may as well be a pure polynomial with some additional information attached, which allows further interpretation of the result. This additional information could for instance be the cofactors with respect to the initial polynomials. Computing a Gröbner basis for a set of "polynomials with cofactors" (see Section 3.13) yields a Gröbner basis plus a representation of each basis polynomial in terms of the initial polynomials.

Independent of this *structure*, we can choose among different *representations for "pure polynomials"*, for instance the distributive list representation (see Section 3.12) or the various representations provided by SACLIB. Whatever structure is used for "polynomials", the representation of a "pure polynomial" will occur as a virtual subdomain.

Thus, the domain "polynomials" as a virtual subdomain of "sets of polynomials" should have the 2-level structure as shown in Figure 5.

4.4.1 Structure of Polynomials

TYPE NAME: `Poly`
SETUP IN: `GB.poly.setup`
DESCRIPTION: A polynomial with arbitrary additional information, which can be utilized in the Gröbner bases algorithm. An example of such additional information attached to a polynomial are the "cofactors" with respect to a set of polynomials (see Section 3.13).

4.4.2 Representation of Pure Polynomials

TYPE NAME: `PPoly`
SETUP IN: `GB.ppoly.setup`
DESCRIPTION: Representation of pure polynomials.

4.5 Polynomial Sets

TYPE NAME: `PS`
SETUP IN: `GB.ps.setup`
DESCRIPTION: A set of polynomials of type `Poly` (see Section 4.4.1) or labeled polynomials (see Section 3.14). Functions for labeled polynomial lists usually carry the tag `lps`.

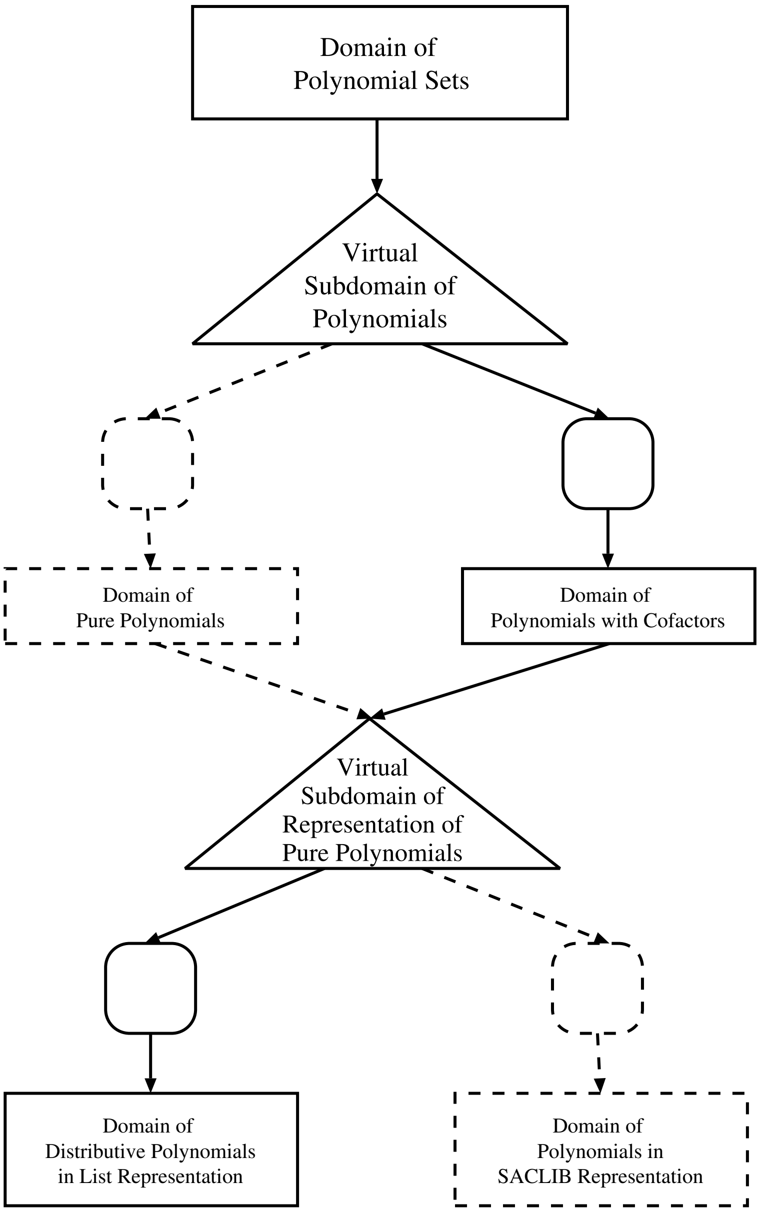

Figure 5: 2-level Structure of Polynomials

4.6 Domain Hierarchy

Figure 6 illustrates the hierarchy of domains in GRÖBNER. Triangles are used for virtual domains whereas rectangles denote actually implemented domains. The small triangles glued to the bottom of each "domain" should represent the subdomains of the respective domain.

5 Further Adjustments

5.1 Term Ordering

GRÖBNER offers the possibility to choose among several term orderings. This ordering determines the ordering of monomials in a polynomial. The result of a Gröbner basis calculation as well as the computing time heavily depends on the chosen ordering of monomials.

In the following, $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ denote exponent vectors, which represent power products (terms) in n variables. We provide the following term orderings:

Definition 1 (Purely Lexical Ordering)

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

Definition 2 (Total Degree Lexical Ordering)

$$u <_{\mathrm{tdeg}} v :\Leftrightarrow (\sum_{i=1}^{n} u_i, u_1, \ldots, u_{n-1}) <_{\mathrm{lex}} (\sum_{i=1}^{n} v_i, v_1, \ldots, v_{n-1})$$

Definition 3 (Total Degree Inverse Lexical Ordering)

$$u <_{\mathrm{tdilex}} v :\Leftrightarrow \begin{array}{l} \sum_{i=1}^{n} u_i < \sum_{i=1}^{n} v_i \text{ or} \\ \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} v_i \text{ and } (v_n, \ldots, v_1) <_{\mathrm{lex}} (u_n, \ldots, u_1) \end{array}$$

Definition 4 (Matrix Ordering)
Let s be an integer and U an $(n \times s)$ -matrix.

$$u <_U v :\Leftrightarrow u \cdot U <_{\mathrm{lex}} v \cdot U$$

Details about the properties of the matrix U, which guarantee that this yields an admissible ordering, are given in [Robbiano, 1985] or [Windsteiger, 1992].

Ordering of monomials is done by the predicate `is_greater_mon`, which is mapped to one of those predicates in the file `GB.to.setup`.

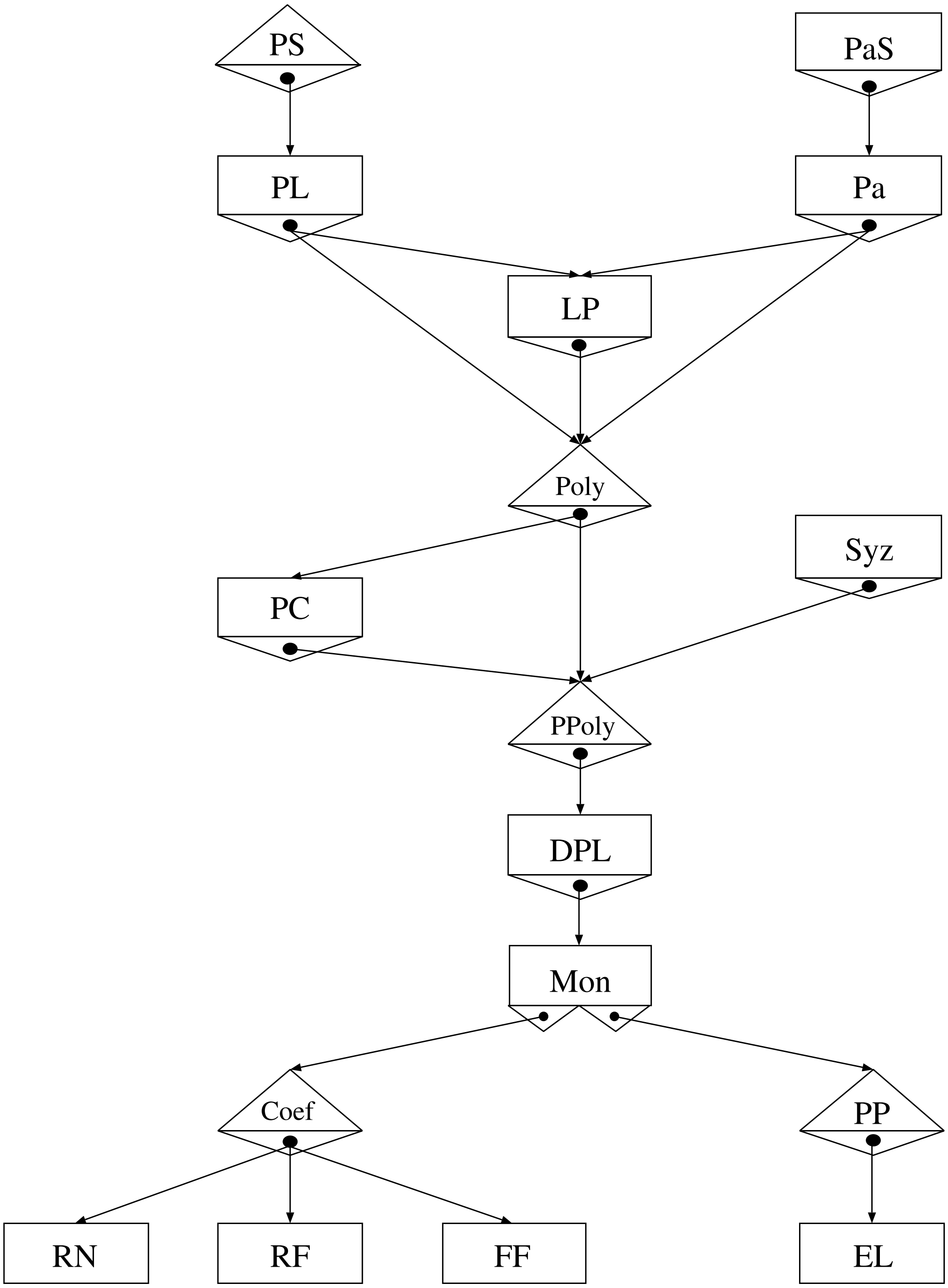

Figure 6: Hierarchy of Domains

5.2 Pair Ordering

The predicate `is_pa_greater` (in the case, where the variant of the Gröbner bases algorithm is used, where criteria are applied when the pair set is created or updated, we use the predicate `is_labeled_pa_greater`) determines the *basic strategy* for the ordering of polynomial pairs in a set of polynomial pairs (see also Section 3.17). Possible strategies are

1. to order pairs in a set ascending w.r.t. the least common multiple of the leading monomials of the two polynomials or
2. to order pairs in a set ascending w.r.t. the leading monomial of the S-polynomial of the two polynomials or
3. ...

(In the present stage, only strategy 1 is implemented.)

Whatever strategy one might choose, some ordering on monomials will also be required in the ordering of the pair set. Since this ordering need not be equal to the ordering chosen for monomials in a polynomial, we have an additional predicate `is_greater_mon_pair_order`, for which we have again the choices listed in Section 5.1.

A possible (i.e. plausible) setting is to order the pair set by ascending *total degree* of the least common multiple of the leading monomials when computing a lexical Gröbner basis. This strategy has turned out to be in some examples significantly faster than having lexical ordering for the least common multiple of the leading monomials as well (see [Windsteiger, 1992], Section on timing statistics). A similar strategy is also described in [Giovini *et al.*, 1991].

Choosing an appropriate setting is done in the file `GB.po.setup`.

5.3 Normal Form Algorithm

When reducing a polynomial p w.r.t. a set of polynomials PS we can choose between *partial reduction* and *complete reduction*.

