An Implementation of Gröbner bases in *Mathematica*

Bruno Buchberger

December 18, 1992

Abstract

In this paper we report on GRÖBNER-*Mathematica*, an experimental implementation for computing *Gröbner bases* in *Mathematica*. This implementation in its current form offers three different versions of the Gröbner bases algorithm and is generic with respect to

- the ordering of power products,
- the domain of coefficients of the polynomials, and
- the representation of power products, monomials, polynomials, and other domains.

1 How To Use GRÖBNER-*Mathematica*

The package GRÖBNER-*Mathematica* implements the author's Gröbner bases algorithm (see [Buchberger 85]) in *Mathematica*. In order to start GRÖBNER-*Mathematica* one has to load the start-up file `Groebner.m` into the *Mathematica* system by entering the command `<<Groebner.m`. This file also contains some settings of global variables to initialize a default version of the Gröbner bases algorithm (global variables always begin with `$`). By changing these global variables one can choose a particular variant of the Gröbner bases algorithm. The global variables and their possible values are listed in Table 1, the exact meaning of the possible values of the global variables is explained in Section 2.5. An evaluation of the performance of the package is given in [Buchberger 91].

Calling the function GB with one parameter, a set of polynomials, causes the system to call the version of the Gröbner bases algorithm that is stored in $GroebnerBasisPN. The result is the reduced Gröbner basis with respect to the ordering $GreaterEL. The normal form algorithm used during computation is the one stored in $NormalFormDNP, interreduction of the basis depends on the value of $InterReducedPN.

To read in a polynomial set, the most comfortable way is to enter a list of polynomials in *Mathematica* format (preferable from a file) and then convert it to a desired

Global Variable	Semantics	Possible Values
$GreaterEL	Ordering of power products	IsMatrixGreater IsLexicalGreater IsPermutedLexicalGreater IsTotalDegreeInverseLexicalGreater IsTotalDegreeLexicalGreater IsPartialLexicalGreater
$NormalFormDNP	Normal form of a polynomial	PartialNormalForm CompleteNormalForm
$InterReducedPN	Interreduction of the basis	CompletelyInterReduced Identity (no interreduction)
$GroebnerBasisPN	Version of the Gröbner Basis algorithm	GroebnerBasisPN GroebnerBasisPNSmallPairSets GroebnerBasisPNAllReduced

Table 1: Global Variables

format with the conversion function C. Readable output is generated by the print function P.

1.1 An Example

Computing the Gröbner basis with the example input polynomial set stored in the file `Exmeaequ.m`.
Input file: $F = \{-x_1^2/2 - x_2^2/2 + x_3^2, x_1 * x_2 - 2 * x_3 + x_1 * x_3, x_1^2 - x_2\}$.

1) Load GRÖBNER:

```
In[1]:= <<Groebner.m
Groebner Package
```

2) Check (change) actual variant of the Gröbner bases algorithm:

```
In[2]:= $GreaterEL

Out[2]= IsPermutedLexicalGreater

In[3]:= $GreaterEL = IsTotalDegreeLexicalGreaterEL

Out[3]= IsTotalDegreeLexicalGreaterEL

In[4]:= $GreaterEL

Out[4]= IsTotalDegreeLexicalGreaterEL
```

```
In[5]:= $NormalFormDNP

Out[5]= CompleteNormalForm

In[6]:= $InterReducedPN

Out[6]= CompletelyInterReduced

In[7]:= $GroebnerBasisPN

Out[7]= GroebnerBasisPN
```

3) Load input file:

```
In[8]:= <<Exmeaequ.m

              2      2
           -x1     x2       2                              2
Out[8]= {---- - --- + x3  , x1 x2 - 2 x3 + x1 x3, x1   - x2}
           2      2
```

4) Convert the list to a "polynomial nest" of *Mathematica* polynomials:

```
In[9]:= F = C[ F, PN[ ListToNest, MP]]

                   2      2
                -x1     x2       2
Out[9]= PN[MP[---- - --- + x3 ],
                 2      2

                                                   2
>     PN[MP[x1 x2 - 2 x3 + x1 x3], PN[MP[x1   - x2], PN[]]]]
```

(For the data structures for polynomials used in the package see Section 2.5.)
5) Convert each polynomial in the nest into a distributive nest polynomial (this representation is used in the Gröbner bases algorithm):

```
In[10]:= F = C[F,PN[NestToNest,DNP[x1,x2,x3]]]

                           1
Out[10]= PN[DNP[Mon[-(-), EL[2, 0, 0]],
                           2

                    1
```

```
>      DNP[Mon[-(-), EL[0, 2, 0]], DNP[Mon[1, EL[0, 0, 2]], DNP[]]]],
                2

>     PN[DNP[Mon[1, EL[1, 1, 0]],

>       DNP[Mon[1, EL[1, 0, 1]], DNP[Mon[-2, EL[0, 0, 1]], DNP[]]]],

>      PN[DNP[Mon[1, EL[2, 0, 0]], DNP[Mon[-1, EL[0, 1, 0]], DNP[]]],

>     PN[]]]]

In[11]:= P[F]
Poly Nest
Distributive Nest Poly
-1/2
EL[2, 0, 0]
-1/2
EL[0, 2, 0]
1
EL[0, 0, 2]
End of Distributive Nest Poly
Distributive Nest Poly
1
EL[1, 1, 0]
1
EL[1, 0, 1]
-2
EL[0, 0, 1]
End of Distributive Nest Poly
Distributive Nest Poly
1
EL[2, 0, 0]
-1
EL[0, 1, 0]
End of Distributive Nest Poly
End of Poly Nest
```

6) Compute the Gröbner basis of F:

```
In[12]:= G = GB[ F]

Out[12]= PN[DNP[Mon[1, EL[0, 2, 0]],

>      DNP[Mon[-2, EL[0, 0, 2]], DNP[Mon[1, EL[0, 1, 0]], DNP[]]]],
```

```
>    PN[DNP[Mon[1, EL[1, 0, 1]],

                 1
>      DNP[Mon[-(-), EL[0, 1, 1]],
                 2

                                                    1
>       DNP[Mon[-1, EL[0, 0, 2]], DNP[Mon[-, EL[0, 1, 0]],
                                                    2

>         DNP[]]]]],

>     PN[DNP[Mon[1, EL[1, 1, 0]],

                1
>       DNP[Mon[-, EL[0, 1, 1]],
                2

>        DNP[Mon[1, EL[0, 0, 2]],

                    1
>         DNP[Mon[-(-), EL[0, 1, 0]], DNP[Mon[-2, EL[0, 0, 1]],
                    2

>           DNP[]]]]]],

>      PN[DNP[Mon[1, EL[2, 0, 0]], DNP[Mon[-1, EL[0, 1, 0]], DNP[]]],

>       PN[DNP[Mon[1, EL[0, 0, 3]],

>         DNP[Mon[-5, EL[0, 1, 1]],

>          DNP[Mon[12, EL[0, 0, 2]],

>           DNP[Mon[-2, EL[0, 1, 0]], DNP[Mon[-6, EL[0, 0, 1]],

>             DNP[]]]]]],

>        PN[DNP[Mon[1, EL[0, 1, 2]],

>          DNP[Mon[6, EL[0, 1, 1]],
```

```
>           DNP[Mon[-18, EL[0, 0, 2]],

>            DNP[Mon[3, EL[0, 1, 0]], DNP[Mon[8, EL[0, 0, 1]],

>              DNP[]]]]]],

>         PN[]]]]]]]
```

7) Convert polynomial nest into a list of *Mathematica* polynomials:

```
In[13]:= G = C[ G, MPL[ x1, x2, x3]]

                       2       2      x2             x2 x3     2
Out[13]= PL[MP[x2 + x2  - 2 x3 ], MP[-- + x1 x3 - ----- - x3 ],
                                      2                2

        -x2                   x2 x3     2         2
>    MP[--- + x1 x2 - 2 x3 + ----- + x3 ], MP[x1  - x2],
         2                      2

                                         2     3
>    MP[-2 x2 - 6 x3 - 5 x2 x3 + 12 x3  + x3 ],

                                        2         2
>    MP[3 x2 + 8 x3 + 6 x2 x3 - 18 x3  + x2 x3 ]]

In[14]:= P[G]
Poly List
       2       2
x2 + x2  - 2 x3
x2           x2 x3     2
-- + x1 x3 - ----- - x3
2              2
-x2                  x2 x3     2
--- + x1 x2 - 2 x3 + ----- + x3
 2                     2
  2
x1  - x2
                                2     3
-2 x2 - 6 x3 - 5 x2 x3 + 12 x3  + x3
                               2         2
3 x2 + 8 x3 + 6 x2 x3 - 18 x3  + x2 x3
```

```
End of Poly List
```

For a description of the many possibilities offered by the conversion function C and the print function P see Section 2.2.

2 Data Structures and Implementation Details

2.1 Lists and Nests

The two main data structures used in GRÖBNER-*Mathematica* are *Lists* and *Nests*. Lists over a domain D are lists in the sense of *Mathematica* (with arbitrary "tags" `tag` instead of `List`), i.e. objects of the form

```
tag[ e1,...,en],
```

where the `ei`'s are in D. Internally, lists are represented by arrays, and therefore some operations like Rest, Prepend, etc., which usually need constant time, are dependent on the length of the list. We introduce "nested lists" ("nests") to simulate the behavior of true "lists" in the usual sense of computer science. A nest over a domain D is an expression of the following form:

```
tag[ e1, tag[ e2, tag[ e3, ..., tag[ ] ...  ]]]  ,
```

where `tag` is an arbitrary "tag" and the `ei`'s are in D and represent the actual list elements. Conversion between the two formats is possible by the conversion function C, for details see Section 2.2.1.

2.2 Conversion and Printing

2.2.1 Conversion

Based on the structure of lists and nests as described in Section 2.1, conversion is possible from list to list, list to nest, nest to list and nest to nest. Furthermore, all entries of a list (nest) can be transformed into another representation. In general, the pattern of a call to the conversion function C is as follows:

```
C[ element, newTag[ direction, elementConversion]] ,
```

where `element` is the element to be converted, `newTag` is the tag of the resulting list (nest), the possible choices for `direction` are "ListToList", "ListToNest", "NestToList", "NestToNest" and "TupleToTuple". `elementConversion` determines the conversion of the individual entries of the list (nest), i.e. each entry e_i is converted into e_i', where $e_i' = \mathtt{C}[e_i, \mathtt{elementConversion}]$. Thus, the conversion parameters may handle the recursive layers of data. When using "TupleToTuple", `elementConversion` can be specified for each element, whereas in the other cases all elements are transformed uniformly.

Example: To convert a list of polynomials in our nest representation into a nest of polynomials in *Mathematica* representation use the following conversion (ListOfDNPs.m is a file with an example list of polynomials):

```
In[1]:= L = <<ListOfDNPs.m

Out[1]= {DNP[Mon[1, EL[3, 1, 1]], DNP[Mon[-1, EL[1, 0, 2]], DNP[]]],

>    DNP[Mon[1, EL[1, 2, 1]], DNP[Mon[-1, EL[1, 1, 1]], DNP[]]]}

In[2]:= C[ L, PN[ ListToNest, MP[x,y,z]]]

                     3          2                        2
Out[2]= PN[MP[x  y z - x z ], PN[MP[-(x y z) + x y  z], PN[]]]
```

In this example the tag of the resulting nest is `PN` (Polynomial Nests), the conversion is from list to nest and each polynomial in DNP representation is converted with the parameter `MP[x,y,z]`, which causes transformation into a *Mathematica* polynomial in three variables, namely `x`, `y` and `z`. See Table 2 for a complete listing of all conversions implemented.

2.2.2 Printing

The actual value of a variable can be displayed by just entering the name of the variable. However, more complicated nested objects are almost unreadable in this format. The print function P can be used to generate readable output. Furthermore, it offers the possibility to vary the grain of the produced output reaching from coarse grain, which mainly describes the object's structure, to fine grain, which prints the whole object. For example, a coarse-grained output of a polynomial in nest representation only contains the number of monomials, whereas a fine-grained output prints the entire polynomial.

The grain of the output is specified by several print options, which can be given as additional parameters to the print function P. If no option is specified P produces fine-grained output. Information about all possible options can be obtained by the parameter "Q". When added in addition to another parameter, the option "Q" causes P to explain the parameter.

