D o C o n

The Algebraic Domain Constructor

V e r s i o n 2.12

U s e r M a n u a l

Sergey D. Mechveliani

Pereslavl - Zalessky, November 2012.

Contents

1 Introduction

1.1 Aim

The first attempt with `DoCon` program [Me1] dates of 1989. The reason for starting the project was absence of any freely available, powerful and generic enough tool for programming mathematics.

The aims of the `DoCon` project are

- to provide the author himself with the computation tool based on categorial approach and functional programming,
- to make this approach to mathematical computation programming freely accessible, comprehensive and transparent at all design levels.

Disclaimer By the word "categorial" we do not mean applying the real *category theory*. The categorial approach in `DoCon` means the following.
1) Using certain concrete classical categories of `Ring, EuclideanRing, Field`, and so on.
2) Inviting the programmer to continue this with the user-defined categories.
3) Arithmetic and some other operations are defined under the very generic assumptions: "over any Euclidean ring", over any field, and so on. The Domain Constructors are supported: `Fraction, Polynomial, Residue ring`, and others, that is certain set of operations are defined automatically, following the domain constructors. So, programmed are the needed algebraic *functors*. This makes programming of mathematics more generic, and makes it to follow the style of the mathematical textbooks of the last hundred years (see below in this section the example with the sphere).

Now, it starts a long introduction. But

(1) read first the paper [Me2] to get the general author's idea about the relation of `Haskell` to computer algebra,

(2) Section 3.4 formulates briefly the whole `DoCon` notion.

So, if the reader is familiar with the `Haskell` type classes and wants to know immediately what really `DoCon` is, one should skip directly to the Section 3.4.

The following example is for the readers not familiar with the "categorial" kind of CA programs. It shows what it is possible to do with the CA system based on the category principle. Suppose we have to represent symbolically an algebraic surface, say a sphere

$$S: \ x^2 + y^2 + z^2 = 1$$

in the rational space $\mathbb{Q}^3$. It is known that the geometry of a surface of such kind is given by the field $K = \mathbb{Q}(S)$ of rational functions on S. Mapping the rational coordinates x, y to S and defining z by an algebraic equation over $\mathbb{Q}(x, y)$ we express K as a composition of the functors `Polynomial, Fraction, Residue ring`:

```
Q(x,y) = Fraction (Z[x,y]);
K      = Q(x,y)[z]/(p);     p = x^2 + y^2 + z^2-1.
```

After scripting this in the `DoCon` constructors, `DoCon` evaluates expressions like

```
(1+z)*(1-z)/(x^2 + y^2)  :: K
```

being able to convert them automatically to the canonical form and to do the arithmetic operations on them. So, applying the constructors, we build automatically the algebraic representation of such surface.

The implementation idea for this is that any domain constructor C is considered as a functor that builds certain algebraically meaningful algorithms for the result domain starting with appropriate algorithms for the argument domains. For example, "to program polynomial arithmetics" means to relate the above functors to the constructor `Pol`. Given the arithmetic and certain other algorithms for the domain `R`, these functors build a certain subset of the aforementioned algorithms for the domain `(Pol R)`.

The representation and processing mechanism for an algebraic domain depends on the programming tool. `DoCon` applies the `Haskell` language [Ha, HFP] for this, and models a category mostly as a *class*, a domain — as several *class instances*, together with certain explicit domain terms.

1.1.1 Acknowledgements

The author is grateful to

- Dr. S.V.Duzhin for his administrative support during many years,
- Marc van Dongen (Cork College, Ireland)
  for the consultation on Gröbner bases and programming matters,
- Russian Foundation for Basic Research:
  in 1998-99 `DoCon` was the part of the project supported by the grant No 98-01-00980,
- INTAS foundation of European Community:
  in 1995-96 `DoCon` was the part of the project supported by INTAS grant (coordinator Professor B.Jacob),
- Computer Algebra Center MEDICIS:
  Unite' Mixte de Service (under the supervision of CNRS and Ecole Polytechnique)
  `<http//www.medicis.polytechnique.fr>`
  for kindly making it possible to compare `DoCon` and other CA systems at the same machine (Section 51).

1.1.2 Programming language

The main `Haskell` language [Ha] features are

- Typed lambda calculus machine as the basic computation model. The program compiles into something like typed lambda expression and runs by reducing to the head normal form.
- Pure functionality.
- High order functions (a consequence of the lambda model).
- Pattern matching.
- 'lazy' (non-strict) evaluation — the head normal form computation strategy.
- Polymorphic values.
  The type of expression is defined by the constructors and type parameters. Say,
  `Ord a => (Char, [a], U a b)` defines the type of triples with the components of different types depending on the parameter types `a, b`, where `Char, []` are the standard constructors for Character and `List, U` introduced by the user.
- Recursive type descriptions. Example:
  `data Tree a = Leaf a | Node a (Tree a) (Tree a)`
- Strong static typing: the types are resolved at the compilation stage.
- Data classes and instances.
  This is something to replace the object oriented programming.
  Mathematically, it is similar to joining algebraic domains into categories.

See [HFP] as the introduction into programming in `Haskell`.
`Haskell` belongs to the so-called `Miranda` family of languages.

(`Miranda` is a trademark of Research Software Ltd.)

The particular features of the language are discussed specially in the Section 1.5; the summary of further requirements to the language is given in Section 48. We also hope for the future extended versions of the language and its implementations.

`DoCon` is programmed in what we call here `Haskell-2-pre`.

`Haskell-2-pre` === `Haskell-2010` + Extension (functional).

`Haskell-2010` is the standard accepted in 2010 [Ha].

`Haskell-2` is the future language extension project (it is delayed, being under a long discussion).

`Haskell-2-pre` is a certain subset of the `Haskell-2` language that was implemented in 1990-ties by at least two systems.

Extension ===
(mp) multi-parameter classes,
(oi) overlapping instances,
(*) composed data constructors and repeated type variables in the instances.

Example with (mp): `class (AddGroup a, Ring r) => LeftModule r a where ...`
Example with (oi):

```
instance GCDRing a => Ring (Fraction a)       where ...
instance              Ring (Fraction Integer) where ...
```

Here the second instance defines some operations in a more special way, since `'a = Integer'` tells more than only `GCDRing a`. Similarly, the domain attributes for the type `EuclideanRing a => ResidueE a` can be computed more precisely for the special case of `a = Field k => UPol k`.

Libraries used:
Among the non-standard libraries, `DoCon` imports `Data.Set, Data.Map` and `Debug.Trace`. (see documentation on [GH]). We hope, `Set` amd `Map` will become standard in the future `Haskell`.

From `Debug.Trace`, `DoCon` uses only the function `trace`. This usage is very slight: for some `DoCon` functions it is appropriate to issue intermediate messages, if the corresponding mode is given.

1.2 Current `DoCon` abilities

Mathematical library:

- Permutation group: composition, inversion, decomposition to cycles.
- Fraction field over a gcd-ring: arithmetic.
- Linear algebra over an Euclidean domain: vector, matrix arithmetic (dense form), reduction to the staircase and diagonal form of matrix, solving a linear system.
- Polynomial arithmetic in $P = R[x_1, \ldots, x_n]$, R a commutative ring. Possible models for P, free module over P, symmetric functions over R are unified into the `class PolLike` and are presented by the constructors
  `UPol` (sparse univariate polynomial),
  `Pol` (multivariate polynomial with dense power products),
  `RPol` ("recursive" form), `EPol` (vector over P), `SymPol` (symmetric function).
- g.c.d. in $R[x_1, \ldots, x_n]$, R a factorial ring with given `gcd` function.
- Gröbner basis, normal form, syzygy generators functions for $P = R[x_1, \ldots, x_n]$, R an Euclidean ring, and in a free module over P [Bu, Mo, MoM].
- Some items for non-commutative polynomials over a commutative ring: arithmetics, reduction to normal form, some others.
- Symmetric function package and partition operations [Ma].
- Factorization in $k[x, y]$ for a finite field k [CG, Me3], building finite extension of k of given dimension, Hensel lift in $R[x]$ for an Euclidean R.
- Special interpolation technique for $k[x]$, k a finite field, its application to determinant computation over $k[x]$.

Category hierarchy expressed partially via the data classes:

- `Set, Semigroup, Group, Ring, LinSolvRing ... LeftModule ...`
- some operations with the description terms of `Subset, Subgroup, Subring, Ideal ...`

Domain constructors:

- `Permutation`.
- `Fraction` field for a gcd-ring.
- Direct product of Sets, (semi)Groups, Rings, Free (vector) module over a ring.
- `Matrix` algebra over a commutative ring.

- `Polynomial` algebra over a commutative ring:
  the `UPol, Pol, RPol` models.
- `Vector` module over a ring P, in some cases, with the Gröbner bases structure.
- `Residue` ring by an ideal:
  the `ResidueI, ResidueE` models for the generic and Euclidean cases.

Property processing:
evaluation of certain small set of the most important algebraic domain property values is supported, such as `Finite, IsCyclicGroup, IsMaxIdeal, Factorial` (ring), and such.
They serve as correctness conditions for various methods, are used to determine dynamically the membership to categories, and also present an important information by themself.

1.3 Examples of possible contribution

The scientific library described in previous subsection has to grow. It is a good idea to write a package, consistent with `DoCon`, adding some methods. Let us list the tasks, that we would like to program in the first order.

- **(an)** Setting an interface to some freely available advanced computational number theory library (maybe, by Lenstra).
  This will join many efficiently implemented methods, such as of the large integer factorization, for integer extensions, and others.
- Permutations implemented via binary trees (for large sets).
- Chinese remainder method for GCD in $\mathbb{Z}[x]$, $R[x_1, \ldots, x_n]$ (so far it is done only for $k[t][x]$, k a finite field).
- Chinese remainder method for various tasks, like solving systems, etc.
- Gröbner bases: optimizations for syzygies, Gröbner walk, change of Gröbner basis, and such.
- Factoring in $R[x_1, \ldots, x_n]$ for generic enough R.
  Primary decomposition of ideal in $R[x_1, \ldots, x_n]$.
  Efficient methods for Primitive element and other attributes for a field extension.
- Fraction ring by a multiplicative subset, local ring by a prime ideal.
- **(tr)** Term rewriting, prover based on it, AC-completion, and so on.

Remark on (tr)
It can be based on the conditional term rewriting, various versions of unfailing completion procedure, equational prover.

Currently the author is developing such a library (`Dumatel`) aiming to link it to `DoCon`. We have implemented the many-sorted (unconditional) term rewriting, completion, AC-completion, with proof search in the predicate logic via this completion, inductive prover, with a certain proof search strategy, and with heuristics for the lemma invention.

And there always remains a huge space for the contributions in this area.

Also there are possible various optimizations in `DoCon`.

Performance comparison

The Section 51 presents certain bit of this, relatively to `Axiom` [Al, Je] and `MuPAD` [Mu] programs.

But any further experiments would be interesting.

1.4 Comparing it to other CA systems. On Aldor, Axiom

The first profound categorial approach to programming of mathematics had started before 1980 and has grown into the `Aldor` language [Al] and `Axiom` system [Je].

`Axiom` can be considered as a large algebraic library for `Aldor`.
In 2002, `Axiom` was said being rewritten to `Aldor`.

And it looks like `Axiom` has somewhat out of date language.

`Aldor` is a language somewhat similar to `ML`, and with some respects — to `Haskell`. But it is extended with the *dependent types* feature.

The algebaic domains are represented as *abstract data types*. The (abstract) types can be treated as regular *values* and their resolution can be delayed to the run-time.

`Aldor` is non-functonal and 'strict'. `Aldor` has separated from the `Axiom` system in about 2000. Before this event, it was hard to understand where in `Axiom` was a language and where a library.

We think, the CA language and system developers and implementors have now a task to provide a tool which

- has the main part of `Aldor` paradigm (like dependent types),
- is functional, 'lazy' and elegant as `Haskell`,
- is extended with some library for a prover: term rewriting, completion equational reasoning,
- has several implementations distributing freely with source.

The `Cayenne` language [Au] steps in this path, but it needs some extension and reliable implementation.

Such a rich type system as `Aldor`'s is 'undecidable'. But it is not neccessary to solve all types at compilation time. Some of them can be solved at the run-time.

Here are the main points of DoCon difference with respect to `Aldor-Axiom`.

- `DoCon` is a library for `Haskell` written in `Haskell`.
  `Axiom` is a library for `Aldor` written in `Aldor`.
- `DoCon` applies functional programming and 'lazy' evaluation — a consquence of using `Haskell`.
- `DoCon` uses the so-called sample argument approach to replace the dependent types tool of `Aldor`, in particular — to model a domain depending on a dynamic value. This leads to some complications in architecture.
- `DoCon` has a comparably small library of algorithms, certain part of commutative algebra, it needs developing.
- `DoCon` distributes freely with source, as well as `Haskell` implementations do.
  In 2003 `Axiom` was announced as open-source and free.
  In April 2002 `Aldor` has become freely-executable
  (but not open-source).
- `Haskell` has 2-3 reliable implementations.
  `Aldor` has one, and we have to test its reliability.

The `Weyl` program, as it is described in [Zi] in 1993, looked like developing in the similar directions — from algebraic point of view. Though `Common Lisp`, its object system, bring in a very different programming style and architecture.

The `MuPAD` system [Mu] claims that it supports the domains and categories. The `C++` language for the kernel and the `MuPAD` language, its *procedures* and other features, show that this system is not functional, and again, assumes a very different programming technology.

Recalling the situation of about 2000, the advanced release of `MuPAD` was commercial. We do not know what is the current status, what open-source release is available.

Not only `DoCon` experiments with the `Haskell` application to algebra. Thus, [Fo, Ka] describe certain plain approach to the subject.

Also it is a good idea to read a paper [Me2] before dealing with `DoCon`.

The main architecture points relatively to [Fo] are:

- In `DoCon`, the Set of a domain is not only a type, it is defined also by the so-called base set term.
- `DoCon` can do some operations with the domain description terms.
- `DoCon` operates with the residue group and residue ring elements and provides in a certain generic case a canonical map modulo subgroup, ideal (so far, for the commutative case only).

- For some reasons, `DoCon` treats the zero, unity, characteristic, and other constants of a domain `'a'` not as formal constants, — say `zero :: a`, — but as the maps, say `zero :: a -> a`, with sample argument. Though, each such function has actually to be constant (Section 3.4, SA).

1.5 More on the programming language

The `Haskell` language is a comprehensive tool for scientific programming — among the bets ones but far from being perfect. A general scheme for the user for such programming in the `DoCon` package environment is to set the algorithms in `Haskell` importing the `DoCon` entities from `DoCon` interface (or its library).

But one needs to take in account certain particular properties of the `Haskell` language.

Functionality, 'laziness' and efficiency

`DoCon` exploits a non-strict computation, for this helps to write clear programs. For example, the function for the staircase form of matrix returns

```
(s, t, sign),
```

where `s` is the resulting staircase matrix, `t` the transformation matrix, `sign` the row permutation sign of transformation.

If the program does not use further the variables `t`, `sign`, then the whole corresponding subtree of computation would never perform.

If evaluation was strict (as, say in Common Lisp) we would have to provide extra (faster) function for the case when `t` is not needed. There are other examples when a non-strict evaluation helps to bring more clarity to the program.

But we have to pay for the advantage. For a long time, no efficient implementations for 'lazy' languages were known. In 1990-ies, a significant progress has been achieved, maybe, due to implementation of the so-called graph reduction model of evaluation [Jo, FH].

Also many people consider pure functional programming as impractical, saying there is a necessity to use a pointer, to update a list "in place" or introduce a mutable array.

Concerning this, `DoCon` insists on the total functionality and tries to prove on practice that the advantages cost the restrictions. And the Section 51 reveals a good practical performance test.

Among usually mentioned benefits of functionality, we stress the one that we consider as the most important:
a functional program, when it is brought to the kernel language, is an explicit symbolic expression (compare it to polynomial in mathematics) which is transformed step by step to the 'reduced' expression — even if not all of arguments are given.

Similarly as $(x+y)*(x-y)$ "simplifies" to x^2-y^2 in algebra, the program term is simplified by the interpreter (and partly, by the compiler) to another, equivalent one. This provides a large field for automatic symbolic optimization of programs.

The main point here is:
the programs defining the same mathematical data map should be considered as equivalent — this should be the intention of a functional tool.

Most 'functional' systems (and Haskell) do not satisfy this property in full. This is why we take sometimes the words "purely functional" in quotes.

The reader may ask: "how a real program can be functional, when input and output of the data is a side effect?"

We think that a program can be "less functional" or "more functional". And qualify `DoCon` as functional, because it is itself free of side effects. The user program P that computes something with `DoCon` and outputs the result is almost functional. Because the most part of the program P + `DoCon` can be a subject of symbolic simplification.

Besides, the `Haskell` approach with *monads* still makes input/output functional (we skip this feature).

On period and dollar denotation
In the sequel, we apply in Haskell programs the usual Haskell denotations of `'.'` and `'$'` for the composition of fuctions and fuction application respectively. This helps to write many composition levels with less parentheses.

Example.

```
map (+1) . map (^2)  :: [Integer] -> [Integer]
```

means composing the above two maps on integer lists. Thus,

```
(map (+1) . map (^2)) [1,2]      :: [Integer]  --> [2,5],
sum $ map (+1) $ map (^2) [2,5]  :: Integer    --> 31
```

Illustration for 'laziness'
The below example with generating of the infinite list of primes and its re-use bases on the 'lazy' evaluation principle
(this program is elegant, but note that for large numbers, there exist more efficient methods).

```
primes' = s [2..]  :: [Int]                        -- sieve method
          where
          s (p: ns) =  p: (s (filter (notm p) ns))
          notm p n = (mod n p) /= 0       -- "n is not a multiple of p"

p n = primes'!!(n+1)                               -- (P1)
      -- (primes'!!n, primes'!!(n+1))              -- (P2)
      -- (primes'!!n, primes'!!(n+1), primes'!!500)   -- (P3)
```

For example, to compute `(p 1000)`, `primes'` evaluates as follows:

```
2: (s (filter (notm 2) [3..]))        = 2:(s $ filter (notm 2) [3..])
2: (s (3:(filter (notm 2) [4..])))
2:3:(s$  filter (notm 3)$ filter (notm 2) [4..])
2:3:(s$ filter (notm 3)$ (5:(filter (notm 2) [6..])))
2:3:(s (5 :(filter (notm 3)$ filter (notm 2) [6..])))
2:3:5:(s$ filter (notm 5)$ filter (notm 3)$ filter (notm 2) [6..])
...
```

An important point: how much may differ the computation costs for the expressions `(P1)`, `(P2)`, `(P3)` ?
`DoCon` relies only on such (reasonable) systems in which `(P2)`, `(P3)` do *not* cost any essentially more than (P1). And why `(P2)`, `(P3)` are not likely to cost essentially more than `(P1)` ?

First, the `(!!)` function invocates not more than 3 times, and this costs much less than the whole program. Second, the first (n+1) values in `primes'` needed in `(P2)` for the first component of the pair also participate in evaluation of the second component; the variable (value) `primes'` being copied. Then, the interpreter re-uses these values — this is "sharing". Commonly, this happens when some program variable `u` computes as some simple expression of `v`. Then the value of `v`, if ready, is re-used to find `u`. If `v` is not needed for the result (not referenced from other data), then its value joins the "garbage".

`DoCon` assumes the programming style in which the expressions like `(P2)`, `(P3)` may often appear. This makes the programs more clear.

More example.
One of my mathematician colleagues needed to check the condition of `(det M) == 0` for a certain concrete matrix `M` over polynomials. He asked me to try `DoCon`, because running the program like `(det M) == 0` in the `Maple` system overfilled the memory after a long computation. I tried a similar program in `DoCon`, and it has resulted in `False` in a couple of seconds. This has happened due to the following reasons.
1) A polynomial is naturally represented (at least, in `DoCon`) as a monomial list ordered by a certain degree comparison, and equality to zero is checked by testing emptyness of the monomial list.
2) `det M` has occured a very long monomial list.
3) `Maple` finds first the whole polynomial `det M`, and then compares it to zero.
4) `Haskell` computes lazily. The leading monomial `m1` has been obtained, and the computation state has occured `null (m1: (f ...))`, where `f` is a certain function that forms the tail of `det mM`. By the 'lazy' computation, this immediately results to `False`.

A bit more generally: consider applying, for example, `null (f x)` for some function `f :: a -> [b]` doing a large computation and returning a list. It is most natural for the user to write such an expression with `'null'`. Quite often, on average, computing it in a 'strict' system will cost extremely much, while computing it in a 'lazy' system will cost extremely small.

Contra Example.
But there also exists unneeded laziness. For example, the program `maximum [1 .. b]`, where

```
-- the below is taken from the Haskell Prelude

maximum [] = error "maximum [] \n"
maximum xs = foldl1 max xs

max x y = if  x < y  then y  else x

foldl1 f (x: xs) = foldl f x xs
foldl1 _ []      = error "foldl1 _ [] \n"

foldl f z []      = z
foldl f z (x: xs) = foldl f (f z x) xs
```

requires, in many implementations, the stack size proportional to `b` to find `maximum [1 .. b]`. Because evaluating it really 'lazily', produces the data sequence

```
foldl1 max [1..b]  =
foldl max 1 [2..] =
foldl max (max 1 2) [3..] =
foldl max (max (max 1 2) 3) [4..] =
...
```

That is the iteration style program may occur 'recursive' when computed lazily. Here the list `[1 .. b]` unwinds lazily all right, but the nested delayed `max` applications are accumulated eagerly.

Changing the evaluation strategy, the interpreter might compute this so that neither the list `[1 .. b]` nor the `max` application terms grow more than to one cell. But it requires a compiler of infinite intellect to find such strategy change in generic case.

At this point, we soothe ourself with the following considerations.

- The evaluation cost is $R + C \cdot A$,
  where R is the number of "reduction steps",
  A number of cell allocations (for new data), C a constant.
  And in the above example, the ratio of the lazy evaluation cost to the best strategy one is $(b + C \cdot b)/b = 1 + C$ — a constant. That is the best strategy costs near the same as the purely lazy one. Only the space expenses may sometimes turn into the proportional time ones. If this is not a generic law, then it would be of great interest to observe any contra example.

- In easy cases, as the aforementioned `maximum`, it is not too hard for the compiler to improve the unneeded laziness.
- The practical average thrust of unneeded laziness does not look great — see the examples in Section 51.

2 Getting started

Here follow some simple examples of how to arrange work with `DoCon`.

First, as the file `install.txt` says, 'make' `DoCon` and run the test. It shows many things, and looking into the modules

`.../demotest/T_*.hs`

you get an impression of how to set computation.

Nevertheless, we give

2.1 Starting example 1

```
module Foo where                                   -- in the file  Foo.hs
import Categs     (Factorization))
import RingModule (FactorizationRing(..))
import Z ()

f :: Integer -> Factorization Integer      -- just define some function
f n =  factor (n + 2)
```

Here the three lines of `'import'` can be replaced by a single one:

`import DExport`

— which means "import all the DoCon interface".

2.1.1 'Making' executable and running

To use the above program create the file `Main.hs`:

```
module Main where
import Foo (f)
main = putStr str
       where          -- example of setting the head function
       n = 10         -- edit this

       str = concat ["f ", shows n " = ", shows (f n) "\n"]
```

To 'make' and run this program under the `GHC-7.0.1` system, command

```
ghc $doconCpOpt --make Main
./Main +RTS -M20m -RTS
```

This prints: `f 10 = [(3,1),(2,2)]`.

The option `'+RTS -M... -RTS'` means restricting the memory space for computation.

For this example, and also for the whole this manual, we presume that the `docon` *package* is installed following the guide `install.txt`, and (according to this guide too) the environment variables are set:

```
setenv doconSource  ... <path to the DoCon directory .../source>
setenv doconCpOpt "-fwarn-unused-matches -fwarn-unused-binds   \
         -fwarn-unused-imports -fno-warn-overlapping-patterns  \
         -XRecordWildCards -XNamedFieldPuns -XFlexibleContexts \
         -XMultiParamTypeClasses -XUndecidableInstances        \
         -XTypeSynonymInstances -XFlexibleInstances -XOverlappingInstances"
```

Thus, our command `ghc $doconCpOpt --make Main` uses this environment variable. Also it uses that `ghc` will find the `DoCon` interface and object library according to the configuration for the `docon` package set in the file `.../source/docon.cabal`.

Note: it does not matter here in what directory the file `Foo.hs` resides. Of course, it is better to put a user program apart from the `DoCon` source.