Partial reduction can be sloppily described as follows: the leading monomial of p is deleted by subtraction of an "appropriate multiple" of a polynomial in PS. This process is iterated as long as there exists a polynomial q in PS s.t. the leading monomial of p is a multiple of the leading monomial of q. Partial reduction yields a polynomial, called the *partial normal form of p w.r.t. PS*, whose leading monomial is *not a multiple* of the leading monomial of any of the polynomials in PS.

Complete reduction *iterates* partial reduction in the sense that when a partial normal form is found, the leading monomial is kept and the remaining polynomial is partially reduced again. This yields a polynomial, the *complete normal form of p w.r.t.*

PS, which does not contain monomials that are multiples of the leading monomial of any of the polynomials in *PS*.

The function that computes the normal form of a polynomial w.r.t. a set of polynomials is called `normal_form_poly` and the desired normal form algorithm is selected in the file `GB.nf.setup`.

6 Global Variables

In Table 7, we describe all global variables used in GRÖBNER.

7 Naming Conventions

In this section, we will sketch the naming conventions we sticked to during the implementation of GRÖBNER. This is very important both for *reading the code* and for *extending the system consistently* by own routines.

These conventions are to be considered as guidelines rather than laws. On occasions, where a deviation from those rules improves the readability of the code, exceptions are permitted.

7.1 Capitalization and Separation of Names

As a general rule, all names in GRÖBNER contain *only lowercase letters*. The only exception of this rule are abbreviations for data types (see Section 7.2) and proper names (e.g. S-polynomial). If a name consists of several words, these words are separated by the underscore character "_".

Example: `complete_normal_form`

7.2 Types and Tags

The name for data types is usually a *sequence of uppercase letters*, where each letter is the initial letter of a word in the description of the data type.

Example: The data type of *rational numbers* is by this rule abbreviated to `RN`.

If a reasonable description of the data type can be given by *only one word* then the name of the data type is a capitalized abbreviation of this word.

Example: The data type representing *monomials* is called `Mon`.

According to our "Simulated Package Principle" (see Section 2) each function name has to carry a tag that identifies for which data domain the function can be applied. This tag corresponds to the name of the data type but is *lowercase*.

Example: The tags for functions for rational numbers and monomials are `rn` and `mon` respectively.

7.3 Functions

The philosophy behind the choice of function names is that the *name of a function should describe the result of the function as accurate as possible.* This implies, that names for functions are in general *not abbreviated.*

We distinguish between two types of functions:

1. functions that return a value and
2. functions that return `void`, i.e. no value (such functions are frequently called "procedures").

We usually choose the *noun(s)*, which describe(s) the result of the function as name of a function of type 1 (on rare occasions we also use a second participle) whereas for procedures we use the *verb* describing the action of the procedure.

Examples: We call the function that multiplies two monomials `product_mon` (not `multiply_mon`), the function that interreduces a polynomial set `inter_reduced_ps`, and the procedure that prints a monomial `print_mon`.

7.4 Predicates

For predicates the same rules as for functions apply. In addition, most of the predicates are prefixed by "is".

Example: The predicate comparing two monomials by the total degree lexical ordering (see Section 5.1, Definition 2) is accordingly called `is_total_degree_lexical_greater_mon`.

7.5 Variables

In most cases, variable names are chosen *as short as possible.* Unless the content of the variable already suggests an appropriate name, we just use the name of the data type in lowercase letters as the name of a variable of this type. When there are more such variables of one type we enumerate them starting with 0, where 0 is usually used for a variable carrying the return value.

Names for *global variables* are marked by a trailing underscore (_). Compared to local variables, global variables have rather long names, which should help to avoid unexpected name conflicts.

Examples: The input parameters to the product function for exponent lists are called `el1` and `el2`, the result is called `el0`, for examples of names for global variables see Table 7.

Coercion Unit	Power Product Representation	Data Type
`CU.pp2el`	Exponent Lists (Section 3.10)	`EL`

Table 3: Available Coercion Units for Power Products

Coercion Unit	Polynomial Structure	Data Type
`CU.poly2ppoly` `CU.poly2pc`	Pure Polynomial Polynomial with Cofactors (Section 3.13)	`PPoly` `PC`

Table 4: Available Coercion Units for Polynomial Structure

Coercion Unit	Representation of Pure Polynomials	Data Type
`CU.ppoly2dpl`	Distributive Polynomials as Lists (Section 3.12)	`DPL`

Table 5: Available Coercion Units for Representation of Pure Polynomials

Coercion Unit	Representation of Polynomial Sets	Data Type
`CU.ps2pl`	Polynomial Lists (Section 3.15)	`PL`

Table 6: Available Coercion Units for Polynomial Sets

Variable	Type	Description
`error_in_groebner_`	`Boolean`	An error flag, which is set to `TRUE` as soon as there occurs an error in any calculation. Attempting to divide a polynomial by the zero-polynomial for instance causes this flag to be set.
`modulus_`	`int`	When using finite fields as coefficient domain, we represent them as $\mathbb{Z}_p$ and this variable contains the value of p. When this variable is set, it is *not* checked whether p is a prime number.
`number_of_field_variables_`	`int`	When using rational functions as coefficient domain, we use SACLIB-polynomials for representing the numerator and the denominator. The arithmetic operations for polynomials in SACLIB take the number of variables as an argument. Hence, we keep it stored in this global variable.
`order_matrix_`	`Word`	When power products are ordered by a matrix (see Section 5.1), this variable contains the matrix.
`label_`	`int`	This variable contains the label that will be assigned to the next polynomial to be labeled. Its initial value is 1.

Table 7: Global Variables

Part II
Reference Manual

This part of the manual is dedicated to a description of each constituent of GRÖBNER. The items are listed lexicographically, where uppercase letters come before lowercase letters. The items are divided into several types, containing

- ◇ functions,
- ◇ predicates,
- ◇ virtual functions,
- ◇ virtual predicates,
- ◇ macros,
- ◇ constants, and
- ◇ global variables.

We particularly want to emphasize the subtle difference between functions and macros taking arguments for further development of the system. Some functions in GRÖBNER have the possibility to take a *pointer to a function* as an argument. Be aware that you cannot pass the name of a function, which is actually implemented as a macro taking arguments, on such an occasion. This fact might be obvious for experienced C programmers, but it might be worth mentioned *explicitly* for those not so familiar with the C language, who should also be able to use and extend GRÖBNER.

`DIVISION`

TYPE: Constant
DEFINED IN: `error.h`

▷ Error code indicating that a polynomial was attempted to be divided by a non-constant polynomial.

`EMPTY`

TYPE: Constant
DEFINED IN: `groebner_globals.h`

▷ Renamed to `BETA`, the representation of the empty list in SACLIB.

`GB_criteria( PS F)`

TYPE: Function
DEFINED IN: `gb_criteria.c.trace`

▷ Returns the reduced Gröbner basis of the ideal generated by the polynomials in the set `F`.
The algorithm used applies the criteria for a-priori detection of zero-reductions immediately before computation of the S-polynomial and keeps the pair set ordered s.t. "lower pairs" (w.r.t. the ordering chosen in `gb.po.setup`) are considered for S-polynomial computation first.

SEE ALSO: Section 5.1, Section 5.2, Section 5.3, `is_greater_mon`, `is_pa_greater`, and `is_greater_mon_pair_order`.

`GB_crude( PS F)`

TYPE: Function

DEFINED IN: `gb_crude.c.trace`

▷ Returns the Gröbner basis of the ideal generated by the polynomials in the set `F`.
The algorithm is not optimized in any respect, this means *no criteria* are applied, the pair set is *unordered*, and no interreduction is made.

`GB_reduce_all( PS F)`

TYPE: Function

DEFINED IN: `gb_reduce_all.c.trace`

▷ Returns the reduced Gröbner basis of the ideal generated by the polynomials in the set `F`.
The algorithm used is the improved version sketched in [Buchberger, 1985], where criteria are applied and, whenever a reduction yields a non-zero polynomial, interreduction is performed *before* this polynomial is adjoined to the basis. Like in `GB_criteria`, pairs are kept ordered.

SEE ALSO: Section 5.1, Section 5.2, Section 5.3, `is_greater_mon`, `is_pa_greater`, `is_greater_mon_pair_order`, `all_reduced`.

`GB_small_pair_set( PS F)`

TYPE: Function

DEFINED IN: `gb_small_ps.c.trace`

▷ Returns the reduced Gröbner basis of the ideal generated by the polynomials in the set `F`.
The algorithm applies the criteria for a-priori detection of zero-reductions already when the pair set is created and cancels all unnecessary pairs immediately. This keeps the pair set in each stage as small as possible. The pair set is totally ordered by `is_labeled_pa_greater`. This refinement of the ordering for pairs is required for correct application of the criteria in this version of the algorithm.

SEE ALSO: Section 3.17, Section 5.1, Section 5.2, Section 5.3, `is_greater_mon`, `is_labeled_pa_greater`, and `is_greater_mon_pair_order`.

GROEBNER_VERSION_NUMBER

TYPE: Constant

DEFINED IN: `groebner.h`

▷ Contains a string representing the actual version number of the library.

NOGB

TYPE: Constant

DEFINED IN: `error.h`

▷ Error code indicating that a set of polynomials is no Gröbner basis where a Gröbner basis was expected.

REST(Word w)

TYPE: Macro

DEFINED IN: `groebner_globals.h`

▷ Renaming of SACLIB function `RED`, which takes a list `w` as input and returns a list containing all but the first element of `w`. Same as `cdr` in LISP.

SACLIB_MEMORY

TYPE: Constant

DEFINED IN: `machine.h`

▷ Determines the number of cells that SACLIB makes available on start-up.

S_polynomial_lp(LP lp1, LP lp2)

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the S-polynomial of the two labeled polynomials `lp1` and `lp2`. The result is *not labeled*.

SEE ALSO: `S_polynomial_poly`.

S_polynomial_poly(Poly p1, Poly p2)

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the S-polynomial of the two polynomials `p1` and `p2`.

ZERODIVISION

TYPE: Constant

DEFINED IN: `error.h`

▷ Error code indicating that there was an attempted division by zero.

`absolute_ff( FF ff)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Returns the absolute value of `ff`. Due to the representation of finite fields, this function simplifies to the identity function on $\mathbb{Z}_p$.

SEE ALSO: Section 3.9.

`absolute_rn( RN rn)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns the absolute value of the rational number `rn`. Is renamed to the SACLIB function `RNABS`.

`all_reduced( PS reducibles, PS reducers, PS basis, PaS pairs)`

TYPE: Function

DEFINED IN: `gb_reduce_all.c.trace`

▷ Performs interreduction during the Gröbner bases algorithm. `reducibles` contains polynomials, whose head term is reducible by one of the polynomials in `reducers`, which contains candidates to be adjoined to the basis. `basis` and `pairs` contain the current basis and pair set respectively.

An element from `reducibles` is reduced by polynomials from `reducers` and the basis. If this reduction yields a non-zero polynomial p, then p is adjoined to `reducers` and all polynomials in `reducers` and in the basis, whose head term can be reduced by p, change to `reducibles`. This can lead to a cascade of reductions until `reducibles` is empty. When this state is reached, the "reducers" are adjoined to the basis and the pair set is updated.

A list containing the basis and the updated pair set is returned.

SEE ALSO: [Buchberger, 1985].

`applied( UF uf, Word l)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Suppose `l` is of the form $(l_1, \ldots, l_n)$. Then the return value is the list $(\texttt{uf}(l_1), \ldots, \texttt{uf}(l_n))$.

SEE ALSO: `applied_iter`, `applied_rec`.

`applied_iter( UF uf, Word l)`

TYPE: Function
DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `applied`.

SEE ALSO: `applied`.

`applied_rec( UF uf, Word l)`

TYPE: Function
DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `applied`.

SEE ALSO: `applied`.

`basis_module_syz( PS gb)`

TYPE: Function
DEFINED IN: `syz.c`

▷ Computes a basis of the module of syzygies of the vector `gb`, i.e. a basis of the module

$$\{(p_1, \ldots, p_n) | \sum_{i=1}^{n} p_i \cdot gb_i = 0\}, \text{ for } n = \text{length}(gb).$$

`gb` must be a Gröbner basis, otherwise an error is reported.