Example: Think of G to be the result of a Gröbner bases computation. The example shows how one one would typically inspect the result:

```
In[1]:= G = GB[F]

Out[1]= PN[DNP[Mon[1, EL[0, 2, 0]],

>     DNP[Mon[-2, EL[0, 0, 2]], DNP[Mon[1, EL[0, 1, 0]], DNP[]]]],

>    PN[DNP[Mon[1, EL[1, 0, 1]],
```

Element	Result	Option
Mathematica float	float (our reresentation)	FP
float (our rep.)	*Mathematica* float	IFP
Mathematica number	finite field element (our rep.)	FF
finite field element (our rep.)	*Mathematica* number	IFF
Mathematica rational function	rational function (our rep.)	MRF
rational function (our rep.)	*Mathematica* rational function	IMRF
power product pp	pp with permuted variables	VarPerm[*perm*]
power product	*Mathematica* polynomial	MP[*variables*]
list	Gödel exponent	GE
Gödel exponent	list	IGE
monomial m	m with permuted variables	VarPerm[*perm*]
monomial	*Mathematica* polynomial	MP[*variables*]
nested polynomial	nested polynomial with permuted variables	VarPerm[*perm*]
nested polynomial	*Mathematica* polynomial	MP[*variables*]
nested polynomial	list polynomial	LP
list polynomial	nested polynomial	DNP
nested polynomial	polynomial over rational functions	RFDNP[...] See Section 3
expression	polynomial in *Mathematica* representation	MP
polynomial in *Mathematica* representation	expression	IMP
polynomial in *Mathematica* representation	nested polynomial	DNP[*variables*]
polynomial	labeled polynomial	Label
labeled polynomial	polynomial	ILabel
labeled polynomial	labeled polynomial with permuted variables	VarPerm[*perm*]
list of polynomials	nest of polynomials	DNPN[*variables*]
polynomial nest	polynomial nest with permuted variables	VarPerm[*perm*]
polynomial nest	list of polynomials in list representation	LPL
polynomial nest	list of polynomials in *Mathematica* representation	MPL
polynomial nest	nest of polynomials over rational functions	RFDNPN[...] See Section 3

Table 2: Conversion Functions (Pattern: Result = C[Element, Option])

```
                  1
>       DNP[Mon[-(-), EL[0, 1, 1]], ...... DNP[]]]]]], PN[]]]]]]]
                  2
```

1) The result in its internal representation is hard to read and therefore one uses the print function P:

```
In[2]:= P[G,Q]
PN Print options: N, S, P, C, Sh, D
```

We first give some examples for using these options. See Table 3 for a complete listing of all implemented print options.
2) After finding out the number of polynomials in the nest one can have a look at the structure of the polynomial nest:

```
In[3]:= P[G,N]
Number of polynomials: 6

In[4]:= P[G,S]
Leading PP:
EL[0, 2, 0]
Maximal Coefficient Size: 20;  Number of Monomials: 3
Leading PP:
EL[1, 0, 1]
Maximal Coefficient Size: 68;  Number of Monomials: 4
Leading PP:
EL[1, 1, 0]
Maximal Coefficient Size: 68;  Number of Monomials: 5
Leading PP:
EL[2, 0, 0]
Maximal Coefficient Size: 20;  Number of Monomials: 2
Leading PP:
EL[0, 0, 3]
Maximal Coefficient Size: 20;  Number of Monomials: 5
Leading PP:
EL[0, 1, 2]
Maximal Coefficient Size: 20;  Number of Monomials: 5
```

3) If one is interested in a particular polynomial of the nest one can study this polynomial in more detail:

```
In[5]:= p1 = FirstPolynomial[G]

Out[5]= DNP[Mon[1, EL[0, 2, 0]],
```

```
>    DNP[Mon[-2, EL[0, 0, 2]], DNP[Mon[1, EL[0, 1, 0]], DNP[]]]]

In[6]:= P[p1,Q]
DNP Print Options: N, S, P, C, Sh, D

In[7]:= P[p1,N]
Number of monomials: 3

In[8]:= P[p1,P]
Distributive Nest Poly: Only Power Products
EL[0, 2, 0]
EL[0, 0, 2]
EL[0, 1, 0]
End of Distributive Nest Poly

In[9]:= lm1 = LeadingMonomial[p1]

Out[9]= Mon[1, EL[0, 2, 0]]

In[10]:= P[lm1,C,MP[x1,x2,x3]]

CS[20]
  2
x2

In[11]:= P[p1]
Distributive Nest Poly
1
EL[0, 2, 0]
-2
EL[0, 0, 2]
1
EL[0, 1, 0]
End of Distributive Nest Poly

In[12]:= cp1 = C[p1,MP[x1,x2,x3]]

                    2       2
Out[12]= MP[x2 + x2  - 2 x3 ]

In[13]:= P[cp1]
       2       2
```

Input for P	Options	Output
anything	Domain Tag Type	tag of *anything* tag of *anything* tag of *anything*
monomial	C C,MP[*variables*]	coefficient of the monomial is just given by its size in bytes coefficient in bytes, power product in *Mathematica* format
nested polynomial	N S P P,MP[*variables*] C C,MP[*variables*] Sh,MP[*variables*] D	number of monomials structure information all power products all power products in *Mathematica* format each monomial is printed with option C each monomial is printed with options C,MP[*variables*] result of *Mathematica* internal function "Short" domain of coefficients
labeled polynomial	Label see "nested polynomial"	label of polynomial see nested polynomials
nest of polynomials	see "nested polynomial"	each polynomial is printed with the corresponding options
nest of pairs of polynomials	N	number of pairs

Table 3: Print options

```
x2 + x2  - 2 x3
```

4) This procedure can then be repeated for every polynomial, from which one wants to have detailed information.

2.3 Principles of Generic Programming in *Mathematica*

"Generic" programming is important for algorithms in hierarchical towers of algebraic domains. For example, in GRÖBNER-*Mathematica* we want to formulate one particular version of the Gröbner bases algorithm for many different coefficient domains (rational numbers, rational functions, finite fields etc.) and various representations of power products (representation as list of exponents, representation as "Gödel numbers" etc.). Genericity in programming means that an algorithm can be formulated once and for all independently of the particular domains over which it should work.

For realizing a generic style of programming in GRÖBNER-*Mathematica* we used the technique described briefly in [Wolfram 88], pp. 208 - 211. We give here more details about this approach because it might be useful also for other algebraic packages.

In fact, the two essential characteristic features of *Mathematica* in their combination yield a very natural realization of generic programming. These two features are:

- *Mathematica*'s fundamental data type is the "expression", which models both nested data and nested function descriptions and encompasses the expression encountered in ordinary mathematics in a very natural way.

- In function definitions, *Mathematica* allows arguments that are "patterns" (i.e. terms) and not only variables.

For example, any selection of the ordinary mathematical rules for "lim" would immediately yield an executable *Mathematica* program:

```
Limes[a_+ b_] := Limes[a] + Limes[b]
Limes[a_ b_] := Limes[a]  Limes[b]
```

Every *Mathematica* expression that matches the "pattern" `Limes[ a_+ b_]`, where `a_` and `b_` stand for arbitrary expressions, would be transformed by the *Mathematica* interpreter according to the first rule. In fact, `Limes[ a_+ b_]` is an abbreviation for `Limes[ Plus[ a_, b_]]`. Even, an entire rule like `Limes[a_+ b_] := Limes[a] + Limes[b]` is in fact a single expression, which in "full form" would be

```
Define[ Limes[ Plus[ Blank[ a], Blank[b]]],
    Plus[ Limes[ a], Limes[ b]]]
```

Generally speaking, except for some atoms, the only data items in *Mathematica* are expressions of the form $f[e_1, \ldots, e_n]$, where $f, e_1, \ldots, e_n$ are again expressions.

This simple concept of "expression and matching" is very powerful. We will now show that it incorporates, in a very natural way, the concept of "generic programming".

The main point how generic programming can be incorporated in *Mathematica* programs is the fact that the f in a *Mathematica* expression $f[e_1, \ldots, e_n]$ can be used as a "tag" for characterizing a data domain. Objects having two different (constant) tags `T1` and `T2` will automatically be analyzed to belong to two different domains and, accordingly, two different sets of rules may be installed for the same operation `Operation`:

```
Operation[ T1[ ...]]  := first right-hand side,
Operation[ T2[ ...]]  := second right-hand side.
```

When finding an expression of the form `Operation[` *expr*`]`, the *Mathematica* interpreter analyzes the expression *expr* and depending on whether *expr* starts with `T1` or `T2` applies the first rule or the second. This simple mechanism can be used to create "generic packages" that handle complicated "towers" of algebraic domains without repetition of code.

One way of organizing the many possible "towers" of domains for GRÖBNER-*Mathematica* is as follows: The top-most domain is the domain of distributive polynomials which I chose to represent in the following "nested" form:

> `DNP[` *m, dnp*`]`,
> where m is a monomial and
> *dnp* is again a distributive polynomial.
> (`DNP[ ]` is the "empty" (zero) polynomial.)

The monomials might be represented in the following form

> `Mon[` *c*, *pp*`]`,
> where c is an element of a coefficient domain and
> *pp* is an element of a power product domain.

We are interested in many different coefficient domains: the built-in rational numbers, the finite fields GF(p), the field of rational functions etc. For example, finite field elements in GF(p) might be represented by

> `FF[` *f*`]`,

where f is a number modulo a prime p, and rational functions may be represented by

> `MRF[` *rf*`]`,

where *rf* might be a rational function in the built-in *Mathematica* representation.

Finally, the power products may be represented as "exponent lists" in the form

> `EL[` $e_1, \ldots, e_n$`]`,

where the $e_1, \ldots, e_n$ are the exponents at the n indeterminates. Alternatively, one may be interested in a "Gödel coding" of the exponents $e_1, \ldots, e_n$ by the natural number $p_1^{e_1} \cdots p_n^{e_n}$, where the $p_1, \ldots, p_n, \ldots$ are the prime numbers. A corresponding representation in *Mathematica* may be

> `GE[` *ge*`]`,

where *ge* is a natural number.

Now we show some parts of the corresponding *Mathematica* code for realizing arithmetic on these domains:

Addition for `DNP`*-polynomials*:

```
dnp_DNP + DNP[] := dnp

DNP[ m1_, dnp1_] + DNP[ m2_, dnp2_] :=
     DNP[ m1, dnp1 + DNP[ m2, dnp2]] /; m1 > m2
```

```
...

DNP[ m1_, dnp1_] + DNP[ m2_, dnp2_] :=
     DNP[ m1 + m2, dnp1 + dnp2] /; IsEquivalent[ m1, m2]
                                   && m1 != -m2
     ...
```

(A rule containing a variable like `dnp_DNP` on the left-hand side must be read as follows: If an argument `dnp` whose tag is `DNP` is encountered then the rule is applied). Note that the "`+`" on the left-hand side of these rule denotes the addition specific for `DNP`-polynomials whereas the "`+`" on the right-hand side is "generic" in the sense that, at run-time, the objects `m1, m2, dnp1, dnp2` etc. are analyzed and, in dependence on their "tag" (in this case "`Mon`" or "`DNP`"), the appropriate rule of the package is selected and applied. Similarly, in this example, also ">", "−", and "`IsEquivalent`" are "generic". (In fact, the actual code in the file `DNP.m` is a little bit different from the one shown above because the above "merging" process that realizes addition of polynomials is formulated once and for all for arbitrary "nests" in the file `Nests.m`. A similar remark applies to the subsequent examples of code.)

Some operations on `Mon`*-monomials*:

```
Mon[ c1_, pp1_] > Mon[ c2_, pp2_] := pp1 > pp2
Mon[ c1_, pp_] + Mon[ c2_, pp_] := Mon[ c1 + c2, pp]
IsEquivalent[ Mon[ c1_, pp1_], Mon[ c2_, pp_]] := pp1 == pp2
```

The operations ">", "+", and "`IsEquvialent`" appearing on the left-hand side of these definitions are the specific realizations for `Mon`-polynomials of the generic operations used in the above domain of `DNP`-polynomials. The operations ">" and "+" on the right-hand side are again "generic".

Let us consider one more "layer" in this example of a "tower of algebraic domains".

">" on `EL`*-power product*:

```
el1_EL > el2_EL :=
   Block[ i, ...,
      For[ i = 1, ...
          ...
          If[ el1[[ i]] > el2[[ i]],
              ...
              ]
          ]
      ]
```

">" on `GE`*-power products*:

```
GE[ e1_> GE[ e2_] := e1 > e2
```

In these two definitions, again, "$>$" on the left-hand side denotes the operation specific for the domain of `EL`-power products and `GE`-power products, respectively. "$>$" on the right-hand side denotes the corresponding generic operation. Typically, the domain of natural numbers will be used as the substitute for these generic domains. However, it is well conceivable that other domains are used as exponent domains. For example, it may turn out that "symbolic exponents" (i.e. polynomial or rational expressions) are an interesting exponent domain.

Addition on rational number coefficients: Arithmetic on these numbers is built-in. In fact, these numbers have the internal tag `Rational`.

Addition on finite field elements:

```
FF[ f1_] + FF[f2_] := FF[ Modul[ f1 + f2, $Prime]]
```

Addition on built-in rational functions:

```
RF[ rf1_] + RF[ rf2_] := RF[ Factor[ rf1 + rf2]]
```

In the last two examples, again, "+" on the left-hand side is specific for the domains "`FF`" and "`RF`", respectively, whereas "+" on the right-hand side is generic. Typically, in these cases, the generic rule will only be applied to the domain of integers and the domain of built-in rational function expressions, respectively.

On top of the `DNP`-polynomials, a number of higher levels in the generic domain hierarchy are constructed in GRÖBNER-*Mathematica*, for example, the domain of sets of polynomials, pairs of polynomials, sets of pairs of polynomials etc.

A package that is constructed according to the above principle can then be used for a huge variety of domain combinations. At run-time, the *Mathematica* pattern matcher will analyze the tags of the data expressions and select the appropriate rules in the rule base (= "program"). Roughly, if in a package with m "layers" of domains there are n domains in each layer then the package can be used for n^m many concrete domains although there are only $m \cdot n$ many pieces of code!