2.1.2 Interpreting the program

To run this program by the interpreter of the `GHC-7.0.1` system, command, for example,

```
ghci  -package docon   Foo   +RTS -M20m -RTS
...
Foo> :set +s
Foo> f 10
[(3,1),(2,2)]
(0.01 secs, 0 bytes)
Foo>
```

Here `ghci` is the interpreter and interactive system of GHC,

Using compiled module:

```
ghc -c $doconCpOpt Foo.hs
...
ghci -package docon  Foo
...
```

Compiling from under `ghci` is done by

```
...> :! ghc -c $doconCpOpt Foo.hs
```

2.2 Starting example 2

This program gives a more definite idea of how to program in DoCon. It

- builds integer matrix `M` of size 3 x 6;
- finds the generic solution of equation `M*x = 0` over the ring `Z` of integers;
- adds rows to `M` to obtain the square matrix `M'`;
- makes a rational matrix `qM'` by mapping `:/1` to the elements of `M'`;
- evaluates `iM' = inverse qM'`;
- finds `iM'*qM'` to test that it is unity;
- builds the residue ring `R = Z/(5)`, finds its property list and evaluates `2/3` in `R`;
- parses from the string the polynomials `f`, `g` in variables `["x","y"]` with the coefficients from `R`,
- evaluates `gcd f g`;
- prints (unparses and outputs) all these results to the file `./result`

```
import DPrelude   (Z, ctr, smParse, mapmap                  )
import Categs     (Subring(..)                              )
import SetGroup   (zeroS                                    )
import RingModule (Ring(..), GCDRing(..), upField, eucIdeal)
import Z          (dZ                                       )
import VecMatr    (Matrix(..), scalarMt                     )
import Fraction   (Fraction(..)                             )
import Residue    (ResidueE(..)                             )
import Pol        (PolLike(..), Pol(), degRevLex, cToPol    )
import LinAlg (solveLinearTriangular, solveLinear_euc, inverseMatr_euc)
--
-- all the above import can be replaced with  import DExport

type Q = Fraction Z   -- rational number field
type R = ResidueE Z   -- for  Z/(b)
type P = Pol R        -- for  R[x,y]
main =
  let
    un  = 1     :: Z
    unQ = 1:/1  :: Q
    zrQ = zeroS unQ                   -- zero of domain of unQ
    dQ  = upField unQ Map.empty       -- domain term of Q
    mM  = [[1,2,3,4,5,6],
           [5,0,6,7,1,0],
           [8,9,0,1,2,3]]  :: [[Z]]
```

```
  v         = [0,0,0] :: [Z]           -- right-hand part of system
  (_, ker) = solveLinear_euc mM v
  rows'     = mM ++ [[0,2,0,4,0,2],[5,0,6,1,2,0],[0,2,0,1,2,1]]
  qM'       = mapmap (:/ un) rows'            -- making rationals
  iM'       = Mt (inverseMatr_euc qM') dQ     -- \m-> Mt m <dom>  makes true matrix
  Mt mPrd _  = (Mt qM' dQ)*iM'              -- matrix product
  unM         = scalarMt qM' unQ zrQ         -- unity matrix of given size
  checkInvM' = mPrd==unM
  -----------------------------------------------------------------
  b  = 5 :: Z                         -- arithmetics in R = Z/(b)
  iI = eucIdeal "bef" b [] [] []      -- makes Ideal(b) in R
  r1 = Rse un iI dZ                   -- unity residue by ideal iI
  dR = upField r1 Map.empty           -- domain term for R = Z/(b)
  (_, rR)    = baseRing r1 dR         -- base ring to which r1 belongs
  rProps     = subringProps rR
  [r2,r3,r4] = map (ctr r1) ([2..4] :: [Z])  -- each n projects to domain of r1
  q = r2/r3
  -----------------------------------------------------------------
  vars = ["x","y"]                           -- for polynomial over R
  ord  = (("drlex",2), degRevLex, [])        -- power product ordering term
  p1    = cToPol ord vars dR r1  :: P        -- coefficient-> polynomial
  [x,y] = varPs r1 p1                                   -- x,y as polynomials
  [f,g] = map (smParse p1) ["x*(x^5+1)", "y*(x+1)^2"]   -- parse by sample p1
  gcdFG = gcD [f,g]
 in
 writeFile "result" $
             -- shows e s  prints any value  e  prepending its string to  s
   concat
   ["kernel generators over Z =\n",                shows (Mt ker dZ) "\n\n",
    "inverse(qM') =\n",                            shows iM'         "\n\n",
    "Checking qM'*(inverse qM') == unityMatrix = ",  shows checkInvM' "\n",
    "properties of R =\n",                         shows rProps "\n\n",
    "2/3 in R = ",                                 shows q     "\n",
    "gcd(f,g) = ",                                 shows gcdFG "\n"
   ]
```

This program (let it reside in the file `Main.hs`) is run under the `ghc-7.0.1` interpreter as follows:

```
ghci -package docon Main
...
Main> main
```

Now, we repeat the `'let'` part giving more comments:

```
main =
  let
    un  = 1     :: Z
    unQ = 1:/1  :: Q
    zrQ = zeroS unQ               -- zero of domain of unQ,
                                  -- unQ is a sample argument
    dQ  = upField unQ Map.empty   -- domain term of Q;  unQ a sample argument
    mM  = [[1,2,3,4,5,6], [5,0,6,7,1,0], [8,9,0,1,2,3]] :: [[Z]]
    v   = [0, 0, 0] :: [Z]
    (_, ker) = solveLinear_euc mM v
    rows ' = mM ++ [[0,2,0,4,0,2], [5,0,6,1,2,0], [0,2,0,1,2,1]]
    qM'    = mapmap (:/un) rows'
    iM'    = Mt (inverseMatr_euc qM') dQ  -- \m-> Mt m <dom>  makes true matrix
                                           -- from the list of rows
    Mt mPrd _  = (Mt qM' dQ)*iM'
    unM        = scalarMt qM' unQ zrQ
    checkInvM' = mPrd == unM
    ----------------------------------------------------------------
    b  = 5 :: Z                                 -- arithmetics in R = Z/(b)
    iI = eucIdeal "bef" b [] [] []   -- Ideal (b) in R.  eucIdeal "bef" completes
                                     -- ideal attributes according to mode "bef"
    r1 = Rse un iI dZ                  -- unity residue by ideal iI
    dR = upField r1 Map.empty          -- the full domain description
                                       -- for the ring of r1 (R = Z/(b))
    (_, rR)    = baseRing r1 dR        -- base ring to which r1 belongs
    rProps     = subringProps rR
    [r2,r3,r4] = map (ctr r1) ([2..4] :: [Z])  -- each n projects to domain of r1,
                                               -- r1 is a sample argument
    q = r2/r3
    ----------------------------------------------------------------
    vars = ["x", "y"]                             -- for polynomial over R
    ord  = (("drlex",2), degRevLex, [])
                                -- Power product ordering description.
                                -- We could put instead of degRevLex any
                                -- admissible function for pp comparison.
    p1 = cToPol ord vars dR r1  :: P
                           -- Mapping from coefficient is the best
                           -- way to initiate some polynomial.
                           -- Here  r1  maps to the unity of  P = R[x,y].
    [x, y] = varPs r1 p1                               -- x,y as polynomials
    [f, g] = map (smParse p1) ["x*(x^5+1)", "y*(x+1)^2"]
                         -- Parses domain elements by the sample p1.
                         -- Alternative: [f,g] = [x*(x^5+p1), y*(x+p1)^2]
    gcdFG = gcD [f, g]   -- this must be (x+1)^2 because x^5+1 = (x+1)^5  over R
  in  ...
```

3 Principles of DoCon

3.1 Partial maps. m format

Consider an example: in DoCon,

`n/m :: Z (= divide n m)`

either evaluates to `q :: Z` or to the program break with the report like
`error: "divide 6 4"`.

But `DoCon` provides also `divide_m,` where the 'm' denotation stands for 'maybe'. For example,

```
divide_m 4 2 -> Just 2
         4 3 -> Nothing
         4 0 -> Nothing
```

This gives a partial map with explicitly processed fail. Example of usage:

```
f x y = case  divide_m x y  of  Just q -> ... something   with q ...
                                _      -> ... other thing with x,y
```

The type

`data Maybe a = Nothing | Just a ...`

its instances, auxiliary functions, like `allMaybes`, `catMaybes` ..., are taken from the `Haskell Prelude, DPrelude` of `DoCon` and the `Haskell` library `Maybe`.

Further, the operation `root n x`
may yield the values

`Just (Just r) | Just Nothing | Nothing`

of type

`MMaybe a = Maybe (Maybe a)`

(see Section 8). This means respectively

- "r is the root of n -th degree of x",
- "such root does not exist in the given domain",
- "could not determine whether such root exists in given domain".

Similarly are treated the `zero`, `unity` operations in semigroup, listing of a subset, and some others. On the other hand, the *property values* attribute may help in choosing between 'm' format and simple format. Thus, `(IsGroup, Yes)` in the property list of an additive semigroup `H` insures that the `neg`, `sub` operations are total on `H`, hence, there is no need to apply `neg_m`, `sub_m`.

Additional reason for wide use of 'm' format is the feature of parametric domains, where the true definition domain of operations may depend on a dynamic parameter (Section 3.4).

3.2 Example of DoCon application. Computing in cubic radical extension.

The following example serves as an introduction to the DoCon design principles. The function

```
\a b k -> cubicExt a b k
```

builds automatically the radical extension tower

```
k -- k(d) -- E = k(d)(u,v,r)
```

for the given field `k`, and coefficients `a`, `b` of irreducible polynomial

$$\mathtt{f} = t^3 + \mathtt{a} \cdot t + \mathtt{b}$$

over `k`, $\mathtt{a} \neq 0$,
`d` stands for the square root of `discriminant(f)`,
`r` square root of -3, `u`, `v` Cardano cubic radicals.

So, `E` contains the field `K' = k(x,y,z)` generated by the roots of `f`.

`cubicExt` applies the operation `root 2 x` to `x` from `k` returning `Just (Just y)` with `y` from `k` such that $\mathtt{y}^2 = \mathtt{x}$, if there is such `y` in `k`.

From the user point of view, one has only to apply

```
cubicExt a b k
```

in the user program to prepare the extension. But to explain the DoCon architecture, we show here how to implement this function in Haskell using DoCon. This implementation needs combining in a certain way the constructors of Polynomial and Residue; the latter presumes, in its turn, the Gröbner basis computation. The scheme is

- `D = discriminant(f) = - 4*a^3 - 27*b^2;`
  `dd = minimalPolynomial(d)`
  here `dd = d - squareRoot(D)`, if `k` contains `squareRoot(D)`, `d^2-D` otherwise;
- build the field `k1 = k(d) = k[d]/(dd)` and the ring `B = k1[u,v,r];`
- set `r`, `u`, `v`, `uv` to be the defining polynomials for the corresponding field elements (first portion of Cardano formulas):
  ```
  r  : r^2  + 3,
  u  : u^3  - (3/2)*d*r + (27/2)*b,
  v  : v^3  + (3/2)*d*r + (27/2)*b,
  uv : u*v  + 3*a ;
  ```
- find `gs = GröbnerBasis [u,v,uv,r]` in `B`
  needed to represent `B/Ideal(u,v,uv,r);`

- build `E = B/I` for `I = Ideal(gs)` in `B`
  — `E` represents an extension of `k1` containing the roots;
- define the roots of `f` in `E` by Cardano formulas:
  `x = (1/3)*(u + v),`
  `y = (1/6)*(r*v - r*u - u - v),`
  `z = (1/6)*(r*u - r*v - u - v)`

Then the programmer may set, for example, the `main` function that applies `cubicExt a b k` for some particular field `k`, and then, performs various computations in terms of `x, y, z` in `E`, such as testing the Viete relations, and so on. The more advanced experiment here might be expressing of `y` as a polynomial of `x` over `k(d)`.

Remarks for a newcomer to `Haskell`:

the below program contains the commentaries starting with '–'. The script has the form

`let` ... `in` *Expression*

familiar to the Lisp programmers. We also use intensively the pattern matching. To understand better the script, recall also that

- identifiers of type parameters, data (functions in particular) start with small letters;
- data and the type constructor names, as `Integer, Z, Pol`,
  start with capital letters;
- `Haskell` encourages omitting the parentheses, say `f x y === ((f x) y)`,
- the indentation rule also helps this and in narrowing the scope of the variables.

```
import Prelude hiding (minimum, maximum)
import DPrelude
...
import GBasis (algRelsPols)

type Q    = Fraction Z        -- rational numbers
type A  k = UPol k            -- for  A  = k[d]
type K1 k = ResidueE (A k)    --      K1 = k[d]/(d_equation)
type B  k = Pol (K1 k)        --      B  = K1[u,v,r]
type E  k = ResidueI (B k)    --      E  = B/I = k1(u,v,r)

cubicExt :: Field k => k -> k -> Domains1 k -> (Domains1 (E k), [E k], k -> E k)
                    -- a    b    dK            dE             [x,y,z] kToE
```

`cubicExt` returns the

* domain description `dE` for the field `E`,
* elements `[x,y,z]` of `E` representing the aforementioned roots,
* embedding function `k -> E`.

Restriction: `char(k)` $\neq 2, 3$.

Here `dK :: Domains1 k` is the domain description (term) for a field `k`. It may include the set term, additive group term, and so on. The simplest way to obtain such domain supplued with all neccessary desciption terms, is to apply an appropriate 'up' function.
For example,

* `dZ :: Domains Z` is 'up' for the domain `Z = Integer`,
  it saturates `dZ` with all things known for `Z`;
* `dK = upField (1:/1) Map.empty`
  saturates `dK` with set, semigroup, ..., ring terms, and so on, using `1:/1` as the domain sample element, `Map.empty` means here the empty initial domain. `1:/1` is a sample argument here, and, for example, `2:/3`, produces an algebraically equivalent result.

Thus,

```
let {un = 1:/1 :: Q;  dK = upField un Map.empty} in  cubicExt un (-un) dK
```

builds the extension `(dE, [x,y,z], kToE)`
expressing $Q — Q(rootsOf(t^3 - t + 1)) = E$.

`Field k =>` means that the parameter domain `k`, considered as the type is an *instance* of the `Field` category (`Field` data class). For example, this implies, that the expression
`x + x/y` is valid for `x`, `y` from `k`.

Now, skip the body of this function and concentrate on how one can use it for constructing the extension of the field `Rational = Q` with the roots of

$$f = t^3 - t + 1$$

and for performing some computations with the corresponding `x`, `y`, `z` values in `E`.

```
module Main where
import ...
...                       -- Example. Build the extension of Q = Fraction Z
main =                    --                          with  f = t^3 - t + 1
  (discr, [roots, fRoots, vieteValues], propsE, rels)
  where
  un                 = 1:/1 :: Q
  (a, b)             = (-un, un)
  dK                 = upField un Map.empty
  (dE, roots, kToE) = cubicExt a b dK
  [x, y, z]          = roots
  Just (D1Ring rE)  = Map.lookup Ring dE  -- look into ring E
  propsE             = subringProps rE     -- - for curiosity

                                          -- example of calculation in E
  discr  = - (4:/1)*a^3 - (27:/1)*b^2
  n1     = fromi x 1                         -- integer -> ring of x
  fRoots = [x^3 - x + n1 | x <- roots]       -- must be [0,0,0] in E
  vieteValues = [x+y+z, x*y+x*z+y*z, x*y*z]  -- test Viete relations

  -- Now, find  y  as a quadratic polynomial in  x  over  k1.
  -- x and y  are the polynomial residues of  x',y' <- B = k1[u,v,r]
  -- modulo I.  We have to find the algebraic relations between  x',y' in
  --                                                          B modulo I.
  [x', y'] = map resRepr [x,y]
  Just hs = idealGens (resIdeal x)
  o       = lexPPO 2                              -- set pp ordering
  rels    = algRelsPols hs ["y", "x"] o [y', x']
          --
          -- generators of algebraic relations for  y', x' viewed  modulo Ideal(hs)
          -- in  k1[u, v, r],  the relations to display in the variables "y", "x"
```

Now, we repeat the 'main =' part supplying it with more comments. Some of the program lines are marked with `'n:'`. This means "see the commentary number n after the program".

```
  main = (discr, [root, [fRoots, vieteValues], propsE, rels)
    where
1:  un                 = 1:/1 :: Q
    (a, b)             = (-un, un)
    dK                 = upField un Map.empty
    (dE, roots, kToE) = cubicExt a b dK
    [x,y,z]            = roots      -- the list is pattern matched to x,y,z values
2:  Just (D1Ring rE) = Map.lookup Ring dE   -- look into ring E
    propsE           = subringProps rE      -- - for curiosity

                                              -- example of calculation in E
    discr  = - (4:/1)*a^3 - (27:/1)*b^2
    n1     = fromi x 1              -- map homomorphically integer to base ring of  x
    fRoots = [x^3 - x + n1 | x <- roots]           -- this must be [0,0,0], 0 of E
    vieteValues = [x+y+z, x*y+x*z+y*z, x*y*z]      -- testing Viete relations

    -- For these a,b, the Galois theory says E' = k1(x,y,z) = k1(x)  and
    -- E':k1 = 3.  In particular,  y  has to express as a quadratic polynomial in
    -- x  over  k1.  Let us test this.  x, y  are the polynomial residues of
    -- x', y' <- B = k1[u,v,r]  modulo the ideal I.  So we have to find the
    -- algebraic relations between x', y' in B modulo I.

3:  [x', y'] = map resRepr [x, y]
4:  Just hs  = idealGens (resIdeal x)
    o        = lexPPO 2           -- set ordering for polynomial power product
    rels    = algRelsPols hs ["y", "x"] o [y', x']
                                -- generators of algebraic relations for y',x'
                                -- viewed modulo Ideal(hs) in  k1[u,v,r],
                                -- relations to display in variables "y","x"
```

To print the results, one also could set here the calls for `shows(n)`, after the fashion of earlier examples. This program, together with `cubicExt`, is also presented in the file `demotest/T_cubeext.hs.`

The result print-out is

```
discr       = -23
roots       = [(1/3)*u + (1/3)*v,
                 (-1/6)*u*r + (-1/6)*u + (1/6)*v*r  + (-1/6)*v,
                 (1/6)*u*r  + (-1/6)*u + (-1/6)*v*r + (-1/6)*v)
                ]
fRoots      = [0, 0, 0]
fRoots      = [0, 0, 0]
vieteValues = [(0, (-1), (-1)]
propsE      = [(IsField, Yes), (Factorial, Yes) ...]

relsl  (algebraic relation generators for y,x) =
  [x^3 -x + 1,
   y + ((3/23)*d)*x^2 + ((9/46)*d + 1/2)*x + ((-2/23)*d)
  ]
```

We are mostly interested in `rels`, and observe that it consists of the source equation on `x` and a non-trivial quadratic expression of `y` in `x` over `k(d)`.

Now, the No -red commentaries to the function 'main':

1:

`:/` is the data constructor for the `Fraction` functor in `DoCon`. The `DoCon` declaration associates it with the type constructor

```
data Fraction a = a :/ a ...
```

As the first approach, we assume that `DoCon` represents
an algebraic category as an `Haskell` data class,
operation from a category as a class method,
(static) algebraic domain as a class instance.

`DoCon` contains the declarations that supply a type `(Fraction a)` with the operations of Set, Additive group, Multiplicative semigroup ... Field categories — provided the parameter type `a` is the instance of the `GCDRing` class described in DoCon. And the usual arithmetic methods for `(Fraction a)` are correct if the operations on `a` satisfy the GCD-ring property. In our case, `a = Z` provides such a ring, and this is why in the further script the expressions like `(-4:/1)*a^3` make sense.

Similarly, the `DoCon` functors `UPol`, `Pol`, `ResidueE`, `ResidueI` are composed and make the ring tower up to `E`, with the instances of the `Field` class. This is why, for example, the expression `\x-> x^3-x+1 :: E` is valid in the further program.

2:

```
Just (D1Ring rE) = Map.lookup Ring dE
propsE           = subringProps rE
```

Here `Map.lookup` extracts the ring term given the key `Ring :: CategoryName`. Say `Map.lookup AddGroup dE` extracts the additive group, and so on. The value `rE` is an explicit description of the ring:

```
Subring {subringChar  :: Maybe Natural,
         subringGens  :: Maybe [a],
         subringProps :: Properties_Subring,
         ...
        }
```

There are several specially designed functions to help constructing such terms from the simplest data: integer domain `dZ`, set from list, ideal from generators, and so on.

3: `[x',y'] = map resRepr [x,y]`

4: `Just hs = idealGens (resIdeal x)`

`x`, `y` are built as the elements of a residue ring `E = B/iI`, `B = k1[u,v,r]`.

`resRepr` extracts a representative from the residue element. For these representatives `x'`, `y'`, the program finds the relations modulo ideal. Another way to extract `x'`, is to pattern match `Rsi x' ...= x`, with the constructor `Rsi` of the residue ring data. `resRepr` is better, because it does not rely on the particular type constructor.

`resRepr, resIdeal` are the operations from the *constructor class* `Residue`, that joins `ResidueE` — for the Euclidean case, and `ResidueI` — for the generic case. `ResidueI` relates to the implementation of a residue ring by an ideal. The ideal term `iI` is inside each residue element `r`, and can be extracted by applying `resIdeal r`.

Further, `idealGens iI` extracts the ideal generator list from `iI`, — if it was set into the `iI` data earlier. For the ideal, `DoCon` needs the generator list with the property value `(IsGxBasis, Yes)` to provide the residue ring with the correct `Ring` instance. So, starting from the residue elements `x`, `y`, we get the ideal, ideal generators, and so on.

`x'` has to be a canonical representative of a residue class. For the polynomial ring over an Euclidean ring, `DoCon` chooses as such representative the normal form modulo the Gröbner basis contained in `iI`.

3.3 Category hierarchy

DoCon deals with the following category graph:

```
                     (D)Show    Eq
                           \   /
     OrderedSet    <- - -   Set
                           /   \
                         /       \
Ordered.. <- - - AddSemigroup    MulSemigroup
...                 /                  \
                  /                  /   \
          AddMonoid                /     MulMonoid
               /                 /          \
             /                 /              \
        AddGroup _ _ _ _  _  /                MulGroup
         /     \           / \ _ _ _
       /         \       /           \
 OrderedAddGroup   \   /             LeftModule
                     |                        \
                    Ring                     LinSolvLModule
                     |
               CommutativeRing --> OrderedRing
                    /  \
                  /      \
             GCDRing      LinSolvRing
               /            \
             /                \
   FactorizationRing         EuclideanRing
               \            /
                 \        /
                   \    /
                   Field     - - ->  RealField
```

The programmer may develop this graph further. There also exist the `Haskell` library classes

`Show, Eq, Ord, Enum, Num, Fractional`

`DoCon` includes them into this tower in order to use their operation names. But other `DoCon` operation definitions has to agree with these ones in a certain way — see the comments to each category.

`Eq`	is a superclass	of `Set`,
`Ord`		of `OrderedSet`,
`Num`, `Fractional`		of `Ring`.

Most of DoCon library is devoted to `GCDRing`, `LinSolvRing`. Some of the category declarations are *empty*, only proclaim in a commentary some additional presumed conditions. See, for example, `Field`.

3.4 What is `DoCon`

The following is a summary of the DoCon architecture.

<u>(MB) Mathematical base</u>

The methods, algorithms (of arithmetic nature or other) are defined for some constructed domain

$$C \texttt{ a b} \ldots$$

via some generic methods for the domains `a`, `b` ... — under appropriate declared conditions. For example,

```
instance GCDRing a => GCDRing (Pol a) where
                                        gcD = ... <gcd for polynomials>
```

expresses the polynomial GCD via arithmetics and GCD on `a`, and relies on the property of `a` being a gcd ring. Developing such methods constitutes the science of computing, they may be taken from books, scientific papers (see the literature **References**), sometimes are invented by the author.

<u>(CT) Category</u>

is defined by a class declaration and presumed, intended condition; the latter is provided as a commentary. **Example:**

```
class Set a => AddSemigroup a where add :: a -> a -> a
  --                                ...
  -- presumed: ‘add’ agrees with (==), is associative, commutative
```

<u>(D) Domain (many-domain)</u>

is defined by several algebraic instance declarations for a given algebraic constructor C, together with the sample element

$$\texttt{s = } C \texttt{ a b ...}$$

This `s` defines finally the true current algebraic domain `D(s)`, and hence, to which categories `D(s)` belongs in fact. (Many-)domain has also an explicit description — a *bundle* — see **(DT)**, **(MD)**.

<u>(DT) Domain term</u>

is an explicit expression describing partly a domain.
Example: the base bundle for `Z = Integer` contains

- `sZ :: OSet Z` which describes the set Z, it contains
  `membership = const True`, `cardinality = Infinity`, other attributes,

• `aZ :: Subgroup Z` contains the generator list `[1]` for an additive group ...

and so on.