`canceled_multiples_pl( PL pl)`

TYPE: Function
DEFINED IN: `pl.c`

▷ Implementation of virtual function `canceled_multiples_ps`.

SEE ALSO: `canceled_multiples_ps`.

`canceled_multiples_ps( PS ps)`

TYPE: Virtual Function
DEFINED IN: `CU.ps2pl`

▷ All polynomials in the set `ps`, whose leading monomial is a multiple of one of the remaining polynomials are deleted from the set.

SEE ALSO: `inter_reduced_ps`.

`chain_criterion( Pa pa, PS basis, Pa pair)`

TYPE: Predicate
DEFINED IN: `pa.c.trace`

▹ The version of the chain criterion (i.e. Criterion1 in [Buchberger, 1985]), which is used in the variant of the Gröbner bases algorithm, where criteria are applied immediately when the pair set is updated. It returns `TRUE` if there is a polynomial p in `basis` s.t. the triangle criterion holds for `pa` and p. The parameter `pair` is passed for "technical reasons", it would lead too far to explain in all detail, why this is required in this case.

SEE ALSO: `triangle_criterion`, `thinned_by_chain_criterion`.

`coefficient( Mon m)`

TYPE: Macro
DEFINED IN: `mon.h`

▹ Selects the coefficient of the monomial `m`.

`coefficient_domain`

TYPE: Constant
DEFINED IN: `GB.coef.setup`

▹ Determines the domain of coefficients of polynomials.

`cofactors_pc( PC pc)`

TYPE: Macro
DEFINED IN: `pc.h`

▹ Selects the vector of cofactors of the polynomial with cofactors `pc`.

SEE ALSO: Section 3.13

`combine_dpl( DPL dpl1, DPL dpl2)`

TYPE: Macro
DEFINED IN: `dpl.h`

▹ Reverses the monomials in `dpl1`. This function was introduced as an analogon to `combine_pc`.

SEE ALSO: Implementation of `complete_normal_form_poly`.

`combine_pc( PC pc1, PC pc2)`

TYPE: Function
DEFINED IN: `pc.c`

▹ Returns a polynomial with cofactors combined from `pc1` and `pc2` by reverting the order of monomials of `pc1` and taking the cofactors of `pc2`. It is needed to plug together the appropriate parts when computing the complete normal form of a polynomial with cofactors.

SEE ALSO: Implementation of `complete_normal_form_poly`.

---

combine_ppoly(PPoly p1, PPoly p2)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Virtual Function for `combine_dpl`.

SEE ALSO: `combine_dpl`.

---

combine_poly(Poly p1, Poly p2)

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`.

▷ Virtual Function for `combine_pc` and `combine_ppoly`.

SEE ALSO: `combine_pc`, `combine_ppoly`.

---

complete_normal_form_poly(Poly p, PS ps)

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the complete normal form of `p` w.r.t. `ps`.

SEE ALSO: Section 5.3,
`partial_normal_form_poly`.

---

completely_reduced_lpl(PL pl)

TYPE: Function

DEFINED IN: `pl.h`

▷ Implementation of virtual function `completely_reduced_lps`.

SEE ALSO: `completely_reduced_lps`.

---

completely_reduced_lps(PS ps)

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Each polynomial in the set of labeled polynomials `ps` is reduced w.r.t. the remaining polynomials. In the algorithm it is assumed that none of the polynomials has a leading monomial that is a multiple of one of the remaining leading monomials.

SEE ALSO: `inter_reduced_ps`, `canceled_multiples_ps`.

---

completely_reduced_pl(PL pl)

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `completely_reduced_ps`.

SEE ALSO: `completely_reduced_ps`.

`completely_reduced_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Analogon to `completely_reduced_lps` a set of unlabeled polynomials `ps`.

SEE ALSO: `completely_reduced_lps`.

`component_wise( BF bf, Word l1, Word l2)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Suppose `l1`$= (l_{11}, \ldots, l_{1n})$ and `l2`$= (l_{21}, \ldots, l_{2n})$. Then the return value is $(\texttt{bf}(l_{11}, l_{21}), \ldots, \texttt{bf}(l_{1n}, l_{2n}))$. `l1` and `l2` are assumed to have equal length. If this is not the case, the longer list is truncated to the length of the shorter list.

SEE ALSO: `component_wise_iter`, `component_wise_rec`.

`component_wise_iter( BF bf, Word l1, Word l2)`

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `component_wise`.

SEE ALSO: `component_wise`.

`component_wise_rec( BF bf, Word l1, Word l2)`

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `component_wise`.

SEE ALSO: `component_wise`.

`contains_pa( Pa pa, PS ps)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Returns `TRUE` if `pa` contains one of the polynomials in `ps`.

`contains_pas( PaS pas, Pa pa)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Returns `TRUE` if `pas` contains the pair `pa`.

`criteria( LP lp1, LP lp2, PS G, PaS B)`

TYPE: Predicate

DEFINED IN: `reductions.c.trace`

▷ Returns `TRUE` if one of the criteria for detecting reduction to zero of the S-polynomial of `lp1` and `lp2` w.r.t. `G` can be successfully applied.

SEE ALSO: [Buchberger, 1985]
`product_criterion`, `explicit_chain_criterion`.

`denominator_rf( RF rf)`

TYPE: Macro

DEFINED IN: `rf.h`

▷ Selects the denominator of the rational function `rf`.

`denominator_rn( RN rn)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Selects the denominator of the rational number `rn`.

`difference_coef( Coef c1, Coef c2)`

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the difference `c1 - c2`, where `c1` and `c2` are elements of the chosen domain of coefficients.

`difference_dpl( DPL p1, DPL p2)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Returns the difference `p1 - p2`, where `p1` and `p2` are distributive polynomials in list representation.

SEE ALSO: Section 3.12

`difference_ff( FF ff1, FF ff2)`

TYPE: Macro
DEFINED IN: `ff.h`

▷ Returns the difference `ff1 - ff2`, where `ff1` and `ff2` are elements of a finite field. It uses the SACLIB function `MIDIF`.

SEE ALSO: Section 3.9, `modulus_`

`difference_mon( Mon m1, Mon m2)`

TYPE: Function
DEFINED IN: `mon.c`

▷ Returns the difference `m1 - m2`, where `m1` and `m2` are monomials. We assume that both monomials have equal power products, otherwise the result would not be a monomial but a polynomial. *No error* is reported in this case but the result is wrong.

`difference_pc( PC pc1, PC pc2)`

TYPE: Function
DEFINED IN: `pc.c`

▷ Returns the difference `pc1 - pc2`, where `pc1` and `pc2` are polynomials with cofactors. The list of cofactors is updated appropriately.

SEE ALSO: Section 3.13

`difference_ppoly( PPoly p1, PPoly p2)`

TYPE: Virtual Function
DEFINED IN: `CU.ppoly2dpl`

▷ Returns the difference of the two polynomials `p1` and `p2`.

SEE ALSO: Section 4.4.2.

`difference_poly( Poly p1, Poly p2)`

TYPE: Virtual Function
DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the difference of the two polynomials `p1` and `p2`.

SEE ALSO: Section 4.4.1.

`difference_rf( RF rf1, RF rf2)`

TYPE: Function
DEFINED IN: `rf.c`

▷ Returns the difference `rf1 - rf2`, where `rf1` and `rf2` are rational functions.

`difference_rn( RN r1, RN r2)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns the difference `r1 - r2`, where `r1` and `r2` are rational numbers. It is renamed to the SACLIB function `RNDIF`.

`difference_syz( Syz s1, Syz s2)`

TYPE: Function

DEFINED IN: `syz.c`

▷ Returns the difference `s1 - s2`, where `s1` and `s2` are syzygies w.r.t. to a fixed vector of polynomials.

SEE ALSO: Section 3.18

`empty_pl`

TYPE: Constant

DEFINED IN: `pl.h`

▷ Empty list of polynomials. Renamed to `EMPTY`.

SEE ALSO: `EMPTY`

`empty_ps`

TYPE: Constant

DEFINED IN: `CU.ps2pl`

▷ Empty set of polynomials.

`error_in_groebner_`

TYPE: Global Variable

DEFINED IN: `error.h`

▷ Error flag raised whenever an error is detected during computation.

SEE ALSO: Section 6.

`expanded( Word element, Word list, BP comp)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ `element` is inserted into `list`, which is in ascending order w.r.t. the binary predicate `comp`.

`expanded_iter( Word element, Word list, BP comp)`

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `expanded`.

SEE ALSO: `expanded`.

`expanded_rec( Word element, Word list, BP comp)`

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `expanded`.

SEE ALSO: `expanded`.

`explicit_chain_criterion( LP lp1, LP lp2, PS G, PaS B)`

TYPE: Predicate

DEFINED IN: `reductions.c.trace`

▷ The version of the chain criterion (i.e. Criterion1 in [Buchberger, 1985]), which is used in the variant of the Gröbner bases algorithm, where criteria are applied before S-polynomial computation. It returns `TRUE` if there is a polynomial p in `G` s.t.

- ◇ the least common multiple of the leading monomials of `lp1` and `lp2` is a multiple of the leading monomial of p,
- ◇ $p \neq lp1$ and $p \neq lp2$, and
- ◇ the pairs $(p, lp1)$ and $(p, lp2)$ are *not contained* in `B`, which means that they have already been considered for S-polynomial computation.

SEE ALSO: [Buchberger, 1985], `criteria`.

`extension_syz( Syz s, PS gb)`

TYPE: Function

DEFINED IN: `syz.c`

▷ `s` is a syzygy of the leading monomials of `gb` and `gb` is a Gröbner basis. The return value is an extension of `s` w.r.t. `gb`, i.e. a syzygy w.r.t. `gb`. See [Windsteiger, 1992] for the detailed definition of an extension of a syzygy.

SEE ALSO: [Möller, 1985], [Windsteiger, 1992].

`extract_info4groebner(int argc, char *argv[])`

TYPE: Function

DEFINED IN: `init.c.trace`

▷ Extracts information contained in the command-line arguments `argv` that are passed when the `main` program is called. Information relevant for GRÖBNER concerns global SACLIB parameters, which are set according to the values of `argv`. Default values are used for parameters that are not explicitely specified.

`argv` is an array of `argc` strings that contain the command-line arguments, one per string, following the convention of C language.

SEE ALSO: [Buchberger *et al.*, 1993].

`first_pair_pas( PaS p)`

TYPE: Macro

DEFINED IN: `pa.h`

▷ Returns the first pair in `p`. Renamed to SACLIB function `FIRST`.

`first_polynomial_pa( Pa p)`

TYPE: Macro

DEFINED IN: `pa.h`

▷ Returns the first polynomial in the pair `p`. Renamed to SACLIB function `FIRST`.

`head_reduced_lp( LP lp, PS ps)`

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Applies `head_reduced_poly` to the polynomial in `lp` and `ps` and leaves the label unchanged.

SEE ALSO: Section 5.3, `head_reduced_poly`.

`head_reduced_poly( Poly p, PS ps)`

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the complete normal form of the polynomial `p` w.r.t. the set of polynomials `ps` in the case it is known that the leading monomial of `p` cannot be further reduced. This implies that the leading monomial remains unchanged and the remaining polynomial is completely reduced.

SEE ALSO: Section 5.3.

`init_groebner_globals()`

TYPE: Function

DEFINED IN: `init.c.trace`

▷ Initialization of global variables needed in GRÖBNER.

---

initial_pairs(PS ps)

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ Returns the set of all pairs that can be built from polynomials in `ps`. The resulting pair set is ordered w.r.t. `is_pa_greater`.

SEE ALSO: Section 5.2,
`updated_pairs`, `is_pa_greater`.

---

initial_pairs_with_criteria(PS ps)

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ Returns the set of all pairs that can be built from polynomials in `ps`, for which zero-reduction of the S-polynomial cannot be predicted by any of the *criteria*. The resulting pair set is ordered w.r.t. `is_labeled_pa_greater`.