2.4 Hierarchy of Data Structures

The various data types available in GRÖBNER-*Mathematica* can be recognized and distinguished by their tags. A hierarchy of data types is established as follows (from high-level to low-level):

Pair Nests (Nests of pairs of polynomials; tag: PaN)

$\downarrow$

Poly Pairs (Pairs of polynomials; tag: PPa)

$\downarrow$

Poly Nests (Nests of polynomials; tag: PN),
Poly Lists (Lists of polynomials; tag: PL)

$\downarrow$

Labeled Polynomials (Polynomials with a label; tag: LabP)

$\downarrow$

Mathematica internal polynomials (tag: MP),
List Polys (polynomials in list-representation; tag: LP),
Distributive Nest Polys (polynomials in nest-representation; tag: DNP)

$\downarrow$

Monomials (tag: Mon)

$\downarrow$

Exponent Lists (tag: EL),
Gödel Exponents (tag: GE),
Rational Numbers (built-in *Mathematica* Rational Numbers, tag: Rational),
Floating Point numbers with discrete zero (tag: FP),
Complex Numbers (built-in *Mathematica* Complex Numbers, tag: Complex),
Finite Fields (tag: FF),
Rational Functions (built-in *Mathematica* Rational Functions; tag: MRF)

$\downarrow$

Nests, Lists

2.5 Implementation Details

As already mentioned in [Buchberger 91] and Section 2.1, *Mathematica* lists do not have the complexity behavior of "real lists" because of their internal representation by arrays. For example, computing `Rest[ l]` (comparable to `cdr` in LISP), `Prepend[ l, a]` (comparable to `cons` in LISP) ..., where `l` is an arbitrary list and `a` is any element, depends on the length of `l` whereas this would be a constant-time operation for "real lists". In the case of Gröbner basescomputation this is a serious drawback because of the heavy usage of such operations, e.g. `RemainingPolynomial`, `RemainingPolynomials`, `RemainingPairs`, building up the list of pairs, from which S-Polynomials have to be computed etc. To overcome this disadvantage we introduce the data structure *"nested lists"* (short: *nests*) by which we can simulate the behavior of lists. The recursive structure of nests is shown in Section 2.1.

The file `Lists.m` contains some operations on lists, which are needed as a consequence of viewing every structure of the form

```
tag[ e1,...,en]
```

as a list and not just those whose tag is `List`. All these "list operations" are available for nests, too, and have the suffix N in this case. As an example, `Length` yields the length of a list whereas `LengthN` yields the length of a nest, i.e. the depth of the nest.

The file `Fields.m` contains all the fields, for which arithmetic is implemented up to now, such as rational numbers, floating point numbers, finite fields and rational functions. These fields can be used as coefficient domains for polynomials. Arithmetic in finite fields is implemented as arithmetic for integers modulo the prime number, which is stored in the global variable `$Modul` and can be set by the function `SetModul`. All other fields are the *Mathematica* built-in fields except that an additional tag is put in front such that they fit into the scheme for generic programming as described in Section 2.3.

The file `PowerProducts.m` contains operations on power products. Power products can either be represented as *"exponent lists"* or as *"Gödel exponents"*. Exponent Lists are lists of the form `EL[e1,...,en]`, representing the power product $x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$. In this case various term orderings can be used, such as purely lexical ordering and total degree lexical ordering. The most general form is the ordering of terms by a matrix that is stored in the global variable `$OrderMatrix`. For details on this kind of ordering see [Robbiano 85] and [Windsteiger 90]. The power product $x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$ can also be represented by its "Gödel exponent", i.e. the integers $p_1^{e_1} \cdot \ldots \cdot p_n^{e_n}$, where p_i is the i-th prime number. Integer ordering on "Gödel exponents" is used as term ordering in this case.

The file `Monomials.m` contains operations on monomials that are represented as lists of the form `Mon[ c, pp]`, where `c` is from the coefficient domain and `pp` is from the power product domain. The file `DNP.m` contains polynomials as used in the Gröbner bases algorithm are represented as nests of monomials. The monomials in a polynomial are sorted such that the first monomial is the greatest with respect to the ordering stored in `$GreaterEL`. Two normal form algorithms for reduction of a polynomial with respect to a set of polynomials are implemented:

- `PartialNormalForm` checks whether the leading monomial of the polynomial is a multiple of a leading monomial of a polynomial in the basis. If this is the case, the leading monomial is deleted and the process is continued recursively with the remaining part of the polynomial.

- `CompleteNormalForm` replaces all monomials that are multiples of a leading monomial of a polynomial in the basis. `CompleteNormalForm` is implemented by an iterative application of `PartialNormalForm`.

For details see file `DNP.m` in Section 3.

In order to provide easier input, the file `ListPolys.m` contains a conversion function from a polynomial represented as a list of monomials into a nested representation. For generating readable output and for using *Mathematica* internal polynomial arithmetic, the file `MathematicaPolys.m` contains operations on the data type `MathematicaPolys`, which is a list of the form `MP[ mp]`, where `mp` is *Mathematica* internal polynomial. For generating a *Mathematica*-like output of a nested polynomial one can convert it into a `MathematicaPoly` and then print it with the print function P.

The file `LabeledPolys.m` contains arithmetic for labeled polynomials, which are polynomials with an integer label for identification. This representation is used if the Chain Criterion is applied during Gröbner bases computation.

For easier input of polynomials the file `PolyLists.m` provides a representation of polynomial sets as lists of polynomials. However, the representation used in the Gröbner bases algorithm is the one given in the file `PolyNests.m`, i.e. a representation of the set as a nest where the polynomials are ordered with respect to ascending leading monomials. In addition, the subalgorithms for interreduction of the basis used in the various versions of the Gröbner bases algorithm can be found in this file. Operations on pairs of polynomials are included in the file `PolyPairs.m`. Operations for pair sets that are represented as nests of pairs and the subalgorithms required in the version of the Gröbner bases algorithm "GroebnerBasisPNSmallPairSets" can be found in the file `PairNests.m`.

All versions of the Gröbner bases algorithm use the criteria for detection of unnecessary reductions of S-polynomials and an order of selection of the pairs for which S-polynomials have to be formed: pairs whose least common multiple of the leading monomials is minimal with respect to the chosen ordering are treated at the beginning. "GroebnerBasisPN" uses no additional knowledge, "GroebnerBasisPNSmallPairSets" applies the criteria immediately when generating pairs after adding a new polynomial to the basis, which keeps the set of pairs as small as possible, and "GroebnerBasisPNAllReduced" is the version equivalent to Algorithm 6.3 as described in [Buchberger 85]. These three versions are stored in the files `PolyNestsGB.m`, `PolyNestsGBSmallPairSets.m` and `PolyNestsGBAllReduced.m`, respectively.

2.6 Experience With Using GRÖBNER-*Mathematica*

After extensive computational experiments with GRÖBNER-*Mathematica*

- we found that *Mathematica* is a great language for fast and well-structured implementation of algebraic packages,
- but we were very disappointed with the computational speed of the system, which makes its use for examples of serious size prohibitive.

We documented our experience in [Buchberger 91].

However, for doing research by considering small examples we found that the package is quite useful, though. Each of the many functions we implemented in GRÖBNER-*Mathematica* (for example, `PartialNormalForm, SPolynomial`, etc.) can be called interactively and individually. Intermediate results can be inspected by the function `P` with an additional parameter that determines the "granularity" of the output. And, of course, it is very easy to modify the system because of the easy-to-read code and the generic structure. As a result, we were able to pursue some worthwhile research on Fermat's ideal by using GRÖBNER-*Mathematica* for small input, see [Buchberger 90]. The package may be useful for similar applications where the emphasis lies on the interactive study of small examples and not on fast processing of examples of large size.

For the study of examples of large size we decided to implement the Gröbner bases method in the frame of Collins' SAC-2 algorithm library in its new C version, see [Buchberger, Collins, Hong 91].