<u>**(MD) Many-domain (bundle)**</u>
is a finite collection of domain terms stored each under its *key*,

```
Key = CategoryName = Set | AddGroup | Ring | ...
```

Example: `dZ` is the bundle for `Z`, it contains terms for the keys from `Set` up to `FactorizationRing`, `EuclideanRing`.

<u>**(E) Element (of domain)**</u>
is a data of kind C `x y ...`
obtained by a type constructor C applied to arguments. Some arguments may be domains (bundles)
(the domains `Char, Integer` are presented as constructors applied to empty argument sequence).
Example:

```
Rse x iI aD
```

is an element of residue ring `a/I` `<-->` `ResidueE a` for Euclidean `a`,
`aD` a bundle for domain `a`, `iI` ideal description.

<u>**(DS) Domain by sample element**</u>
Any element

`s =` C `a b ...`

can serve as a sample defining its many-domain. For each known to `DoCon` constructor C, `DoCon` puts individually how the parameters `a`, `b` ... of C define a domain. Partly, this is set by assumption, partly — by explicit definition of several base-operations: `baseSet`, `baseAddSemigroup` ... — see **(BO)**.
Example: for the ideal descripions `iI = idealOf(7 in Z)`, `iJ = idealOf(8 in Z)`, the data

```
r1 = Rse 0 iI dZ,  r2 = Rse 0 iJ dZ
```

are of the same type `ResidueE Z` but of different domains `Z/(7), Z/(8)`.
Further,

```
bs1 = snd $ baseSet r1 Map.empty,  bs2 = snd $ baseSet r2 Map.empty
```

build the set terms for `Z/I`, `Z/J`. One could see, for example (at the run-time),

```
osetCard bs1 = 7,   osetCard bs2 = 8.
```

If further domain terms have been formed by `upAddGroup` ... `upRing` ...,
then other domain attributes can be tested at the run time. For example, `isField(Z/I)` can be checked by looking into the *property list* of the ring term.

(SA) Sample argument

Some class operations, functions in `DoCon` need a sample argument to specify the domain for the operation (function). For example,

```
zeroS $ Vec [2],  zeroS $ Vec [2,0]
```

are the zeroes of different groups — defined by samples of different domains. If a function `f` is said to have a sample argument `s`, taking value in some `Set`, then `f` must not depend on the choice of `s` inside the same domain (see **(DS)**). **Examples:**

```
fromi (1,1) 4   == fromi (2,1) 4    == (4,4)
zeroS $ Vec [2] == zeroS $ Vec [-1]  /= zeroS $ Vec [2,2],
unity (2:/3)    == unity ((-3):/1).
```

Sample arguments are sometimes commented with "– SA" or "– sa". Often they are taken from surrounding program context, otherwise, can be easily built recursively, applying the constructor to the simpler samples.

(CS) Cast by sample

Construct the initial sample `s :: T` for a domain in the type `T`
and apply the casting functions

```
ct s,  ctr s,  fromi s,  smParse s
```

to construct other elements of domain from simpler data.
Example 1: in **(DS)** example, it is better to write

`ct r1 4` than `Rse 4 iI dZ`,   `ctr r1 11` than `Rse 11 iI dZ`.

`ct s`, `ctr s` cast to domain of `s`. These functions are polymorphic. For example, they cast to polynomial from coefficient, from monomial, from monomial list.
Example 2: `fromi s: Z -> domain of s` maps homomorphically from `Integer`.
Thus, in **(DS)**, `fromi (r1,r2) : Z -> (Z/(7), Z/(8))`

(DE) Extractors

It is a good style to use specially designed operations, functions to extract the parts of domain element. For this purpose there are also provided the classes

`Dom (sample, dom),   Residue (resRepr, resPIdeal ...),    PolLike ...`

Example 1: in example of **(DS)**, `resRepr r1`, `dom r1`, `resPIdeal r1` extract the representative from the element of residue ring `Z/I`, argument domain `dZ`, ideal term for `I`. This is more invariant than matching against `Rse x iI d`.

Example 2: for a polynomial `f` from $R[x]$, $R[x_1, \ldots, x_n]$, which can be represented in various ways,

`sample f,  dom f,  pVars f,  lc f`

yield respectively the coefficient sample, term of R, variable list, leading coefficient.

(BO) Base-operations

`baseSet,  baseAddSemigroup,  ... baseRing,  baseEucRing`

and some other, build the corresponding domain terms from a sample `s` and a given current *bundle* `dD`.

- Example 1: `baseSet s dD --> (dD',bs) :`
  if the Set term `bs` is in `dD`, then `dD` remains and `bs` extracts,
  otherwise, `bs` is constructed via `s, dD` and put into `dD` too.
  Other base-operations act similarly.
- Example 2: for the example from **(DS)**, `bS = snd $ baseSet r1 Map.empty`
  builds the subset term for `Z/I`. As `r1 = Rse _ iI dZ`,
  `bS` is defined via the bundle for `Z` and the ideal `iI`.

(UP) Up-functions

A *base bundle* can be built from *initial* one (maybe, from empty) by the `DoCon` functions

`upAddSemigroup, upGroup, upRing`, and so on.

Up-function is a composition of several *base-operations.*
Example: in `(DS)` example, the program

```
let bd = upRing r1 Map.empty in upEucRing r1 bd'
```

adds the terms, up to the description of Euclidean ring, to the bundle obtained from `upRing`.

(BSE) Base set. Equality.

`(==)` has to be an equivalence relation on a *base set* BS.

BS is understood as the set of canonical representatives by `(==)`. This is always a `DoCon` assumption for a base domain of given sample. In easy cases the subset BS in a type is defined completely by the the membership function contained in the subset term made via `baseSet s _`. And often, the subset is presumed.

All the further base domains for `s` are considered as having the *support* BS.

Example:

`r1, r2` from example of `(DS)` define the finite subsets BS1, BS2 in `ResidueE Z`, with empty intersection of BS1, BS2.

`Rse 6 iI dZ` belongs to BS1. `r3 Rse 7 iI dZ` does not — the representative being not reduced canonically by `I`. `r3/r3` will pass through compilation, but may evaluate wrongly, because `DoCon` presumes `r3` to belong to BS1.

(IN) Meaning of a category instance declaration

Here we formulate how the programmer should view the class instances when applying the advanced part of DoCon. This concerns a parameterized domain `D(p)` inside a type `(C a)`. For example, `D(m) = Integer/(m)` inside `ResidueE Integer`.

The following semantics is proposed for the category (algebraic class) instance with respect to the parametric domain `D(p)` given by a sample `s :: C a`. This instance meaning differs in its last point from traditional Haskell.

- (1) For a non-parametric domain the meaning remains. Examples:
  `Integer`, `(Fraction Integer, UPol (UPol Integer))`.
- (2) For a parametric domain `D(p)` related to type `(C a)` the meaning of *absence* of instance declaration `(...) =>` *Category* `(C a)` remains: no domains `D(p)` are considered as belonging to *Category*. In the cases (1),(2), categories for `D` are detected at compilation time.
- (3) For a parametric domain `D(p)` related to a type `(C a)` the presence of declaration

  `instance` *Context* `a =>` *Category* `(C a)`

  means that

  1. (as usual) for `D(p)` to be of *Category*, the instances (*Context* `a`) must match the type `a` (this is detected at compilation time) AND
  2. `s` must contain a parameter value `p` satisfying the so-called 'detect' condition. This condition check is optional and often performs at the run-time.
     The programmer may apply explicitly appropriate detecting function to `s`.

The mentioned detecting function can always be presented as a composition of base-functions and access to the domain attributes produced by base-functions.
Example:
`DoCon` contains a declaration

```
instance EuclideanRing a => Field (ResidueE a)   -- (resPIdeal r) must be prime
```

According to the general rule, this means: a residue data `r = Rse x iI aD  :: ResidueE a` belongs to a ring `R = a/I` of residues of a ring `a`. And `R` is a field when the ideal `iI` is prime. The latter condition can be checked by applying

`isPrimeIdeal iI` or `isPrimeIdeal $ resPIdeal r`

or `isField rR`, with a base ring term `rR` for `r` formed earlier by `(baseRing r ...)`. And again, `baseRing` would analyze `iI`, `aD` when forming `rR`.

Note the role of compiler here: it checks the static *class instance* `EuclideanRing a` for the type `a` but does not check further the primality of `iI`, nor the condition `isField (a/I)`.

(CC) Correctness condition for method

A polymorphic method defined under the instance context, has the correctness condition that depends not only on the context declaration but also on the sample (true domain) attributes. This is a consequence of **(IN)**, **(DS)**.
Example: let the user put

```
det_Gauss :: Field a => [[a]] -> a
det_Gauss mM = ... <determinant via the Gauss method>
```

and apply it to `a = ResidueE Integer`. This will compile all right, and at the run-time, the result correctness depends on which residues are put in the matrix. With `(Rse _ ideal(3) _)`, `det_Gauss` it is correct, with `(Rse _ ideal(4) _)`, may be wrong.

`DoCon` usually checks such conditions before starting to really evaluate expensive enough methods: determinant, gcd, and such. Similarly, the user program decides each time, what to check.

(BF) Bundle field in constructor

Bundle field in a domain constructor provides a reliable control on the cost of base-fuctions and domain attribute access.
Example: a residue element for an Euclidean ring is

`r = Rse x iI aD` $\in$ `a/iI = R = base ring of  r`

If in some loop the attributes of `R` are processed many times, they form dependently on `iI`, `aD`, `aD` the bundle for `a`. But `iI`, `aD` are ready, their values shared by the elements of `R`.

If, say for this example, there appears further the ring `R[x]`, then the constructor `UPol` requires a bundle `dR` for `R`, `dR` is formed after `iI`, `aD`, and then, the ready value for `dR` is shared among elements of `R[x]`. And so on, up the constructors applied.

This looks like a situation when each student of university keeps this university in one's pocket. This is all right for functional programming, due to most natural features of any reasonable implementattion, such as data sharing, references and 'lazy' copying.

3.5 Further explanations on principles

3.5.1 Presumed condition for category

Still, the "presumed condition" — see (Section 3.4 (CT)) — is sometimes provided automatically, due to the constructor properties. Thus, the `DoCon` declaration

```
instance (AddSemigroup a, AddSemigroup b) => AddSemigroup (a,b)
  where
  add (x,y) (x1,y1) = (add x x1, add y y1)
     ...
```

automatically guarantees that the semigroup law holds for the domain `(a, b)` if it holds for `a` and `b`.

3.5.2 Sample element

This approach is taken partly because in `Haskell` one cannot relate a constant to a domain in a simplest way. For example, declaring

```
class WithCard a where cardinality :: Natural
```

would not do any good. Because seeing, say `2*cardinality` in a program, the compiler cannot solve, who's cardinality is taken. But it appears, the sample element also helps to identify the presumed dynamic domain and to map between domains. For each constructor C, `DoCon` documentation declares individually how the subset is defined in the type
`(`C` a b ...)` (Section 3.4 DS).

Example: `DoCon` specifies the following assumptions:

- `s :: Z` defines in the base set Integer — it coincides with the type.
- `s = k:/l :: F a = GCDRing a => Fraction a`
  defines the subset of `(F a)` of all the expressions `n:/d`, `d /= zero, n, d` cancelled canonically via gcd (Section 34).
- `s = Vec xs :: Vector a`
  defines the subset `V = {Vec ys :: Vector a | length ys == length xs}`
  of vectors over `a` with the given number of components.

3.5.3 Base set. Equality

`Haskell` wisely treats the relation `(==)` not as syntactic but as "algebraic" one. So that the programmer can define equality according to one's undesrtanding of a given algebraic domain. For example, the polynomial data

```
f = UPol [] 1 "x" dZ,   g = UPol [] 2 "x" dZ
```

in `Z[x]` are syntactically different, but `f==g = True` in DoCon.
Because in `(UPol _ c _ _)` `c` is treated as a sample element.

3.5.4 Cast by sample

For the data `s :: T`,
`fromi s   : Z      -> T` maps homomorphically from Integer,
`smParse s : String -> T` parses a string after a sample,
`ctr s     : a      -> T` casts from domain `a` after a sample.

Example: `s = Pol monomials coefSample ppOrd variables coefDomain`
is a format for a multivariate polynomial. But the programmer is not supposed to write things like `(Pol ms c o vs dD)`.
Rather one sets `ctr s <coefficient>`, `ctr s <monomial>`,
`ctr s <monomialList>` (Section 3.4 (CS)).

Apply similar technique to other constructors: `RPol`, `ResidueE`, `ResidueI`, ...

See in this manual the programs with these operations, their instances declared here for the constructors, examples in the files ...`demotest/T_*.hs`

3.5.5 Static and dynamic domain

As we see from Section 3.4,
the static notion of a domain is given by the class `instances`,
a dynamic domain is a bundle of domain terms.

A bundle is used mostly for operating with the domain attributes: cardinality, properties, and such. The `instances` are needed to use the category operations: `+, *`, `gcD` ...

Base-operations (Section 3.4 BO) present an important relation between a static domain and a bundle.

3.5.6 Base-operations

The needed domain term can be obtained from a sample element `s` and a current bundle `dm` either by

`Map.lookup <key> dm` (`s` not needed) or by `base<Category> s dm`

Consider, for example, `(Map.lookup Set dm)` and `(baseSet s dm)`.
The latter is more generic: if it does not find the needed term in `dm`, it starts forming it from `s, dm`.
Example 1:
for the domain `Char`, `DoCon` declares something like

```
instance Set Char where
  ...
  baseSet _ dm = case  Map.lookup Set dm
                 of
                 Just (D1Set o) -> (dm, o)
                 _              -> (Map.insert Set (D1Set o) dm, o)
                     where
                     o = OSet {osetSample  = 'a', membership = (\_ _->True), ...}
```

For `Char`, the function `baseSet` ignores the sample argument. Similar is `Integer`.
Example 2:
The `Fraction` constructor uses the sample as follows.

```
instance GCDRing a => Set (Fraction a)  where
  ...
  baseSet (n:/_) dm = case  (Map.lookup Set dm, upGCDRing n Map.empty)  of

                (Just (D1Set sbs), _    ) -> (dm, sbs)
                (_,                 aDom) -> (Map.insert Set (D1Set sbs) dm, sbs)
                                    where
                                    sbs = ... build a base set for  Fraction A
                                    from the bundle  aDom  of A
```

Here the bundle `aDom` is built for the numerator of the sample, — for the domain `a`, then `aDom` is used to build the destination term.

3.5.7 Bundle, domain

Sometimes we mean by 'domain' a bundle: several algebraic structures related to the *set*. In other case, 'domain' means one of these structures.

A type, instances related to it, and a bundle define completely what a domain is. If, for example, the call `Map.lookup Ring dD` returns `Just s`, this means that `DoCon` is ready to deal with `dD` as with a true ring, it may look into the property list, and so on. Otherwise, `dD` is not known as a true ring. For example,

```
dZ :: Domains1 Z
```

is the fullest known bundle for `Integer`, and

```
(Map.lookup AddGroup dZ, lookupFM MulGroup dZ, lookupFM EuclideanRing dZ)
-->
(Just (D1Group sG), Nothing, Just (D1EucR rE))
```

extracts the descriptions of `Z` as of additive group, and as an Euclidean ring. The same request for the multiplicative group returns `Nothing` — because `dZ` does not put such term there, for obvious reason.

3.5.8 Up-functions

(Section 3.4, UP).
The *domain descriptions do not include into each other.*
For example, a `Subring rR` is also an additive `Subgroup`, and the corresponding term for this subgroup is extracted not from `rR` but from the bundle.
Example.
To create a bundle "up to `Ring`" from the empty bundle `Map.empty`, one needs to create `OSet`, put it to bundle under the key `Set`, create `Semigroup` and put it to bundle under the name `AddSemigroup`, create another `Semigroup` and put it to bundle under the name `MulSemigroup`, create `Subring` and put it to bundle under the name `Ring`.
The call `upRing s dom` performs all this. Naturally, it applies `upAddGroup` and some other 'ups'. According to the category hierarchy, one up-functions apply some others. And they are the compositions of base-functions. footnotesize

```
type ADomDom a = a -> Domains1 a -> Domains1 a

                      -- up<category> :: <category> a => ADomDom a

upAddSemigroup :: AddSemigroup a => ADomDom a
upAddGroup     :: AddGroup     a => ADomDom a
...
upFactrLinSolvRing :: (FactorizationRing a, LinSolvRing a) => ADomDom a

upAddSemigroup a = fst . baseAddSemigroup a . fst . baseSet a
upAddGroup     a = fst . baseAddGroup     a . upAddSemigroup a
...
upRing a = fst . baseRing a . fst . baseMulSemigroup a . upAddGroup a
...
upField = upEucFactrRing   -- here the matter is not in bundle
                           -- but in presumed conditions
```

It is also important that creating some domain term to add to a bundle `dD` the program may use the ready terms in `dD`. For example, the property value `v` in `(IsPrimeGroup, v)` for the group term may somtimes be derived as

```
let v = Fin n = osetCard sS in isPrime n
```

for the subset term `sS` from `dD`.

Which domain terms `t` have to be placed in a bundle for the domain defined by a given sample `s` ?

`t` is placed there when `s` is of the type that matches the corresponding class C *and* the domain D of `s` may, — for some parameter values, — occur of the category C.

Example:

let `t = GCDRingTerm {..}`, `s1 = (1,2)` $\in$ `(Z,Z)`, `s2 = (UPol..)` $\in$ `(Z/(m))[x]`.
Then

- `t` is not placed in the bundle of `s1` (of `(Z,Z)`), because the type of `(Z,Z)` is not an instance of `GCDRing` class.
- `t` goes to the bundle of `s2` and it has to contain `(IsGCDRing, v)` in the property list, where `v = Yes` for a prime `m`, `No` — for composed `m`, and when this property is hard to detect, put `v = Unknown`. And the GCD category memership is defined finally by this property value. See Section 3.5.11.

3.5.9 Domain by sample

(Section 3.4 DS).

Most commonly, a base domain appears as in the below example. It shows also an approach to residue ring in Euclidean case.

```
instance EuclideanRing a => Set (ResidueE a)  where
  ...
  baseSet (Rse x iI aDom) rDom = case  Map.lookup Set rDom  of

    Just (D1Set o) -> (rDom, o)
    _              -> (Map.insert Set (D1Set o) rDom, o)
                   where
                   o = let  b = pirCIBase iI
                       in ... build OSet for the residue ring a/I, the bundle
                          for ‘a’ given by  aDom,  b the generator of ideal I
```

This is important: each residue element of domain `a/I` contains the domain description `aDom` for domain `a` and the ideal description `iI`.

So, `baseSet` does not need to build the bundles for the argument domain by new. Rather it has to combine the ready terms `aDom` and `iI`. So, in this example, `x::a` is a representative of residue, and it can be used in `o` via `(baseSet x aDom)` to obtain a set term for `a`. But this would share the ready value from `aDom`. Because the base-operations are designed so that they proceed to construct the term according to the sample only if the given bundle does not contain this term.

But what if `aDom` from `(Rse..)` does not contain the set term?

When the programmer deals with the residue domain, one has first to create the descriptions `aDom, iI` as shown above; then, to copy these *variables* to the expressions destined to this particular domain `a/iI`. The best way to do this is as follows:

```
let  u    = cToUPol (1:/1) "x" dQ           -- sample for Q[x],  Q = Fraction Z
     aDom = upEuclideanRing u Map.empty            -- represents Q[x]
     f    = smParse u "x^3 + ... "          -- ideal generator
     iI   = eucIdeal "be" f [] [] [(f,1)]   -- create ideal
     r    = Rse u iI aDom                   -- initial sample for residue
in   (ctr r f1, ctr r f2, fromi r 9, smParse r "(x^2+1)*x") ...
```

This operates with the residues in `R = Q[x]/I,  I = (f).`

`aDom` is created from `Map.empty` — this is done once. The initial element `r` for `R` is set explicitly. Other expressions for the values in `R` are made after the sample `r`

(Section 3.4 CS). Also this example shows the three ways to map values to the domain

`R = a/I` of `r` from other domains — `Z`, `String`, `a`.

`ctr r g` extracts `iI` from `r`, reduces canonically `g` by `iI` to `g'`, "wraps" `g'` into the residue after the sample `r`.

`ct` does the same, only without reduction of `g`.

And what if the programmer forms the residues in another style, or maybe, forgets to feed `aDom` with domains: skips the above `upEuclideanRing` ... ?

Then, either `(baseSet r _)` reports at the run-time

```
error "...  such and such thing not found in  aD  for  Rse aI aD ..."
```

or it constructs the needed part of `aDom` by new, according to the representation sample `x` contained in `r`. The latter spends some computation cost.

This style, with *sample*, up-functions, copying variables of domain terms, cast by sample,

supported by DoCon, is so natural that we hardly ever can imagine avoiding it.

3.5.10 Domain field in constructor. Cost of bundles

As an element often contains a domain bundle in it, creating the sample needs up-function to build a bundle, some functions imply searching in such bundle, maybe, inside a program loop, — may this ruin the performance?

No. Because

- a sample is created once for a domain, and then, its attributes are *shared* (Section 3.4 (BF)),
- bundle is built "lazily",
- `Map.lookup, Map.insert` cost $O(\log N)$, for a bundle, $N < length(CategoryName - list)$, so, in DoCon, this cost is not more than $\log_2 20$ key comparisons,
- all the important domain constructors C have a data field for an argument domain bundle (Section 3.4 (BF)).

We would only add that the bundle processing is *less desirable in the inner loops.*

Example: consider the program of kind

`let rs =` $[r_1 .. r_{1000}]$ `:: [T] in (sum rs, foldl1 gcD rs)`

For `T = Z`, bundles never appear here, only `(+)`, `gcd` run in the loop.

For `T = UPol (ResidueE Z) <-> (Z/(p))[x]`, — polynomial over residue, — `gcD` of `DoCon` gets each time into the sample, into the sample of coefficient, finds the ideal generator `p`, checks the property value `isField` for the sample for `Z/(p)`. But even here, this does not increase the cost any essentially. Because the `isField` value is shared, and `(+)`, `gcD` for a polynomial over `Z/(p)` cost much more on average than this `isField` test. Besides, `(+)` in `Z/(p)` needs the reduction by `p` in any case, `gcD` needs in any case the inversion for several coefficients in `Z/(p)` — this costs much more that the ideal generator extraction.

So, we have an idea of the relative cost of bundles.

3.5.11 Meaning of an instance declaration

Certain small *property value list* supported by DoCon, may help to define which of the category instances are correct for the sample. The cost of such check is usually small — see Section 3.5.10.

Example 1.

For the Euclidean residue constructor `Rse`, `DoCon` defines the addition (of AddSemigroup) in `a/I` as

```
instance EuclideanRing a => AddSemigroup (ResidueE a) where
  ...
  add r1 r2 = ifCEuc r1 "A" (("add r1 r2,"++) . showsWithDom r1 "r1" "") $
                     ctr r1 $ add (resRepr r1) (resRepr r2)
```

The last line `add ...` forms the true result: sums the residue class representatives and casts to the residue ring by the sample `r1`. This casting is done by taking of the Euclidean remainder `x -> remEuc 'r' x b` by the ideal base `b` contained in `r1`.

The `ifCEuc ...` line tests whether the ring is c-Euclidean: extracts the Euclidean ring term from `r1` and looks there into the properly value list. What happens if we skip the `ifCEuc ...` part?

Then, it might look like

```
careless_add r =  ctr r . add (resRepr r) . resRepr
```

Then, for example, for `a = P = P(2) = (Z/(m))[x],   m = 2,`
`careless_add` causes `remEuc 'r' (f+g) b` — the remainder division of polynomials. The result is correct.

For `m = 4`, `P` still has formally the instance of `EuclideanRing` class. And here `remEuc 'r' (f+g) b` may yield an incorrect result
(why apply Euclidean residue to `P(4)` if `P(4)` is not Euclidean?).

And for more safety, `DoCon` tests this correctness condition in `add` and in some other operations for `a/I`.

Each time `x+y` applies in `a/I`, the property value is searched in certain list extracted from `x`. Can this ruin the performance?

It cannot. Because

- of 'sharing' of a property value in a domain term (Section 3.4 (BF)),
- the property list is small,
- for a polynomial ring `a`, the remainder division costs not less than the search in the property list, and usually, costs much more.
- for `a = Z`, even this fast test is not necessary: `DoCon` declares special category instances for `ResidueE Z`, there the `add` definition skips `ifCEuc`.

Example 2: `DoCon` contains the declaration

```
instance EuclideanRing a => Field (ResidueE a)
```

It means that for the residue ring `R = a/iI` of an Euclidean ring defined by a sample
`r = Rse x iI aD :: ResidueE a`, it depends on the primality of `iI`, whether `R` is a *field*. This primality can be checked fast by looking into the part of `iI` presenting the generator factorization.