SEE ALSO: Section 5.2,
`initial_pairs_with_product_criterion`,
`thinned_by_chain_criterion`.

---

initial_pairs_with_product_criterion(PS ps)

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ Returns the set of all pairs that can be built from polynomials in `ps`, for which zero-reduction of the S-polynomial cannot be predicted by the *product criterion*. The resulting pair set is ordered w.r.t. `is_labeled_pa_greater`.

SEE ALSO: Section 5.2,
`updated_pairs_with_product_criterion`.

---

initialized_dpl(DPL p, int pos, int total)

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `initialized_ppoly`. Returns `p`. This function was introduced as an analogon to `initialized_pc` where the parameters `pos` and `total` are required.

SEE ALSO: `initialized_pc`, `initialized_ppoly`.

---

initialized_pc(PPoly p, int pos, int total)

TYPE: Function

DEFINED IN: `pc.c`

▷ Initialization of a polynomial with cofactors before Gröbner basis computation. `p` is assumed to be at position `pos` in a set of `total` polynomials. The cofactors refer to this set of polynomials, hence, the appropriate list of cofactors has the unit polynomial at position `pos` and zero at all other positions.

SEE ALSO: Section 3.13.

`initialized_ppoly( PPoly p, int pos, int total)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Initializes the `pos`'th polynomial `p` in a set of `total` polynomials before Gröbner basis computation.

SEE ALSO: Section 4.4.2.

`initialized_pl( PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `initialized_ps`.

SEE ALSO: `initialized_ps`.

`initialized_poly( PPoly p, int pos, int total)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Initializes the `pos`'th polynomial `p` in a set of `total` polynomials before Gröbner basis computation.

SEE ALSO: Section 4.4.1.

`initialized_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Initializes a set of polynomials `ps` before Gröbner basis computation.

SEE ALSO: `initialized_poly`.

`insert_lpl( LP lp, PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `insert_lps`.

SEE ALSO: `insert_lps`, `expanded`.

`insert_lps( LP lp, PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ A labeled polynomial `lp` is inserted into an ordered set of labeled polynomials `ps`. The set is in ascending order w.r.t. `is_leading_monomial_greater_lp`.

SEE ALSO: Section 4.5.

`insert_ppoly_pl( PPoly p, PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `insert_ppoly_ps`.

SEE ALSO: `insert_ppoly_ps`, `expanded`.

`insert_ppoly_ps( PPoly p, PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ A polynomial `p` of type `PPoly` is inserted into an ordered set of polynomials `ps`. The set is in ascending order w.r.t. `is_leading_monomial_greater_ppoly`.

SEE ALSO: Section 4.5, Section 4.4.2.

`insert_pl( Poly p, PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `insert_ps`.

SEE ALSO: `insert_ps`, `expanded`.

`insert_ps( Poly p, PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ An unlabeled polynomial `p` is inserted into an ordered set of unlabeled polynomials `ps`. The set is in ascending order w.r.t. `is_leading_monomial_greater_poly`.

SEE ALSO: Section 4.5.

---

inter_reduced_ps(PS ps)

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Polynomials whose leading monomial is a multiple of one of the other polynomials are deleted from the set `ps`. Each of the remaining polynomials is completely reduced w.r.t. `ps`.

SEE ALSO: `canceled_multiples_ps`, `completely_reduced_ps`.

---

inverse_coef(Coef c)

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff` or `CU.coef2rf` or `CU.coef2rn`

▷ Returns the inverse element of `c` in the selected field of coefficients.

SEE ALSO: Section 4.2.

---

inverse_ff(FF ff)

TYPE: Macro

DEFINED IN: `ff.h`

▷ Returns the inverse element of `ff` in a finite field. It is renamed to the SACLIB function `MIINV`.

SEE ALSO: Section 3.9.

---

inverse_rf(RF rf)

TYPE: Function

DEFINED IN: `rf.c`

▷ Returns the inverse of `rf` in the field of rational functions.

SEE ALSO: Section 3.8.

---

inverse_rn(RN r)

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns the inverse of `r` in the field of rational numbers. It is renamed to the SACLIB function `RNINV`.

---

is_disjoint_el(EL el1, EL el2)

TYPE: Predicate

DEFINED IN: `el.c`

▷ Implementation of virtual predicate `is_disjoint_pp`.

SEE ALSO: `is_disjoint_pp`.

`is_disjoint_pp( PP pp1, PP pp2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if *all variables* that occur in `pp1` *vanish* in `pp2` and vice versa, or, with other words, if all variables with non-zero exponent in `pp1` have exponent 0 in `pp2` and vice versa.

SEE ALSO: Section 4.3.

`is_empty_pas( PaS pas)`

TYPE: Macro

DEFINED IN: `pa.h`

▷ Returns `TRUE` if `pas` is the empty (pair) set.

`is_empty_pl( PL pl)`

TYPE: Macro

DEFINED IN: `pl.h`

▷ Implementation of virtual predicate `is_empty_ps`.

SEE ALSO: `is_empty_ps`.

`is_empty_ps( PS ps)`

TYPE: Virtual Predicate

DEFINED IN: `CU.ps2pl`

▷ Returns `TRUE` if `ps` is the empty (polynomial) set.

SEE ALSO: Section 4.5.

`is_equal_coef( Coef c1, Coef c2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns `TRUE` if `c1` = `c2`, where `c1` and `c2` are elements of the chosen coefficient domain.

SEE ALSO: Section 4.2.

`is_equal_dpl( DPL p1, DPL p2)`

TYPE: Predicate

DEFINED IN: `dpl.c`

▷ Returns `TRUE` if `p1` = `p2`, where `p1` and `p2` are distributive polynomials in list representation.

SEE ALSO: Section 3.12.

`is_equal_el( EL el1, EL el2)`

TYPE: Predicate

DEFINED IN: `el.c`

▷ Returns `TRUE` if `el1` = `el2`, where `el1` and `el2` are exponent lists.

SEE ALSO: Section 3.10.

`is_equal_ff( FF ff1, FF ff2)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Returns `TRUE` if `ff1` = `ff2`, where `ff1` and `ff2` are elements of a finite field. It uses the SACLIB function `ICOMP`.

SEE ALSO: Section 3.9.

`is_equal_int( Int i1, Int i2)`

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns `TRUE` if `i1` = `i2`, where `i1` and `i2` are arbitrarily long integers. It uses the SACLIB function `ICOMP`.

SEE ALSO: Section 3.6.

`is_equal_lp( LP lp1, LP lp2)`

TYPE: Predicate

DEFINED IN: `lp.c`

▷ Returns `TRUE` if `lp1` = `lp2`, where `lp1` and `lp2` are labeled polynomials.

SEE ALSO: Section 3.14.

`is_equal_pa( Pa pa1, Pa pa2)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Returns `TRUE` if `pa1` = `pa2`, where `pa1` and `pa2` are pairs of polynomials.

SEE ALSO: Section 3.16.

`is_equal_pc( PC pc1, PC pc2)`

TYPE: Predicate

DEFINED IN: `pc.c`

▷ Returns `TRUE` if `pc1` = `pc2`, where `pc1` and `pc2` are polynomials with cofactors.

SEE ALSO: Section 3.13.

`is_equal_poly( Poly p1, Poly p2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `TRUE` if `p1` = `p2`, where `p1` and `p2` are polynomials.

SEE ALSO: Section 4.4.1.

`is_equal_pp( PP pp1, PP pp2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp1` = `pp2`, where `pp1` and `pp2` are power products.

SEE ALSO: Section 4.3.

`is_equal_ppoly( PPoly p1, PPoly p2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.ppoly2dpl`

▷ Returns `TRUE` if `p1` = `p2`, where `p1` and `p2` are pure polynomials.

SEE ALSO: Section 4.4.2.

`is_equal_rf( RF rf1, RF rf2)`

TYPE: Predicate

DEFINED IN: `rf.c`

▷ Returns `TRUE` if `rf1` = `rf2`, where `rf1` and `rf2` are rational functions.

SEE ALSO: Section 3.8.

`is_equal_rn( RN r1, RN r2)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns `TRUE` if `r1` = `r2`, where `r1` and `r2` are rational numbers. It uses the SACLIB function `RNCOMP`.

SEE ALSO: Section 3.7.

`is_equal_syz( Syz s1, Syz s2)`

TYPE: Predicate

DEFINED IN: `syz.c`

▷ Returns `TRUE` if `s1` = `s2`, where `s1` and `s2` are syzygies.

SEE ALSO: Section 3.18.

`is_greater_mon( Mon m1, Mon m2)`

TYPE: Virtual Predicate

DEFINED IN: `GB.to.setup`

▷ Returns `TRUE` if `m1` > `m2` w.r.t. to the chosen term ordering. This predicate determines the ordering of monomials in a polynomial.

SEE ALSO: Section 5.1,
`is_lexical_greater_mon`,
`is_matrix_greater_mon`,
`is_total_degree_inverse_lexical_greater_mon`,
`is_total_degree_lexical_greater_mon`.

`is_greater_mon_pair_order( Mon m1, Mon m2)`

TYPE: Virtual Predicate

DEFINED IN: `GB.po.setup`

▷ Returns `TRUE` if `m1` > `m2` w.r.t. to the chosen ordering for polynomial pairs. This predicate determines the ordering of monomials to be used for ordering polynomial pairs.

SEE ALSO: Section 5.2,
`is_lexical_greater_mon`,
`is_matrix_greater_mon`,
`is_total_degree_inverse_lexical_greater_mon`,
`is_total_degree_lexical_greater_mon`.

`is_greater_rn( RN r1, RN r2)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns `TRUE` if `r1` > `r2`, where `r1` and `r2` are rational numbers. It uses the SACLIB function `RNCOMP`.

SEE ALSO: Section 3.7.

`is_labeled_lcm_greater( Pa p1, Pa p2)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Returns `TRUE` if the *lcm* of the leading monomials of `p1` is greater than the *lcm* of the leading monomials of `p2`. In case the *lcm*s are equal it returns `TRUE` if the pair of labels of `p1` is lexicographically greater than the pair of labels of `p2`. It is an implementation of virtual predicate `is_labeled_pa_greater`.

SEE ALSO: Section 5.2, `is_labeled_pa_greater`.

`is_labeled_pa_greater( Pa p1, Pa p2)`

TYPE: Virtual Predicate

DEFINED IN: `GB.po.setup`

▷ This predicate determines the strategy for ordering a set of polynomial pairs in the case the variant of the Gröbner bases algorithm is chosen, where criteria are applied when the pair set is created or updated.

SEE ALSO: Section 5.2, `is_labeled_lcm_greater`.

`is_lcm_greater( Pa p1, Pa p2)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Returns `TRUE` if the *lcm* of the leading monomials of `p1` is greater than the *lcm* of the leading monomials of `p2`. It is an implementation of virtual predicate `is_pa_greater`.

SEE ALSO: Section 5.2, `is_pa_greater`.

`is_leading_monomial_greater_dpl( DPL p1, DPL p2)`

TYPE: Predicate

DEFINED IN: `dpl.c`

▷ Implementation of virtual predicate `is_leading_monomial_greater_ppoly`.

SEE ALSO: `is_leading_monomial_greater_ppoly`.

`is_leading_monomial_greater_lp( LP lp1, LP lp2)`

TYPE: Predicate

DEFINED IN: `lp.c`

▷ Returns `TRUE` if the leading monomial of `lp1` is greater than the leading monomial of `lp2` w.r.t. the chosen ordering on monomials `is_greater_mon`.

SEE ALSO: `is_leading_monomial_greater_poly`.

`is_leading_monomial_greater_pc( PC pc1, PC pc2)`

TYPE: Predicate

DEFINED IN: `pc.c`

▷ Implementation of virtual predicate `is_leading_monomial_greater_poly`.

SEE ALSO: `is_leading_monomial_greater_poly`.