3 Source Code

```
Print[ "Groebner Package"]

(* Additional directories for searching external files: *)

$Path = Append[ $Path, "//tanne/user_data/groebner/versionrew/InOutput"]

(* Systems Parameters: *)

$RecursionLimit = Infinity

(* Initialize a Default Version: *)

$GreaterEL = IsLexicalGreater
$GreaterPPa = IsTotalDegreeLexicalGreater
$NormalFormDNP = CompleteNormalForm
$InterReducedPN = CompletelyInterReduced
$GroebnerBasisPN = GroebnerBasisPNAllReduced

(* Load Files: *)

<<Objects.m
<<Lists.m
<<Nests.m
<<Fields.m
<<PowerProducts.m
<<Monomials.m
<<DistributiveNestPolys.m
<<ListPolys.m
<<MathematicaPolys.m
<<LabeledPolys.m
<<PolyLists.m
<<PolyNests.m
<<PolyPairs.m
<<PairNests.m
<<PolyNestsGB.m
<<PolyNestsGBSmallPairSets.m
<<PolyNestsGBAllReduced.m
<<PrintGlobalVariables.m

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain ‘‘Objects’’

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

Objects are arbitrary Mathematica Expressions.

Operations on Objects:
```

```
---------------------
*)

Type[ object_] := Head[ object]

Typed[ object_, type_] :=
  Block[{ typedObject = object},
    typedObject[[ 0]] = type;
    typedObject
    ]

Sorted[ x_] := Sort[ x]

P[ object_, Domain] := Print[ Type[ object] ]

P[ object_, Tag] := Print[ Type[ object] ]

P[ object_, Type] := Print[ Type[ object] ]

P[ st_String] := Print[ st]

Unprotect[ C]

C[ object_, Identity] := object

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain ``Lists''

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

   Lists over a domain D  are lists
   in the sense of MATHEMATICA
   (with arbitrary tags instead of List),
   i.e. objects of the form tag[ e1, ..., en],
   where the ei are in D.

*)

ComponentWise[ function_, l_] :=
  Map[ function, l]

ComponentWise[ function_, l1_, l2_] :=
  Block[{ i, l0},
    l0 = Table [function [l1 [[i ]], l2[[ i]]],
          {i, Length[ l1]}
          ];
    Typed[ l0, Type[ l1]]
    ]

DoForAll[ l_, function_] :=
  Scan[ function, l]
```

```
ForAll[ l_, predicate_] :=
  Block[{ i},
    For[ i=1, i <= Length[ l], i++,
      If[ !predicate[ l[[ i]]],
        Return[ False],
        Continue[ ]
        ]
      ];
    Return[ True]
    ]

ForAll[ l1_, l2_, predicate_] :=
 Block[{ i},
    For[ i=1, i <= Length[ l1], i++,
      If[ !predicate[ l1[[ i]], l2[[ i]]],
        Return[ False],
        Continue[ ]
        ]
      ];
    Return[ True]
    ]

IsEmpty[ tag_ [ ]] := True
IsEmpty[ tag_ [ elements__]] := False

Delete[ l_, n_] :=
  Block[{  l0},
    l0 =
      Join[ Take[ l, n-1],
       Take[ l, {n+1, Length[ l]}]
       ];
    Typed[ l0, Type[ l]]
    ]

Changed[ l_, position_, element_] :=
  Block[{ l0 = l}, l0[[ position]] = element; l0]

IsLexicalGreater[ l1_, l2_] :=
 Block[{ i, e1, e2},
    For[ i=1, i <= Length[ l1], i++,
      e1 = l1[[ i]]; e2 = l2[[ i]];
      If[ e1 != e2, Return[ e1 > e2], Continue[ ]]
      ];
    Return[ False]
    ]

IsPermutedLexicalGreater[ l1_, l2_, permutation_] :=
 Block[{ i, e1, e2},
    For[ i=1, i <= Length[ l1], i++,
      e1 = l1[[ permutation[[ i]] ]];
```

```
      e2 = l2[[ permutation[[ i]] ]];
      If[ e1 != e2, Return[ e1 > e2], Continue[ ]]
      ];
    Return[ False]
    ]

Permuted[ l_, permutation_] :=
  Block[{ l0, i},
    l0 = Table[ l[[ permutation[[ i]] ]], {i, Length[ l]}];
    Typed[ l0, Type[ l]]
    ]

Inverted[ permutation_] :=
  Block[{ p0, i},
    p0 = Table[ Position[ permutation, i] [[ 1, 1]],
      {i, Length[ permutation]}
      ];
    Typed[ p0, Type[ permutation]]
    ]

P[ l_, List] :=
  DoForAll[ l, P[ #] &]

P[ l_, List, function_] :=
  DoForAll[ l, P[ function[ #]] &]

C[ tuple_, newTag_ [ TupleToTuple, elementTags___]] :=
  (* Convert tuple = oldtag[ e1, ..., en] into
     newTag[ e1', ..., en'],
     where  ei' = C[ ei, elementTags[[i]]].
  *)
  Block[{ t, i, et = newTag[ Tuple, elementTags]},
    t =
      Table[ C[ tuple[[ i]], et [[ i+1]] ],
        {i, Length[ tuple]}
        ];
    Typed[ t, newTag]
    ]

C[ list_, newTag_[ ListToList, elementTag_]] :=
  (* Convert list = oldtag[ e1, ..., en] into
     newTag[ e1', ..., en'],
     where ei' = C[ ei, elementTag].
  *)
  Block[{ l, i},
    l =
      Table[ C[ list[[ i]], elementTag], {i, Length[ list]}];
    Typed[ l, newTag]
    ]
```

```
C[ list_, newTag_[ ListToNest, elementTag_]] :=
  (* Convert list = oldtag[ e1, ..., en] into
     newTag[ e1', newTag[ e2', ... newTag[] ... ]],
     where ei' = C[ ei, elementTag].
  *)
  newTag[ C[ First[ list], elementTag],
    C[ Rest[ list], newTag[ ListToNest, elementTag]]
    ]

C[ oldTag_[ ], newTag_[ ListToNest, elementTag_]] :=
  newTag[ ]

C[ list_, tag_, ListToNest] :=
  C[ list, tag[ ListToNest, Identity]]

C[ list_, ListToNest] := C[ list, T, ListToNest]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor ``tag-Nests''

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

tag-Nests over a domain D are expressions
of the following form:

    tag []  or

    tag [ e, n]

       where e is an object from D and
       n is again a tag-Nest.

Example:

    T[ e1, T[ e2, T[]]]

    is a T-Nest.

*)

ComponentWiseN[ function_, tag_[ e_, n_]]:=
  (* Result of ComponentWise application of
    function to all elements in a tag-Nest.
    *)
  tag[ function[ e], ComponentWiseN[ function, n]]

ComponentWiseN[ function_, tag_[ ]] :=
  tag[ ]

DoForAllN[ tag_[ e_, n_], function_] :=
  Block[ {}, function[ e]; DoForAllN[ n, function]]
```

```
DoForAllN[ tag_[], function_] := Null

ForAllN[ tag_[ e_, n_], predicate_] :=
  predicate[ e] && ForAllN[ n, predicate]

ForAllN[ tag_[], predicate_] := True

ForAllN[ tag_[ e1_, n1_], tag_[ e2_, n2_], predicate_] :=
  predicate[ e1, e2] && ForAllN[ n1, n2, predicate]

ForAllN[ tag_[], tag_[ e2_, n2_], predicate_] := False

ForAllN[ tag_[ e1_, n1_], tag_[], predicate_] := False

ForAllN[ tag_[], tag_[], predicate_] := True

ExistsN[ tag_[ e_, n_], predicate_] :=
  predicate[ e] || ExistsN[ n, predicate]

ExistsN[ tag_[], predicate_] := False

EqualN[ n1_, n2_] := ForAllN[ n1, n2, Equal]

IsEmptyN[ tag_[ e_, n_]] := False

IsEmptyN[ tag_[ ]] := True

EmptyN[ tag_[ e___]] := tag[ ]

PrependN[ tag_[ e0___], e_] := tag[ e, tag[ e0]]

FirstN[ tag_[ e_, n_]] := e

RestN[ tag_[ e_, n_]] := n

LengthN[ tag_[ e_, n_]] := LengthN[ n] + 1

LengthN[ tag_[ ]] := 0

ReverseN[ n_] := RevN[ n, EmptyN[ n]]

RevN[ tag_[ e_, n_], newn_] := RevN[ n, tag[ e, newn]]

RevN[ tag_[], newn_] := newn

PartN[ nest_, n_Integer] :=
  PartCountN[ nest, n, 1]

PartCountN[ tag_[element_, nest_], n_, actual_] :=
  PartCountN[ nest, n, actual+1]

PartCountN[ tag_[element_, nest_], n_, n_] :=  element

PartCountN[ tag_[], n_, actual_] :=
  "PartN: n outside range of nest."

PartN[ nest_, l_List] :=
  Block[ {i, nest0 = EmptyN[ nest]},
    For[ i=1, i<=Length[ l], i++,
      nest0 = PrependN[ nest0,
```

```
              PartN[ nest, l[[i]]]
              ]
      ];
    ReverseN[ nest0]
    ]

RangeN[ n_, tag_] :=
  ReverseN[ DescendingRangeN[ n, tag]]

DescendingRangeN[ n_, tag_] :=
  tag[ n, DescendingRangeN[ n-1, tag]]

DescendingRangeN[ 0, tag_] := tag[ ]

RangeN[ n_] := RangeN[ n, T]

DescendingRangeN[ n_] := DescendingRangeN[ n, T]

DeleteN[ tag_[ e_, n_], predicate_] :=
  If[ predicate[ e],
    DeleteN[ n, predicate],
    tag[ e, DeleteN[ n, predicate]]
    ]

DeleteN[ tag_[], predicate_] := tag[ ]

ComplementarySubSetsN[ n_, predicate_] :=
  ComplementarySubSetsN$[ Type[ n] [], Type[ n] [], n, predicate]

ComplementarySubSetsN$[
  nPositive_, nNegative_, tag_[ e_, n_], predicate_
  ] :=
  If[ predicate[ e],
    ComplementarySubSetsN$[ tag[ e, nPositive],
      nNegative, n, predicate
      ],
    ComplementarySubSetsN$[ nPositive,
      tag[ e, nNegative], n, predicate
      ]
    ]

ComplementarySubSetsN$[
  nPositive_, nNegative_, tag_[ ], predicate_
  ] :=
  { ReverseN[ nPositive], ReverseN[ nNegative]}

InsertN[ e_, tag_ [ e0_, n_], predicate_] :=
  (* tag-Nest with e inserted at first position where
     predicate holds or at last position if predicate
     never holds.
     *)
  If[ predicate[ e, e0],
```

```
    tag[ e, tag[ e0, n]],
    tag[ e0, InsertN[ e, n, predicate]]
    ]
InsertN[ e_, tag_ [ ], predicate_] :=
  tag[ e, tag[ ]]
IsMemberInN[ e_, tag_[ e0_, n_], predicate_] :=
  If[ predicate[ e, e0],
    True,
    IsMemberInN[ e, n, predicate]
    ]
IsMemberInN[ e_, tag_[ ], predicate_] := False
MergeN[
  tag_ [ e1_, n1_], tag_ [ e2_, n2_],
    isFirstGreater_, isSecondGreater_, areComplementary_,
    function_
    ] :=
  (* Assumption: Input tag-Nests are
     sorted with greatest element first.
     *)
  Which[
     isFirstGreater[ e1, e2],
     tag[
       e1,
       MergeN[ n1, tag[ e2, n2],
         isFirstGreater, isSecondGreater, areComplementary, function
         ]
       ],
     isSecondGreater[ e2, e1],
     tag[
       e2,
       MergeN[ tag[ e1, n1], n2,
         isFirstGreater, isSecondGreater, areComplementary, function
         ]
       ],
     areComplementary[ e1, e2],
     MergeN[ n1,  n2,
       isFirstGreater, isSecondGreater, areComplementary, function
       ],
     True,
     tag[
       function[ e1, e2],
       MergeN[ n1, n2,
         isFirstGreater, isSecondGreater, areComplementary, function
         ]
       ]
```

```
    ]
MergeN[ tag_ [ ], n2_,
  isFirstGreater_, isSecondGreater_, areComplementary_, function_
  ] :=  n2
MergeN[ n1_, tag_ [ ],
  isFirstGreater_, isSecondGreater_, areComplementary_, function_
  ] :=  n1
MergeN[ n1_, n2_, greater_] :=
  (* Assumption: Input tag-Nests are
     sorted with greatest element first.
     *)
  MergeN[ n1, n2,
    (greater[#1,#2] || #1 == #2)&, greater, (False)&, (Null)&
    ]
UnionN[ n1_, n2_, greater_] :=
  (* Assumption: Input tag-Nests are
     sorted with greatest element first
     and have no multiple occurrences of
     elements.
     *)
  MergeN[ n1, n2, greater, greater, (False)&, (#1)&]
SortN[  n_, greater_] :=
  Block[ { l, s},
    l = C[ n, NestToList];
    s = Sort[ l, greater];
    C[ s, Type[ n], ListToNest]
    ]
P[ n_, Nest] :=
  DoForAllN[ n, P[ #] &]
P[ n_, Nest, function_] :=
  DoForAllN[ n, P[ function[ #]] & ]
C[ oldTag_[ e1_, restOfNest_], newTag_[ NestToList, elementTag_]] :=
  (* Convert oldTag[ e1, oldTag[ e2, ... oldTag[]...]]]
     into newTag[ e1', ..., en'],
     where ei' = C[ ei, elementTag].
  *)
  Prepend[ C[ restOfNest, newTag[ NestToList, elementTag]],
    C[ e1, elementTag]
    ]
C[ oldTag_[ ], newTag_[ NestToList, elementTag_]] :=
    newTag[ ]
C[ oldTag_[ e1_, restOfNest_], newTag_[ NestToNest, elementTag_]] :=
```

```
  (* Convert oldTag[ e1, oldTag[ e2, ... oldTag[]...]]]
     into newTag[ e1', newTag[ e2', ... newTag[]...]]],
     where ei' = C[ ei, elementTag].
  *)
  newTag[ C[ e1, elementTag],
    C[ restOfNest, newTag[ NestToNest, elementTag]]
    ]

C[ oldTag_[ ], newTag_[ NestToNest, elementTag_]] :=
  newTag[ ]

C[ nest_, NestToList] := C[ nest, List[ NestToList, Identity]]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Rational Numbers"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

   Rational Numbers are the built-in rational numbers.

*)

Zero[ r_Integer]:= 0
Zero[ r_Rational]:= 0/1

One[ r_Integer]:= 1
One[ r_Rational]:= 1/1

Unprotect[ LCM, GCD]
LCM[ r1_Rational, r2_] := r1 * r2
LCM[ r1_, r2_Rational] := r1 * r2

GCD[ r1_Rational, r2_] := 1
GCD[ r1_, r2_Rational] := 1

P[ r_Integer] := Print[ r]
P[ r_Rational] := Print[ InputForm[ r]]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Floating Point numbers with discrete zero"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
--------------

Floating Point numbers with discrete zero
are expressions of the following form:

   FP[ fp],
```

```
   where fp is a Mathematica floating point number.
*)

$Epsilon = 10^(-6)

FP/:
FP[ fp1_] == FP[ fp2_] :=
   Abs[ fp1 - fp2] < $Epsilon

FP/:
FP[ fp1_] != FP[ fp2_] :=
   Abs[ fp1 - fp2] >= $Epsilon

Zero[ FP[ fp_]] := FP[ 0.0]

One[ FP[ fp_]] := FP[ 1.0]

FP/:
FP[ fp1_] + FP[ fp2_] := FP[ fp1 + fp2]

FP/:
- FP[ fp_] := FP[ -fp]

FP/:
fp_FP - fp_FP := Zero[ fp]

FP/:
FP[ fp1_] * FP[ fp2_] := FP[ fp1 * fp2]

FP/:
Power[ FP[ fp_], n_] := FP[ Power[ fp, n]]

FP/:
FP[ fp1_] < FP[ fp2_] := fp1 < fp2

FP/:
FP[ fp1_] <= FP[ fp2_] := fp1 <= fp2

FP/:
FP[ fp1_] > FP[ fp2_] := fp1 > fp2

FP/:
FP[ fp1_] >= FP[ fp2_] := fp1 >= fp2

Unprotect[ LCM, GCD]

FP/:
LCM[ fp1_FP, fp2_FP] := fp1 fp2

FP/:
GCD[ fp1_FP, fp2_FP] := One[ fp1]

P[ FP[ fp_]] := Print[ fp]

Unprotect[ C]

C[ fp_, FP] := FP[ fp]
```

```
C[ FP[ fp_], IFP] := fp

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain ``Finite Fields''

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
--------------

Elements of a finite field are expressions of the following
form:

   FF[ n],

   where n is an integer satsifying 0 <= n < $Modul.