Further, if the above declaration included the definition for some specific `Field` operation, for example, `dimensionOverPrimeField`, then the aforementioned attribute in `iI` would be actually a *correctness condition* for an algorithm put for the `dimensionOverPrimeField` definition.

A drawback:
in DoCon, a category declaration has a more complex meaning, it puts less static restrictions than it is in the purely static approach to domain. In particular, it is often on the programmer, not to mix wrongly the data from different domains inside the same type.

Still `DoCon` keeps on with the half-dynamic, sample-dependent domains. Because with the purely static model, there are impossible, for example, nice programs for the Chinese remainder method, and other useful algorithms. The above method needs constructing of many residue rings `a/(p)`, and sometimes one cannot even predict, how many different `a/(p)` would suffice. With DoCon, the user program has only to form dynamically the samples `r(p) = Rse _ iI(p) aD` for the residues.

More example:
dynamic change of the *power product ordering* (Section 38.1) solves certain problems for the multivariate polynomials, and also often helps to reduce the computation cost. And in DoCon, this ordering is a part of a *polynomial sample*. Its change means changing to another isomorphic copy of a given polynomial ring.

3.5.12 Static domain alternative

Suppose we add the constructor classes `SIdeal, MaximalIdeal` for the 'static' ideals `I` in an Euclidean ring `a`,
SIdeal — with, say the ideal base operation,
MaximalIdeal — with, say inversion operation modulo `I`.
Then, for the constructor `i`, the `Ring, Field` instances for

```
R = StaticResidue i a   <-->  a/I
```

can be defined depending on the existence of the static instances `SIdeal, MaximalIdeal` for `i`.

With this approach, quite natural for `Haskell`, the correctness of the `Field a` instance is solved statically. And the meaning of the category declaration becomes straight.

Still DoCon remains only with the dynamic ideals.

Besides the reasons for this explained in previous subsection, recall that an ideal in the above domain `a` is defined usually by the base `b`. After `b` is defined in the program as, say `b = f x`, declaring

```
instance Maximaldeal I1 a
```

for the ideal `I1 = (b)` in `a` presumes that `b` is prime. Such declaration makes sense only when it is known statically that `b` is prime. And often this is not the case.

In other words, the static representation of an ideal fits better the `Haskell` language nature, but does not reflect the dynamic nature of correspondence between the ideals and ideal generators.

3.5.13 Treating properties

`DoCon` operates with certain small set of algebraic domain properties. The property values, such as `IsGroup = Yes | No | Unknown`, are set in the domain terms.
These property lists are introduced because
(a) such property evaluation is itself an important function of system,
(b) domains are often parametric, usually, a property list helps to determine the true computation domain.

Many of the property requirements only accompany the program texts as comments — to prevent the programmer from applying functions to mathematically incorrect data. But often `DoCon` derives these property values automatically, following the construction of the domain, properties of the argument domains, and using certain restricted set of rules. These rules are simple and follow classic algebra. They are like this:

```
IsField      ==> Euclidean,       Euclidean ==> not HasZeroDivisor,
IsMaxIdeal I <==> IsField (R/I),
Factorial R  <==> Factorial (R[x1,...,xn]),   and so on.
```

3.6 Subdomain

How does one compute in (over) a subdomain, for example, subgroup of even integers? Or maybe, factor a polynomial over some subfield of a field K?

Commonly — by introducing a constructor and defining new domain, new instances — see "New sub-domain" below.

Induced subdomain is viewed as a subdomain bundle together with the category operations induced from the base domain. It is implicit.

Examples:

`sPI :: Subsemigroup Z` may describe the *Positive Integer* subsemigroup in `Z`. The restriction of `add` from `Z` is correctly described by `sPI`, but the restriction of `neg_m` does not fit `sPI`. With this, factoring polynomial over a sub-domain is impossible.

New subdomain is simply another base domain, with its new constructor and related instances.

3.7 Printing to `String`. The `DShow` class

In addition to the Haskell library class `Show`, `DoCon` provides the

```
class DShow a
  where
  dShows    :: ShowOptions -> a -> String -> String
  dShow     :: ShowOptions -> a -> String
  showsList :: ShowOptions -> [a] -> String -> String
               -- showList is of Haskell-2010 Prelude,  showsList is of DShow.

  dShow opts a = dShows opts a ""

  showsList opts xs =  <... the default implementation>
```

It aims to somehow improve the `Show` class of `Haskell-2010`. It uses the options for printing the data field separators, for verbosity, for removing parentheses.

For each printable user's data type, the user must define the instance of `DShow`. The simplest way to do this is to define the `Show` instance and then, to apply it:

```
instance DShow T where dShows _ = shows
```

The simplest way to print a `DoCon` data is either
1) to apply `show` or `shows` of `Haskell`, or
2) to apply `showsn` or `shown` or `showsUnparensUpper` or `showsWithPreNLIf`.
— let us call them the *d-show* functions or operations.

The function `showsn :: DShow a => Verbosity -> a -> String -> String`
prints a data `a` to string under the verbosity `verb`, where the verbosity type is defined as
`type Verbosity = Integer`.

Example. For the polynomials

```
f = b1^2*c0 - b1*c1*b0 - b1*c1*c0 + c1^2*b0 + b0^2,
g = a1*b0 - a1*c0 - b1*a0 + b1*c0 + c1*a0 - c1*b0,
h = a1*b1*c0 - a1*c1*c0 - b1*c1*a0 + c1^2*a0 + a0*b0 - a0*c0 - b0*c0  + c0^2,
```

the result of `show [f, g, h] = shown 0 [f, g, h]`

is the string which is visible on the screen as

```
[b1^2*c0 - b1*c1*b0 - b1*c1*c0 + c1^2*b0 + b0^2,a1*b0 - a1*c0 - b1*a0 + b1*c0 + c1*a0
- c1*b0,a1*b1*c0 - a1*c1*c0 - b1*c1*a0 + c1^2*a0 + a0*b0 - a0*c0 - b0*c0 + c0^2]
```

(printed to a single line). And `shown 1 [f, g, h]`
prints it as

```
[b1^2*c0 - b1*c1*b0 - b1*c1*c0 + c1^2*b0 + b0^2,
a1*b0 - a1*c0 - b1*a0 + b1*c0 + c1*a0 - c1*b0,
a1*b1*c0 - a1*c1*c0 - b1*c1*a0 + c1^2*a0 + a0*b0 - a0*c0 - b0*c0 + c0^2]
```

Also the `DoCon` library instances for `DShow` are so that any d-show function usually prints (recursively) a sub-expression in an expression `e` under less verbosity than it uses for `e`. For example,

```
showsn 2 [ (a1, [b(1,1),...,b(1,20)]), ..., (a20, [b(20,1),...,b(20,20)]) ] ""
```

prints `a1` under the verbosity `1`, `b(1,1)` — under the verbosity `0`.

All the d-show functions are defined via the operation

```
dShows :: ShowOptions -> a -> String -> String,
```

which also can be used separately.

The simplest setting for its first argument is `defaultShowOptions`.

The functions `showWithPreNLIf`, `showsWithPreNLIf`
print a data with breaking the line when it reaches the right margin given by a global `DoCon` constant

```
pageWidth = 72
```

(the user can edit this value in the `DoCon` source).

More examples of applying the d-show functions can be found 1) further in this manual, 2) in the examples in `demotest/*`, 3) in the `DoCon` library source.

3.8 Advancing with category declarations

The category hierarchy is expressed naturally by declarations like the following.

```
class (Eq a, Show a, DShow a) => Set a                      -- partially ordered set
  where
  compare_m  :: a -> a -> Maybe CompValue                   -- partial ordering
  showsDomOf :: Verbosity -> a -> String -> String          -- for messages
                               --sa
  fromExpr   :: a -> Expression String -> ([a], String)     -- parsing
               --sa  e                          messg

  baseSet    :: a -> Domains1 a -> (Domains1 a, OSet a)     -- description of a set
```

This means that any instance of the Equality and Printable `Haskell` classes can be made an algorithmic `Set` instance by implementing the operations `fromExpr`, `compare_m`, `baseSet`. Here

- `showsDomOf verbty smp` prints to `String` a short description of domain defined by the sample `smp`. It is used in error messages.
- `fromExpr`, together with the d-show functions, serve to parse and print between a domain element and a string.
- `baseSet` is explained in (Section 3.4 (BO),(BSE)) and in further sections.

Further,

```
class Set a => AddSemigroup a
  where              -- REQUIRED: 'add' is associative, commutative, agreed with (==)
  add     :: a -> a -> a
  zero_m  :: a -> Maybe a   -- sa
  neg_m   :: a -> Maybe a
  sub_m   :: a -> a -> Maybe a
  times_m :: a -> Z -> Maybe a
  baseAddSemigroup :: a -> Domains1 a -> (Domains1 a, Subsemigroup a)
                   -- sa
```

So `AddSemigroup` inherits by default the operations from `Set`, (and of supercategories `(D)Show`, `Eq` of `Set`), and exports the operations of its own. It proclaims that a `Set` instance can be made an additive semigroup instance by adding the above operations. Zero (`zero_m`) and opposite element `neg_m` are presented as partial maps. `zero_m s` has a sample argument `s` (Section 3.4 SA).

In this manner, `DoCon` adds `AddGroup, MulSemigroup` categories. Then, it declares

```
class (AddGroup a, MulSemigroup a, Num a, Fractional a) => Ring a
  where
  ...
```

which puts that the operations `(+) = add`, `(*) = mul`, `0 = zeroS _`, `1 = unity _` form a `Ring` — under certain presumed conditions for these operations. And so on, down to the leaves of the category tree shown on the picture in Section 3.3.

Some of the category declarations show no operations. For example,

```
class (EuclideanRing a, FactorizationRing a) => Field a
```

only asserts in its accompanying commentary that `inv_m` of `(MulSemigroup a)` has to satisfy a certain property.

3.9 Algebraic data and functors

Example 1.
`(1, 'b')` is an element of the direct product of domains `Z`, `Char`.
`(,)` is the data constructor from `Haskell`. `DoCon` relates to it a finite set of instance declarations

```
instance (Set a, Set b) => Set (a, b) where ...

instance (AddSemigroup a, AddSemigroup b) => AddSemigroup (a, b)
  where
  add (x, y) (x', y') = (add x x', add y+y')
  ...
```

— up to `Ring ... LinSolvRing`. Here `add x x'`, `add y y'` apply the `add` operation from the `AddSemigroup` instance for `a` and `b` respectively. This relates to the data constructor `(,)` the functor of the direct product of sets, functor of the direct product of additive semigroups, and so on. But the instances

```
... => EuclideanRing (a, b), ... Field (a, b)
```

are *not* declared. For, evidently, the direct product of rings cannot be an Euclidean ring. So, `DoCon` adds several `DoCon` instances to the `Haskell` library instances for `(,)`. The effect of these declarations is that setting in the program the expressions like

```
let p = (a1, b1) in  p + (c, d)*(c, d)
```

with `a1, c :: Ring a => a`, `b1, d :: Ring b => b`, causes the operations `+`, `*` to be resolved in the sense of the aforementioned instances for `(a, b)`.

Example 2. `DoCon` introduces an univariate polynomial as

```
data UPol a = UPol [UMon a] ...
```

— a name `UPol` serves both as a type and data constructor. The declaration

```
instance CommutativeRing a => AddSemigroup (UPol a)
  where
  add (UPol mons ...) (UPol mons' ...) = UPol (addpol mons mons') where ...
```

defines for any commutative ring `a` the corresponding additive semigroup instance on `a[x]`. The addition is expressed by the implementation part `addpol` which applies `zero_m`, `add` of the coefficient domain `a` to sum the coefficients. And so on, up to

```
instance Field a => EuclideanRing (UPol a) ...,
```

and some other instances. This brings the usual mathematical meaning to expressions like `f/(gcD f g)` with `f, g :: Field a => UPol a`.

More example: for `f, g :: UPol (Z, Fraction Z)`,
`f - f*g` also has usual algebraic meaning, it evaluates via the operations in the ring `(Z ⊕ Q)[x]` — the coefficients are from the direct sum of the integer and rational number rings. This is due to the instances that `DoCon` declares for the domain constructors of `Z`, `Fraction`, `(,)`, `UPol`.

3.10 User program design

With DoCon, it consists mostly of adding the data types, categories, implementing for these data the category instances (new, or of DoCon, or of the `Haskell` library), adding instances to the data constructors of DoCon, or `Haskell`. Naturally, the user items (data, functions, instances ...) may refer to the `DoCon` ones. For example, `DoCon` does not provide the trigonometric expressions. The user design for this might be

```
data TrigExpr a = TrigGround a | Sin (Trig a) | Cos (Trig a) ...

instance RealField a => Eq (TrigExpr a)
  where
  (TrigGround x) == (TrigGround y) =  x==y
  (Sin e1      ) == (Sin e2      ) =  ... implement some reasonable (partial)
                                          equality on the type TrigExpr a
...
instance RealField a => Set (TrigExpr a)  where ...
instance RealField a => AddSemigroup (TrigExpr a)
  where
  add (TrigGround x) (TrigGround y) = (TrigGround (x+y))
  add (Sin ...)      ...            = ...
```

— and so on

3.11 df and ndf domain constructors

The ndf `DoCon` constructors ("no domain field") are

`Char, Z, Permutation, [], (,), Vector, Fraction`

They do not provide a data field for an argument domain.

All the rest `DoCon` constructors are df — "with domain field".

The difference is in the cost control of the bundle builders. For ndf, they build the domains mostly by new, and recursively. For df — mostly, by taking the domains from the parameter fields of constructors.

`DoCon` could avoid ndf at all. For example, treat a direct product as

```
data Pair a b = Pr a b (Domains1 a) (Domains1 b)
```

But we think, it is better to remain with the mentioned above ndf. The reasons for this are that this
(a) simplifies denotations,
(b) is unlikely to cause much additional computation.

The latter is explained as follows.

- `Char, Z, Permutation` do not need a domain parameter at all.
- `[,]` makes only a `Set`, `DoCon` does not need any more instances for it.
- `Fraction` cannot apply twice for different arguments in any tower of ndf constructors. Because
  `Fraction [a], Fraction (a,b)` are senseless,
  `Fraction (Fraction a) === Fraction a` (why apply it twice?).
- `(,), Vector` often produce the final domain, not subjected to further constructors.

3.12 Algorithms for constructors

The underlaying mathematics for the constructor implementation is classical.
Let us sketch it.

Fraction

As usual, a fraction can be cancelled correctly over the (factorial) GCD-ring `R`. In particular, `R` has the operation `gcd`, and has `canInv` for the canonical invertible element [BL, Da]. This gives rise to the arithmetic of the fraction field `Fraction R`.

Pol

For the `CommutativeRing R` and the polynomial ring

$$P = R[vars] = R[x_1, \ldots, x_n]$$

defined by the sample element

```
Pol <monomials> r ord vars dR,   r :: R, vars :: [PolVar],
```

the `Set` instance (and all the extensions) depends on the indeterminate list `vars` and the power product ordering `ord`. The arithmetic `(+), (-), (*), (^), (/)` is evident.
The gcd operation is given, say by the subresultant algorithm — provided `R` is a gcd-ring — see Section 38.5.1. Factorization in $R[x], R[x, y]$ is provided for some R — see Section 38.5.2.
`UPol, RPol` are similar. `UPol` also has

```
instance Field k => EuclideanRing (UPol k) ...
```

Residue ring

For

- a ring `R` of category `LinSolvRing` possessing the attribute `(IsGxRing, Yes)` (Section 18),
- and ideal `I` in `R` supplied with generators `rs` and possessing (IsGxBasis, Yes) in its property list,

any sample element `Rsi x di dR,  x :: R`
defines the ring of residues.
`di` contains the ideal term `iI`.
`x` has to be a canonical representative modulo `iI`. Such representative can be found by reducing by the generators contained in `iI`, under the condition of `(IsGxBasis, Yes)`. Here follow the particular cases, when `DoCon` can compute such generators for `iI`.

For the so-called c-Euclidean ring (Section19), any `rs = [r]` with non-zero `r` is a GxBasis, and reduction is represented by Euclidean remainder `remEuc 'c'`.

For $P = R[x_1, \ldots, x_n]$, R a c-Euclidean ring, a gx-basis is a weak reduced Gröbner basis.

Further, for the direct sum $R \oplus U$, the gx-bases are defined via the generator lists that have the gx-bases as their projections. In all these cases, the canonical form for a residue element is evident, as well as the algorithms for the operations `+, -, *, ^` — see Section 49.

A partial operation `(divide_m b a)` is generally treated as solution of linear equation $a * x = b$. It has to find *any* solution or return `Nothing`. Together with `syzygyGens`, this yields the generic solution. For a c-Euclidean ring R, such division in $R/(d)$ can be done via the extended gcd algorithm:

```
u*d + v*b = g = gcd(a,b);           q = a/g;
quotient = canonicRemainder(q*u, b)
```

For a gx-ring, say polynomials over a field, we use similarly `gxBasis`, which can be considered as the generalization for the mentioned above extended gcd method. See Sections 18, 49.

And so on, for other constructors.

3.13 No limitations. Generality

`DoCon` is very generic. Thus,
(1) integer, rational number, list or vector size, power product and its entity — may be arbitrary large,
(2) polynomials are considered under any indeterminate list and arbitrary comparison function for the power products,
(1) sorting takes in the argument any comparison function and returns also the permutation sign,
and so on.

3.14 Parsing, unparsing

Unparsing is writing to `String`, it is by the instances of the classes `Show, DShow` — see Section 3.7.

Parsing

is reading from `String`. This is more complex. `DoCon` uses for this the operation

```
fromExpr ::   a -> Expression String -> ([a], String)
           -- sa
```

of the class `Set`. But before this, the instances of `Set, Show, DShow` must be declared.

The generic *parsing by sample*

`smParse`

is the composition of the functions `lexLots, infixParse, fromExpr`.

- `lexLots` splits a string to lexemes.
- `infixParse` takes a lexeme list and produces a syntactic tree, — `Expression`, — with the lexemes in the leaves. At this stage, the infix operations, as `+, *`, and others, have no more sense beyond their signature and precedence.
- `fromExpr` gives an interpretation to the `Expression` according to the sample element of domain. It must be programmed for each domain.

`smParse` is defined as follows:

```
smParse :: Set a => a  ->  String -> a
smParse             sample s      =
  case
      infixParse parenTable opTable $ lexLots s
  of
    ([e], "" ) -> case  fromExpr sample e  of
                               ([x],"" ) -> x
                               (_   ,msg) -> error (..."bad string for sample"...)
    (_   , msg) -> error (..."infixParse:  "++msg++"\n")
```

Example. For `str = " (1+2, 1 ) "`,

```
smParse ((1,1) :: (Z,Z)) str              --(infix . lexLots)-->

(E "," (E "+" "1" "2") "1") =  e   :: Expression String      -->

fromExpr ((1,1)::(Z,Z)) e                                     -->  (3,1)
```

Further, `smParse ((1:/1, 1) :: (Fraction Z, Z)) str --> (3:/1, 1)` — changing the sample changes the parser for the domain. This is because the `fromExpr` instance changes. Very naturally, `fromExpr` is defined recursively, following the type (or domain) construction.

More example: for a polynomial, a sample element contains also the list of indeterminates (variables) which are recognized in the `Expression` by `fromExpr` function. Also `smParse` uses these indeterminates to display the polynomial. See `Set`, Section 47, and the files `source/parse/princip.txt`, `Iparse_.hs`.

Why do we need this intermediate `fromExpr` and conversion to `Expression` ? Is not it simpler to parse straight from `String`, exploiting `lex` ? For example, to parse `(n:/m) :: Fraction a` from `str`, parse the element of `a` from `str` (recursion); then `lex` gets `:/`, then parse element of `a` — ?

We tried this, but met the difficulty with parsing the elements of domains like $((R[x])[y])[z]$. It is hard to define for the strings like `" y*2*x + z "`, where the "coefficient" or "monomial" ends with `"y"`, or maybe, with `"z"` respectively.

Instead, `infixParse` sets the parentheses systematically, and then, `fromExpr` is simple to define.

4 Usage of Haskell Prelude

DoCon exploits the Haskell Prelude items and its libraries. But for some of operations it adds its own counterparts. It coexists peacefully with the Haskell Prelude and its libraries, and never changes the Prelude operation meaning for the Prelude domains or constructors.

Here follow some points concerning the interaction with Prelude.

No Int : DoCon uses only `type Z = Integer` for the ring of integers.

No fromInteger :

DoCon relies instead on operation `fromi s` — map from Integer by sample.

Examples:

```
(2, 3) + 4 :: (Z, Z) --> error "... Undefined member: fromInteger",

let  p          = (2, 3) :: (Z, Z)
     (p1:p2:_) = map (fromi p) [1 ..]
in   p + p2*p2                         --> (6, 7)
```

List processors `minimum, maximum, sort, sortBy.`

are re-implemented. In particular, to import the DoCon definitions for them, the program has to specify the import like this:

```
import List hiding (minimum ...)
import DPrelude    (minimum ...)
```

This is the choice of DoCon's taste. For example, DoCon prefers "merge" algorithm for sorting, and some Haskell implementations prefer "quickSort" (faster on average, and $O(N^2)$ in the worst case). But the situation may change. Besides, this issue is not so important. The user is free to import the variant one likes.

sum1, product1 instead of sum, product

For Z, there is no difference. But for other lists `xs :: Ring a => [a]`, apply `sum1 xs`, `product1 xs`. They are for non-empty lists.

If `xs` may occur empty, then apply

`sum1 ((zeroS s):xs)` or `product1 ((unity s):xs)`

respectively — with any sample element `s` for the needed domain. This is because the Haskell Prelude initiates `sum`, `product` with `fromInteger 0`, `fromInteger 1`, while DoCon has `fromi <sample>` instead of `fromInteger`.

Example:

```
sum  [(1,2),(3,4)]  :: (Z,Z)  --> "error: Undefined member: fromInteger"
sum1 [(1,2),(3,4)]  :: (Z,Z)  --> (4,6)
```

Classes Prelude.Eq, Prelude.Show :

DoCon relies on them (but it often uses its own class DShow instead of Show). See (Section 3.4 (BSE)) on Eq.

Read :

instead it, DoCon applies smParse, fromExpr — see Section 3.14.

Prelude.Ord

includes into the DoCon hierarchy with the condition of compare_m to agree with compare:

```
class (Ord a, Set a) => OrderedSet a
  -- Presumed:
  -- on the base set BS,  compare_m  possesses (OrderIsTotal, Yes),
  -- and agrees with 'compare': (compare_m x y)==(Just $ compare x y)
```

For other domains, of Set, but not of Ord, compare_m may differ, for example, may be defined as compareTrivially. Note also that the *property list* of the base set contains certain information on compare_m.

On unlucky Num, Fractional

We think (see the paper in [Me2]) that the Haskell-2010 library categories
Num, Fractional, Integral are not reasonably organized from mathematical point of view.

But Haskell relates to them the names +, -, *, ^, / ...,
And DoCon likes to use these names. So, it includes these classes into its categories in a following way. Any domain with the Ring instance has the operations

```
add,  neg,  sub,  mul,  power,  divide,
```

some of them — with their _m counterparts.
DoCon puts Num, Fractional to be the superclasses for Ring:

```
class (AddGroup a, MulSemigroup a, Num a, Fractional a) => Ring a
  where
  ...
  -- presumed:
  -- ... (+) = add, (*) = mul  obey the ring laws,  (/) = divide
```

Concerning Haskell's abs, signum, fromInteger:
DoCon uses them, for example, for Z, but in general case, it relies on

```
absValue :: AddGroup a => a -> a
absValue x = if less_m x $ zeroS x  then  neg x  else  x
```

and on times(_m) of AddSemigroup, fromi of Ring.

Unlucky Prelude.Integral:

The `Haskell-2010` Prelude provides this class, probably, as the merely a holder for `Integer` and `Int`.

`DoCon` applies sometimes the operations from `Integral`, say `quotRem` — but *only* for `Integer`. For the general purpose (for `Integer` too), it has the `EuclideanRing` category with its operations `divRem`, `eucNorm`.

Caution: the `Haskell` class `Integral` does not express what is called 'integral domain' in classic algebra. Rather it intends the meaning 'isomorphic to Integer'.

For example, `Haskell`'s `Integral a => Ratio a` applies `divMod, sign, (<)` to cancel fractions — which is not so appropriate for the polynomials over a field.