`is_leading_monomial_greater_poly( Poly p1, Poly p2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `TRUE` if the leading monomial of `p1` is greater than the leading monomial of `p2` w.r.t. the chosen ordering on monomials `is_greater_mon`.

SEE ALSO: Section 4.4.1.

`is_leading_monomial_greater_ppoly( PPoly p1, PPoly p2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.ppoly2dpl`

▷ Returns `TRUE` if the leading monomial of `p1` is greater than the leading monomial of `p2` w.r.t. the chosen ordering on monomials `is_greater_mon`.

SEE ALSO: Section 4.4.2.

`is_leading_monomial_multiple_dpl( DPL p1, DPL p2)`

TYPE: Predicate

DEFINED IN: `dpl.c`

▷ Implementation of virtual predicate `is_leading_monomial_multiple_ppoly`.

SEE ALSO: `is_leading_monomial_multiple_ppoly`.

`is_leading_monomial_multiple_lp( LP lp1, LP lp2)`

TYPE: Predicate

DEFINED IN: `lp.c`

▷ Returns `TRUE` if the leading monomial of `lp1` is a multiple of the leading monomial of `lp2`.

SEE ALSO: `is_leading_monomial_multiple_poly`.

---

is_leading_monomial_multiple_pc(PC pc1, PC pc2)

TYPE: Predicate
DEFINED IN: `pc.c`

▷ Implementation of virtual predicate `is_leading_monomial_multiple_poly`.

SEE ALSO: `is_leading_monomial_multiple_poly`.

---

is_leading_monomial_multiple_poly(Poly p1, Poly p2)

TYPE: Virtual Predicate
DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `TRUE` if the leading monomial of `p1` is a multiple of the leading monomial of `p2`.

SEE ALSO: Section 4.4.1.

---

is_leading_monomial_multiple_ppoly(PPoly p1, PPoly p2)

TYPE: Virtual Predicate
DEFINED IN: `CU.ppoly2dpl`

▷ Returns `TRUE` if the leading monomial of `p1` is a multiple of the leading monomial of `p2`.

SEE ALSO: Section 4.4.2.

---

is_lexical_greater_el(EL el1, EL el2)

TYPE: Predicate
DEFINED IN: `el.c`

▷ Implementation of virtual predicate `is_lexical_greater_pp`.

SEE ALSO: `is_lexical_greater_pp`.

---

is_lexical_greater_mon(Mon m1, Mon m2)

TYPE: Predicate
DEFINED IN: `mon.c`

▷ Implementation of virtual predicates `is_greater_mon` and `is_greater_mon_pair_order`.

SEE ALSO: Section 5.1,
`is_lexical_greater_pp`.

---

is_lexical_greater_pp(PP pp1, PP pp2)

TYPE: Virtual Predicate
DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp1` is lexical greater than `pp2`.

SEE ALSO: Section 5.1.

---

is_matrix_greater_el(EL el1, EL el2)

TYPE: Predicate
DEFINED IN: `el.c`

▷ Implementation of virtual predicate `is_matrix_greater_pp`.

SEE ALSO: `is_matrix_greater_pp`.

---

is_matrix_greater_mon(Mon m1, Mon m2)

TYPE: Predicate
DEFINED IN: `mon.c`

▷ Implementation of virtual predicates `is_greater_mon` and `is_greater_mon_pair_order`.

SEE ALSO: Section 5.1,
`is_matrix_greater_pp`.

---

is_matrix_greater_pp(PP pp1, PP pp2)

TYPE: Virtual Predicate
DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp1` is greater than `pp2` in the term ordering induced by the matrix `order_matrix_`.

SEE ALSO: Section 5.1,
`order_matrix_`.

---

is_multiple_el(EL el1, EL el2)

TYPE: Predicate
DEFINED IN: `el.c`

▷ Implementation of virtual predicate `is_multiple_pp`.

SEE ALSO: `is_multiple_pp`.

---

is_multiple_mon(Mon m1, Mon m2)

TYPE: Predicate
DEFINED IN: `mon.c`

▷ Returns `TRUE` if `m1` is a multiple of `m2`.

SEE ALSO: `is_multiple_pp`.

`is_multiple_pp( PP pp1, PP pp2)`

TYPE: Virtual Predicate

DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp1` is a multiple of `pp2`.

`is_negative_coef( Coef c)`

TYPE: Virtual Predicate

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`.

▷ Returns `TRUE` if `c` is negative, where `c` is an element of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

`is_negative_ff( FF ff)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Due to the representation of finite fields, this predicate returns always `FALSE`.

SEE ALSO: Section 3.9.

`is_negative_int( Int i)`

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns `TRUE` if `i < 0`, where `i` is an arbitrarily long integer. It uses the SACLIB function `ICOMP`.

SEE ALSO: Section 3.6.

`is_negative_mon( Mon m)`

TYPE: Predicate

DEFINED IN: `mon.c`

▷ Returns `TRUE` if the coefficient of `m` is negative.

SEE ALSO: `is_negative_coef`.

`is_negative_rf( RF rf)`

TYPE: Predicate

DEFINED IN: `rf.c`

▷ Returns `TRUE` if the leading coefficient of the numerator of `rf` is negative.

SEE ALSO: Section 3.8.

`is_negative_rn( RN r)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns `TRUE` if `r < 0`, where `r` is a rational number. It uses the SACLIB function `RNCOMP`.

SEE ALSO: Section 3.7.

`is_one_coef( Coef c)`

TYPE: Virtual Predicate

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns `TRUE` if `c` equals the unit element of the chosen coefficient domain.

SEE ALSO: Section 4.2.

`is_one_el( EL el)`

TYPE: Predicate

DEFINED IN: `el.c`

▷ Implementation of the virtual predicate `is_one_pp`.

SEE ALSO: `is_one_el`.

`is_one_ff( FF ff)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Returns `TRUE` if `ff = 1`, where `ff` is an element of a finite field. It uses the SACLIB function `ICOMP`.

SEE ALSO: Section 3.9.

`is_one_int( Int i)`

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns `TRUE` if `i = 1`, where `i` is an arbitrarily long integer.

`is_one_pp( PP pp)`

TYPE: Virtual Predicate

DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp` equals the unit element of the domain of power products.

SEE ALSO: Section 4.3.

---

is_one_rf(RF rf)

TYPE: Predicate

DEFINED IN: `rf.c`

▷ Returns `TRUE` if `rf = 1`, where `rf` is a rational function.

SEE ALSO: Section 3.8.

---

is_one_rn(RN r)

TYPE: Macro

DEFINED IN: `rn.h`

▷ Returns `TRUE` if `r = 1`, where `r` is a rational number.

SEE ALSO: Section 3.7.

---

is_pa_greater(Pa p1, Pa p2)

TYPE: Virtual Predicate

DEFINED IN: `GB.po.setup`

▷ This predicate determines the strategy for ordering a set of polynomial pairs in the case the variant of the Gröbner bases algorithm is chosen, where criteria are applied immediately before computation of the S-polynomial.

SEE ALSO: Section 5.2, `is_lcm_greater`.

---

is_positive_coef(Coef c)

TYPE: Virtual Predicate

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns `TRUE` if `c` is positive, where `c` is an element of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

---

is_positive_ff(FF ff)

TYPE: Macro

DEFINED IN: `ff.h`

▷ Due to the representation of finite fields, this predicate returns always `TRUE`.

SEE ALSO: Section 3.9.

---

is_positive_rf(RF rf)

TYPE: Predicate
DEFINED IN: `rf.c`

▷ Returns `TRUE` if the leading coefficient of the numerator of `rf` is positive.

SEE ALSO: Section 3.8.

---

is_positive_rn(RN r)

TYPE: Macro
DEFINED IN: `rn.h`

▷ Returns `TRUE` if `r > 0`, where `r` is a rational number. It uses the SACLIB function `RNCOMP`.

SEE ALSO: Section 3.7.

---

is_total_degree_inverse_lexical_greater_el(EL el1, EL el2)

TYPE: Predicate
DEFINED IN: `el.c`

▷ Implementation of virtual predicate `is_total_degree_inverse_lexical_greater_pp`.

SEE ALSO: `is_total_degree_inverse_lexical_greater_pp`.

---

is_total_degree_inverse_lexical_greater_mon(Mon m1, Mon m2)

TYPE: Predicate
DEFINED IN: `mon.c`

▷ Implementation of the virtual predicates `is_greater_mon` and `is_greater_mon_pair_order`.

SEE ALSO: Section 5.1, `is_total_degree_inverse_lexical_greater_pp`.

---

is_total_degree_inverse_lexical_greater_pp(PP pp1, PP pp2)

TYPE: Virtual Predicate
DEFINED IN: `CU.pp2el`

▷ Returns `TRUE` if `pp1` is total degree inverse lexical greater than `pp2`.

SEE ALSO: Section 5.1.

---

is_total_degree_lexical_greater_el(EL el1, EL el2)

TYPE: Predicate
DEFINED IN: el.c

▷ Implementation of virtual predicate is_total_degree_lexical_greater_pp.

SEE ALSO: is_total_degree_lexical_greater_pp.

---

is_total_degree_lexical_greater_mon(Mon m1, Mon m2)

TYPE: Predicate
DEFINED IN: mon.c

▷ Implementation of the virtual predicates is_greater_mon and is_greater_mon_pair_order.

SEE ALSO: Section 5.1, is_total_degree_lexical_greater_pp.

---

is_total_degree_lexical_greater_pp(PP pp1, PP pp2)

TYPE: Virtual Predicate
DEFINED IN: CU.pp2el

▷ Returns TRUE if pp1 is total degree lexical greater than pp2.

SEE ALSO: Section 5.1.

---

is_zero_coef(Coef c)

TYPE: Virtual Predicate
DEFINED IN: CU.coef2ff, CU.coef2rf or CU.coef2rn

▷ Returns TRUE if c equals zero in the chosen domain of coefficients.

SEE ALSO: Section 4.2.

---

is_zero_ff(FF ff)

TYPE: Macro
DEFINED IN: ff.h

▷ Returns TRUE if ff = 0, where ff is an element of a finite field.

SEE ALSO: Section 3.9.

`is_zero_int( Int i)`

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns `TRUE` if `i = 0`, where `i` is an arbitrarily long integer.

`is_zero_lp( LP lp)`

TYPE: Predicate

DEFINED IN: `lp.c`

▷ Returns `TRUE` if `lp = 0`, where `lp` is a labeled polynomial.

SEE ALSO: Section 3.14, `is_zero_poly`.

`is_zero_mon( Mon m)`

TYPE: Predicate

DEFINED IN: `mon.c`

▷ Returns `TRUE` if `m = 0`, where `m` is a monomial.

SEE ALSO: Section 3.11, `is_zero_coef`.

`is_zero_pc( PC pc)`

TYPE: Predicate

DEFINED IN: `pc.c`

▷ Returns `TRUE` if `pc = 0`, where `pc` is a polynomial with cofactors.

SEE ALSO: Section 3.13, `is_zero_ppoly`.

`is_zero_poly( Poly p)`

TYPE: Virtual Predicate

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `TRUE` if `p` is the zero polynomial.

SEE ALSO: Section 4.4.1.

`is_zero_ppoly( PPoly p)`

TYPE: Virtual Predicate

DEFINED IN: `CU.ppoly2dpl`

▷ Returns `TRUE` if `p` is the zero polynomial.

SEE ALSO: Section 4.4.2.

`is_zero_rf( RF rf)`

TYPE: Predicate
DEFINED IN: `rf.c`
▷ Returns `TRUE` if `rf = 0`, where `rf` is a rational function.
SEE ALSO: Section 3.8.

`is_zero_rn( RN r)`

TYPE: Macro
DEFINED IN: `rn.h`
▷ Returns `TRUE` if `rn = 0`, where `rn` is a rational number.
SEE ALSO: Section 3.7.

`is_zero_syz( Syz s)`

TYPE: Predicate
DEFINED IN: `syz.c`
▷ Returns `TRUE` if `s` is zero in the module of syzygies.
SEE ALSO: Section 3.18.