*)

FF/:
FF[ ff1_] == FF[ ff2_] :=  ff1 == ff2

FF/:
ff1_FF != ff2_FF := Not[ ff1 == ff2]

Zero[ FF[ ff_]] := FF[ Zero[ ff]]

One[ FF[ ff_]] := FF[ One[ ff]]

FF/:
FF[ ff1_] + FF[ ff2_] :=
  FF[ Mod[ ff1 + ff2, $Modul]]

FF/:
- FF[ ff_] := FF[ Mod[ -ff, $Modul]]

FF/:
ff_FF - ff_FF := Zero[ ff]

FF/:
FF[ ff1_] * FF[ ff2_] := FF[ Mod[ ff1 ff2, $Modul]]

FF/:
Power[ FF[ ff_], -1] :=
  Block[ { ff1 = ExtendedGCD[ ff, $Modul][[ 2, 1]]},
    FF[ Mod[ ff1, $Modul]]
    ]

FF/:
Power[ ff_FF, n_] :=
  Power[ ff, n/2] * Power[ ff, n/2] /; EvenQ[ n]

FF/:
Power[ ff_FF, n_] :=
  ff * Power[ ff, n-1] /; OddQ[ n]
```

```
FF/:
Power[ ff_FF, 1] := ff

FF/:
FF[ ff1_] < FF[ ff2_] := ff1 < ff2

FF/:
FF[ ff1_] <= FF[ ff2_] := ff1 <= ff2

FF/:
FF[ ff1_] > FF[ ff2_] := ff1 > ff2

FF/:
FF[ ff1_] >= FF[ ff2_] := ff1 >= ff2

FF/:
GCD[ ff1_FF, ff2_FF ] := One[ ff1]

FF/:
LCM[ ff1_FF, ff2_FF] := ff1 ff2

P[ FF[ ff_]] := StringForm[ "[`` mod ``]", ff, $Modul]

Unprotect[ C]

C[ n_?NumberQ, FF] :=
  FF[ Mod[ n, $Modul]]

C[ FF[ ff_], IFF] := ff

(*
Maintenance of Parameter:
*)

SetModul[ p_] :=
  If[ PrimeQ[p], $Modul = p,
    Print["Error in SetModul: p should be a prime number"]
    ]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

   Domain: Mathematica Rational Functions

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
--------------

Mathemtaica Rational Functions are expressions of the following
form:

   MRF[ mrf],

   where mrf is a Mathematica rational function.

*)
```

```
MRF/:
MRF[ mrf1_] == MRF[ mrf2_] := TrueQ[ mrf1 == mrf2]

MRF/:
mrf1_MRF != mrf2_MRF := !(mrf1 == mrf2)

Zero[ mrf_MRF] := MRF[ 0]

One[ mrf_MRF] := MRF[ 1]

MRF/:
MRF[ mrf1_] + MRF[ mrf2_] := MRF[ Factor[ mrf1 + mrf2]]

MRF/:
- MRF[ mrf_] := MRF[ -mrf]

MRF/:
mrf_MRF - mrf_MRF := Zero[ mrf]

MRF/:
MRF[ mrf1_] * MRF[ mrf2_] := MRF[ Factor[ mrf1 mrf2]]

MRF/:
Power[ MRF[ mrf_], -1] := MRF[ mrf^(-1)]

MRF/:
GCD[ mrf1_MRF, mrf2_MRF] := One[ mrf1]

MRF/:
LCM[ mrf1_MRF, mrf2_MRF] := mrf1 mrf2

P[ MRF[ mrf_]] :=  Print[ mrf]

Unprotect[ C]

C[ mrf_, MRF] :=  MRF[ Factor[ mrf]]

C[ MRF[ mrf_], IMRF] := mrf

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Exponent Lists"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
--------------

Exponent Lists over a domain D have the form

      EL[e1,...,en],

where the "exponents" ei are in D.

*)

ExpandedByIndeterminate[ el_EL] :=
  Append[ el, Zero[ el[[ 1]] ]]
```

```
NewIndeterminate[ el_EL] :=
  Append[ One[ el], One[ el [[1 ]] ]]

EL/:
el1_EL == el2_EL :=
  ForAll[ el1, el2, Equal]

EL/:
el1_EL != el2_EL :=
  Not[ el1 == el2]

One[ el_EL]:=
  Block[{ el0},
    el0=  Table[ Zero[ el[[ 1]]], {Length [el]}];
    Typed[ el0, EL]
    ]

EL/:
el1_EL + el2_EL :=
  Block[ { el10 = el1, el20 = el2},
    el10[[ 0]] = el20[[ 0]] = List;
    Typed[ el10 + el20, EL]
    ]

EL/:
el1_EL * el2_EL :=
  ComponentWise[ Plus, el1, el2]

EL/:
el1_EL / el2_EL :=
  ComponentWise[ Subtract, el1, el2]

Unprotect[ LCM]

EL/:
LCM[ el1_EL, el2_EL] :=
  ComponentWise[ Max, el1, el2]

IsMultiple[ el1_EL, el2_EL] :=
  ForAll[ el1, el2, GreaterEqual]

IsDisjoint[ el1_EL, el2_EL]:=
  ForAll[ el1, el2, EitherZero]

EitherZero[ e1_, e2_] :=  e1 == 0 || e2 == 0

EL/:
el1_EL > el2_EL :=
  $GreaterEL[ el1, el2]
  (*
    Alternatives:
      IsMatrixGreater
      IsLexicalGreater
```

```
      IsPermutedLexicalGreater
      IsTotalDegreeInverseLexicalGreater
      IsTotalDegreeLexicalGreater
      IsPartialLexicalGreater.
  *)

EL/:
el1_EL < el2_EL :=
  el2 > el1

TotalDegree[ el_EL] :=
  Apply [Plus, el]

IsMatrixGreater[ el1_EL, el2_EL] :=
  (* Global variable: $OrderMatrix. *)
  Block[
    { i,
      elDiff = ComponentWise[ Subtract, el1, el2],
      test
    },
    For[ i = 1, i <= Length[ $OrderMatrix], i++,
      test = elDiff . $OrderMatrix[[i]];
      If[ test != 0, Return[ Positive[ test]]]
      ];
    False
    ]

EL/:
el_EL . vector_ :=
  Block[ { el0 = Typed[ el, List]},
    el0 . vector
    ]

IsPermutedLexicalGreater[ el1_EL, el2_EL] :=
  (* Global input variable: $ELPermutation. *)
  IsPermutedLexicalGreater[ el1, el2, $ELPermutation]

IsPartialLexicalGreater[ el1_EL, el2_EL] :=
  (* Global input variables:
     $ELFirstIndices, $ELSecondIndices,
     $ELFirstOrdering, $ELSecondOrdering.
  *)
  Block[ {
    el11 = Select[ el1, $ELFirstIndices],
    el12 = Select[ el1, $ELSecondIndices],
    el21 = Select[ el2, $ELFirstIndices],
    el22 = Select[ el2, $ELSecondIndices],
    },
    $ELFirstOrdering[ el11, el21] ||
    el11 == el21  &&  $ELSecondOrdering[ el12, el22]
```

```
    ]

EL/:
Select[ el_EL, indices_] :=
  Block[ { },
    el0 = Table[ el[[ indices[[ i]] ]], {i, Length[ indices]}];
    Typed[ el0, EL]
    ]

NumberOfVariables[ el_EL] := Length[ el]

P[ el_EL]:= Print[ el]

C[ el_EL, permutation_VarPerm] := Permuted[ el, permutation]

C[ el_EL, variables_MP] :=
  Block[{ ii, vs = variables},
    If[ IsEmpty[vs], vs = StandardVariables[ Length[ el]]];
    MP[ Product[
          vs[[ ii]] ^ el[[ ii]],
          {ii, Length[ vs]}
          ]
      ]
    ]

StandardVariables[ n_] :=
  Table[ x[ i], {i, n}]

C[ el_EL, RFPP[ variableIndices_, variables_]] :=
  Block[{ i, vs = variables, l = Length[ variableIndices]},
    If[ IsEmpty[vs], vs = StandardVariables[ l]];
    C[
      Product[
        vs[[ i]] ^ el[[ variableIndices[[i]] ]],
        {i,l}
        ],
      MRF
      ]
    ]

ThinnedPP[ el_EL, variableIndices_] :=
  Block[{ el0, indices},
    indices = Complement[ Range[ Length[ el]], variableIndices];
    el0 = Table[ el[[ indices[[ i]] ]], {i, Length[ indices]}];
    Typed[ el0, EL]
    ]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Graded Lexical Orderings"
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
*)

IsTotalDegreeLexicalGreater[ x1_, x2_] :=
  Block[
    { td1 = TotalDegree[ x1],
      td2 = TotalDegree[ x2]
    },
    If[ td1 == td2,
      IsLexicalGreater[ x1, x2],
      td1 > td2
      ]
    ]

IsTotalDegreeInverseLexicalGreater[ x1_, x2_] :=
  Block[
    { td1 = TotalDegree[ x1],
      td2 = TotalDegree[ x2]
    },
    If[ td1 == td2,
      IsLexicalGreater[ x2, x1],
      td1 > td2
      ]
    ]
(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Goedel Exponents"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

   A Goedel Exponent has the form

      GE[ g],

   where g is a natural number.

*)

GE/:
GE[ ge1_] == GE[ ge2_] :=  ge1 == ge2

GE/:
ge1_GE != ge2_GE := Not[ ge1 == ge2]

One[GE[ ge_]] := GE[ 1]

GE/:
GE[ ge1_] * GE[ ge2_] := GE[ ge1 ge2]

GE/:
```

```
GE[ ge1_] / GE[ge2_] := GE[ ge1 / ge2]

GE/:
LCM[ GE[ ge1_], GE[ ge2_]] :=
  GE[ LCM[ ge1, ge2]]

IsMultiple[ GE[ ge1_], GE[ ge2_]] :=
  Mod[ ge1, ge2] == 0

IsDisjoint[ GE[ ge1_], GE[ ge2_]] :=
  GCD[ ge1, ge2] == 1

GE/:
GE[ ge1_] > GE[ ge2_] := ge1 > ge2

GE/:
GE[ ge1_] < GE[ ge2_] := ge1 < ge2

P[ GE[ ge_]] := Print[ GE[ ge]]

C[ list_, GE] :=
  GE[ GoedelNumber[ list]]

GoedelNumber[ list_] :=
  Block[{ i},
    Product[ Power[ Prime[ i], list[[i ]]], {i, Length[ list]}]
    ]

C[ GE[ ge_], IGE] :=
  ListFromGoedelNumber[ 1, ge]

ListFromGoedelNumber[ i_, ge_] :=
  Block[ { exp, newge},
    If[ ge == 1,
      {},
      {exp, newge} = PrimeExponent[ i, ge];
      Return[
        Prepend[ ListFromGoedelNumber[ i+1, newge],
          exp
          ]
        ]
      ]
    ]

PrimeExponent[ i_, n_] :=
  Block[{ pi = Prime[ i], exp, newn},
    If[ Mod[ n, pi] != 0,
      Return[{ 0, n}],
      {exp, newn} = PrimeExponent[ i, Quotient[ n, pi]];
      Return[{ exp + 1, newn}]
      ]
    ]
```

```
(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Domain Constructor "Monomials"
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data structure:
---------------
A Monomial over a coefficient domain and
a power product domain is an expression of the form

   Mon[ c, pp],

where c is from the coefficient domain and
pp is from the power product domain.
*)

Unprotect[ Coefficient]
Coefficient[ Mon[ c_, pp_]] := c

PowerProduct[ Mon[ c_, pp_]] := pp

ExpandedByIndeterminate[ Mon[ c_, pp_]] :=
  Mon[ c, ExpandedByIndeterminate[ pp]]

NewIndeterminate[ Mon[ c_, pp_]] :=
  Mon[ One[ c], NewIndeterminate[ pp]]

NumberOfVariables[ Mon[ c_, pp_]] :=
  NumberOfVariables[ pp]

Mon/:
Mon[ c1_, pp1_] == Mon[ c2_, pp2_] :=
  c1 == c2  &&  pp1 == pp2

Mon/:
m1_Mon != m2_Mon := Not[ m1 == m2]

One[ Mon[ c_, pp_]] :=
  Mon[ One[ c], One[ pp]]

Mon/:
Mon[ c1_, pp_] + Mon[ c2_, pp_] :=
   Mon[ c1 + c2, pp]

Mon/:
- Mon[ c_, pp_] := Mon[ -c, pp]

Mon/:
Mon[ c1_, pp1_] * Mon[ c2_, pp2_] :=
  Mon[ c1 *  c2, pp1 * pp2]

Mon/:
```

```
Mon[ c1_, pp1_] /  Mon[ c2_, pp2_] :=
  Mon[ c1 / c2, pp1 / pp2]

Mon/:
LCM[ Mon[ c1_, pp1_],  Mon[ c2_, pp2_]] :=
  Mon[ LCM[ c1, c2], LCM[ pp1, pp2]]

LCMOfPowerProducts[ Mon[ c1_, pp1_],  Mon[ c2_, pp2_]] :=
  LCM[ pp1, pp2]

HasZeroCoefficient[ Mon[ c_, pp_]] := c == Zero[ c]

IsPowerProductEqual[ Mon[ c1_, pp1_], Mon[ c2_, pp2_]] :=
  pp1 == pp2

IsMultiple[ Mon[ c1_, pp1_], Mon[ c2_, pp2_]] :=
  IsMultiple[ pp1, pp2]

IsDisjoint[ Mon[ c1_, pp1_], Mon[ c2_, pp2_]] :=
  IsDisjoint[ pp1, pp2]

Mon/:
Mon[ c1_, pp1_] > Mon[ c2_, pp2_] :=
  pp1 > pp2

Mon/:
Mon[ c1_, pp1_] < Mon[ c2_, pp2_] :=
  pp1 < pp2

Normalized[ Mon[c_, pp_]] :=  Mon[ One[ c], pp]

Normalizer[ Mon[c_, pp_]] :=  Mon[ 1/c, One[ pp]]

CoefficientSize[ Mon[c_, pp_]] :=
  ByteCount[ c]

P[ mon_Mon] :=  P[ mon, List]

P[ mon_Mon, C, Q] :=
  Print[ "Prints monomial with Coefficient size."]

P[ mon_Mon, C] :=
 Block[ {},
    P[ Coefficient[ mon], CS];
    P[ PowerProduct[ mon]]
    ]

P[ coefficient_, CS] := Print[ CS[ ByteCount[ coefficient]]]

P[ mon_Mon, C, variables_MP] :=
  Block[ {},
    P[ Coefficient[ mon], CS];
    P[ C[  PowerProduct[ mon], variables]]
    ]

C[ mon_Mon, perm_VarPerm] :=
```

```
  C[ mon, Mon[ TupleToTuple, Identity, perm]]

C[ Mon[c_, pp_], variables_MP] :=
  MP[ c] C[ pp, variables]

C[ Mon[c_, pp_], RFMon[ variableIndices_, variables_]] :=
  Mon[ C[ c, MRF] C[ pp, RFPP[ variableIndices, variables]],
    ThinnedPP[ pp, variableIndices]
    ]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Distributive Nest represented Polys"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

A Distributive Nest represented Poly over a domain
of monomials is a Nest of the form

   DNP[ ]

   or

   DNP[ m,  dnp]

   where m is from the domain of monomials
   dnp is a Distributive Nest represented Poly
   and m is >  than all monomials
   occurring in dnp.

*)

LeadingMonomial[ DNP[ m_, dnp_]] := m

RemainingPolynomial[ DNP[ m_, dnp_]] := dnp

LeadingPowerProduct[ DNP[ m_, dnp_]] := PowerProduct[ m]

LeadingPolynomial[ DNP[ m_, dnp_]] := DNP[ m, DNP[ ]]

NumberOfVariables[ DNP[ m_, dnp_]] :=
  NumberOfVariables[ m]

ExpandedByIndeterminate[ dnp_DNP] :=
  ComponentWiseN[ ExpandedByIndeterminate, dnp]

RabinovichExpanded[ dnp_DNP] :=
  ExpandedByIndeterminate[ dnp] *
  NewIndeterminate[ LeadingMonomial[ dnp]] -
  ExpandedByIndeterminate[ One[ dnp]]

DNP/:
DNP[ m1_, dnp1_] == DNP[ m2_, dnp2_] :=
```

```
  m1 == m2  &&   dnp1 == dnp2

DNP/:
DNP[ ] == DNP[ m_, dnp_] := False

DNP/:
DNP[ m_, dnp_] == DNP[ ] := False

DNP/:
DNP[ ] == DNP[ ] := True

DNP/:
dnp1_DNP != dnp2_DNP := Not[ dnp1 == dnp2]

Zero[ dnp_DNP] := DNP[ ]

One[ DNP[ m_, dnp_]] :=
  DNP[ One[ m], DNP[ ]]

DNP/:
dnp1_DNP + dnp2_DNP :=
  MergeN[ dnp1, dnp2, Greater, Greater, (#1 == - #2)&, Plus]

DNP/:
-  dnp_DNP := ComponentWiseN[ (- #)&, dnp]

DNP/:
dnp_DNP - dnp_DNP := Zero[ dnp]

DNP/:
DNP[ m1_, dnp1_] *  dnp2_DNP :=
  m1 * dnp2 +  dnp1 * dnp2

DNP/:
DNP[ ] * dnp_DNP := DNP[ ]

DNP/:
dnp_DNP * DNP[ ] := DNP[ ]

DNP/:
monomial_  * dnp_DNP :=
  ComponentWiseN[  (monomial * #)&, dnp]

DNP/:
Power[ dnp_DNP, n_Integer] :=
  (* Assumption: n > 0. *)
  Block[ { r = dnp, i},
    For[ i=1, i<n, i++, r = r dnp];
    r
    ]

SPolynomial[
  DNP[ m1_, dnp1_],
  DNP[ m2_, dnp2_]
  ] :=
```

```
  Block[
    { lcm = LCM[ m1, m2], u1, u2},
    u1 = lcm / m1;
    u2 = lcm / m2;
    u1  dnp1 -  u2 dnp2
    ]

Normalized[ DNP[ m_, dnp_]] :=
  DNP[ Normalized[ m], Normalizer[m] dnp]

Labeled[ dnp_DNP] :=  LabP[ NewLabel[], dnp]

NormalForm[ dnp_, pn_] :=
  $NormalFormDNP[ dnp, pn]
  (*
    Alternatives:
      PartialNormalForm [ dnp, pn]
      CompleteNormalForm [ dnp, pn].
   *)

PartialNormalForm[ DNP[ m_, dnp_], pn_PN] :=
  (* Block[ { dnp0 = DNP[ m, dnp]}, P[ dnp0];
       PartialNormalForm$[ dnp0, pn, pn]
       ]
     *)
  PartialNormalForm$[ DNP[ m, dnp], pn, pn]

PartialNormalForm[ DNP[ ], pn_PN] :=
  (* Block[ {}, P[ DNP[]]; DNP[ ]] *)
  DNP[ ]

PartialNormalForm$[ DNP[ m_, dnp_], PN[ pRed_, pnRest_],  pn_] :=
  Block[ {lm = LeadingMonomial[ pRed],
    rp = RemainingPolynomial[ pRed]
    },
    Which[
      IsMultiple[ m, lm],
        (* Print["use "]; PLabel[ pRed]; *)