Instead, `DoCon` defines the categories `GCDRing`, `EuclideanRing`, and so on — which allow the generic `Fraction` constructor. Concerning `divRem`, the main difference is that it does not rely on the ordering (sign of remainder and such attributes). And `divRem` extends naturally to a polynomial domain $k[x]$ for a field k, some quadratic integer rings, and so on.

No Ratio :

instead, `DoCon` introduces the `Fraction` constructor, with its operations `:/`, `canFr`.

`DoCon` uses slightly different algorithms for the fraction sum, product, and such. Also `Fraction` is related to the `GCDRing` category of `DoCon` and the property values as `IsGCDRing`, and so on. See Section 34.

Sorry for 'floating point' numbers:

`DoCon` does not provide any instances for them
(for the types `Float`, `Double`, `Complex` ...), just never got around.

Caution: these types of `Haskell-2010` have the operations `+, *, 0, 1` of class `Num`, but their definition does not satisfy the `AddSemigroup` law, or `MulSemigroup`, and such. So, it hardly makes sense to declare such instances for them. Probably, they need some other treating.

Generally, the paper [Me2] suggests the Prelude algebra improvement for future `Haskell`.

4.1 Avoiding name clashes with Haskell

zero is certain pre-defined *monad* from Haskell.

DoCon uses zero_m, zeroS to form zero element.

partition of Haskell library breaks a list by a predicate.

DoCon uses partitionN for the repeated partition.
And the type Partition refers to Young diagrams.

gcD, lcM are of DoCon GCDRing category, they take a list of elements in argument.
And gcd, lcm are of Haskell Prelude.

The most clash-dangerous global names from DoCon are

- L, E data constructors for Expression,
- char :: Ring a => a -> Maybe Z for characteristic of a Ring,
- add, sub, neg, mul, inv, ct, ctr

Qualified names
(of kind <moduleName>.<itemName>). They help to avoid name clashes. For example,

```
import qualified AlgSymmF ( to_e                              )
import AlgSymmF           ( <all needed items, except to_e> )

f    =  ... AlgSymmF.to_e ...
to_e =  <another to_e>
```

5 What is not used from Haskell

In the `Haskell-2-pre` language, `DoCon` ignores the following features.

- `Ratio` — see Section 4.
- **arrays** — `DoCon` prefers more explicit `Map.Map` data (exploiting a binary tree).
- **existential types**
  — it is not clear, so far, how they work and whether `DoCon` needs them.
- **monadic programming style**
  — we do not see so far, how monads may help DoCon.
- **input-output**
  — except `putStr, writeFile` in the end of some example programs, `getChar` in the Test program.
  Small exception: sometimes it may be used the `SymmDecMessageMode` for intermediate messages.

5.1 About strictness annotations

Generally, `DoCon` avoids them. There is only a single place where the strictness annotation (of `'seq'`) is applied.

6 Demonstration, test, benchmark

The source directory `<D>/docon/source/demotest/T_*`
contains the automatic test function `T_.test` for all the important `DoCon` facilities. The functions from `T_*` can run as a demonstration and as a benchmark — it depends on which part of their result are taken by the user function. See `install.txt`, `T_.hs`.

Let us proceed now with the systematic description of the `DoCon` parts.

7 Program modules of DoCon

DoCon keeps one module in one .hs file. DoCon has a small number of the “open” modules — the ones that are expected to be imported by the user program. They reside in <D>/docon/source/ :

```
AlgSymmF.hs    Fraction.hs    Pol.hs        Z.hs
Categs.hs      GBasis.hs      Residue.hs    DExport.hs
LinAlg.hs      RingModule.hs  DPair.hs      Partition.hs
SetGroup.hs    DPrelude.hs    Permut.hs     VecMatr.hs
```

Here DExport only reexports all the other open modules, and many items from GHC library too.

The rest of DoCon modules are ‘hidden’ — not expected to be imported by the user program. The hidden modules contain the most part of implementation. Their names end with ‘_’, say auxil/Pol__.hs, and such. They reside in

```
<D>/docon/source/auxil/  demotest/    lin/   parse/  pol/  residue/
                         pol/factor/  pol/symmfunc/
```

DPrelude contains generic items for the whole DoCon, such as

```
class Cast,
data PropValue, data InfUnn(..)
types  Z, Natural, MMaybe, CompValue, Comparison,
functions   tuple31, tuple32 ... alteredSum, sortBy ...
instance Set Char,  instance ... => Set [a] ...
```

Categs contains the data structures for domain term, bundle:

```
class Dom,
data CategoryName(..), Domain1, Domain2,  OSet, Subsemigroup ...
Submodule ...
types  Domains1,Domains2 ...
```

SetGroup contains the category declarations for

```
Set, AddSemigroup, MulSemigroup, AddMonoid, AddGroup, MulGroup ...,
```

polymorphic functions

```
zeroS, times, unity, power, divide, compareTrivially, isFiniteSet,
listToSubset, isoOSet, upAddGroup, upMulSemigroup, upMulGroup ...
```

RingModule contains the categories

```
Ring ... GCDRing ... Field, LeftModule(..),
```

some functions for `Submodule`, some instances of `LeftModule`,
the 'Chinese' ideal data `PIRChinIdeal(..)`, many auxiliary functions for `Ring, GCDRing, EuclideanRing` ...:

```
quotEuc, remEuc, eucGCDE, multiplicity, isPowerOfNonInv, isoRing ...
```

Z
declares category instances for `Z = Integer` — in addition to the `Haskell` Prelude ones:

```
Set ... AddGroup ... EuclideanRing ... FactorizationRing,
```

defines a domain bundle `dZ` for `Integer`.

DPair
declares instances for the constructor `(,)` of the direct product — in addition to the `Haskell` Prelude ones.

Permut defines some instances and functions for `Permutation` constructor.

Fraction
contains various items for the construction `Fraction a`, `a` a gcd-Ring: some generic functions: `num`, `denom`, `canFr`, instances `Set` ... `OrderedField`.

VecMatr
declares items for `Vector a, Matrix a, SquareMatrix a`:
category instances `Set ... AddGroup` for all the three, instances,
`Ring` ... for `Vector a`, `Ring` for `SquareMatrix a`,
several generic functions:

```
vecSize, vecHead, vecTail, constVec, scalProduct,
isLowTriangMt, vandermondeMt, resultantMt
```

Pol declares items for the polynomial constructors

```
UPol, Pol, EPol, RPol, VecPol,
```

declares `type PowerProduct`, with its related functions `lexComp, degLex ...`,
classes `PolLike, Dom`, their instances for `UPol`, `Pol`, `EPol`, `RPol`,
instances for `UPol`: ... `EuclideanRing, FactorizationRing`,
instances for `Pol`: ... `LinSolvRing, FactorizationRing`,
instances for `RPol`: ...`GCDRing`,
instances

```
... => LinSolvLModule (Pol  a) (EPol a),
       LinSolvLModule (Pol  a) (Vector (Pol  a)),
```

and such,
numerous functions for these kinds of polynomial, various conversions between them, and so on.

Residue declares the `Residue` class and the items for

- `ResidueG a` — residue by the subgroup of a commutative group,
- `ResidueE a` — residue by an ideal of an Euclidean ring,
- `ResidueI a` — `a/I`, residue by an ideal `I` of a `LinSolvRing`, `I` given by gx-basis.

For `ResidueG`, it contains the instances `Set` ... `AddGroup`,
for others — `Set ... Field`.

It also declares some items related to `Ideal, PIRChinIdeal`.

LinAlg
defines several functions for the staircase form of matrix (Gauss method), diagonal form, determinant, linear system solution ...

GBasis
defines the functions of the Gröbner reduction to normal form, Gröbner basis, syzygy generator list, algebraic relation generator list
— for the vector domain over $a[x_1, \ldots, x_n]$, a an Euclidean ring.

Partition
defines operations with the partitions (Young diagrams), shapes, skew hooks,
horizontal bands:

```
partition conjugation, union, ..., subtracting h-bands, s-hooks,
previous partition, hook, band, ... ,
```

Kostka number, character matrix for $S(n)$, and some others.

AlgSymmF
defines arithmetic for symmetric functions — sym-polynomials,
transformations between various bases of the symmetric function algebra [Ma]:
`m, s, p, h, e`.

Now, proceed with the description of the `DoCon` components.

8 DoCon prelude

It consists of a number of the general purpose items exported from the module

DPrelude.hs

— see the export list of this module in Section 50.
Some of these items are implemented in auxiliary modules

`Prelude_`, `Common_`, `Char_`, `List_`, `Iparse_` ...

`DPrelude` also exports the `Set` instances for the domains `Char`, `List`:

```
instance Set Char         where ...
instance Set a => Set [a] where ...
```

— to add to the `Haskell` Prelude instances for `Char`, `[]`.

Describing here the `DoCon` Prelude, and further, describing other parts of DoCon, we partly repeat the contents of the relevant modules. But many implementation details are skipped. And the Section 50 is specially devoted to listing of each module export.

```
module DPrelude ...

type Natral = Integer
type Z      = Integer              -- IGNORE Int !

toZ :: Integral a => a -> Z
toZ =  toInteger
fromZ :: Num a => Z -> a
fromZ =  fromInteger

type Natural = Integer              -- use it for values >= 0

tuple31 (x, _, _) = x
tuple32 (_, x, _) = x
tuple33 (_, _, x) = x
                               --- and other  tuple<i><j>,  i,j <- [3,4,5]
  -- There comes a language extension where these  tuple  things  are
  -- done in a nicer way.

compose :: [a -> a] -> a -> a
compose =  foldr (.) id

sublists :: [a] -> [[a]]
sublists []      =  [[]]
sublists (x: xs) =  (map (x :) ls) ++ ls  where  ls = sublists xs
```

```
listsOverList :: Natural -> [a] -> [[a]]
                      -- list of all lists of length n with the elements from xs.
                      -- It does Not take notice of repetitions in  xs.

partitionN :: (a -> a -> Bool) -> [a] -> [[a]]
                          -- break list into groups by the Equivalence relation p
partitionN _ []      = []
partitionN p (x: xs) = (x: ys): (partitionN p zs) where
                                                  (ys, zs) = partition (p x) xs
               --
               -- but for the case of equivalent items being *neighbours* use groupBy

-- BEGIN   suggestion for the Haskell library? *************

mapmap :: (a -> b) -> [[a]] -> [[b]]
mapmap    f        =  map (map f)

fmapmap :: Functor c => (a -> b) -> c [a] -> c [b]
fmapmap                 f        =  fmap (map f)

mapfmap :: Functor c => (a -> b) -> [c a] -> [c b]
mapfmap                 f        =  map (fmap f)

fmapfmap :: (Functor c, Functor d) => (a -> b) -> c (d a) -> c (d b)
fmapfmap                              f        =  fmap (fmap f)

                                     -- rewritten from  Maybes  of 'data'  to
                                     -- avoid mentioning non-standard Maybes
allMaybes :: [Maybe a] -> Maybe [a]
allMaybes []           = Just []
allMaybes (Nothing: _ ) = Nothing
allMaybes (Just x : ms) = case allMaybes ms of  Nothing -> Nothing
                                                Just xs -> Just (x: xs)

takeAsMuch, dropAsMuch  :: [a] -> [b] -> [b]
--
-- the following two implementations are nicer (and, probably, faster)
-- than combining 'length' with 'take' and 'drop'.
--
takeAsMuch (_: xs) (y: ys) =  y: (takeAsMuch xs ys)
takeAsMuch _       _       =  []

dropAsMuch xs ys =  case (xs, ys) of ([],      _     ) -> ys
                                     (_,       []    ) -> []
                                     (_: xs', _: ys') -> dropAsMuch xs' ys'
```

```
eqListsAsSets :: Eq a => [a] -> [a] -> Bool
eqListsAsSets xs ys =  all (`elem` xs) ys  &&  all (`elem` ys) xs

zipRem :: [a] -> [b] -> ([(a,b)], [a], [b])
                    -- zip preserving remainders. Example:
                    -- for  xs = [1,2]  zipRem xs (xs++[3]) = ([(1,1),(2,2)], [], [3])

zipRem []      ys      = ([], [], ys)
zipRem xs      []      = ([], xs, [])
zipRem (x: xs) (y: ys) = ((x, y): ps, xs', ys') where
                                                (ps, xs', ys') = zipRem xs ys

delBy :: (a -> Bool) -> [a] -> [a]
delBy _ []      = []
delBy p (x: xs) = if  p x  then  xs  else  x: (delBy p xs)

separate :: Eq a => a -> [a] -> [[a]]
                         -- break list to lists separated by given separator
                         -- Example:  ';' -> "ab;cd;;e f " -> ["ab", "cd", "", "e f "]
separate _ [] = []
separate a xs = case span (/= a) xs of (ys, []  ) -> [ys]
                                       (ys, _:zs) -> ys:(separate a zs)
pairNeighbours :: [a] -> [(a,a)]
pairNeighbours (x:y:xs) = (x,y):(pairNeighbours xs)
pairNeighbours _        = []

removeFromAssocList :: (Eq a) => [(a,b)] -> a -> [(a,b)]
removeFromAssocList []              _  = []
removeFromAssocList ((x',y):pairs)  x  =
                        if x==x' then pairs  else (x',y):(removeFromAssocList pairs x)

addToAssocList_C :: Eq a => (b -> b -> b) -> [(a,b)] -> a -> b -> [(a,b)]
                    -- c
                                  -- combines with the previous binding: when a key is
                                  -- in the list, it binds with  (c oldValue newValue)
addToAssocList_C c pairs key value = add pairs
  where
  add []        = [(key,value)]
  add ((k,v):ps) = if k /= key then (k,v):(add ps) else (k, c v value):ps

addListToAssocList_C :: Eq a => (b -> b -> b) -> [(a,b)] -> [(a,b)] -> [(a,b)]
addListToAssocList_C c ps binds = foldl addOne ps binds
                                          where
                                          addOne ps (k,v) = addToAssocList_C c ps k v
-- END ************************************************************
```

```
compBy :: Ord b => (a -> b) -> Comparison a
compBy f x y = compare (f x) (f y)
  --
  -- Usable tool to define comparison. Examples:
  -- compBy snd          compares pairs by second component,
  -- compBy (abs . snd)  -              by absolute value of second component

mulSign :: Char -> Char -> Char
mulSign    x       y    =  if  x==y  then '+'  else '-'

invSign '+'  = '-'
invSign '-'  = '+'

evenL :: [a] -> Char  -- '+' ('-') means the list has even (odd) length
evenL []     = '+'
evenL (_:xs) = invSign (evenL xs)

cubeList_lex :: (Show a, Ord a, Enum a) => [(a,a)] -> [[a]]
                                           --bounds
  -- Lists in the lex-increasing order all the vectors  [a(1)..a(n)]
  -- over 'a' in the cube   a(i) <- [l(i) .. h(i)],  1 <= i <= n,
  -- defined by  bounds = [(l(1),h(1))..(l(n),h(n))],  l(i) <= h(i)
  -- Example: [(0,2),(0,3)] ->
  --                     [ [0,0],[0,1],[0,2],[0,3],[1,0],[1,1],[1,2],[1,3],
  --                       [2,0],[2,1],[2,2],[2,3] ]

factorial :: Z -> Z
factorial 0 = 1
factorial n = if n < 0 then error $ ("factorial "++)$ shows n ":  negative argument\n"
              else          product [1..n]
```

```
sum1, product1, alteredSum ::  Num a => [a] -> a
```

`alteredSum` $[x_1..x_n] = x_1 - x_2 + x_3 - x_4 \ldots$
These functions are for *non-empty* list.
For non-integer, apply `sum1` instead of `sum`, `product1` instead of `product`.
If `xs` may occur empty, then apply `sum1 ((zeroS s):xs)`
or `product1 ((unity s):xs)` respectively — with any sample element `s` for the domain `a` — see `sum1` in Section 4.

```
binomCoefs :: Natural -> [Natural]
                    -- binomial coefficients [C(n,k)..C(n,0)], k <= n/2+1
```

```
data InfUnn a = Fin a | Infinity | UnknownV  deriving(Eq, Show, Read)

--- to represent domain ‘a’ extended with the Infinity and Unknown
values.
Examples: Fin n  :: InfUnn Z   means a ‘finite’ integer,

          let  {(_,sChar)= baseSet 'a' Map.empty;  (_,sZ) = baseSet 0 dZ}
          in
          (osetCard sChar, osetCard sZ)    --> (Fin 256, Infinity)

instance Functor InfUnn  where  fmap f (Fin a)  = Fin (f a)
                                fmap _ Infinity = Infinity
                                fmap _ UnknownV = UnknownV
type MMaybe a = Maybe (Maybe a)
  --
  Example of usage:
  subsmgUnity sS -> Just (Just u)  means here the unity  u  is found in sS,
                    Just Nothing   - sS does not contain unity,
                    Nothing        - cannot determine whether such  u  exists

type CompValue = Ordering               -- ‘Ordering’ is from Haskell-2010
                                        -- and it does not sound well
type Comparison a = a -> a -> CompValue
data PropValue = Yes | No| Unknown  deriving (Eq, Ord, Enum, Show, Read)

not3 :: PropValue -> PropValue
not3 Yes =  No
not3 No  =  Yes
not3 _   =  Unknown

and3, or3 :: PropValue -> PropValue -> PropValue

and3 Yes Yes = Yes
and3 No  _   = No
and3 _   No  = No
and3 _   _   = Unknown

or3 ...

boolToPropV :: Bool -> PropValue
boolToPropV b =  if  b  then Yes  else No

propVToBool :: PropValue -> Bool
propVToBool Yes =  True
propVToBool _   =  False
```

```
compBy :: Ord b => (a -> b) -> Comparison a
compBy f x y = compare (f x) (f y)
                        -- Usable tool to define comparison. Examples:
                        -- compBy snd          compares pairs by second component,
                        -- compBy (abs . snd)  - by absolute value of second component

antiComp :: CompValue -> CompValue
antiComp LT =  GT
antiComp GT =  LT
antiComp _  =  EQ
         -- It holds: (flip compare) x y == flip compare x y == antiComp $ compare x y
         -- Use  'flip'.  Though,  antiComp  occurs sometimes useful too.

less_m, lessEq_m, greater_m, greaterEq_m, incomparable :: Set a => a -> a -> Bool

less_m       x y = case  compare_m x y  of  Just LT -> True
                                            _       -> False
greater_m    x y = case  compare_m x y  of  Just GT -> True
                                            _       -> False
incomparable x = not . isJust . compare_m x

lessEq_m    x y = x==y || less_m    x y
greaterEq_m x y = x==y || greater_m x y

lexListComp :: (a -> b -> CompValue) -> [a] -> [b] -> CompValue
                     -- Compare lists lexicographically according to the given
                     -- element comparison cp. The lists may differ in type and length
lexListComp cp = lcp where
                     lcp []     []     = EQ
                     lcp []     _      = LT
                     lcp _      []     = GT
                     lcp (x:xs) (y:ys) = case  cp x y  of  EQ -> lcp xs ys
                                                           v  -> v

minBy, maxBy :: Comparison a -> [a] -> a  --minimum, maximum by the given comparison
                                         -- Example: minBy compare        [2,1,3,1] = 1
                                         --          minBy (flip compare) [2,1,3,1] = 3

minPartial, maxPartial :: Eq a => (a -> a -> Maybe CompValue) -> [a] -> Maybe a
  -- Minimum (maximum) by Partial ordering.
  -- The result maybe
  --           Just m   - for  m <- xs & m <= x  for all  x  from xs,
  --           Nothing  - if there is no such  x  in  xs.

isOrderedBy :: Comparison a -> [a] -> Bool
                            -- Examples:  isOrderedBy compare        [1,2,2] -> True,
```

```
                          --          isOrderedBy (flip compare) [1,2,2] -> False

minAhead, maxAhead :: Comparison a -> [a] -> [a]
                                    -- put ahead the minimum (maximum)
                                    -- without changing the order of the rest

mergeBy :: Comparison a -> [a] -> [a] -> [a]    -- merge lists ordered by cp
mergeBy _  []     ys     = ys
mergeBy _  xs     []     = xs
mergeBy cp (x:xs) (y:ys) = case cp x y of GT -> y:(mergeBy cp (x:xs) ys)
                                          _  -> x:(mergeBy cp xs (y:ys))

mergeE :: Comparison a -> [a] -> [a] -> ([a],Char)
                     -- extended merge: permutation sign '+' | '-' is also accumulated

sortE :: Comparison a -> [a] -> ([a],Char)
                     -- Extended sort: permutation sign '+' | '-' is also accumulated.
                     -- The cost is still  O( n*log(n) ).

propVOverList :: Eq a => [(a,PropValue)] -> a -> PropValue -> [(a,PropValue)]
propVOverList ps _  Unknown = ps                              -- update property value
propVOverList ps nm v       = pov ps
  where
  pov []            = [(nm,v)]
  pov ((nm1,v1):ps) = if  nm1/=nm  then  (nm1,v1):(pov ps)
                      else               (nm ,v ):ps

updateProps :: Eq a => [(a,PropValue)] -> [(a,PropValue)] -> [(a,PropValue)]
updateProps ps ps' = foldl update ps ps'
                                    where  update ps (nm,v) = propVOverList ps nm v

mbPropV :: Maybe PropValue -> PropValue
mbPropV    (Just v)       =  v
mbPropV    _              =  Unknown

lookupProp :: Eq a => a -> [(a,PropValue)] -> PropValue
lookupProp            a =  mbPropV . lookup a

addUnknowns :: Eq a => [(a,PropValue)] -> [a] -> [(a,PropValue)]
addUnknowns props = foldl addOne props
  where
  addOne  []               nm = [(nm,Unknown)]
  addOne  ps@((nm',v):ps') nm = if nm==nm' then  ps  else  (nm',v):(addOne ps' nm)


                         smParse :: Set a => a -> String -> a
```

Generic parsing by sample. Applies `infixParse, fromExpr`.
See Section 3.14, function `Iparse.infixParse`,
operation `fromExpr` of `Set` category.

```
smParse sample s = case infixParse parenTable opTable$ lexLots s  of

  ([e], "" ) -> (case  fromExpr sample e
                 of
                   ([x],"" ) -> x
                   (_  ,msg) -> error $ ("fromExpr sample str:  bad string.\n"++) $
                                showsWithDom sample "sample" ""
                                ('\n':(msg++"\n"))
                )
  (_  , msg) -> error ("infixParse:  "++msg++"\n")
---------------------------------------------------------------------
-- for the following 3 operations with the association lists, we
-- could not find a good replacement in the Haskell libraries

removeFromAssocList :: Eq a => [(a,b)] -> a -> [(a,b)]
addToAssocList_C    :: Eq a => (b -> b -> b) -> [(a,b)] -> a -> b -> [(a,b)]
                               -- c
                                  -- combines with the previous binding: when a key is
                                  -- in the list, it binds with  (c oldValue newValue)

addListToAssocList_C :: Eq a => (b->b->b) -> [(a,b)] -> [(a,b)] -> [(a,b)]
addListToAssocList_C c ps binds = foldl addOne ps binds where
                                     addOne ps (k,v) = addToAssocList_C c ps k v
```

```
                   class Cast a b where  cast :: Char -> a -> b -> a
```

`cast mode a b` means to cast `b` to domain of `a`.
`a` is used as a *sample* for the destination domain.
`mode = 'r'` means the additional correction to perform,

other value means to cast "as it is".

Example 1.
map coefficient $\mathtt{c} \in R$ to domain $R[x_1, \ldots, x_n]$:
`cast mode pol c`, `mode = 'r'` means the check `c == zero` is needed, other mode is usually set when `c` is known as non-zero.

Example 2.
Projection $R \to R/I$ to residue ring,
say $R = Z,\ I = Ideal(g),\ g > 0,$ `res` $\in R/I$, `n` $\in R$, and `cast mode res n` projects `n` to R/I.
Here `mode = 'r'` takes first the remainder by g (g contained in `res` data); other mode has sense when it is known that $0 \leq$ `n` $< g$.

```
ct, ctr :: Cast a b => a -> b -> a
ct  = cast '_'
ctr = cast 'r'
```

These functions are explained in Sections (3.4 (CS)), 3.5.4.