`label( LP lp)`

TYPE: Macro
DEFINED IN: `lp.h`
▷ Returns the label of the labeled polynomial `lp`.
SEE ALSO: Section 3.14.

`label_`

TYPE: Global Variable
DEFINED IN: `lp.h`
▷ Contains the label that will be used for the *next* polynomial to be labeled.
SEE ALSO: Section 3.14, Section 6.

`labeled_pl( PL pl)`

TYPE: Function
DEFINED IN: `pl.c`
▷ Implementation of virtual function `labeled_ps`.
SEE ALSO: `labeled_ps`, `label_`.

`labeled_poly( Poly p)`

TYPE: Function

DEFINED IN: `lp.c`

▷ Returns `p` labeled with the current value of `label_`, which is then increased by 1.

`labeled_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Returns a set of polynomials with each polynomial labeled, starting with 1 for the first polynomial in the set.

SEE ALSO: Section 4.5.

`lcm_coef( Coef c1, Coef c2)`

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the least common multiple of `c1` and `c2`, where `c1` and `c2` are elements of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

`lcm_denominator_rf( RF rf1, RF rf2)`

TYPE: Function

DEFINED IN: `rf.c`

▷ Suppose `rf1` and `rf2` are of the form n_1/d_1 and n_2/d_2 respectively. Then this function returns the rational function $1/\mathrm{lcm}(d_1, d_2)$.

`lcm_el( EL el1, EL el2)`

TYPE: Function

DEFINED IN: `el.c`

▷ Implementation of virtual function `lcm_pp`.

SEE ALSO: `lcm_pp`.

`lcm_ff( FF ff1, FF ff2)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `lcm_coef`. The least common multiple can be defined arbitrarily in a field, we define it to be `ff1`.

SEE ALSO: `lcm_coef`.

`lcm_mon( Mon m1, Mon m2)`

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the least common multiple of the monomials `m1` and `m2`.

`lcm_pp( PP pp1, PP pp2)`

TYPE: Virtual Function

DEFINED IN: `CU.pp2el`

▷ Returns the least common multiple of the power products `pp1` and `pp2`.

SEE ALSO: Section 4.3.

`lcm_rf( RF rf1, RF rf2)`

TYPE: Macro

DEFINED IN: `rf.h`

▷ Implementation of virtual function `lcm_coef`. The least common multiple can be defined arbitrarily in a field, we define it to be `rf1`.

SEE ALSO: `lcm_coef`.

`lcm_rn( RN r1, RN r2)`

TYPE: Macro

DEFINED IN: `rn.h`

▷ Implementation of virtual function `lcm_coef`. The least common multiple can be defined arbitrarily in a field, we define it to be `r1`.

SEE ALSO: `lcm_coef`.

`leading_coefficient( Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the leading coefficient of `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.1.

`leading_coefficient_dpl( DPL dpl)`

TYPE: Macro

DEFINED IN: `dpl.h`

▷ Implementation of virtual function `leading_coefficient_ppoly`.

SEE ALSO: `leading_coefficient_ppoly`.

`leading_coefficient_lp( LP lp)`

TYPE: Macro

DEFINED IN: `lp.h`

▷ Returns the leading coefficient of the labeled polynomial `lp` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 3.14.

`leading_coefficient_pc( PC pc)`

TYPE: Macro

DEFINED IN: `pc.h`

▷ Implementation of virtual function `leading_coefficient`.

SEE ALSO: Section 3.13, `leading_coefficient`.

`leading_coefficient_ppoly( PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the leading coefficient of the polynomial `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.2.

`leading_monomial( Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the leading monomial of `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.1.

`leading_monomial_dpl( DPL dpl)`

TYPE: Macro

DEFINED IN: `dpl.h`

▷ Implementation of virtual function `leading_monomial_ppoly`.

SEE ALSO: `leading_monomial_ppoly`.

---

leading_monomial_lp(LP lp)

TYPE: Macro

DEFINED IN: `lp.h`

▷ Returns the leading monomial of the labeled polynomial `lp` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 3.14.

---

leading_monomial_pc(PC pc)

TYPE: Macro

DEFINED IN: `pc.h`

▷ Implementation of virtual function `leading_monomial`.

SEE ALSO: Section 3.13, `leading_monomial`.

---

leading_monomial_ppoly(PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the leading monomial of the polynomial `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.2.

---

leading_power_product(Poly p)

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the leading power product of `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.1.

---

leading_power_product_dpl(DPL dpl)

TYPE: Macro

DEFINED IN: `dpl.h`

▷ Implementation of virtual function `leading_power_product_ppoly`.

SEE ALSO: `leading_power_product_ppoly`.

---

leading_power_product_lp(LP lp)

TYPE: Macro

DEFINED IN: `lp.h`

▷ Returns the leading power product of the labeled polynomial `lp` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 3.14.

`leading_power_product_pc( PC pc)`

TYPE: Macro

DEFINED IN: `pc.h`

▷ Implementation of virtual function `leading_power_product`.

SEE ALSO: Section 3.13, `leading_power_product`.

`leading_power_product_ppoly( PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the leading power product of the polynomial `p` w.r.t. the chosen ordering `is_greater_mon`.

SEE ALSO: Section 4.4.2.

`merged( Word l1, Word l2, BP comp, BF f, UP cancel)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Returns a merged list from `l1` and `l2`. `l1`, `l2`, and the resulting list are ordered w.r.t. the binary predicate `comp`.

Suppose $e_1 \in$ `l1` and $e_2 \in$ `l2`. If $e_1 > e_2$ w.r.t. `comp` then e_1 will occur *before* e_2 in the resulting list. If $e_1 = e_2$ then `f`(e_1, e_2) is a candidate for the new list. If the candidate *does not* satisfy the predicate `cancel` it is inserted into the resulting list, otherwise it is canceled and the next elements are tested.

`merged_disjunct( Word l1, Word l2, BP comp)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Returns a merged list from `l1` and `l2`, where `l1` and `l2` *do not* contain elements that are equal w.r.t. `comp`. `l1`, `l2`, and the resulting list are ordered w.r.t. the binary predicate `comp`.

SEE ALSO: `merged`.

`merged_disjunct_iter( Word l1, Word l2, BP comp)`

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `merged_disjunct`.

SEE ALSO: `merged_disjunct`.

merged_disjunct_rec(Word l1, Word l2, BP comp)

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `merged_disjunct`.

SEE ALSO: `merged_disjunct`.

merged_iter(Word l1, Word l2, BP comp, BF f, UP cancel)

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `merged`.

SEE ALSO: `merged`.

merged_rec(Word l1, Word l2, BP comp, BF f, UP cancel)

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `merged`.

SEE ALSO: `merged`.

modulus_

TYPE: Global Variable

DEFINED IN: `ff.c`

▷ Global variable containing the modulus in case finite fields are chosen as coefficient domain.

SEE ALSO: Section 3.9, Section 6.

negative_coef(Coef c)

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the negative of `c`, where `c` is an element of the coefficient domain.

SEE ALSO: Section 4.2.

negative_dpl(DPL p)

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `negative_ppoly`.

SEE ALSO: `negative_ppoly`.

---

negative_ff(FF ff)

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `negative_coef`. It returns `-ff` modulo `modulus_` and uses the SACLIB function `MINEG`.

SEE ALSO: Section 3.9.

---

negative_int(i)

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns `-i`, where `i` is an arbitrarily long interger.

---

negative_mon(Mon m)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns `-m`, where `m` is a monomial.

---

negative_pc(PC pc)

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `negative_poly`.

SEE ALSO: Section 3.13, `negative_poly`.

---

negative_poly(Poly p)

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `-p`, where `p` is a polynomial.

SEE ALSO: Section 4.4.1.

---

negative_ppoly(PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns `-p`, where `p` is a polynomial.

SEE ALSO: Section 4.4.2.

`negative_rf( RF rf)`

TYPE: Function
DEFINED IN: `rf.c`

▷ Implementation of virtual function `negative_coef`.

SEE ALSO: Section 3.8, `negative_coef`.

`negative_rn( RN r)`

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of virtual function `negative_coef`. It uses the SACLIB function `RNNEG`.

SEE ALSO: Section 3.7, `negative_coef`.

`negative_syz( Syz s)`

TYPE: Function
DEFINED IN: `syz.c`

▷ Returns the negative of the syzygy `s`.

SEE ALSO: Section 3.18.

`new_pair( LP lp1, LP lp2)`

TYPE: Function
DEFINED IN: `pa.c.trace`

▷ Returns a pair consisting of `lp1` and `lp2`, s.t. the first polynomial in the pair has lower label than the second.

SEE ALSO: Section 3.14.

`new_pairs( Poly p, PS ps)`

TYPE: Function
DEFINED IN: `pa.c.trace`

▷ Returns the set of all pairs consisting of `p` and a polynomial in `ps`.

`normal_form_algorithm`

TYPE: Constant
DEFINED IN: `GB.nf.setup`

▷ Determines which normal form algorithm should be used.

SEE ALSO: Section 5.3.

`normal_form_lp( LP lp, PS ps)`

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the normal form of `lp` w.r.t. `ps`. The result is *not* labeled.

SEE ALSO: Section 5.3, `normal_form_poly`.

`normal_form_poly( Poly p, PS ps)`

TYPE: Virtual Function

DEFINED IN: `GB.nf.setup`

▷ Returns the normal form of `p` w.r.t. `ps`.

SEE ALSO: Section 5.3,
`complete_normal_form_poly`, `partial_normal_form_poly`.

`normalized_dpl( DPL p)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `normalized_ppoly`.

SEE ALSO: `normalized_ppoly`.

`normalized_pc( PC pc)`

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `normalized_poly`.

SEE ALSO: `normalized_poly`.

`PL normalized_pl( PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `normalized_ps`.

SEE ALSO: `normalized_ps`.

`normalized_poly( Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Suppose the leading coefficient of `p` is c. Then it returns `normalizer_coef(`c`)`·`p`.

SEE ALSO: Section 4.4.1, `normalizer_coef`.

`normalized_ppoly( PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Suppose the leading coefficient of `p` is c. Then it returns `normalizer_coef(`c`)`·`p`.

SEE ALSO: Section 4.4.2, `normalizer_coef`.

`normalized_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Returns a set of polynomials derived from `ps` by applying `normalized_poly` to each of the polynomials in `ps`.

SEE ALSO: Section 4.5, `normalized_poly`.

`normalized_rf( RF rf)`

TYPE: Function

DEFINED IN: `rf.c`

▷ Returns an eqivalent representation of `rf` s.t. the leading coefficient of the denominator of `rf` is positive.

SEE ALSO: Section 3.8.

`normalizer_coef( Coef c)`

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ If the coefficient domain is a field, it returns the inverse of `c`. It is used for making the leading coefficient of a polynomial 1. In the case where the coefficient domain is not a field (which does not occur in the current implementation) one has to make up ones mind about a reasonable return value of this function.

SEE ALSO: Section 4.2.

`normalizer_ff( FF ff)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `normalizer_coef`.

SEE ALSO: Section 3.9, `normalizer_coef`.

`normalizer_rf( RF rf)`

TYPE: Function
DEFINED IN: `rf.c`

▷ Implementation of virtual function `normalizer_coef`.

SEE ALSO: Section 3.8, `normalizer_coef`.

`normalizer_rn( RN r)`

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of virtual function `normalizer_coef`.

SEE ALSO: Section 3.7, `normalizer_coef`.

`number_of_field_variables_`

TYPE: Global Variable
DEFINED IN: `rf.c`

▷ Global variable containing the number of field variables in case rational functions are chosen as coefficient domain.

SEE ALSO: Section 3.8, Section 6.

`numerator_rf( RF rf)`

TYPE: Macro
DEFINED IN: `rf.h`

▷ Selects the numerator of the rational function `rf`.

`numerator_rn( RN r)`

TYPE: Macro
DEFINED IN: `rn.h`

▷ Selects the numerator of the rational number `r`.

`one_coef( Coef c)`

TYPE: Virtual Function
DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the unit element of the coefficient domain containing `c`.