        PartialNormalForm[ dnp - (m / lm) rp, pn],
      lm > m, DNP[ m, dnp],
      True, PartialNormalForm$[ DNP[ m, dnp], pnRest, pn]
      ]
    ]

PartialNormalForm$[ dnp_, PN[ ], pn_] := dnp

CompleteNormalForm[ dnp_DNP, pn_] :=
  CompleteNormalForm$[ PartialNormalForm[ dnp, pn], pn]

CompleteNormalForm$[ DNP[ m_, dnp_], pn_] :=
  (* Complete Normal Form under the assumption that
```

```
     the first argument is already partially reduced
     w.r.t. pn.
  *)
  DNP[ m, CompleteNormalForm[ dnp, pn]]

CompleteNormalForm$[ DNP[ ], pn_] := DNP[ ]

HeadReduced[ DNP[ m_, dnp_], pn_] :=
  DNP[ m, CompleteNormalForm[ dnp, pn]]

DNP/:
Sort[ dnp_DNP] := SortN[ dnp, Greater]

P[ dnp_DNP] :=
  Block[ {},
    Print[ "Distributive Nest Poly"];
    P[ dnp, Nest];
    Print[ "End of Distributive Nest Poly"]
    ]

P[ dnp_DNP, Q] :=
  Print[ "DNP Print Options: N, S, P, C, Sh, D"]

P[ dnp_DNP, N, Q] :=
  Print[ "Prints Number of monomials"]

P[ dnp_DNP, N] :=
  Print[ "Number of monomials: ", LengthN[ dnp]]

P[ dnp_DNP, S, Q] :=
  Print[ "Prints Structure (leading PP, maximal coefficient size,"]
  Print[ "number of monomials)."]

P[ dnp_DNP, S] :=
  Block[ {
    lpp = PowerProduct[ LeadingMonomial[ dnp]],
    mcl = MaximalCoefficientSize[ dnp],
    n = LengthN[ dnp]
    },
    Print[ "Leading PP: "]; P[ lpp];
    Print[ "Maximal Coefficient Size: ", mcl, ";Number of Monomials: ", n]
    ]

MaximalCoefficientSize[ DNP[ m_, dnp_]] :=
  Max[ CoefficientSize[ m], MaximalCoefficientSize[ dnp]]

MaximalCoefficientSize[ DNP[ ]] := 0

P[ dnp_DNP, P, Q] :=
  Print[ "Prints all Power products."]

P[ dnp_DNP, P] :=
  Block[ {},
    Print[ "Distributive Nest Poly: Only Power Products"];
```

```
    P[ dnp, Nest, PowerProduct];
    Print[ "End of Distributive Nest Poly"]
    ]

P[ dnp_DNP, P, variables_MP] :=
  Block[ {},
    Print[ "Distributive Nest Poly: Only Power Products"];
    P[ dnp, Nest, C[ PowerProduct[ #], variables] & ];
    Print[ "End of Distributive Nest Poly"]
    ]

P[ dnp_DNP, C, Q] :=
  Print[ "Prints all monomials with Coefficient sizes."]

P[ dnp_DNP, C] :=
 Block[ {},
    Print[ "Distributive Nest Poly: Monomials with Coefficient Size"];
    DoForAllN[ dnp,  P[ #, C] & ];
    Print[ "End of Distributive Nest Poly"]
    ]

P[ dnp_DNP, C, variables_MP] :=
  Block[ {},
    Print[ "Distributive Nest Poly: Monomials with Coefficient Size"];
    DoForAllN[ dnp, P[ #, C, variables] & ];
    Print[ "End of Distributive Nest Poly"]
    ]

P[ dnp_DNP, Sh, Q] :=
  Print[ "P[ dnp, Sh, MP[ var1, ...]] prints Mathematica Short form."
    ]

P[ dnp_DNP, Sh, variables_MP] :=
  Print[ Short[ C[ C[ dnp, variables], IMP]]]

P[ dnp_DNP, D, Q] :=
  Print[ "Prints Domain of coefficients."]

P[ dnp_DNP, D] :=
  Block[ {},
    Print[ "Coefficient Domain: "];
    P[ Coefficient[ LeadingMonomial[ dnp]], Domain]
    ]

C[ dnp_DNP, perm_VarPerm] :=
  C[ dnp, DNP[ NestToNest, perm]]

C[ DNP[ m_, dnp_], variables_MP] :=
  C[ m, variables] + C[ dnp, variables]

C[ DNP[ ], variables_MP] := MP[ 0]

C[ dnp_DNP, LP] :=
```

```
  C[ dnp, LP[ NestToList, Identity]]

C[ dnp_DNP, RFDNP[ variableIndices_, variables_]] :=
  MonomialCondensed[
    Sort[
      C[ dnp,
        DNP[ NestToNest, RFMon[ variableIndices, variables]]
        ]
      ]
    ]

MonomialCondensed[ DNP[ m1_, DNP[ m2_, dnp_]]] :=
  MonomialCondensed[
    DNP[ m1 + m2, dnp]
    ] /; IsPowerProductEqual[ m1, m2]

MonomialCondensed[ DNP[ m1_, DNP[ m2_, dnp_]]] :=
  DNP[ m1,
   MonomialCondensed[ DNP[ m2, dnp]]
   ]  /; ! IsPowerProductEqual[ m1, m2]

MonomialCondensed[ DNP[ m_, dnp_]] :=
  MonomialCondensed[ dnp] /; HasZeroCoefficient[ m]

MonomialCondensed[ DNP[ m_, DNP[ ]]] := DNP[ m, DNP[ ]]

MonomialCondensed[ DNP[ ]] := DNP[ ]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain "List Polys"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
--------------

List Polys are expressions of the
following form:

   LP[ m1, ..., mn]

where the mi are monomials.

*)

P[ lp_LP] :=
  Block[ {},
    Print[ "Distributive List Poly"];
     P[ lp, List];
    Print[ "End of Distributive List Poly"]
    ]

C[ lp_LP, DNP] :=
```

```
  C[ lp, DNP[ ListToNest, Identity]]
(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Mathematica Polynomials"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
--------------

Mathematica Polynomials are expressions of the
following form:

   MP[ polynomial_],

   where expression is a Mathematica ``polynomial''
   in expanded form, i.e. an expression
   of the following structure

   Plus[ Times[ Power[  ], ...], ... Times[ Power[  ]]
   containing only variables and rational numbers.

*)

MP/:
MP[ mp1_] == MP[ mp2_] := TrueQ[ mp1 == mp2]

MP/:
mp1_MP != mp2_MP := !(mp1 == mp2)

Zero[ mp_MP] := MP[ 0]

One[ mp_MP] := MP[ 1]

MP/:
MP[ mp1_] + MP[ mp2_] := MP[ Expand[ mp1 + mp2]]

MP/:
- MP[ mp_] := MP[ Expand[ -mp]]

MP/:
mp_MP - mp_MP := Zero[ mp]

MP/:
MP[ mp1_] * MP[ mp2_] := MP[ Expand[ mp1 mp2]]

Unprotect[ LCM, GCD]

MP/:
GCD[ MP[ mp1_], MP[ mp2_]] := MP[ GCD[ mp1, mp2]]

MP/:
LCM[ MP[ mp1_], MP[ mp2_]] :=
  MP[ Expand [Factor [ mp1 mp2 / GCD[ mp1, mp2]]]]

P[ MP[ mp_]] := Print[ mp]
```

```
PBeginSymbol[ mp_MP] := Null (* Print[ "Mathematica Poly"] *)

PEndSymbol[ mp_MP] :=
  Print[ "----------------------------------------------------------"]
  (* Print[ "End of Mathematica Poly"]  *)

Unprotect[ C]

C[ e_, MP] :=  MP[ Expand[ e]]

C[ MP[ mp_], IMP] := mp

C[ MP[ p_], variables_DNP] :=  ConvertMP[ p, variables]

ConvertMP[ Plus[ e1_, e2_], variables_DNP] :=
  ConvertMP[ e1, variables] + ConvertMP[ e2, variables]

ConvertMP[ Times[ e1_, e2_], variables_DNP] :=
  ConvertMP[ e1, variables] ConvertMP[ e2, variables]

ConvertMP[ Power[ e1_, n_Integer], variables_DNP] :=
  ConvertMP[ e1, variables]^n

ConvertMP[ variable_, variables_DNP] :=
  Block[ {expList, position},
    position = Position[ variables, variable][[1,1]];
    expList =
      Table[ If[ i==position, 1, 0],
        {i, Length[ variables]}
        ];
    expList[[ 0]] = EL;
    DNP[ Mon[ 1, expList], DNP[]]
    ]   /;  ( AtomQ[ variable] || IsIndexedVariable)  &&
           MemberQ[ variables, variable]

IsIndexedVariable[ variable_] := Length[ variable] == 1

ConvertMP[ constant_, variables_DNP] :=
  Block[ {expList},
    expList = Table[ 0, { Length[ variables]}];
    expList[[ 0]] = EL;
    DNP[ Mon[ constant, expList], DNP[]]
    ] /; NumberQ[ constant]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Labeled Polys"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:

   A Labeled Poly has the form

   LabP[ label, polynomial].
```

```
*)
NewLabel[] := Block[ {}, $Label++; $Label]

Unprotect[ Label]

Label[ LabP[ label_, p_]] := label

Polynomial[ LabP[ label_, p_]] := p

LeadingMonomial[ LabP[ label_, p_]] :=
  LeadingMonomial[ p]

RemainingPolynomial[  LabP[ label_, p_]] :=
  RemainingPolynomial[ p]

LeadingPowerProduct[ LabP[ label_, p_]] :=
  LeadingPowerProduct[ p]

LeadingPolynomial[ LabP[ label_, p_]] :=
  LeadingPolynomial[ p]

Unlabeled[ LabP[ label_, p_]] := p

LabP/:
LabP[ label1_, p1_] == LabP[ label2_, p2_] :=
  label1 == label2

LabP/:
lp1_LabP != lp2_LabP := Not[ lp1 == lp2]

LabP/:
p1_ + LabP[ label_, p2_] :=  p1 +  p2

LabP/:
- LabP[ label_, p_]  := - p

LabP/:
x_ * LabP[ label_, p_] := x p

SPolynomial[ LabP[ label1_, p1_], LabP[ label2_, p2_]]:=
  SPolynomial[ p1, p2]

NormalForm[ LabP[ label_, p_], pn_] :=
  NormalForm[ p, pn]

HeadReduced[ LabP[ label_, p_], pn_] :=
  LabP[ label, HeadReduced[ p, pn]]

LabP/:
Sort[ LabP[ label_, p_]] := LabP[ label, Sort[ p]]

Normalized[ LabP[ label_, p_]] := LabP[ label, Normalized[ p]]

NumberOfVariables[ LabP[ label_, p_]] :=
  NumberOfVariables[ p]

P[ LabP[ label_, p_]] :=
```

```
  Block[{ },  Print[ "[", label, "]"]; P[ p]]

P[ LabP[ label_, p_], Label] :=
  Print[ "[", label, "]"]

P[ LabP[ label_, p_], N] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, N]]

P[ LabP[ label_, p_], S] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, S]]

P[ LabP[ label_, p_], P] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, P]]

P[ LabP[ label_, p_], P, variables_MP] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, P, variables]]

P[ LabP[ label_, p_], C] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, C]]

P[ LabP[ label_, p_], C, variables_MP] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, P, variables]]

P[ LabP[ label_, p_], Sh, variables_MP] :=
  Block[{ },  Print[ "[", label, "]"]; P[ p, Sh, variables]]

P[ LabP[ label_, p_], D] := P[ p, D]

C [ p_, Label] := LabP[ NewLabel[], p]

C [ LabP[ label_, p_], ILabel] := p

C[ LabP[ label_, p_], permutation_VarPerm] :=
  LabP[ label, C[ p, permutation]]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Poly Lists"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

A Poly List is a list of
the form

   PL[ p1, ..., pn ]

   where the pi are polynomials.

*)

GB[ pl_PL] :=
  (* pl ... a Poly List consisting of Mathematica
            polys.
     Global input variable: $MathematicaVariables.
```

```
  *)
  GroebnerBasis[ C[ pl, List[ ListToList, Identity]],
    $MathematicaVariables
    ]

P[ pl_PL] :=
  Block[ {},
    Print[ "Poly List"];
    P[ pl, List];
    Print[ "End of Poly List"]
    ]

C[ pl_PL, DNPN[ variables__]] :=
  C[ pl, PN[ ListToNest, DNP[ variables]]]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Constructor "Poly Nests"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

A Poly Nest is a nest of
the form

   PN[ ]

   or

   PN[ p, pn]

   where p is a polynomial and pn is a Poly Nets.

*)

FirstPolynomial[ PN[ p_, pn_]] := p
RemainingPolynomials[ PN[ p_, pn_]] := pn

IsEmpty[ pn_PN] := IsEmptyN[ pn]
Empty[ pn_PN] := PN[ ]

Singleton[ p_, pn_PN] := PN[ p, PN[ ]]

PN/:
pn1_PN == pn2_PN :=  EqualN[ pn1, pn2]

Adjoin[ p_, pn_PN] := PN[ p, pn]

PN/:
Insert[ p_, pn_PN] := InsertN[ p, pn, IsLeadingMonomialLess]

PN/:
Union[ p1_PN, p2_PN] :=
```

```
  UnionN[ p1, p2, IsLeadingMonomialLess]

ExpandedByIndeterminate[ pn_PN] :=
  ComponentWiseN[ ExpandedByIndeterminate, pn]

RabinovichExpanded[ p_, pn_PN] :=
  Adjoin[ RabinovichExpanded[ p], ExpandedByIndeterminate[ pn]]

PN/:
Sort[ pn_PN] := SortN[ InnerSort[ pn],  IsLeadingMonomialLess]

IsLeadingMonomialLess[ p1_, p2_] :=
  LeadingMonomial[ p1] < LeadingMonomial[ p2]

Augmented[ pn_PN, p_] :=
  InsertN[ p, pn, IsLeadingMonomialLess]

InnerSort[ PN[ p_, pn_]] :=
  PN[ Sort[ p], InnerSort[ pn]]

InnerSort[ PN[ ]] := PN[ ]

Normalized[ PN[ p_, pn_]] :=
  PN[ Normalized[ p], Normalized[ pn]]

Normalized[ PN[ ]] := PN[ ]

InterReduced[ pn_PN] :=
  $InterReducedPN[ pn]
  (*
    Alternatives:
      Identity[ pn]
      CompletelyInterReduced[ pn].
  *)

CompletelyInterReduced[ pn_PN] :=
  (* Assumption: pn is a Groebner basis. *)
  CompletelyReduced[ CanceledMultiples[ pn]]

CanceledMultiples[ pn_PN] :=
  CanceledMultiples$[{ pn, PN[ ]}]

CanceledMultiples$[{ PN[ p_, pnhigher_], pnlower_}] :=
  If[ IsLeadingMonomialMultiple[ p, pnlower],
    CanceledMultiples$[{ pnhigher, pnlower}],
    PN[ p, CanceledMultiples$[{ pnhigher, PN[ p, pnlower]}]]
    ]

CanceledMultiples$[{ PN[ ], pnlower_}] := PN[ ]

IsLeadingMonomialMultiple[ p_, PN[ plower_, pn_]] :=
  IsMultiple[
    LeadingMonomial[ p],
    LeadingMonomial[ plower]
    ]  ||
```

```
  IsLeadingMonomialMultiple[ p, pn]

IsLeadingMonomialMultiple[ p_, PN[ ]] := False

CompletelyReduced[ pn_PN] :=
  CompletelyReduced$[ pn, pn]

CompletelyReduced$[ PN[ p_, pn_], pnreducing_] :=
  (* Assumption: The polynomials in the basis are
       in ascending order w.r.t. their leading monomials.
       This implies that it suffices to head-reduce
       the polynomials w.r.t. to the lower polynomials.
       *)
  PN[ HeadReduced[ p, LowerPart[ pnreducing, p]],
    CompletelyReduced$[ pn, pnreducing]
    ]

CompletelyReduced$[ PN[ ], pnreducing_] := PN[ ]

LowerPart[ PN[ p1_, pn_], p2_] :=
  PN[ ]  /; p1 == p2

LowerPart[ PN[ p1_, pn_], p2_] :=
  PN[ p1, LowerPart[ pn, p2]]  /; p1 != p2

Labeled[ pn_PN] :=
  ComponentWiseN[ Labeled, pn]

Unlabeled[ pn_PN] :=
  ComponentWiseN[ Unlabeled, pn]

NumberOfVariables[ PN[ p_, pn_]] :=
  NumberOfVariables[ p]

(* Algorithms for Version AllReduced: *)

AllReduced[ reducibles_, reducers_,  basis_, pairs_] :=
  Block[
    { newReducibles = reducibles, newReducers = reducers,
      newBasis = basis, newPairs = pairs,
      h, b, nf, reducerPairs, lnf
      },
    While[ !IsEmpty[ newReducibles],
      h = FirstPolynomial[ newReducibles];
      newReducibles = RemainingPolynomials[ newReducibles];
      b = Union[ newBasis, newReducers];
      nf = NormalForm[ h, b];
      If[ nf != Zero[ nf],
        { reducibleBasis, newBasis} =
            Multiples[ nf, newBasis];
        { reducibleReducers, newReducers} =
            Multiples[ nf, newReducers];