9 Domain descriptions

Their general meaning is explained in (Section 3.4 (D), (DT) ...).
Their data types are mostly exported from the module `Categs`.

```
module Categs
...
-- some prelude

data AddOrMul = Add | Mul deriving (Eq,Show,Ord,Enum)  -- additive | multiplicative
                                                       -- subsemigroup
type {-Ring a=>-}  Factorization a = [(a,Z)]

  -- example: 8*49*3 = 2^3*7^2*3   expresses as  [(2,3),(7,2),(3,1)]:: Factorization Z

newtype Vector a = Vec [a] deriving (Eq)    -- Presumed: NON-empty list under Vec
vecRepr (Vec l) = l

type PowerProduct = Vector Z
type PPComp       = Comparison PowerProduct       -- power product comparison
```

9.1 Bundle

```
data CategoryName =
     Set | AddSemigroup | AddGroup | MulSemigroup | MulGroup | Ring
     | LinSolvRing | GCDRing  | FactorizationRing | EuclideanRing |
     IdealKey | LeftModule | LinSolvLModule
     -- may extend ...
     deriving (Eq, Ord, Enum, Show)
```

```
type Domains1 a = Map.Map CategoryName (Domain1 a)
```

Represents a domain or a bundle — see (Section 3.4 (D), (DT), (MD))
— with one parameter `a`.

```
type Domains2 a b = Map.Map CategoryName (Domain2 a b)

data Domain1 a =
     D1Set      (OSet a)            | D1Smg    (Subsemigroup a)  |
     D1Group    (Subgroup a)        | D1Ring   (Subring a)       |
     D1GCDR     (GCDRingTerm a)     | D1FactrR (FactrRingTerm a) |
     D1LinSolvR (LinSolvRingTerm a) | D1EucR   (EucRingTerm a)   |
     D1Ideal    (Ideal a)
     -- ...
```

— see (Section 3.4 (DT)).

```
data Domain2 a b = D2Module (Submodule a b) | D2LinSolvM (LinSolvModuleTerm a b)}
-- may extend ...
```

9.2 Dom class

```
class Dom c where
            dom    :: c a -> Domains1 a  -- extracts the description of domain
                                         -- over which the constructor c acts
                                         -- - description of argument domain of c
            sample :: c a -> a   -- extracts sample element for argument domain
```

Examples:

for `f = UPol _ 0 _ dZ` — univariate polynomial over `Z` —
`dom f = dZ,  sample f = 0`;

for `r = Rse 2 iI dZ` — residue modulo `I` of `Z` —
`dom r = dZ,  sample f = 2`. See (Section 3.4 (CS), (DE)).

9.3 Domain terms

See first (Section 3 (DT)).
The rest of the module `Categs` describes the domain term data:
`OSet, Subsemigroup, ..., GCDRingTerm, ..., Submodule, ...`
We describe them separately, in the sections devoted to each particular category — `Set`, `AddSemigroup`, and so on.

Now, proceed with these categories.

10 Set

10.1 Set category

See the module SetGroup, OSet data declaration in the module Categs.

```
class (Eq a, Show a, DShow a) => Set a            -- partially ordered set
  where
  compare_m  :: a -> a -> Maybe CompValue           -- partial ordering
  showsDomOf :: a -> String -> String               -- for messages
              --sa
  fromExpr   :: a -> Expression String -> ([a], String)      -- for parsing
              --sa   e                           messg

  baseSet    :: a -> Domains1 a -> (Domains1 a, OSet a)
              --sa                           description of set
```

baseSet
provides the main information on set. It returns the description of the base subset related to the given sample element. See (Section 3.4 (DS), (SA), (BO)).

```
baseSet x dom --> (dom', o)
```

dom' = Map.insert Set (... o) dom.
o is either found in dom or built according to the construction of x and dom contents.

compare _m
Any Set is considered as partially ordered by compare_m.
At least, the trivial ordering compare_m = compareTrivially
can be specified:

```
compareTrivially x y = if x==y then Just EQ else Nothing
```

All the subsets of the base set are considered relatively to compare_m.
compare_m must define a transitive and anti-symmetric relation. It takes the values in

```
Just LT | Just GT | Just EQ | Nothing
```

— the latter value means the incomparability of elements.
Further information on this ordering is contained in the base subset description bS — for example, bS = snd $ baseSet <sample> Map.empty.

Recall also of compare of Haskell Prelude. In principle, for some domains of Set and Ord instances, compare and compare_m may disagree. But starting from OrderedSet, they have to agree.

showsDomOf x str --> str'

prints to String the short description of domain defined by sample `x`.
It is used in error messages: see Section 10.5.

fromExpr

is reading from expression by sample — see Section 3.14, function `DPredude.smParse`, the files `parse/princip.txt, Iparse_.hs`.

If `fromExpr` succeeds, the result is `([elementOf_a], "")`,
otherwise, it is `(_, message)`, where `message` contains some explanation of why the expression `e` does not fit to represent an element of domain (defined by sample).

10.2 Subset

```
data OSet a =
     OSet {osetSample  :: a,                    -- sample data for type
           membership  :: Char -> a -> Bool,
           osetCard    :: InfUnn Z,             -- cardinality
           osetPointed :: MMaybe a,             -- pointed element of set
           osetList    :: Maybe [a],
           osetBounds  :: (MMaybe a, MMaybe a, MMaybe a, MMaybe a),
           osetProps   :: Properties_OSet,
           osetConstrs :: [Construction_OSet a],
           osetOpers   :: Operations_OSet a
          }
type Properties_OSet = [(Property_OSet,PropValue)]
data Property_OSet   =
     Finite | FullType | IsBaseSet | OrderIsTrivial | OrderIsTotal |
     OrderIsNoether | OrderIsArtin  -- more?
     deriving(Eq, Ord, Enum, Show)

data Construction_OSet a = Interval (Maybe a) Bool (Maybe a) Bool
                           --
                           -- Union [OSet a], Intersection [OSet a] ...?
type Operations_OSet a = ...   -- DUMMY, so far
```

Base set and subset

A value `o :: OSet a` describes a subset S = S(o) of the base set BS inside a type `a`.

For this section, we keep on with the denotations S, BS.

BS is defined, for example, by the sample element `<sample> = osetSample o :: a`.
And S is a subset of S' = S'(o) = `{x <- a | membership o 'r' x == True}`.

According to the type constructors involved in `<sample>`, `DoCon` puts individually, what BS is precisely presumed. See Sections (3.4 (DS), (SA), (BSE)), 3.5.2.

For some constructors, S = S', and `membership o 'r'` is really an algorithm solving the membership to S. For others, it is not, this is the matter of a given constructor: `Integer`, `Vector`, `Pol`, `ResidueI`, or other.

All the further possible base domains for `<sample>` (additive group, ring, ...) are considered as having the *support* BS. For example, for the base additive group, `(+), neg` are considered as restricted to the set BS.

`osetProps o` contains `(IsBaseSet, v)`,
`v` may be `Yes` (means S = BS), `No` (S $\neq$ BS), `Unknown`.

The letter `'O'` in `OSet` stands for "ordered partially".

`membership :: Char -> a -> Bool`

is the above predicate for S'(o). In the function `belongs = membership o`,
in `belongs mode x`,
`mode = 'r'` means to test the membership recursively, down all the constructors in `x`.
Example: for the `Set (a,b)` instance, `DoCon` puts

```
belongs 'r' (x,y) = bel1 'r' x  &&  bel2 'r' y
belongs _   _     = True,
```

where `bel1, bel2` are the corresponding predicates for `a`, `b`.

`osetCard :: InfUnn Z`

`osetCard o` is cardinality(S(o)) = `Infinity | UnknownV | Fin n`,
`n` a non-negative integer.

`osetPointed :: MMaybe a`

is any chosen element in S. It may be

- `Just (Just x)` — "this `x` belongs to S",
- `Just Nothing` — "S is empty",
- `Nothing` — "could not find any element in S.

`osetList :: Maybe [a]` may be

- `Just xs` — `xs` is a finite list of all the elements of S, free of repetition,
- `Nothing` — "could not provide such list".

`osetBounds` :: `(MMaybe a, MMaybe a, MMaybe a, MMaybe a)`

`= (lower, upper, infinum, supremum)`

These are the attributes of the partial ordering `compare_m` restricted to S. `infinum` and the membership predicate of S may give the minimum of S. Similar is maximum.

It holds `lower <= infinum,  upper >= supremum`
— when these data have definite values.

`lower =`

`Just (Just l)` — means `l` $\in$ S is the lower bound for S,

`Just Nothing` — such bound does not exist in S,

`Nothing` — could not find such bound in S.

Similar are `upper, infinum, supremum`.

`osetProps` :: `Properties_OSet`

is an *association list* with the `Property_OSet` key.
The pairs (or keys) in this list may be:

- `(IsBaseSet, Yes)` means S = BS — a base set.
- `FullType` — S = full set of the *type*.
- `OrderIsTrivial` — `(compare_m x y)==Nothing` for all `x /= y` in S.
- `OrderIsTotal` — `(compare_m x y) /= Nothing` for all `x, y` in S.
- `OrderIsNoether` — S has not any infinite increasing by `compare_m` sequence $x_1 < x_2 < \ldots$
- `OrderIsArtin` — S has not any infinite decreasing by `compare_m` sequence.

`osetConstrs` :: `[Construction_OSet a]`

is the list of algebraic constructions known to produce S.
This feature is reserved, not really used, so far.

Possible construction: `Interval l l_closed r r_closed`
expresses S = interval (l,r) in a partially ordered set.
This is for the case `(OrderIsTotal, Yes)`.

`l = Just l'` means `l'` is a lower bound for S,

`r = Just r'` — `r'` is an upper bound,

`l (r) = Nothing` — there is no lower (upper) bound.

`l_closed (r_closed) = True` means that the lower (upper) bound belongs to the interval.

`osetOpers` :: `Operations_OSet a`

Generally, `<dom>Opers` contains several additional operation descriptions for a domain — set, group, ... Such operation can be extracted by applying the `lookup` function.
For example,

```
lookup DimOverPrimeField $ subringOpers rR
```

may yield `DimOverPrimeField' (...)` or `Nothing`.

On `Constructions, Operations` parts of domain description:
in DoCon, they are DUMMY, except
`Interval` construction for `OSet`,
`DimOverPrimeField` operation for `Operation_Subring`,
`GenFactorizations` for `Construction_Ideal`,
`IdealRank` for `Operation_ideal`

Example 1: DoCon declares for the domain Char

```
instance Set Char
  where
  showsDomOf _ = ("Char"++)

  fromExpr _ (E (L "'") [] [L [c]]) = ([c], "")
  fromExpr _ e                        =
                     ([], "(fromExpr <Char> e):  wrong e = " ++ (shows e ""))
  compare_m x = Just . compare x

  baseSet _ dm = case  Map.lookup Set dm  of

    Just (D1Set o) -> (dm, o)
    _              -> (Map.insert Set (D1Set o) dm, o)
     where
     o = OSet {osetSample  = 'a',
               membership  = (\_ _-> True),
               osetCard    = Fin (n2-n1+1),
               osetPointed = Just (Just 'a'),
               osetList    = Just list,
               osetBounds  = (Just (Just minC), Just (Just maxC),
                              Just (Just minC), Just (Just maxC)
                             ),
               osetProps   = props,
               osetConstrs = [(Interval (Just minC) True (Just maxC) True)],
               osetOpers   = []
              }
     (minC, maxC) = (minBound, maxBound) :: (Char, Char)
     [n1  , n2  ] = map (toZ . fromEnum) [minC, maxC]
     list         = [minC .. maxC]
     props        = [(Finite,Yes),(FullType,Yes),(IsBaseSet,Yes),(OrderIsTrivial,No),
                       (OrderIsTotal,Yes),(OrderIsNoether,Yes),(OrderIsArtin,Yes)]
```

Example 2: zero ideal in a polynomial ring.
For the ring `P = Z[x]`, ideal `I` in `P`, any residue element in `P/I` is represented as `Rsi f (_, iD) dP`, where `iD` the bundle for the domain `I`. In particular, `iD` contains the subset term `sI`.

Suppose now that `I` $= \{0\}$. Then, `sI` has to describe the set $\{0\}$:

```
sI = OSet {osetSample = f :: P,           membership = (\_ g-> isZero g)...
           osetList   = Just [zeroS f],   ...
          }
```

Here the base set $\mathrm{BS(f)} =$ `P` — the whole polynomial ring. This is put by DoCon for the constructor `UPol` in `f = UPol _ _ "x" dZ`.

And S inside BS is defined by `(membership sI):  sI = {(zeroS f)}`.

A good way to form the `(iI,iD)` terms for the ideal description is to apply the `DoCon` function
`gensToIdeal` which starts with the ideal generator list and ring bundle `dP` and produces automatically `sI` and other parts of `iD`, `iI`. As in our case `I = {0}`, `gensToIdeal` may apply in its turn `listToSubset` to build the subset term `sI`.

Commonly, the domain descriptions are constructed by the `DoCon` library functions rather than "by hand".

10.3 Examples of using domain properties

```
f subset = let  ps = osetProps subset
           in
           (case  lookup Finite ps  of  Just Yes -> g
                                        Just _   -> h
                                        _        -> error (..."property skipped"...)
           )
           where  g = ...    h = ...
```

Also this can be scripted more shortly as

```
f subset = let ps  = osetProps subset
               fin = fromMaybe (error (.."property skipped"..) $ lookup Finite ps
           in  if fin==Yes then ... else ...
```

And often we program it like this:

```
               f subset = let  ps  = osetProps subset
                               fin = lookupProp Finite props
                          in   if  fin==Yes  then ... else ...
```

— see DPrelude.lookupProp. Consider also

```
      f1 :: Set a => a -> a
      f1 x = let {(_, xS) = baseSet x Map.empty;   ps = osetProps xS}
             in  ... like above in  f.
```

Here the sample `x` defines a base set, and `f1` analyses the attributes of this set. For several most usable properties `DoCon` provides shorter access. Say,

```
isFiniteSet :: OSet a -> PropValue
```

Such functions are defined simply by composing the functions `osetProps` and `lookup`, and the programmer can arrange the same for the properties that one consideres as the most usable.

Semigroup, Group, and other categories, treat the properties similarly. Only the set of the property names is different.

10.4 Usable functions for subset

```
isBaseSet, isFiniteSet :: OSet a -> PropValue

isBaseSet   = fromMaybe Unknown . lookup IsBaseSet . osetProps
isFiniteSet = ...
intervalFromSet :: OSet a -> Maybe (Construction_OSet a)
                                                  -- extract first interval construction
card :: Set a => a -> InfUnn Z  -- make set from sample and extract cardinality
card a = osetCard s  where  (_, s) = baseSet a Map.empty

                     -- example:  card 'a' == card '0' == Fin 256,  card 2 == Infinity

ofFiniteSet :: Set a => a -> PropValue
ofFiniteSet            a =  isFiniteSet s  where (_, s) = baseSet a Map.empty

isoOSet :: (a -> b) -> (b -> a) -> OSet a -> OSet b
           -- f        f_inv
   -- For given oset S on type `a', the maps  f: a -> b and its inverse f_inv
   -- produce an isomorphic copy of S on `b'.  f  must be injective.

listToSubset :: Set a =>  [a] -> [Construction_OSet a] -> OSet a -> OSet a
                        -- xs    conss
  -- Make a Listed Proper subset in the base set.
  -- Only those elements of  xs  remain which belong to the Base.
  -- conss  overwrites the construction list (is often put []).
  -- See implementation in  Set_.hs
```

10.5 Printing domains

This section is on the `showsDomOf` operation (see Section 10.1).

Example

The inversion `inv` in a semigroup applies `inv_m` and either extracts the result from it or breaks the program with the error message:

```
----------------------------------------------------------------- BAD design
invBAD :: MulSemigroup a => a -> a
invBAD                    x = fromMaybe (error $ ("inv "++) $ show x) $ inv_m x

-- Here the error message provides too few information.  For example,

  invBAD (2::Z)      -> "Fail:  (inv x)  failed ... inv 2"
  invBAD (fromi r 2) -> "...                        inv 2"
  invBAD (fromi f 2) -> "...                        inv 2"
                                    where
                                    r = ... describes the ring Z/(4)
                                    f = ...                    Z[x]
-----------------------------------------------------------------
```

The second and the third lines try to inverse the image of 2 in the rings `Z/(4), Z[x,y]` respectively (we omit the program details). The element 2 of these rings represents internally as `Rse 2 <..> dZ` and `Pol [(2,..)] _ _ _ dZ` respectively. But in the output string the `shows` operation shows it still as "2". `DoCon` defines `shows` so for the same reason as why the mathematicians prefer to write `(2, x*y + 1)` rather than `(2*x^0*y^0, x*y + 1)`.

But then, the output has to print the short domain description — in order to give the user a more definite idea of where the computation was taking place:

```
inv :: MulSemigroup a => a -> a                          -- Real program
inv x = fromMaybe (error msg) $ inv_m x
                     where
                     msg = ("inv x  failed,"++) $ showsWithDom x "x" "" "\n"
```

Here `showsWithDom` prints `x` and applies `(showsDomOf x)` too.

Examples.

For `r <- Z/(4)`, the program `inv (fromi r 2)`
(map 2 into `Z/(4)` and find there the inverse) reports

`" Fail: inv x  failed,    x = 2  <-  (Z/<4>) "`

Here `" <- (Z/<4>)"` means "belongs to domain `Z/Ideal(4)`".

For the residue ring `R = (Z/(3))[x,y] / (x-y, y^2, x^3, x*y)`,
`inv (0 <- R)` would report something like

```
Fail: inv x  failed,
x = (0)
<-  ((Z/<3>)["x","y"] / I<(x-y)... a_3>)
```

Now, it is clear, what is `showsDomOf`, how to define it for various constructors, and how to apply `showsWithDom`.

10.5.1 Domain output syntax

Its denotations follow the mathematical tradition and often differ from the corresponding data constructor names.

It uses the additional '`<`' and '{' parentheses — to help visual parsing of complex domains.

`List` is used instead of `[,]`,

'`[...]`' is used for the polynomial variable listing.

`<...>` denotes the ideal in an Euclidean ring represented by the `Rse` constructor.

`I<...>` denotes the generic ideal (`Rsi` constructor).

`(A x B)` — direct product (constructor `(,)`).

Example: it may occur `(3, (2,2*y))` $\in$ `"(Z x (Z x Z[y]))"`

`(List D)`
— domain of lists over domain described by a string `D` (constructor `[,]`).
Examples: `"(List Z)"`, `"(List (Z/<2>))"`

`{Vec n D}`
— vector domain `D` – `n`-times (constructor `Vec`).
Examples: 1. `(Vec [1,4,0] :: Vector Z) <-  "{Vec 3 Z}"` ;
2. `"{Vec 4 Z[x1,x2]}"` shows the fourth direct product power of `Z[x1,x2]`.

`(L n D)` — matrix `n` $\times$ `n` domain over `D` (constructor `SqMt`)

`(L n m D)` — matrix `n` x `m` domain over `D` (constructor `Mt`).
Examples:

```
(Mt   [[1,2],[3,4]] dZ  :: Matrix Z)       <- "(L 2 2 Z)",
(SqMt [[1,2],[3,4]] dZ  :: SquareMatrix Z) <- "(L 2 Z)"
```

`(Fr D)` — fraction domain (constructor `:/`).
Example:

(t:/3 :: Fraction (UPol Z)) <- "(Fr Z[t])"

(D/<b>)
— residue ring of Euclidean ring D by Ideal(b) (constructor Rse).
Examples: (Rse 0 iI dZ :: ResidueE Z) <- "(Z/<4>)", if iI = <4>.
(Rse x^2 iI dQ :: ResidueE (UPol (Fraction Z))) <-
" ((Fr Z)[x] / < -3*x^3 +2*x^2 - 1 >) ",
where
dQ describes Fraction Z, iI describes Ideal(-3*x^3 +2*x^2 -1).

(D/I<gs>)
— (generic) residue ring of D by I = Ideal<gs> (constructor Rsi)
— see Section 44.
If a finite generator list gs = $[a_1, \ldots, a_n]$ is *not* detected for I, then gs prints as "?",
Otherwise, it prints as a_1-*expression Tail*,
where *Tail* = "" for n = 1, "... a_n" otherwise.
Examples:
Z[x,y]/(x^2 -y)   prints as "(Z[x,y]/I<x^2 - y>)"
Z[x,y]/(x^2, x^3-y, x*y-1) prints as "(Z[x,y]/I<x^2, ..., a_3>)"

{D/Subgroup<gs>}
— residue group of D by a normal subgroup H (constructor Rsg) — see Section 43.
It prints similar as D/I<gs>, only
(a) it is embraced with { }, (b) Subgroup replaces 'I',
(c) gs are the subgroup generators.

Examples:
the quotient group Z/(2) prints as "{Z/Subgroup<2>}",
the quotient additive group (Fraction Z)/<1:/2,1:/3> may be printed as
"{(Fr Z)/Subgroup<1:/2, ..., a_2>}"

S(n)
— permutation group on n elements (constructor Pm).
Example: (Pm [-2,6,4] :: Permutation) $\in$ "S(3)"

D[variables]
— polynomial domain over D (constructors UPol, Pol, RPol, RPol').
For the univariate polynomial domain (UPol) and for the 'usual' multivariate polynomial (Pol), variables is the list as usual. For example,
"Z[x]", "Z[y2]" may mean (UPol Z) or (Pol Z) representation,
"Z[t12,r31]" means only (Pol Z) data.

For the r-polynomial domain (recursively represented polynomial),

```
variables = prefix ranges
```

describes the set of variables. See Sections 40.2, 40.3.

For example, `" Z[vv [(1,3),(0,2)]] "` denotes the r-polynomial domain `P` over `Z` with the variable set `{vv_i_j | 1 <= i <= 3, 0 <= j <= 2}`,
`D[]` denotes the `RPol'`-polynomial domain — see Section 40.3.

`{EPol variables D}`

— E-polynomial domain over `D[variables]` (constructor `EPol`).

This is certain representation for the vectors over polynomials — see Section 39.1.2.

`Example:` `"{EPol [x,y] Z}"` denotes a free module $P \oplus P \oplus \ldots$ over $P = Z[x, y]$, the vectors represented as e-pols.

`{VecPol variables D}`

— `VecPol` representation for the module $P \oplus P \oplus \ldots$ over $P =$ `D[variables]` (constructor `VecPol`) — see Section 39.1.1.

Example: `"{VecPol [x,y] Z}"` denotes the module $P \oplus P \oplus \ldots$ over $P = Z[x, y]$.

`{SymPol D}`

— symmetric function domain over `D` (constructor `SymPol`) — see Section 45.3.

Example:

$$2m_{[3]} = 2 \cdot \sum_{i=1}^{\infty} x_i^3$$

(in its appropriate `SymPol` representation) belongs to `"SymPol Z"`

11 OrderedSet

```
class (Ord a, Set a) => OrderedSet a
  -- Presumed:
  -- on the  base set,  compare_m
  -- possesses  (OrderIsTotal, Yes)  and agrees with ‘compare’:
  --                                 (compare_m x y) == (Just (compare x y))
```

Example:

```
f :: OrderedSet a => [a] -> a
f (x:y:xs) = if  x < y  then  x
             else             if  fromJust $ compare_m y x ...
```

uses both `(<)` (defined via `compare`) and `compare_m` and applies `fromJust` — knowing that `compare_m` yields here only `(Just _)`.

‘`OrderedSet a =>`’ says here that `f` deals with the elements of the base set — on which `compare_m` is a total ordering.

12 Semigroup

See first Secrtions 10.1, 10.2.
See the module `SetGroup` and `Subsemigroup` data in the module `Categs`.

12.1 Subsemigroup term

```
data Subsemigroup a =
     Subsemigroup {subsmgType    :: AddOrMul,       -- operation name
                   subsmgUnity   :: MMaybe a,
                   subsmgGens    :: Maybe [a],
                   subsmgProps   :: Properties_Subsemigroup,
                   subsmgConstrs :: [Construction_Subsemigroup a],
                   subsmgOpers   :: Operations_Subsemigroup a
                  }
data AddOrMul = Add | Mul  deriving (Eq, Show, Ord, Enum)

type Properties_Subsemigroup = [(Property_Subsemigroup, PropValue)]
data Property_Subsemigroup   =
     Commutative | IsCyclicSemigroup | IsGroup | IsMaxSubsemigroup
     | IsOrderedSubsemigroup  deriving (Eq, Ord, Enum, Show)

data Construction_Subsemigroup a = ... DUMMY
type Operations_Subsemigroup   a = ... DUMMY
data OpName_Subsemigroup         = ... DUMMY
data Operation_Subsemigroup a    = ... DUMMY
```

A subsemigroup H is considered relatively to a base semigroup G.
See below the semigroup categories. So, to make a true `Subsemigroup`, we need first to have

- an instance of the `Set` category, with its base set BS,
- an instance of the `Add(Mul)Semigroup` category, with its base semigroup BH,
- a subset S of BS closed under the operation `add` (`mul`).

Together with the above instances, a bundle dH containing the terms sS, sH for above S, H respectively, is a complete representation of a `Subsemigroup`.

Similar construction principle applies to `Subgroup`, `Subring`, `Ideal`, and so on. In particular, the attributes usually related to subsemigroup may sometimes be accessible in other parts of the bundle dH. For example, the sample element and cardinality reside in the subset term in dH.