SEE ALSO: Section 4.2.

one_dpl(DPL p)

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `one_ppoly`.

SEE ALSO: Section 3.12, `one_ppoly`.

one_el(EL el)

TYPE: Function

DEFINED IN: `el.c`

▷ Implementation of virtual function `one_pp`.

SEE ALSO: Section 3.10, `one_pp`.

one_ff(FF ff)

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `one_coef`.

SEE ALSO: Section 3.9, `one_coef`.

one_int(Int i)

TYPE: Macro

DEFINED IN: `int.h`

▷ Returns the integer 1.

one_mon(Mon m)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the unit element in the domain of monomials containing `m`.

SEE ALSO: Section 3.11, `one_coef`, `one_pp`.

one_ppoly(PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the unit polynomial in the domain of `p`.

SEE ALSO: Section 4.4.2.

one_pp(PP pp)

TYPE: Virtual Function
DEFINED IN: `CU.pp2el`

▷ Returns the unit element in the domain of power products containing `pp`.

SEE ALSO: Section 4.3.

one_rf(RF rf)

TYPE: Function
DEFINED IN: `rf.c`

▷ Implementation of virtual function `one_coef`.

SEE ALSO: Section 4.2, `one_coef`.

one_rn(RN r)

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of virtual function `one_coef`.

SEE ALSO: Section 4.2, `one_coef`.

order_matrix_

TYPE: Global Variable
DEFINED IN: `el.h`

▷ Contains the matrix that is used for ordering power products, when this kind of ordering is selected.

SEE ALSO: Section 5.1, Section 6, `is_matrix_greater_el`.

pair_ordering

TYPE: Constant
DEFINED IN: `GB.po.setup`

▷ Determines which ordering is used for ordering pairs of polynomials. This has an impact on the sequence of S-polynomial computations.

SEE ALSO: Section 5.1, Section 5.2.

pair_set(PS ps)

TYPE: Function
DEFINED IN: `pa.c`

▷ Returns the unordered set of pairs of polynomials of `ps`.

`pair_strategy`

TYPE: Constant

DEFINED IN: `GB.po.setup`

▷ Determines the strategy used for ordering pairs of polynomials. This has an impact on the sequence of S-polynomial computations.

SEE ALSO: Section 5.2.

`partial_normal_form_poly( Poly p, PS ps)`

TYPE: Function

DEFINED IN: `reductions.c.trace`

▷ Returns the partial normal form of the polynomial `p` w.r.t. to the set of polynomials `ps`.

SEE ALSO: Section 5.3.

`poly_extracted_pl( PL pl)`

TYPE: Function

DEFINED IN: `pl.c`

▷ Implementation of virtual function `poly_extracted_ps`.

SEE ALSO: `poly_extracted_ps`.

`poly_extracted_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ Extracts the polynomial part of each element in `ps`.

SEE ALSO: `polynomial_poly`.

`poly_structure`

TYPE: Constant

DEFINED IN: `GB.poly.setup`

▷ Determines the structure of polynomials.

SEE ALSO: Section 4.4.1.

`polynomial_dpl( DPL p)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `polynomial_ppoly`.

SEE ALSO: `polynomial_ppoly`.

`polynomial_lp( LP lp)`

TYPE: Macro

DEFINED IN: `lp.h`

▷ Selects the polynomial of a labeled polynomial.

SEE ALSO: Section 3.14.

`polynomial_pc( PC pc)`

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `polynomial_poly`.

SEE ALSO: Section 3.13, `polynomial_poly`.

`polynomial_ppoly( PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ This function is introduced as an implementation of the virtual function `polynomial_poly`, which simplifies to the identity function in this case.

SEE ALSO: Section 4.4.2.

`polynomial_poly( Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Selects the "pure" polynomial of `p`.

SEE ALSO: Section 4.4.1.

`prepend_monomial( Mon m, Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns a polynomial with `m` prepended to the polynomial part of `p`.

SEE ALSO: Section 4.4.1.

---

prepend_monomial_dpl(Mon m, DPL p)

TYPE: Macro

DEFINED IN: `dpl.h`

▷ Implementation of virtual function `prepend_monomial_ppoly`.

SEE ALSO: `prepend_monomial_ppoly`.

---

prepend_monomial_pc(Mon m, PC pc)

TYPE: Macro

DEFINED IN: `pc.h`

▷ Implementation of virtual function `prepend_monomial`. It prepends the monomial `m` as leading monomial to the polynomial of `pc` and leaves the cofactors unchanged.

SEE ALSO: `prepend_monomial`.

---

prepend_monomial_ppoly(Mon m, PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ It prepends the monomial `m` as leading monomial to the polynomial `p`.

SEE ALSO: Section 4.4.2.

---

product_coef(Coef c1, Coef c2)

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns `c1` · `c2`, where `c1` and `c2` are elements of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

---

product_coef_dpl(Coef c, DPL p)

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `product_coef_ppoly`.

SEE ALSO: `product_coef_ppoly`.

---

product_coef_mon(Coef c, Mon m)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns `c` · `m`, where `c` is an element of the coefficient domain of the monomial `m`.

`product_coef_pc( Coef c, PC pc)`

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `product_coef_poly`.

SEE ALSO: `product_coef_poly`.

`product_coef_poly( Coef c, Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns `c` · `p`, where `c` is an element of the coefficient domain of the polynomial part of `p`.

SEE ALSO: Section 4.4.1.

`product_coef_ppoly( Coef c, PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns `c` · `p`, where `c` is an element of the coefficient domain of the polynomial `p`.

SEE ALSO: Section 4.4.1.

`product_criterion( LP lp1, LP lp2)`

TYPE: Predicate

DEFINED IN: `reductions.c.trace`

▷ Returns `TRUE` if $lp1$ and $lp2$ satisy the product criterion, i.e. $\mathrm{lcm}(lp1, lp2) = lp1 \cdot lp2$.

SEE ALSO: [Buchberger, 1985].

`product_dpl( DPL p1, DPL p2)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `product_ppoly`.

SEE ALSO: `product_ppoly`.

`product_el( EL el1, EL el2)`

TYPE: Function

DEFINED IN: `el.c`

▷ Implementation of virtual function `product_pp`.

SEE ALSO: `product_pp`.

`product_ff( FF ff1, FF ff2)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `product_coef`. It uses the SACLIB function MIPROD.

SEE ALSO: `product_coef`.

`product_mon( Mon m1, Mon m2)`

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the product of the monomials `m1` and `m2`.

`product_mon_dpl( Mon m, DPL p)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `product_mon_ppoly`.

SEE ALSO: `product_mon_ppoly`.

`product_mon_pc( Mon m, PC pc)`

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `product_mon_poly`.

SEE ALSO: `product_mon_poly`.

`product_mon_ppoly( Mon m, PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the product of the monomial `m` and the polynomial `p`.

SEE ALSO: Section 4.4.2.

`product_mon_poly( Mon m, Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the product of the monomial `m` and the polynomial `p`.

SEE ALSO: Section 4.4.1.

product_monic_mon(Mon m1, Mon m2)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the product of the *monic* monomial `m1` (i.e. it has coefficient 1) and the monomial `m2`.

product_monic_mon_dpl(Mon m, DPL p)

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `product_monic_mon_ppoly`.

SEE ALSO: `product_monic_mon_ppoly`.

product_monic_mon_pc(Mon m, PC pc)

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `product_monic_mon_poly`.

SEE ALSO: `product_monic_mon_poly`.

product_monic_mon_ppoly(Mon m, PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the product of the *monic* monomial `m` (i.e. it has coefficient 1) and the polynomial `p`.

SEE ALSO: Section 4.4.2.

product_monic_mon_poly(Mon m, Poly p)

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the product of the *monic* monomial `m` (i.e. it has coefficient 1) and the polynomial `p`.

SEE ALSO: Section 4.4.1.

product_neg_monic_mon(Mon m1, Mon m2)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the product of the *negative monic* monomial `m1` (i.e. it has coefficient -1) and the monomial `m2`.

`product_neg_monic_mon_dpl( Mon m, DPL p)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `product_neg_monic_mon_ppoly`.

SEE ALSO: `product_neg_monic_mon_ppoly`.

`product_neg_monic_mon_pc( Mon m, PC pc)`

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `product_neg_monic_mon_poly`.

SEE ALSO: `product_neg_monic_mon_poly`.

`product_neg_monic_mon_ppoly( Mon m, PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the product of the *negative monic* monomial `m` (i.e. it has coefficient -1) and the polynomial `p`.

SEE ALSO: Section 4.4.2.

`product_neg_monic_mon_poly( Mon m, Poly p)`

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the product of the *negative monic* monomial `m` (i.e. it has coefficient -1) and the polynomial `p`.

SEE ALSO: Section 4.4.1.

`product_ppoly( PPoly p1, PPoly p2)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the product of the polynomials `p1` and `p2`.

SEE ALSO: Section 4.4.2.

---

product_pp(PP pp1, PP pp2)

TYPE: Virtual Function
DEFINED IN: `CU.pp2el`

▷ Returns the product of the power products `pp1` and `pp2`.

---

product_rf(RF rf1, RF rf2)

TYPE: Function
DEFINED IN: `rf.c`

▷ Implementation of virtual function `product_coef`.

SEE ALSO: Section 3.8.

---

product_rn(RN r1, RN r2)

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of virtual function `product_coef`. It uses the SACLIB function `RNPROD`.

SEE ALSO: Section 3.7.

---

pure_poly_representation

TYPE: Constant
DEFINED IN: `GB.ppoly.setup`

▷ Determines the representation of pure polynomials.

SEE ALSO: Section 4.4.2.

---

quotient_coef(Coef c1, Coef c2)

TYPE: Virtual Function
DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the quotient of `c1` and `c2`, where `c1` and `c2` are elements of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

---

quotient_dpl(DPL p1, DPL p2)

TYPE: Function
DEFINED IN: `dpl.c`

▷ Implementation of the virtual function `quotient_ppoly`.

SEE ALSO: Section 3.12, `quotient_ppoly`.

---

quotient_el(EL el1, EL el2)

TYPE: Function

DEFINED IN: `el.c`

▷ Implementation of the virtual function `quotient_pp`.

SEE ALSO: `quotient_pp`.

---

quotient_ff(FF ff1, FF ff2)

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of the virtual function `quotient_coef`. It uses the SACLIB function `MIQ`.

SEE ALSO: `quotient_coef`.

---

quotient_monic_mon(Mon m1, Mon m2)

TYPE: Function

DEFINED IN: `mon.c`

▷ Returns the quotient of the monomial `m1` and the *monic* monomial `m2` (i.e. it has coefficient 1).

---

quotient_ppoly(PPoly p1, PPoly p2)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the quotient of the polynomial `p1` and the polynomial `p2`. This quotient is only defined if `p2` is a constant, otherwise an error is reported.

SEE ALSO: Section 4.4.2.

---

quotient_pp(PP pp1, PP pp2)

TYPE: Virtual Function

DEFINED IN: `CU.pp2el`

▷ Returns the quotient of the power products `pp1` and `pp2`.

---

quotient_rf(RF rf1, RF rf2)

TYPE: Function

DEFINED IN: `rf.c`

▷ Implementation of the virtual function `quotient_coef`.

SEE ALSO: `quotient_coef`.

---

quotient_rn(RN r1, RN r2)

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of the virtual function `quotient_coef`. It uses the SACLIB function `RNQ`.

SEE ALSO: `quotient_coef`.

---

rational_function(Word n, Word d)

TYPE: Macro
DEFINED IN: `rf.h`

▷ Returns the rational function n/d built from the SACLIB integral polynomials `n` and `d`. It is assumed that `n` and `d` are relatively prime.

SEE ALSO: Section 3.8.

---

remaining_pairs_pas(PaS pas)

TYPE: Macro
DEFINED IN: `pa.h`

▷ Returns a set of pairs containing all pairs of `pas` but the first one. It is renamed to `REST`.

SEE ALSO: `REST`.