```

```
        newReducibles =
          Union[ Union[ newReducibles, reducibleBasis],
            reducibleReducers
            ];
        newPairs = CanceledPairs[ reducibleBasis, newPairs];
        lnf = Labeled[ Normalized[ nf]];
        newReducers =
          Insert[ lnf, newReducers];
(*        P[ lnf, S]    *)
        ]
      ];
    reducerPairs = InitialPairs[ newReducers];
    newPairs = Union[ newPairs, reducerPairs];
    newPairs = UpdatedPairs[ newPairs, newBasis, newReducers];
    newBasis = Union[ newBasis, newReducers];
    newBasis = CompletelyReduced[ newBasis];
    { newBasis, newPairs}
    ]

Multiples[ p_,  pn_PN] :=
  Block[ { lp = LeadingPowerProduct[ p]},
    ComplementarySubSetsN[
      pn, IsMultiple[ LeadingPowerProduct[ #], lp] &
      ]
    ]

(* End of Algorithms for Version AllReduced: *)

P[ pn_PN] :=
  Block[ {},
    Print[ "Poly Nest"];
    P[ pn, Nest];
    Print[ "End of Poly Nest"]
    ]

P[ pn_PN, Q] :=
  Print[ "PN Print options: N, S, P, C, Sh, D"]

P[ pn_PN, N, Q] :=
  Print[ "Prints Number of polynomials."]

P[ pn_PN, N] :=
  Print[ "Number of polynomials: ", LengthN[ pn]]

P[ pn_PN, S, Q] :=
  Print[ "Prints Structure (leading PP, maximal coefficient size,"]
  Print[ "number of monomials)  of the polynomials)."]

P[ pn_PN, S] :=
  DoForAllN[ pn, P[ #, S] & ]

P[ pn_PN, P, Q] :=
```

```
  Print[ "Prints all Power products of the polynomials."]
P[ pn_PN, P] :=
  DoForAllN[ pn,  P[ #, P] & ]
P[ pn_PN, P, variables_MP] :=
  DoForAllN[ pn,  P[ #, P, variables] & ]
P[ pn_PN, C, Q] :=
  Print[ "Prints monomials with Coefficient sizes of all polynomials."
    ]
P[ pn_PN, C] :=
  DoForAllN[ pn,  P[ #, C] & ]
P[ pn_PN, C, variables_MP] :=
  DoForAllN[ pn,  P[ #, C, variables] & ]
P[ pn_PN, Sh, Q] :=
  Print[ "P[ pn, Sh, MP[ var1, ...]] prints Mathematica Short "]
  Print[ "forms of the polynomials." ]
P[ pn_PN, Sh, variables_MP] :=
  DoForAllN[ pn,  P[ #, Sh, variables] & ]
P[ pn_PN, D, Q] :=
  Print[ "Prints Domain of coefficients."]
P[ pn_PN, D ] :=
  Block[ {},
    Print[ "Coefficient Domain: "];
    P[ Coefficient[ LeadingMonomial[ PartN[ pn, 1]]], Domain]
    ]
C[ pn_PN, permutation_VarPerm] :=
  C[ pn, PN[ NestToNest, permutation]]
C[ pn_PN, LPL] :=
  C[ pn, PL[ NestToList, LP]]
C[ pn_PN, MPL[ variables___]] :=
  C[ pn, PL[ NestToList, MP[ variables]]]
C[ pn_PN, RFDNPN[ variableIndices_, variables_]] :=
  C[ pn, PN[ NestToNest, RFDNP[ variableIndices, variables]]]
C[ mpl_List, PN[ variables___]] :=
  C[ C[ mpl, PN[ ListToNest, MP]],
    PN[ NestToNest, DNP[ variables]] ]
(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Domain "Poly Pairs"
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
Data structure:
---------------

A Polynomial Pair has the form

   PPa[ p1, p2],

   where p1 and p2 are polynomials.
*)

FirstPolynomial[ PPa[ p1_, p2_]] := p1
SecondPolynomial[ PPa[ p1_, p2_]] := p2

PPa/:
PPa[p11_, p12_] == PPa[ p21_, p22_] :=
  p11 == p21 && p12 == p22

Occurs[ p_, PPa[ p1_, p2_]] :=
  p == p1 || p == p2

PPa/:
LCM[ PPa[ p1_, p2_]] :=
  LCM[ PowerProduct[ LeadingMonomial [ p1]],
       PowerProduct[ LeadingMonomial [ p2]]
     ]

IsLCMLess[ ppa1_PPa, ppa2_PPa] :=
  $GreaterPPa[ LCM[ ppa2], LCM[ ppa1]]

ProductCriterion[ PPa[ p1_, p2_]] :=
    IsDisjoint[ LeadingMonomial[ p1], LeadingMonomial[ p2]]

(* Algorithms for Version "Small Pair Sets": *)

NewPair[ p1_, p2_] :=
  If[ Label[ p1] < Label[ p2], PPa[ p1, p2], PPa[ p2, p1]]

TrivialPair [ ] := PPa[ LabP[ 1, {}], LabP[ 1, {}]]

IsLabeledLCMLess[ ppa1_PPa, ppa2_PPa] :=
  Block[ { pp1 = LCM[ ppa1], pp2 = LCM[ ppa2]},
    $GreaterPPa[ pp2, pp1] ||
    pp1 == pp2 && IsLabeledPairLess[ ppa1, ppa2]
    ]

IsLabeledPairLess[ PPa[ p11_, p12_], PPa[ p21_, p22_]] :=
  Label[ p11] < Label[ p21] ||
  Label[ p11] == Label[ p21] && Label[ p12] < Label[ p22]

TriangleCriterion[ PPa[ p1_, p2_], p_, pair_] :=
  p != p1 &&
  p != p2 &&
  IsMultiple[ LCM[ PPa[ p1, p2]], LeadingPowerProduct[ p]] &&
  ( IsLabeledLCMLess[ NewPair[ p1, p], PPa[ p1, p2]] ||
    ProductCriterion[ PPa[ p1, p]] ||
```

```
    NewPair[ p1, p] == pair ) &&
  ( IsLabeledLCMLess[ NewPair[ p, p2] , PPa[ p1, p2]] ||
    ProductCriterion[ PPa[ p, p2]] ||
    NewPair[ p, p2] == pair )

(* End of Algorithms for Version "Criteria At Update Time": *)

P[ PPa[ p1_, p2_]] :=
  Block[
    { pp1 = PowerProduct[ LeadingMonomial[ p1]],
      pp2 = PowerProduct[ LeadingMonomial[ p2]]
    },
    Print[ "[", Label[ p1], ",",  Label[ p2], "]",
      LCM[ pp1, pp2]
      ];
    (*
    (* Print[ "Poly Pair"]; *)
    P[ p1, Label]; (* P[ pp1];  *)
    P[ p2, Label]; (* P[ pp2];  *)
    P[ LCM[ pp1, pp2]];
    (* Print ["End of Poly Pair"] *)
    *)
    ]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain "Pair Nests"

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data structure:
---------------

A Pair Nest has the form

   PaN[ ]

   or

   PaN[ pa, pan],

   where pa is a  pair and pan is a Pair Nest.

*)

FirstPair[ PaN[ pa_, pan_]] := pa
RemainingPairs[ PaN[ pa_, pan_]] := pan

PaN/:
Union[ pan1_PaN, pan2_PaN] := UnionN[ pan1, pan2, IsLCMLess]

InitialPairs[ pn_PN] :=
  InitialPairs$[ PaN[ ], pn]
```

```
InitialPairs$[ pan_, PN[ p_, pn_]] :=
  InitialPairs$[ UpdatedPairs[ pan, pn, p], pn]

InitialPairs$[ pan_, PN[]] := pan

UpdatedPairs[ pan_, PN[ p0_, pnRest_], p_LabP] :=
  UpdatedPairs[
    InsertN[ NewPair[p, p0], pan, IsLCMLess], pnRest, p
    ]

UpdatedPairs[ pan_, PN[ ], p_LabP] := pan

UpdatedPairs[ pan_, pn1_, PN[ p_, pn2_]] :=
  UpdatedPairs[ UpdatedPairs[ pan, pn1, p], pn1, pn2]

UpdatedPairs[ pan_, pn1_, PN[ ]] := pan

MultipleCriterion[ PPa[ p1_, p2_], p_, pairs_] :=
  p != p1 &&
  p != p2 &&
  IsMultiple[ LCM[ PPa[ p1, p2]], LeadingPowerProduct[ p]] &&
  (!IsInPairs[ NewPair[ p1, p], pairs]) &&
  (!IsInPairs[ NewPair[ p, p2], pairs])

EmptyPairs[ pn_] := PaN[ ]

IsVoid[ pan_PaN] :=
  IsEmptyN[ pan]

IsInPairs[ pa_, pan_PaN] :=
  IsMemberInN[ pa, pan, Equal]

CanceledPairs[ p_LabP, pan_PaN] :=
  DeleteN[ pan, Occurs[ p, #] &]

CanceledPairs[ PN[ p_, pn_], pan_PaN] :=
  CanceledPairs[ pn, CanceledPairs[ p, pan]]

CanceledPairs[ PN[ ], pan_PaN] :=  pan

(* Algorithms for Version "Criteria At Update Time": *)

InitialPairsWithCriteria[ pn_PN] :=
  ThinnedByChainCriterion[
    InitialPairsWithProductCriterion[ pn], pn, TrivialPair[ ]
    ]

UpdatedPairsWithCriteria[ pan_PaN, pair_, pn_PN, p_] :=
  ThinnedByChainCriterion[
    UpdatedPairsWithProductCriterion[ pan, pn, p], pn, pair
    ]

InitialPairsWithProductCriterion[ pn_PN] :=
  InitialPairsWithProductCriterion$[ PaN[ ], pn]

InitialPairsWithProductCriterion$[ pan_, PN[ p_, pn_]] :=
```

```
  InitialPairsWithProductCriterion$[
    UpdatedPairsWithProductCriterion[ pan, pn, p], pn
    ]

InitialPairsWithProductCriterion$[ pan_, PN[]] := pan

UpdatedPairsWithProductCriterion[ pan_, PN[ p0_, pnRest_], p_] :=
  Block[ { newPair = NewPair[ p0,p]},
    If[ ProductCriterion[ newPair],
      UpdatedPairsWithProductCriterion[ pan, pnRest, p],
      UpdatedPairsWithProductCriterion[
        InsertN[ newPair, pan, IsLabeledLCMLess], pnRest, p
        ]
      ]
    ]

UpdatedPairsWithProductCriterion[ pan_, PN[ ], p_] := pan

ThinnedByChainCriterion[  PaN[ pa_, pan_], pn_, pair_] :=
  If[ ChainCriterion[ pa, pn, pair],
    ThinnedByChainCriterion[ pan, pn, pair],
    PaN[ pa, ThinnedByChainCriterion[ pan, pn, pair]]
    ]

ThinnedByChainCriterion[  PaN[ ], pn_, pair_] := PaN[ ]

ChainCriterion[ pa_, PN[ p_, pn_], pair_] :=
  Which[ TriangleCriterion[ pa, p, pair], True,
    $GreaterPPa[ LeadingPowerProduct[ p], LCM[ pa]], False,
    True, ChainCriterion[ pa, pn, pair]
    ]

ChainCriterion[ pa_, PN[ ], pair_] := False

(* End of Algorithms for Version "Criteria At Update Time": *)

P[ pan_PaN] :=
  Block[ {},
    Print[ "Pair Set"];
    P[ pan, Nest];
    Print[ "End of Pair Set"]
    ]

P[ pan_PaN, N] :=
   Print[ "Number of Pairs: ", LengthN[ pan]]

(*

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Poly Nests:
   Groebner Bases Algorithm

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
*)

GB[ pn_PN] :=
  (* pn ...  a Poly Nest consisting of
             of distributive nest polys.
  *)
  $GroebnerBasisPN[ pn]
  (* Alternatives:
     GroebnerBasisPN
     GroebnerBasisPNSmallPairSets
     GroebnerBasisPNAllReduced
  *)

GroebnerBasisPN[ pn_PN] :=
  Block[
    { basis,
      pairs, pair, sp, nf
    },

    $Label = 0;
    basis = Labeled[ Normalized[ Sort[ pn]]];
    P[ basis];
    PrintGlobalVariables[ ];

    pairs = InitialPairs[ basis];
(*
    P[ pairs];
*)
    While[ !IsVoid[ pairs],
      pair = FirstPair[ pairs];
 (*
      P[ pair];
*)
      p1 = FirstPolynomial[ pair];
      p2 = SecondPolynomial[ pair];
      pairs = RemainingPairs[ pairs];

      If[ !Criteria[ pair, pairs, basis],
        nf = NormalForm[ SPolynomial[ p1, p2], basis];

        If[ nf != Zero[ nf],

          nf = Labeled[ Normalized[ nf]];

          Print["New Poly"]; P[ nf, Label];
          P[ PowerProduct[ LeadingMonomial[ nf]]];

          pairs = UpdatedPairs[ pairs, basis, nf];
          basis = Augmented[ basis, nf],

          Print["Zero Reduction"]
          ]
```

```
       ];
 (*
      P[ pairs]
 *)
      ];
    basis = InterReduced[  basis];
    basis = Unlabeled[ basis]
    ]

Criteria[ pair_, pairs_, basis_] :=
  Which[
    ProductCriterion[ pair],
      Print[ "Product Criterion"];   Return[ True],
    ExplicitChainCriterion[ pair, basis, pairs],
      Print[ "Chain Criterion"];   Return[ True],
    True, False]

ExplicitChainCriterion[ pair_, pn_, pairs_] :=
  ExistsN[ pn, MultipleCriterion[ pair, #, pairs] &]

(*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Domain Poly Nests:
   Groebner Bases Algorithm with
   Criterion Application Before Pair Set Update

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
*)

GroebnerBasisPNSmallPairSets[ pn_PN] :=
  Block[
    { basis,
      pairs, pair, sp, nf
    },
    $Label = 0;
    basis = Labeled[ Normalized[ Sort[ pn]]];
    P[ basis];
    PrintGlobalVariables[ ];
    pairs = InitialPairsWithCriteria[ basis];
    P[ pairs];  Print[ "------------------------"];
    While[ !IsVoid[ pairs],
      pair = FirstPair[ pairs];
      P[ pair];
      p1 = FirstPolynomial[ pair];
      p2 = SecondPolynomial[ pair];
      pairs = RemainingPairs[ pairs];
```

```
      nf = NormalForm[ SPolynomial[ p1, p2], basis];
      If[ nf != Zero[ nf],
        nf = Labeled[ Normalized[ nf]];
        Print["New Poly"]; P[ nf, Label];
        P[ PowerProduct[ LeadingMonomial[ nf]]];
        pairs = UpdatedPairsWithProductCriterion[ pairs, basis, nf];
        basis = Augmented[ basis, nf];
        pairs = ThinnedByChainCriterion[ pairs, basis, pair],
        Print["Zero Reduction"]
        ];
      P[ pairs]; Print[ "-------------------------"];
      ];
    basis = InterReduced[ basis];
    (* P[basis];  *)
    basis = Unlabeled[ basis]
    ]
(*
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
Domain Poly Nests:
   Groebner Bases Algorithm with
   InterReduction Before S-Polynomials Consideration
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
*)
GroebnerBasisPNAllReduced[ pn_PN] :=
  Block[
    { reducibles , reducers, basis, pairs,
      pair, p1, p2, nf, reducibleReducers
      },
    $Label = 0;
    basis = Labeled[ Normalized[ Sort[ pn]]];
    P[ basis];
    PrintGlobalVariables[ ];
    reducibles = basis; reducers = Empty[ basis];
    basis = Empty[ basis]; pairs = EmptyPairs[ basis];
    {basis, pairs} =
       AllReduced[ reducibles, reducers, basis, pairs];
(*
    P[ basis, S]; (* Print[ "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"]; *)
    P[ pairs];  Print[ "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"];
*)
```

```
    While[ ! IsVoid[ pairs],
       pair = FirstPair[ pairs];
(*
       P[ pair];
*)
       p1 = FirstPolynomial[ pair];
       p2 = SecondPolynomial[ pair];
       pairs = RemainingPairs[ pairs];

       If[ !Criteria[ pair, pairs, basis],
         nf = NormalForm[ SPolynomial[ p1, p2], basis];

         If[ nf != Zero[ nf],
           nf = Labeled[ Normalized[ nf]];

           Print["New Poly"]; P[ nf, Label];
           P[ PowerProduct[ LeadingMonomial[ nf]]];
           Print[ "--------------------"];

           {reducibles, basis} = Multiples[ nf, basis];
           pairs = CanceledPairs[ reducibles, pairs];
           {basis, pairs} =
             AllReduced[
               reducibles, Singleton[ nf, basis], basis, pairs
               ],
           Print["Zero Reduction"]
           ]

         ];
(*
       P[ basis, S]; (* Print[ "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"];*)
       P[ pairs];  Print[ "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"];
*)
       ];

    basis = Unlabeled[ basis]
    ]
```