<u>`subsmgType ::  AddOrMul`</u> =

`Add` means a semigroup by operation called `add (+)`,

`Mul` ... by `mul (*)`.
`Add` is for the commutative case only.

<u>`subsmgUnity :: MMaybe a`</u> =

`Just (Just z)` means `z` is an unity for H, both left and right,
`Just Nothing` — no such unity exist in H,
`Nothing` — could not find unity in H.
For the additive case, 'unity' means zero.

<u>`subsmgGens :: Maybe [a]`</u> =

`Just gs` means `gs` is a finite generator list for H,
`Nothing` — could not provide such list.

<u>`subsmgProps :: Properties`</u>`_Subsemigroup`

is the association list for the properties of H.
The meaning of the properties listed in `Property_Subsemigroup` are evident.
For example, `(IsGroup, Yes)` means `add` restricted to H satisfies the Group laws.
`IsOrderedSubsemigroup` means that
(1) H is commutative,
(2) `compare_m` (of `Set`) is a total ordering on set(H) agreed with `add (mul)`, `zero_m` (`unity_m`).

12.2 AddSemigroup category

```
class Set a => AddSemigroup a
 where
 baseAddSemigroup :: a -> Domains1 a -> (Domains1 a, Subsemigroup a)

 add     :: a -> a -> a
 zero_m  :: a -> Maybe a      -- sa
 neg_m   :: a -> Maybe a
 sub_m   :: a -> a -> Maybe a
 times_m :: a -> Z -> Maybe a

 zero_m x = let  sH = snd $ baseAddSemigroup x Map.empty
            in
            case  subsmgUnity sH  of  Just (Just z) -> Just z
                                      _             -> Nothing
 sub_m x = maybe Nothing (Just . add x) . neg_m
 times_m = timesbin                    -- default definition via 'add'
```

Presumed:
`add` is associative, commutative, agreed with `(==)`, `zero_m`, `neg_m`.

For example: if for some `x` `zero_m x = Just z`,
then for all `y` either `neg_m y = Nothing` or
`neg_m y == Just y'` and `add y y' == z`.

The semigroup class operations relate to the domain of the *base semigroup* bH in the type `'a'`.

Group and Semigroup categories

`AddGroup` category declares only `baseAddGroup` operation.
The mathematicians usually relate to `AddGroup` the operations `0`, `neg(ate)`. But `DoCon` leaves them to `AddSemigroup` — in the form of `zero_m`, `neg_m`. They are partial: may return `Nothing`, if H is not a group.

Similar approach is applied to `Ring` and other categories.
There are also "polymorphic" operations (not from the class). Thus,

```
neg x = case  neg_m x  of  Just x -> x
                           _      -> error (... "neg  failed" ...)
```

never cause a break for H, if H occurs a `Group`, but otherwise, may cause it.

```
class (OrderedSet a, AddSemigroup a) => OrderedAddSemigroup a

   -- Presumed:  base AddSemigroup  contains  (IsOrderedSubsemigroup, Yes)
   --            - see Subsemigroup constructor.
```

12.3 Several usable functions for semigroup

```
zeroS :: AddSemigroup a => a -> a         -- zero of a semigroup
                      --   sa

zeroS x = fromMaybe (error msg) $ zero_m x
            where
            msg = ("(zero x)  failed,"++) $ showsWithDom x "x" "" ""

isZero :: AddSemigroup a => a -> Bool
isZero                      a =  case zero_m a of  Just z -> a==z
                                                   _      -> False
neg :: AddSemigroup a => a -> a
neg                      x =  fromMaybe (error ...) $ neg_m x

sub :: AddSemigroup a => a -> a -> a
sub                      x    y =  fromMaybe (error ...) $ sub_m x y
```

```
times :: AddSemigroup a => a -> Z -> a
times                     x    n =  fromMaybe (error ...) $ times_m x n
```

The above functions yield Break + message when fail to find zero or the opposite element in a semigroup.

```
isGroup, isCommutativeSmg :: Subsemigroup a -> PropValue

isGroup          = lookupProp IsGroup     . subsmgProps
isCommutativeSmg = lookupProp Commutative . subsmgProps

isoSemigroup :: (a->b) -> (b->a) -> Subsemigroup a -> Subsemigroup b

  -- given a  Subsemigroup H with the base set X on a type 'a',
  --   a map  f: a -> b  injective on X,  f_inv  inverse to  f  on X,
  -- produce  Subsemigroup H'  on the base set f(X),  such that
  -- f_restrictedTo_X  is isomorphism between H and H'

trivialSubsemigroup :: Subsemigroup a -> Subsemigroup a
  --
  -- a trivial subsemigroup ({0} or {1}) inside a non-trivial base monoid)
```

12.4 Multiplicative semigroup

It is similar to AddSemigroup, only it may be non-commutative.

```
class Set a => MulSemigroup a
  where
  baseMulSemigroup :: a -> Domains1 a -> (Domains1 a, Subsemigroup a)

  mul       :: a -> a -> a           -- multiplication
  unity_m   :: a -> Maybe a          -- sa
  inv_m     :: a -> Maybe a          -- inversion
  divide_m  :: a -> a -> Maybe a
  divide_m2 :: a -> a -> (Maybe a, Maybe a, Maybe (a,a))
  power_m   :: a -> Z -> Maybe a
  root      :: Z -> a -> MMaybe a    -- root of n-th degree

  unity_m x = -- sa
            case  subsmgUnity $ snd $ baseMulSemigroup x Map.empty  of  Just u -> u
                                                                    _      -> Nothing
  inv_m  x = maybe Nothing (\un -> divide_m un x) $ unity_m x
  power_m  = powerbin                            -- binary method to power via 'mul'
```

Presumed: mul is associative, agreed with (==), unity_m, inv_m. So it makes a base multiplicative semigroup BH.

`divide (_m)`
is for the left-side quotient: for solving for `x` of the equation `x*a = b` in a semigroup. The solution may occur not unique. Returned is some `x`, not necessarily the 'best' one. For a `Ring`, we can only rely (in the general case) on that `x-y` is a *z*ero divisor for any solutions `x`, `y`. Thus, `x` is unique if a ring has not zero divisors.

Similar is `inv(_m)`.

`divide (_m2)`
generalizes `divide_m`. It yields left, right, and bi-sided maybe-quotient.

Example: in `FreeMonoid[a,b,c]`

```
divide_m2 abccb cb =  (Just abc, Nothing, Just (abc, []))
divide_m2 abccb cc =  (Nothing,  Nothing, Just (ab,b)   )
```

`root n x` may yield `Just (Just r) | Just Nothing | Nothing`.
This means respectively
"this `r` from BH is the root of n-th degree of `x`",
"such a root does not exist in BH",
"could not find such a root in BH".

Some functions related to `MulSemigroup`:

```
unity :: MulSemigroup a => a -> a    -- sa
unity                     x =  fromMaybe (error msg) $ unity_m x
                 where
                 msg = ("unity x   failed,"++) $ showsWithDom x "x" "" "\n"

inv :: MulSemigroup a => a -> a
inv                     x =  fromMaybe (error ...) $ inv_m x

divide :: MulSemigroup a => a -> a -> a
divide                     x    y =  fromMaybe (error ...) $ divide_m x y

invertible :: MulSemigroup a => a -> Bool
invertible = isJust . inv_m

divides :: MulSemigroup a => a -> a -> Bool
divides                     x    y =  isJust $ divide_m y x

power :: MulSemigroup a => a -> Z -> a
power                     x    n =  fromMaybe (error ...) $ power_m x n
```

Use `power (_m)` rather than '^':
the former is more generic, the latter is of the Haskell library.

13 Monoid

```
class AddSemigroup a => AddMonoid a

class (OrderedAddSemigroup a, AddMonoid a) => OrderedAddMonoid a

class MulSemigroup a => MulMonoid a
class MulSemigroup a => OrderedMulSemigroup a

class (OrderedMulSemigroup a, MulMonoid a) => OrderedMulMonoid a
```

For a dynamic parametric domain *D(p)* inside a type `a`, declaring

```
instance ... => AddMonoid D(p)
```

means that for `x :: a` containing appropriate value of `p` a base semigroup of `x` has zero. And in such case `zeroS x` gives this zero element, and `zero_m` becomes unnecessary.

14 Group

This category is implemented in the module `SetGroup.hs`.
The `Subgroup` expression is defined in the modle `Categs.hs`.

14.1 AddGroup, MulGroup categories

```
class AddMonoid a => AddGroup a
  where
  baseAddGroup :: a -> Domains1 a -> (Domains1 a, Subgroup a)

  -- Presumed:  zeroS, add, neg  of base semigroup BH satisfy the group axioms.
  -- This condition corresponds to  (IsGroup, Yes)  contained in the
  -- property list of BH.

class (AddGroup a, OrderedAddMonoid a) => OrderedAddGroup a
               --
               -- base AddGroup  contains  (IsOrderedSubgroup, Yes)
               -- - see the Subgroup constructor.

class (MulMonoid a) => MulGroup a
  where
  baseMulGroup :: a -> Domains1 a -> (Domains1 a, Subgroup a)

class (OrderedMulMonoid a, MulGroup a) => OrderedMulGroup a
```

14.2 Subgroup term

```
  data  {-Add(Mul)Group a=>-}  Subgroup a =

       Subgroup {subgrType    :: AddOrMul,
                 subgrGens    :: Maybe [a],
                 subgrCanonic :: Maybe (a -> a),
                 subgrProps   :: Properties_Subgroup,
                 subgrConstrs :: [Construction_Subgroup a],
                 subgrOpers   :: Operations_Subgroup a
                }
  type Properties_Subgroup = [(Property_Subgroup, PropValue)]
  data Property_Subgroup   = IsCyclicGroup | IsPrimeGroup | IsNormalSubgroup
                           | IsMaxSubgroup | IsOrderedSubgroup
                           deriving (Eq, Ord, Enum, Show)
data Construction_Subgroup a = ... DUMMY
type Operations_Subgroup a   = ... DUMMY
data OpName_Subgroup         = ... DUMMY
data Operation_Subgroup a    = ... DUMMY
```

This expresses a `Subgroup` H in a base group `G`. Unity, zero are obtained by applying of `unity x`, `zeroS x` respectively.

<u>`subgrType :: AddOrMul`</u>
— additive — multiplicative is like in `Subsemigroup`.

<u>`subgrGens :: Maybe [a]`</u>

`Just gs` means `gs` is a finite list of the group generators for H.
`Nothing` means fail to provide such a list.

<u>`subgrCanonic :: Maybe (a -> a)`</u> =

- `Just cn` means `cn: G -> G` is a canonical map for the congruence classes `x*H`:
  `cn(x)-x` ∈ `H` `And` `x-y` ∈ `H` ⇔ `cn(x) = cn(y)`
  `And` `cn(zero+H) = zero`
  — similar `*-` denotations are for the `Mul` case.
- `Nothing` means fail to provide such a map.

<u>`subgrProps :: Properties_Subgroup`</u>

The property names in this list have the usual algebraic meaning.
And `(IsOrderedSubgroup, Yes)` means that
(1) `Subsemigroup` for H has `(IsOrderedSubsemigroup, Yes)`,
(2) `compare_m` agrees with `neg` — in `Add` case, with `mul` — in `Mul` case.

14.3 Several usable functions for subgroup

```
absValue :: AddGroup a => a -> a     -- this is correct for (IsOrderedGroup, Yes)
absValue                  x =  if  less_m x (zeroS x)  then  neg x  else  x

trivialSubgroup :: a -> Subgroup a -> Subgroup a
                                -- make trivial subgroup in non-trivial base group
trivialSubgroup zeroOrUnity gG =
  Subgroup
     {subgrType    = subgrType gG,  subgrGens  = Just [zeroOrUnity],
      subgrCanonic = Just id,       subgrProps = props,
      subgrConstrs = [],            subgrOpers = []
     }
    where
    props = [(IsCyclicGroup, Yes), (IsPrimeGroup, No), (IsNormalSubgroup, Yes),
             (IsMaxSubgroup, No), (IsOrderedSubgroup, isOrderedGroup gG)
            ]

isoGroup:: (a -> b) -> (b -> a) -> Subgroup a -> Subgroup b

  -- Given a   Subgroup G with the base set X on a type 'a',
  --    a map  f: a -> b  injective on X,  f_inv  inverse to  f  on X,
  -- produce the  Subgroup G' on the base set f(X),  such that
  -- f_restrictedTo_X  is an isomorphism between G and G'.
```

15 Ring

This category is exported from the module `RingModule`,
implemented in the `Ring*` modules, `Subring` — in the module `Categs`.

15.1 Ring category

```
class (AddGroup a, MulSemigroup a, Num a, Fractional a) => Ring a
  where
  baseRing :: a -> Domains1 a -> (Domains1 a, Subring a)

  fromi_m  :: a -> Z -> Maybe a   -- standard homomorphism from integer
            --sa

  fromi_m x n = maybe Nothing (\u-> times_m u n) $ unity_m x
```

Presumed:
(1) non-zero `baseAddSemigroup` A and `baseMulSemigroup` H have the same base set BS,
(2) `add, mul` obey the ring laws on BS,

(3) `add, mul, divide` agree with the `Haskell` intances of `Num, Fractional`:

`(+) = add, (*) = mul, (/) = divide`.

15.2 Subring term

```
data Subring a = Subring {subringChar    :: Maybe Natural,
                          subringGens    :: Maybe [a],
                          subringProps   :: Properties_Subring,
                          subringConstrs :: [Construction_Subring a],
                          subringOpers   :: Operations_Subring a
                         }
type Properties_Subring     = [(Property_Subring, PropValue)]
data Construction_Subring a = ... DUMMY
type Operations_Subring a   = [(OpName_Subring, Operation_Subring a)]
data OpName_Subring         = WithPrimeField -- | ...
                                        deriving(Eq, Ord, Enum, Show)
```

This describes a subring R of a base ring bR.

<u>`subringChar :: Maybe Natural`</u>
is the characteristic of a ring. It may be
`Just n` — which means characteristic `n` ≥ 0,
`Nothing` — unknown characteristic.

<u>`subringGens :: Maybe [a]`</u> =
`Just gs` — means `gs` is a finite ring generator list for R,
`Nothing` — means `DoCon` could not provide such generators.
Example:
for a bundle `dP` of domain `Z[x]`, `subringGens dP` may be `Just [1,x]`,
and for `Rational[x]`, it has to be `Nothing`. Because, for example, `1/2` cannot be generated from `1` and `x`.

<u>`subringOpers :: Operations`</u>`_Subring a`

```
data OpName_Subring = WithPrimeField  deriving(Eq, Ord, Enum, Show)
```

— so far, it has one member.
Example of application:
DoCon uses it in factorization programs for the polynomials over arbitrary finite field, when finds a primitive generator for a field extension, and so on.

```
data Operation_Subring a =
     WithPrimeField' {frobenius            :: (a -> a, a -> MMaybe a),
                      dimOverPrime         :: InfUnn Z,
                      primeFieldToZ        :: a -> Z,
                      primeFieldToRational :: a -> Fraction Z,
                      primitiveOverPrime   :: ([a], [UMon a], a-> [UMon a])
                     }
     -- | ...
```

It has sense in the subring term for R only when the `Integer` ring image

`Z' = F(Z)`, where `F(n) = times unity n`,

extends to the field inside R. This field is called a prime field PF.

Thus, for a field R of $\mathrm{char}(R) = 0$, PF is a copy of the field `Rational`. The case $\mathrm{char} = p > 0$ makes $PF = \mathrm{GaloisField}(p)$. In both cases R is a vector space over PF.

<u>`frobenius :: (a -> a, a -> MMaybe a)`</u>

is `undefined` for `p = char R = 0`, otherwise, it is `(pc, pcInv)`.

`pc = (^p) :: a -> a` is a ring homomorphism, `pcInv` its inverse. `pcInv` has the same format as `MulSemigroup.root`, `(^p)` may be not invertible for some domains `a`.

<u>`dimOverPrime :: InfUnn Z`</u> is the dimension over a prime field.

`Examples:`

`Z` has to contain `UnknownV` in its `dimOverPrime`,
`Fraction Z` and `Z/(5)` have `Fin 1`,
`(Z/(5))[x] /(x^2+2)` has `Fin 2`, `(Fraction Z)[x]` has `Infinity`.

<u>`primeFieldToZ :: a -> Z`</u>

is `undefined` for `p = char = 0`.

For `p > 0`, the restriction of `primeFieldToZ` to PF has to be the inverse map for `fromi` on `[0 .. p-1]`.

<u>`primeFieldToRational :: a -> Fraction Z`</u>

is `undefined` for `p = char > 0`.

For `p = 0`, it must be the isomorphism: PF $\rightarrow$ Rational numbers.

<u>`primitiveOverPrime :: ([a], [UMon a], a -> [UMon a])`</u> `=`

`(powers, mp, toPol)`

To skip this 'primitive element' construct, put `powers = []`.

Otherwise, `powers = [g^i | i <- [1 ..]]`,

with `g` some primitive generator for a ring `a` over PF,

`mp` minimal polynomial for `g` over PF, given as a sparse list of monomials (`[]` for the dummy), with the coefficients from PF,
`toPol :: a -> [UMon a]` is either `const []` (for dummy) or represents canonically each element of `a` as the value `f(g)` of polynomial `f` over PF.

Here `g = head powers` is a primitive generator, `deg f < deg mp`, any element of `a` is `f(g)` for some polynomial `f` from PF[x], the representing polynomial `f(g)` in `toPol` is given by a sparse list of monomials over `PF`.

Finding primitive element for field extension

There exist clever methods for this — for example, for the case of a finite field. But DoCon finds a primitive `g` by brute force. This can be improved in future. But even as it is now, this is not so bad. Because starting from $x, x+1, \ldots,$ a primitive element `g` is found fast enough, on average. After this, `g` is set in the ring term, and it becomes ready for the repeated use.

This is exploited by the factorization function for $GF(q)[x_1, \ldots, x_n]$, $q = p^m$.

Given a finite field tower K (prime) -- F -- ... -- E,
`DoCon` finds a generator `g` for E over K and factors the given polynomial over E by reducing to the case of a canonic finite field K[g]/minimalPolynomial(g).

15.3 Subring properties

```
data Property_Subring =
     IsField | HasZeroDiv | HasNilp | IsPrimaryRing | Factorial
     | PIR | IsOrderedRing | IsRealField | IsGradedRing
     deriving (Eq, Ord, Enum, Show)
                          -- IsGradedRing  refers to Operations_Subring
```

Below, we describe the properties meaning only for a *base ring* R.

`PIR` === R is an abstract principal ideal ring.
`HasZeroDiv` === R has a zero divisor.
`HasNilp` === R has a nilpotent.
`IsPrimaryRing` === each zero divisor is a nilpotent.
`Factorial` === abstract unique-factorization ring.

`IsGradedRing` === the triplet `(Grading' cp weight forms)` from `operations` satisfies the grading laws for the `weight: R -> Z^n`, `Z^n` ordered by this `cp`,
`forms: R -> [R]` yields the list of homogeneous forms.

`IsRealField` === `IsField` And -1 is not a sum of squares in R.

`IsOrderedRing` ===
`Commutative`, with unity $\neq$ zero, additive subgroup is ordered by `compare_m`,

`compare_m` yields the positive-negative partition $R = \{0\} \cup P \cup (-P)$ satisfying $x, y \in P \implies x + y,\ x * y \in P$

Lemma. IsOrderedRing(R) $\implies$ R has not zero divisors,
and there is an induced 'Ordered' structure on Fraction(R).

DoCon pays attention mostly to the case of a base ring, considering other case as exotic. Though, the proper subring appears when the ideal bundle is formed.

15.4 Several usable functions for ring

```
fromi :: Ring a => a -> Z -> a                    -- map from integer by sample
fromi              x    n =  fromMaybe (error msg) $ fromi_m x n
  where
  msg = "\nfromi x " ++ (shows n $ showsWithDom x "x" "" "\n")

char :: Ring a => a -> Maybe Natural
char              a =  subringChar $ snd $ baseRing a Map.empty
                            -- characteristic of ring defined by given sample

property_Subring_list = [IsField ..]

dimOverPrimeField :: Subring a -> InfUnn Z
dimOverPrimeField rR = case  lookup DimOverPrimeField$ subringOpers rR
                       of
                       Nothing                      -> UnknownV
                       Just (DimOverPrimeField' v) -> v
    -- examples:
    -- Z, Z[x] --> UnknownV,  Z/(3) --> Fin 1,   (Z/(3))[x] --> Infinity

isField, isOrderedRing :: Subring a -> PropValue

isPrimeIfField :: Subring a -> PropValue  -- find, whether R is a prime field
                                          -- (correct to apply only for a field)

zeroSubring :: a -> Subring a -> Subring a   -- zero subring in a Non-zero base ring
zeroSubring zr _ =
     Subring {subringChar  = Just 0,              subringGens    = Just [zr],
              subringProps = props_Subring_zero,  subringConstrs = [],
              subringOpers = []}

multiplicity :: CommutativeRing a =>  a -> a -> (Z, a)
                                   -- x    y     m  q
  -- Multiplicity of  x  in  y  in a factorial ring - correct for (Factorial,Yes)
  -- q = y/(x^m).  x, y  non-zero,  x  must not be invertible.
```

```
multiplicity x y
  | isZero x || isZero y || isJust (inv_m x) = error ...
  | otherwise                                =
                      if  not $ divides x y  then  (0,y)  else  mlt x y 0
                  where
                  mlt x y m = maybe (m,y) (\q-> mlt x q (m+1)) $ divide_m y x


isoRing :: (a -> b) -> (b -> a) -> Subring a -> Subring b
           -- f        fInv        rA           rB
  -- Given a ring term  rA  with the base set X on a type 'a',
  --       a map  f: a -> b  injective on X,  f_inv inverse to f on X,
  -- produce the ring term  rB  on the base set f(X),  such that
  -- f_restrictedTo_X  is an isomorphism between these rings.
```

16 GCDRing

This category is exported from RingModule, implemented in the Ring* modules, GCDRingTerm — in the module Categs.

```
class Ring a => CommutativeRing a
                -- Presumed:  Commutative==Yes  for base multiplicative semigroup

class (CommutativeRing a, OrderedAddGroup a) => OrderedRing a
                                   -- Presumed: (IsOrderedRing, Yes) for the base ring

class (CommutativeRing a, MulMonoid a) => GCDRing a    -- presumed: (HasZeroDiv, No)
  where
  baseGCDRing  :: a -> Domains1 a -> (Domains1 a, GCDRingTerm a)
  canAssoc     :: a -> a                  -- canonical associated element
  canInv       :: a -> a                  -- canonical invertible factor
  gcD          :: [a] -> a                -- gcd, lcm  for a list
  lcM          :: [a] -> a
  hasSquare    :: a -> Bool               -- "has a multiple prime factor"
  toSquareFree :: a -> Factorization a

  canAssoc x = x/(canInv x)

  lcM []     = error "lcM [] \n"
  lcM [x]    = x
  lcM [x,y]  = y*( x/(gcD [x,y]) )
  lcM (x:xs) = lcM [x, lcM xs]

  -- the correctness of these operations depend on the
  -- WithCanAssoc, WithGCD  values - see below
```

```
data GCDRingTerm a = GCDRingTerm {gcdRingProps :: Properties_GCDRing
                                  -- this is all, so far
                                 } deriving (Show)
type Properties_GCDRing = [(Property_GCDRing, PropValue)]
data Property_GCDRing   = WithCanAssoc | WithGCD  deriving(Eq, Ord, Enum, Show)
```

`(WithCanAssoc, Yes)`
means that `canAssoc`, `canInv` are correct algorithms for the canonical associated element and the canonical invertible factor.

`(WithGCD, Yes)`
means `(Factorial, Yes)` and that `gcD` is a correct algorithm for the greatest common divisor.

`toSquareFree :: a -> Factorization a`

returns `[`$(a_1, 1)$ ... (a_m, m)`]`, where `(canAssoc a)` = $a_1^1 \cdot \ldots \cdot a_m^m$,
each a_i is square free and gcd $a_i\ a_j = 1$ for $i \neq j$,
invertible a_i are skipped, in particular, `[]` is returned for invertible `a`.

```
isGCDRing :: GCDRingTerm a -> PropValue
isGCDRing = lookupProp WithGCD . gcdRingProps
```

The property like `WithGCD` for the category `GCDRing` may look like a tautology. But recall of the parametric domains.

Example 1 `DoCon` contains the declaration

`instance GCDRing a => GCDRing (Pol a) where` ...

which looks quite natural.

Example 2 It also contains

```
instance EuclideanRing a => GCDRing (ResidueE a)
 where
 canInv, canAssoc, gcD   are defined trivially
```

— which may look strange.