---

remaining_polynomial(Poly p)

TYPE: Virtual Function
DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the remaining polynomial of `p`, i.e. `p` without its leading monomial.

SEE ALSO: Section 4.4.1.

---

remaining_polynomial_dpl(DPL p)

TYPE: Macro
DEFINED IN: `dpl.h`

▷ Implementation of virtual function `remaining_polynomial_ppoly`. It is renamed to `REST`.

SEE ALSO: `REST`, `remaining_polynomial_ppoly`.

---

remaining_polynomial_lp(LP lp)

TYPE: Macro
DEFINED IN: `lp.h`

▷ Returns the (unlabeled) remaining polynomial of the polynomial of `lp`.

SEE ALSO: Section 3.14, `remaining_polynomial`.

`remaining_polynomial_pc( PC pc)`

TYPE: Macro

DEFINED IN: `pc.h`

▷ Implementation of virtual function `remaining_polynomial`. The cofactors of `pc` are left unchanged, since this is valid in our applications.

SEE ALSO: Section 3.13, `remaining_polynomial`.

`remaining_polynomial_ppoly( PPoly p)`

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the remaining polynomial of `p`, i.e. `p` without its leading monomial.

SEE ALSO: Section 4.4.2.

`singleton( Word w)`

TYPE: Macro

DEFINED IN: `groebner_globals.h`

▷ Returns a list containing `w`.

`split( Word element, Word list, BP rel)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Returns a list (A, B), s.t. A contains all elements e of `list`, for which `rel(`e`, element)` holds and B contains the remaining elements of `list`.

`split_iter( Word element, Word list, BP rel)`

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `split`.

SEE ALSO: Section 3.4, `split`.

`split_rec( Word element, Word list, BP rel)`

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `split`.

SEE ALSO: Section 3.5, `split`.

`start_groebner()`

TYPE: Function

DEFINED IN: `init.c.trace`

▷ Prints relevant information about the current setup of the library.

`step( BF bf, Word o, Word l)`

TYPE: Virtual Function

DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ Suppose $l = (l_1, \ldots, l_n)$. Then the return value is $(\mathtt{bf}(o, l_1), \ldots, \mathtt{bf}(o, l_n))$.

`step_iter( BF bf, Word o, Word l)`

TYPE: Function

DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `step`.

SEE ALSO: Section 3.4, `step`.

`step_rec( BF bf, Word o, Word l)`

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `step`.

SEE ALSO: Section 3.5, `step`.

`sum_coef( Coef c1, Coef c2)`

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns `c1 + c2`, where `c1` and `c2` are elements of the chosen domain of coefficients.

SEE ALSO: Section 4.2.

`sum_dpl( DPL p1, DPL p2)`

TYPE: Function

DEFINED IN: `dpl.c`

▷ Implementation of virtual function `sum_ppoly`.

SEE ALSO: `sum_ppoly`.

`sum_ff( FF ff1, FF ff2)`

TYPE: Macro
DEFINED IN: `ff.h`

▷ Implementation of virtual function `sum_coef`. It uses the SACLIB function `MISUM`.

SEE ALSO: `sum_coef`.

`sum_mon( Mon m1, Mon m2)`

TYPE: Function
DEFINED IN: `mon.c`

▷ Returns the sum of the monomials `m1` and `m2`. The power products of `m1` and `m2` are assumed to be equal, otherwise the result is not correct, but *no error* is reported.

`sum_pc( PC pc1, PC pc2)`

TYPE: Function
DEFINED IN: `pc.c`

▷ Implementation of virtual function `sum_poly`.

SEE ALSO: Section 3.13, `sum_poly`.

`sum_ppoly( PPoly p1, PPoly p2)`

TYPE: Virtual Function
DEFINED IN: `CU.ppoly2dpl`

▷ Returns the sum of the polynomials `p1` and `p2`.

SEE ALSO: Section 4.4.2.

`sum_poly( Poly p1, Poly p2)`

TYPE: Virtual Function
DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the sum of the polynomials `p1` and `p2`.

SEE ALSO: Section 4.4.1.

`sum_rf( RF rf1, RF rf2)`

TYPE: Function
DEFINED IN: `rf.c`

▷ Implementation of the virtual function `sum_coef`.

SEE ALSO: Section 3.8, `sum_coef`.

---

sum_rn(RN r1, RN r2)

TYPE: Macro
DEFINED IN: `rn.h`

▷ Implementation of the virtual function `sum_coef`.

SEE ALSO: Section 3.7, `sum_coef`.

---

sum_syz(Syz s1, Syz s2)

TYPE: Function
DEFINED IN: `syz.c`

▷ Returns the sum of the syzygies `s1` and `s2`.

SEE ALSO: Section 3.18.

---

thinned_by_chain_criterion(PaS pair_set, PS basis, Pa pair)

TYPE: Function
DEFINED IN: `pa.c.trace`

▷ All pairs of `pair_set`, for which the chain criterion w.r.t. `basis` and `pair` is satisfied, are deleted from `pair_set`.

SEE ALSO: `chain_criterion`.

---

thinned(Word element, Word list, BP rel)

TYPE: Virtual Function
DEFINED IN: `CU.list2iter` or `CU.list2rec`

▷ All elements e from `list`, s.t. `rel`(e, `element`) is satisfied, are deleted from `list`.

---

thinned_iter(Word element, Word list, BP rel)

TYPE: Function
DEFINED IN: `list_iter.c`

▷ Iterative implementation of virtual function `thinned`.

SEE ALSO: Section 3.4, `thinned`.

---

thinned_pairs(PS ps, PaS pairs)

TYPE: Function
DEFINED IN: `pa.c.trace`

▷ All pairs in `pairs`, which contain at least one polynomial of `ps`, are deleted from `pairs`. It uses the function `thinned`.

SEE ALSO: `thinned`.

`thinned_rec( Word element, Word list, BP rel)`

TYPE: Function

DEFINED IN: `list_rec.c`

▷ Recursive implementation of virtual function `thinned`.

SEE ALSO: Section 3.5, `thinned`.

`triangle_criterion( Pa pa, LP lp, Pa pair)`

TYPE: Predicate

DEFINED IN: `pa.c.trace`

▷ Suppose `pa` is of the form (p_1, p_2). `triangle_criterion` returns `TRUE` if

- ◇ `lp` $\neq p_1$ and `lp` $\neq p_2$ and
- ◇ $\mathrm{lcm}(\mathrm{lm}(p_1), \mathrm{lm}(p_2))$ is a multiple of lm(`lp`), where lm stands for "leading monomial", and
- ◇ the pairs (p_1, `lp`) and (`lp`, p_2) have already been considered for reduction. It would lead too far to explain why the pair `pair` has to be passed as an argument.

SEE ALSO: `chain_criterion`.

`trivial_pair( )`

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ Returns a pair of labeled polynomials which is *not equal* to any other pair of polynomials. It is used by `initial_pairs_with_criteria`.

SEE ALSO: `initial_pairs_with_criteria`.

`unlabeled_lp( LP lp)`

TYPE: Function

DEFINED IN: `lp.c`

▷ Removes the label from the labeled polynomial `lp`.

SEE ALSO: Section 3.14.

`unlabeled_pl( PL pl)`

TYPE: Function

DEFINED IN: `pl.h`

▷ Implementation of virtual function `unlabeled_ps`.

SEE ALSO: `unlabeled_ps`.

`unlabeled_ps( PS ps)`

TYPE: Virtual Function

DEFINED IN: `CU.ps2pl`

▷ `unlabeled_lp` is applied to each polynomial in the set `ps`.

SEE ALSO: `unlabeled_lp`.

`updated_pairs( PaS pairs, PS ps, LP lp)`

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ All pairs consisting of `lp` and a polynomial of `ps` are adjoined to `pairs`.

`updated_pairs_with_product_criterion( PaS pairs, PS ps, LP lp)`

TYPE: Function

DEFINED IN: `pa.c.trace`

▷ All pairs consisting of `lp` and a polynomial of `ps` that cannot be canceled by the product criterion are adjoined to `pairs`.

SEE ALSO: `product_criterion`.

`zero_coef( Coef c)`

TYPE: Virtual Function

DEFINED IN: `CU.coef2ff`, `CU.coef2rf` or `CU.coef2rn`

▷ Returns the zero element of the coefficient domain containing `c`.

SEE ALSO: Section 4.2.

`zero_dpl( DPL p)`

TYPE: Macro

DEFINED IN: `dpl.h`

▷ Implementation of virtual function `zero_ppoly`. Returns the empty list.

SEE ALSO: `EMPTY`, `zero_ppoly`.

`zero_ff( FF ff)`

TYPE: Macro

DEFINED IN: `ff.h`

▷ Implementation of virtual function `zero_coef`.

SEE ALSO: Section 3.9, `zero_coef`.

---

zero_pc(PC pc)

TYPE: Function

DEFINED IN: `pc.c`

▷ Implementation of virtual function `zero_poly`.

SEE ALSO: Section 3.13, `zero_poly`.

---

zero_ppoly(PPoly p)

TYPE: Virtual Function

DEFINED IN: `CU.ppoly2dpl`

▷ Returns the zero polynomial in the same representation like `p`.

SEE ALSO: Section 4.4.2.

---

zero_poly(Poly p)

TYPE: Virtual Function

DEFINED IN: `CU.poly2pc` or `CU.poly2ppoly`

▷ Returns the zero polynomial in the same structure like `p`.

SEE ALSO: Section 4.4.1.

---

zero_rf(RF rf)

TYPE: Function

DEFINED IN: `rf.c`

▷ Implementation of virtual function `zero_coef`.

SEE ALSO: Section 3.8, `zero_coef`.

---

zero_rn(RN r)

TYPE: Macro

DEFINED IN: `rn.h`

▷ Implementation of virtual function `zero_coef`.

SEE ALSO: Section 3.7, `zero_coef`.

---

zero_syz(Syz s)

TYPE: Function

DEFINED IN: `syz.h`

▷ Returns the zero element in the module of syzygies containing `s`.

SEE ALSO: Section 3.18.

Acknowledgment

This work was sponsored by the Gröbner bases project of the *Bundesministerium für Wissenschaft und Forschung, Austria*, and by the *ESPRIT-POSSO project*.

References

[Buchberger *et al.*, 1993] B. Buchberger, G. E. Collins, M. J. Encarnacion, H. Hong, Johnson J. R., W. Krandick, R. Loos, A. M. Mandache, A. Neubacher, and H. Vielhaber. SACLIB User's Guide. Technical Report RISC-Linz Series 93-19, Univ. Linz, RISC, Linz, Austria, 1993.

[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, 1990] B. Buchberger. An Implementation of Gröbner Bases in Mathematica. Technical Report RISC-Linz Series 90-58, Univ. Linz, RISC, Linz, Austria, 1990.

[Buchberger, 1991] 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.

[Giovini *et al.*, 1991] A. Giovini, T. Mora, G. Niesi, L. Robbiano, and C. Traverso. "One sugar cube, please" or Selection Strategies in the Buchberger Algorithms. In S. M. Watt, editor, *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation (ISSAC 91)*, pages 49–54, Bonn, July 15–17 1991.

[Hong *et al.*, 1993] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of SACLIB/PACLIB Kernels. In *DISCO-93 (Design and Implementation of Symbolic Computation Systems)*, Gmunden, Austria, 1993. To appear.

[Jenks and Sutor, 1992] R. D. Jenks and R. S. Sutor. *AXIOM, the Scientific Computation System*. Springer Verlag, 1992.

[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.

[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, 1992] W. Windsteiger. Gröbner Bases: A Characterization by Syzygy Completeness and an Implementation[3]. Master's thesis, RISC-Linz, University of Linz, Austria, 1992.

[Windsteiger, 1993] W. Windsteiger. GRÖBNER-IO: An Input/Output Libray for GRÖBNER. Technical Report in preparation, RISC-Linz, University of Linz, 1993.

[3] http://www.risc.uni-linz.ac.at:/people/wwindste/Public/Reports/master-thesis.ps.gz

Index