Acknowledgment. The research described in this paper was sponsored by the Austrian Ministry for Science and Research in the frame of the "Gröbner-Bases Project 1990". I am also grateful to my student Wolfgang Windsteiger who helped in coding part of the package and in writing this report.

References

[Buchberger 85] B.Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. Chapter 6, pp. 184-232 in: N.K. Bose: *Multidimensional Systems Theory*, D. Reidel Publishing Company, 1985.

[Buchberger 90] B.Buchberger. Fermat's ideal can be generated by the first three Fermat polynomials. *RISC-Series* 90-44, University of Linz, Austria, 1990.

[Buchberger 91] B.Buchberger. Gröbner bases in Mathematica: enthusiasm and frustration. Technical Report, *RISC-Linz*, University of Linz, Austria, 1991, submitted to the IFIP Workshop on Scientific Computation, Karlsruhe, September 1991.

[Buchberger, Collins, Hong 91] B.Buchberger, G.E.Collins, H.Hong. Design principles of the new SAC-2C algorithm library. *RISC-Series* 91-11, University of Linz, Austria, 1991.

[Robbiano 85] L.Robbiano. Term orderings on the polynomial ring. *EUROCAL '85*, Springer LNCS Vol.204, 1985.

[Windsteiger 90] W.Windsteiger. Term orderings by matrices. Internal Report, *RISC-Linz*, University of Linz, Austria, 1990.

[Wolfram 88] S.Wolfram. *Mathematica. A System for Doing Mathematics by Computers.* Addison Wesley, Redwood City, 1988.