Consider a residue ring `rR = a/(b)` of an Euclidean ring `a`. Evidently, this instance has sense only for `rR` being a field (that is for a prime `b`). And the aforementioned three operations have to be defined trivially.

For not a prime `b`, `rR = a/(b)` still fits the `GCDRing` class declaration.
But `isGCDRing rR --> No` would show that `rR` is not actually a gcd-Ring. This is why the above properties are introduced.

Similar approach is applied to other categories.

Examples with GCDRing:

```
canAssoc (2 :: Z) = 2;    canAssoc (-2) = 2;   canInv (-2) = -1;
canAssoc (2:/3 :: Fraction Z) =  1:/1
canAssoc ((2:/3)*x^2 + 1)     =  x^2 + (3:/2)   in  (Fraction Z)[x]
gcD [12,6,9] = 3
```

17 FactorizationRing

This category is exported from the module `RingModule`, implemented in `Ring*` modules, `FactrRingTerm` is from the module `Categs`.

```
type {-Ring a => -}  Factorization a = [(a, Natural)]

  -- Example:  8*49*3 = 2^3*7^2*3
  --           expresses as  [(2,3),(7,2),(3,1)]  :: Factorization Integer

class GCDRing a => FactorizationRing a    -- presumed: (WithGCD, Yes)
  where
  baseFactrRing :: a -> Domains1 a -> (Domains1 a, FactrRingTerm a)
  isPrime       :: a -> Bool
  factor        :: a -> Factorization a
  primes        :: a -> [a]               -- sa

  isPrime x = case  factor x  of  [(_,1)] -> True
                                  _       -> False

data FactrRingTerm a =
     FactrRingTerm {factrRingProps :: Properties_FactrRing  -- this is all, so far
                   } deriving (Show)

type Properties_FactrRing = [(Property_FactrRing, PropValue)]
data Property_FactrRing   = WithIsPrime | WithFactor | WithPrimeList
                            deriving (Eq, Ord, Enum, Show)

-- (WithIsPrime  , Yes) means  isPrime    is the correct primality test
-- (WithFactor   , Yes)        factor     is the correct factorization
-- (WithPrimeList, Yes)        (primes _) is the correct list of primes

isRingWithFactor :: FactrRingTerm a -> PropValue
isRingWithFactor = lookupProp WithFactor . factrRingProps
```

`primes` *sample*
is some infinite list of primes p_i, free of repetitions, each p_i in `canAssoc` form;
it returns `[]` for the fail.

Examples.

(1) `primes _ :: [Natural] = [2, 3, 5, 7, 11, ...]`.

(2) For any `f` from `(Integer/(3))[x]`

`primes f = [x, x+1, x+2, x^2+1, x^2+x+2, ..., x^3+2*x+2, ...]`

See also `demotest/T_pfactor.hs` for the polynomial factorization examples.

17.1 Several operations with factorizations

```
unfactor :: MulSemigroup a => Factorization a -> a
                                        -- example:  [(a,1),(b,2)] -> a*b^2
unfactor []        = error "unfactor []\n"
unfactor [(a,k)]   = power a k
unfactor ((a,k):f) = mul (power a k) $ unfactor f

isPrimeFactrz, isPrimaryFactrz, isSquareFreeFactrz :: Factorization a -> Bool
isPrimaryFactrz [_] = True
isPrimaryFactrz _   = False

isPrimeFactrz [(_, 1)] = True
isPrimeFactrz _        = False

isSquareFreeFactrz f = (not $ null f) && all ((==1) . snd) f

factrzDif :: Eq a => Factorization a -> Factorization a -> Maybe (Factorization a)
  --
  -- Difference of factorizations.  The order of factors immaterial.
  -- Examples:  [(a,1),(b,2)] [(a,1),(b,2)] -> Just []
  --            [(a,1),(b,2)] [(b,1)]       -> Just [(a,1),(b,1)]
  --            [(a,1),(b,2)] [(b,3)]       -> Nothing

eqFactrz :: Eq a => Factorization a -> Factorization a -> Bool
                              -- Equality. The order of factors is immaterial

gatherFactrz :: Eq a => Factorization a -> Factorization a
  --
  -- Bring to true factorization by joining the repeated factors.
  -- Example:  [(f,2),(g,1),(f,3)] -> [(f,5),(g,1)]
```

18 Syzygy solvable ring

This category is exported from `RingModule`, implemented in `Ring*` modules, `LinSolvRingTerm` — in module `Categs`.

```
class (CommutativeRing a, MulMonoid a) => LinSolvRing a
  where
  baseLinSolvRing :: a -> Domains1 a -> (Domains1 a, LinSolvRingTerm a)

  gxBasis :: [a] -> ([a], [[a]])
           --xs      gs   mt

  moduloBasis :: String -> [a] -> a -> (a,[a])
  syzygyGens  :: String -> [a] -> [[a]]
```

This means a `Ring` with the *solvable linear equations and solvable ideal inclusion* (see first Section 49).

`DoCon` decides that these operations make sense and are solvable when the corresponding `LinSolvRingTerm` data contains the pair `(IsGxRing, Yes)` in its property list.

18.1 gxBasis

`gxBasis xs` yields `(gs, mt)`,
where `gs` is a gx-basis for `I = Ideal(xs) = Ideal(gs)`, that is
`gs` does not contain zeroes,
`moduloBasis "g" gs` is a detaching map modulo `I`,
`moduloBasis "cg" gs` is a canonic map modulo `I`.
`mt` is the transformation matrix.

If `xs` consists of zeroes, then `gxBasis` has to yield `([], Mt [])`.
Example: for the polynomials over an c-Euclidean ring R,
`gs` is a strong reduced Gröbner basis — for R a field,
weak reduced Gröbner basis — otherwise (see (Section 38.5.3 Point pol.a.gr.g)).

18.2 moduloBasis

`moduloBasis mode basis f -> (rem, quot)`
is a zero-detaching reduction modulo the given generators `basis` (of ideal I).
`mode = cMode ++ gMode,  cMode = "" | "c",  gMode = "" | "g"`
`cMode = "c"` means a canonic reduction.
`cMode = ""` means only the reduction in which the zero modulo I is detached.

`gMode = "g"` means a gx-basis.
other value means that nothing is known about the basis.
In this latter case, it is applied the composition of `gxBasis` and `moduloBasis (_:'g')`.

Example:
for the polynomials $f = 2x^2$, $g = y + x$, $h = y + 2x$ from $K[x, y]$, K a field, and any admissable power product ordering with $y > x$,

`reduceByX mode = fst .  moduloBasis mode [x],`
`reduceByX "g"` maps f to 0 and leaves g, h unchanged,
`reduceByX "cg"` maps f to 0, g, h to y.

18.3 syzygyGens

`syzygyGens mode xs -> relationVectors`

`mode = ""` means the generic case,
`"g"` is for a polynomial ring $A = R[x_1, \ldots, x_n]$ over an Euclidean ring R.
`"g"` means that `xs` is a (weak) Gröbner basis — the evaluation will be more direct in such case.

Example:
`syzygyGens "" ([3,4,5,6] ::  Z) = [[6,-6,0,1], [5,-5,1,0], [4,-3,0,0]]`
acts as a solution of a generic homogeneous linear system over an Euclidean ring.

18.4 LinSolvRingTerm

```
data LinSolvRingTerm a =
     LinSolvRingTerm {linSolvRingProps :: Properties_LinSolvRing
                       -- this is all, so far
                     } deriving (Show)

type Properties_LinSolvRing = [(Property_LinSolvRing, PropValue)]
data Property_LinSolvRing   =
     ModuloBasisDetaching | ModuloBasisCanonic | WithSyzygyGens | IsGxRing
                                                 deriving (Eq, Ord, Enum, Show)
isGxRing :: LinSolvRingTerm a -> PropValue
isGxRing = lookupProp IsGxRing . linSolvRingProps
```

`(ModuloBasisDetaching, Yes)`
means that for any `xs`, `gs = fst $ gxBasis xs,  I = Ideal(gs)`
the map `moduloBasis (`*anycMode*`++"g") gs y --> (r, q)`
satisfies the ideal detachability: `y` $\in$ `I` $\Leftrightarrow$ `r == zero`

`(ModuloBasicCanonic, Yes)`
means `(ModuloBasisDetaching, Yes)` and that the `'c'` mode is correct.

That is for `gs = fst $ gxBasis xs,  I = Ideal(gs)`,
the map `moduloBasis "cg" gs y -> (r,q)` has a canonic remainder in its result:
`y-z` $\in$ `Ideal(xs)` $\Longleftrightarrow$ `r(y) == r(z)`.

`(WithSyzygyGens, Yes)`
means that `syzygyGens` *anyMode* is a correct algorithm to find the linear relation generators.

`(IsGxRing, Yes)`
means `(ModuloBasisDetaching, Yes)` and that
`gxBasis,  moduloBasis "cg",  syzygyGens` satisfy the laws of a gx-ring (see Section 49). In particular, it holds `(WithSyzygyGens, Yes)` and the transformation matrix and quotient vector are correctly evaluated.

19 EuclideanRing

This category is exported from `RingModule`, implemented in `Ring*` modules, `EucRingTerm` — in the module `Categs`.

```
class (GCDRing a, LinSolvRing a) => EuclideanRing a
  where
  baseEucRing :: a -> Domains1 a -> (Domains1 a, EucRingTerm a)
  eucNorm     :: a -> Z
  divRem      :: Char -> a -> a -> (a, a)        -- division with remainder

data EucRingTerm a = EucRingTerm {eucRingProps :: Properties_EucRing
                                  -- only this, so far
                                 } deriving (Show)
type Properties_EucRing = [(Property_EucRing, PropValue)]
data Property_EucRing   = Euclidean | DivRemCan | DivRemMin
                          deriving (Eq, Ord, Enum, Show)
```

In `divRem mode x y -> (quotient, remainder)`
`mode = 'm'` means the minimal-norm remainder,
`'c'` means that for each `b /= 0` `x -> divRem 'c' x b` is a canonical map for the residues modulo `b`.

The correctness of the operations `eucNorm, divRem` depends on the property values for `Euclidean, DivRemCan, DivRemMin`.

`(Euclidean, Yes)`
means that `eucNorm, (divRem` *anyMode*`)` are correct algorithms on R for the Euclidean ring structure.

That is, denoting `e = eucNorm : R\{0} -> NonNegativeInteger`, the following holds for any `b` $\in$ `R\{0}`

(1) `e(a*b)` $\geq$ `e(a)` for any `a` $\neq 0$,

(2) for any `a` $\in$ R and `(q,r) = divRem _ a b`

it holds `a = q*b + r`, where either `r = 0` or `e(r) < e(b)`.

`DivRemCan, DivRemMin`

are the correctness conditions for the `divRem` modes of `'c'` and `'m'` respectively.

Examples: In DoCon,

`divRem 'm' (-2) 5 = (0, -2),`   `divRem 'c' (-2) 5 = (-1, 3)`;

`divRem` on $K[x]$ for a field K does not depend on `mode`.

The condition for canonical remainder for `mode = 'c'` is easy to satisfy for simple cases and somewhat harder in esoteric ones. In particular, the simple cases are presented by the rings $\mathbb{Z}$, $k[x]$ with a field k, quadratic integer ring $\mathbb{Z}[root(d)]$ with negative d. $\mathbb{Z}[root(d)]$ with positive d presents a more complex case.

19.1 Several usable functions for Euclidean ring

```
isEucRing, isCEucRing :: EucRingTerm a -> PropValue

isEucRing  = lookupProp Euclidean . eucRingProps
isCEucRing = lookupProp DivRemCan . eucRingProps

quotEuc, remEuc :: EuclideanRing a => Char -> a -> a -> a
quotEuc mode x = fst . divRem mode x
remEuc  mode x = snd . divRem mode x

eucGCDE :: EuclideanRing a => [a] -> (a, [a])          -- extended GCD
  --
  -- xs -> (d, qs),    d = gcd[x1...xn] =  q1*x1 +...+ qn*xn
  --
  -- - qi  are any elements satisfying this equation.

eucGCDE xs =    -- this is also an example of programming:
  case  xs
  of
  []  -> error "eucGCDE []\n"
  x:_ -> gc xs
      where
      (zr, un)   = (zeroS x, unity x)
      gc (x: xs) =                      -- reduce to  gcd2
         case (xs, x == zr)
         of
        ([],    _   ) -> (x, [un])
        (_,    True) -> (d, zr: qs)  where  (d, qs) = gc xs
        (y: ys, _   ) -> let (d, u, v) = gcd2 (un, zr, x) (zr, un, y)
                        in
                        if  null ys  then  (d, [u,v])
                        else               (d', (u*q): (v*q): qs)
                                              where
                                              (d', q: qs) = gc (d: ys)
      gcd2 (u1, u2, u3) (v1, v2, v3) =
                -- It starts with  (un, zr, x) (zr, un, y),  x /= zr.
                -- Euclidean GCD is applied to u3, v3, operations on
                -- u1, u2, v1, v2 perform parallelwise to ones of u3, v3
                --
            if  v3 == zr  then  (u3, u1, u2)
            else                gcd2 (v1, v2, v3) (u1-q*v1, u2-q*v2, r)
                                              where
                                              (q,r) = divRem '_' u3 v3
```

20 Field

```
class (EuclideanRing a, FactorizationRing a) => Field a   -- needs (IsField, Yes)
class Field a => RealField a                              -- needs (IsRealField, Yes)
class (RealField a, OrderedRing a) => OrderedField a
```

A field R is implicit: it does not declare any extra operations.
R must be an EuclideanRing and satisfy the property of a field:

$$\texttt{(inv\_m x) == Nothing} \Leftrightarrow \texttt{x == 0}$$

This corresponds to the attribute (IsField, Yes) in the data `subringProps rR`.

21 Ideal

21.1 Preface

An ideal in `DoCon` is *not* given by a category instance (see Sections 3.5.11, 3.5.12). Unlike with `Ring`, `DoCon` represents an ideal only as an explicit, regular data — ideal term or, maybe, a bundle.

`DoCon` can really compute only with the ideal given by a finite generator list.
Setting an ideal means sometimes to set some additional attributes, like factorization or some property value.

The attributes of an ideal are used intensively by the functions related to the residue ring.

Ideal builders:
apply the functions `eucIdeal, gensToIdeal` to construct an ideal from generators,
from the short description.

For computing in R/I, the decription for I can be prepared once by such function, and further, its attributes are shared by the elements of R/I — see (Section 3.4 BF).

Ideal and its generators
Different generator lists may define the same ideal. `DoCon` mostly operates with the generators, but sometimes keeps in mind the ideals. For example, canonic reduction by generators occurs the reduction by the ideal, if the generators form a gx-basis (Sections 18.4, 49). A notion of gx-ring ('gx') exploited by `DoCon` corresponds to the ring with the solvable ideal inclusion, a solvable residue ring.

Summary:

`DoCon` represents an ideal mostly as a data term `iI`.

The main in `iI` is the generator list.

`DoCon` abilities depend greatly on what generator list is put in the description `iI` and what property values are put there.

21.2 Special ideal for Euclidean ring

These items are exported from `RingModule`, implemented in the `Ring*` modules.

```
data PIRChinIdeal a =
  PIRChinIdeal
    {pirCIBase      :: a,                -- b:  Ideal = (b)
     pirCICover     :: [a],              -- bs: (b) = intersect[..]
     pirCIOrtIdemps :: [a],              -- es: idempotents
     pirCIFactz     :: Factorization a   -- ft = [(p1,m1)...(pw,mw)]
    }
  deriving (Show, Read)
```

— a special representation for a non-trivial ideal `I = (b)` in a *principal ideal ring* R.

`b` is a generator for `I`. Other parts are optional — can be set as `[]`. But providing the factorization for `b`, or the *orthogonal idempotents* for the decomposition of `I` may enforce and optimize many operations on `I, R/I`.

The requirements for the data `iI :: PIRChinIdeal a` are

- `b` is neither zero nor invertible.
- If `bs == []` then `es == []`
- If `bs /= []` then
  (1) `(b) = idealIntersection` $[(b_j) \mid b_j \in bs]$,
  (2) $(b_i) + (b_j) = (1)$ for any i /= j — which means reciprocally prime ideals.
- If `es /= []` then
  `es` $= [e_1, \ldots, e_w]$, $e_i \in$ `a` are the Lagrange orthogonal idempotents:
  e_i *modulo* b_j is 1 for $i = j$ and 0 otherwise; hence $e_i^2 = e_i$, $e_1 + \ldots + e_w = 1$
- If `ft /= []` then
  `a` is a factorial ring, `ft =` $[(p_1, m_1), \ldots, (p_w, m_w)]$ is the factorization:
  `b = product bs,` $b_i = p_i^{m_i}$, $i \in$ `[1 .. w]` (b_i and p_i listed in the same order).

Example: the ideal `(24)` in `Z` can be described as

```
PIRChinIdeal {pirCIBase = 24,           pirCICover = [3,8],
              pirCIOrtIdemps = [16,9],  pirCIFactz = [(3,1),(2,3)]
             }
```

The function

```
eucIdeal :: (FactorizationRing a, EuclideanRing a) =>

     String -> a -> [a] -> [a] -> Factorization a -> PIRChinIdeal a
    -- mode    base cover  ids    factz
```

sets an ideal in an Euclidean ring. This is an easy way to build the ideal description from incomplete parts. The parts are listed as in the `PIRChinIdeal` data fields.
Applying `eucIdeal` completes the description.

Conditions.
In the most complex case, it needs a c-Euclidean ring with the factorization algorithm. In this case, it is presumed that the elements set in `cover` are the canonical remainders modulo `b`, and this function returns the orthogonal idempotents `ids'` — as the canonical remainders too (if `mode` contains `'e'`).

`mode = bMode++eMode++fMode` must be a substring of `"bef"`;

`'b' <- mode` means to update the `pirCICover` of the ideal term, other case means to leave it as it is ;

`'e', 'f'` correspond to the parts `pirCIOrtIdemps, pirCIFactz`.

Method for `eucIdeal` :
the main part is to obtain the Lagrange idempotents `es` from `bs = cover`.
It is done via applying `eucGCDE` to decompose $1 = d_{j,k} + d_{k,j}$ ...

Example:

```
eucIdeal "bef" (24::Z) [] [] [] =

PIRChinIdeal {pirCIBase      = 24,      pirCICover = [3,8],
              pirCIOrtIdemps = [16,9],  pirCIFactz = [(3,1),(2,3)]
             }
===
eucIdeal "b" (24::Z) [] [16,9] [(3,1),(2,3)]
```

— the latter description is cheaper to compute.

See the demonstration modules `T_*.hs` for the examples of computation in R/I.

Further, the `Functor` instance

```
instance Functor PIRChinIdeal
  where
  fmap f i = PIRChinIdeal {pirCIBase      = f a,
                           pirCICover     = map f cs,
                           pirCIOrtIdemps = map f ids,
                           pirCIFactz     = [(f p, e) | (p,e) <- ft]}
    where
    (a, cs, ids, ft) = (pirCIBase i, pirCICover i, pirCIOrtIdemps i, pirCIFactz i)
```

enables to 'map' between the ideals over different domains R, usually, between the isomorphic copies of R.

21.3 Generic ideal

`data Ideal` is exported from `Categs`, its related items are exported from `RingModule, Residue`, implemented in `Ring*.hs, IdealSyz_.hs`.

```
data Ideal a = Ideal {idealGens     :: Maybe [a],
                      idealProps    :: Properties_Ideal,
                      idealGenProps :: Properties_IdealGen,
                      idealConstrs  :: [Construction_Ideal a],
                      idealOpers    :: Operations_Ideal a
                     }
type Properties_Ideal    = [(Property_Ideal   , PropValue)]
type Properties_IdealGen = [(Property_IdealGen, PropValue)]
data Property_Ideal      = IsMaxIdeal | Prime | Primary
                                                deriving (Eq, Ord, Enum, Show)
data Property_IdealGen  = IsGxBasis  deriving (Eq, Ord, Enum, Show)
type Operations_Ideal a = [(OpName_Ideal, Operation_Ideal a)]
data OpName_Ideal       = IdealRank  deriving (Eq, Ord, Enum, Show)

newtype Operation_Ideal a    = IdealRank' (InfUnn Z)  deriving (Eq, Show)
newtype Construction_Ideal a = GenFactorizations [Factorization a]
                                           -- | Intersection of ideals ... ?
```

This presents an ideal I is a ring R.

<u>`idealGens :: Maybe [a]`</u> =
`Just gs` — means a finite list `gs` of generators for I,
`Nothing` — means "could not provide such generators".

<u>`idealProps :: Properties`</u>`_Ideal`
Here `IsMaxIdeal, Prime, Primary` have the usual meaning as in classic algebra ([La] Chapter 2).

<u>`idealGenProps :: Properties`</u>`_IdealGen`
Here `(IsGxBasis, Yes)` is the condition for the canonic reduction by `I`, and for the algorithmic operation correctness in `R/I`. See the gx-basis notion in the Sections 49, 18.4.

<u>`idealConstrs :: [Construction`</u>`_Ideal a]`
So far, it may provide only `GenFactorizations`, and only for the factorial ring.
In the data `GenFactorizations fts` `fts` is either `[]` or `(map factor gens)`.
Still `factor` may return `[]` for some elements — if `WithFactor /= Yes`
(see `FactorizationRing.factor`).

<u>`idealOpers :: Operations`</u>`_Ideal a`

So far only one 'operation' is available: the key of `IdealRank` extracts `IdealRank' r`, where `r` may occur

`Fin n` (finite rank) or `Infinity` or `UnknownV`.

21.3.1 Several usable functions for generic ideal

```
rankFromIdeal :: Ideal a -> InfUnn Natural
rankFromIdeal iI =  case  lookup IdealRank $ idealOpers iI
                    of
                    Just (IdealRank' v) -> v
                    _                   -> UnknownV

isMaxIdeal, isPrimeIdeal, isPrimaryIdeal :: Ideal a -> PropValue
isMaxIdeal = lookupProp IsMaxIdeal . idealProps
...
genFactorizationsFromIdeal :: Ideal a -> Maybe [Factorization a]
                              -- extract factorization list from the
                              -- first GenFactorizations construction

zeroIdeal :: Properties_Ideal -> Subring a -> Ideal a
             -- givenProps
  -- Zero ideal in a non-zero base ring.
  -- givenProps  contains some hints for the ideal properties.

unityIdeal :: a -> Subring a -> Ideal a     -- unity ideal in given base ring
            --unity
                          -- this is also example of ideal setting:
unityIdeal un _ =
             Ideal {idealGens     = Just [un],          idealProps   = props,
                    idealGenProps = [(IsGxBasis,Yes)],  idealConstrs = [],
                    idealOpers    = operations
                   }
              where  operations = [(IdealRank, IdealRank' (Fin 1))]
                     props      = [(IsMaxIdeal,No), (Prime,No), (Primary,No)]
```

21.3.2 Ideal bundle. Ideal from generators

In mathematics, an ideal `I` in a ring `R` is also a subring and a subgroup, and so on. Therefore, to build an ideal `I` and residue ring `R/I`, `DoCon` needs a bundle for the ideal, not only its term. For example, the function `gensToIdeal` takes
(a) a finite list of generators for `I`, (b) a short ideal description,
(c) a bundle `dR` for `R`, (d) a current bundle `dm` for `I`
and returns the new bundle `dm'` and ideal term.

```
gensToIdeal :: LinSolvRing a => [a]                ->   -- igens
                                [Factorization a]  ->   -- fts
                                Properties_IdealGen ->  -- bProps
                                Properties_Ideal   ->   -- iProps
                                Domains1 a         ->   -- dR
                                Domains1 a         ->   -- dm
                                (Domains1 a, Ideal a)
```

It makes the description for an ideal `I` from a finite generator list `igens` in a base ring `R` possessing the property `(ModuloBasisDetaching, Yes)`.

`dR` is a base domain filled up to `LinSolvRingTerm`.

`dm` is the current ideal bundle (usually empty) to be loaded further with the ideal and, maybe, other terms.

`iProps` is the sublist of the full property list.

`fts`
is ignored if it is given empty or `Factorial /= Yes` in `R` or `(Prime, Yes)` is set for `I`.

Otherwise, `fts` contains the given factorizations for each basis element
(see `FactorizationRing.factor`), each unknown factorization denoted by `[]`,
in this case `igens` remains as it is given.

Attention:
if it is known that `igens` is a gx-basis (See Sections 49, 18.4), then better set this explicitly in `bProps : (IsGxBasis, Yes)`. Otherwise, `gensIdeal` it will still compute `gxBasis` and set it in ideal in place of initial one.

For the most important ideal properties, such as
`Prime, Primary, IsMaxIdeal`, it is better to set them explicitly in `iProps`,
if the value is known.

`gensToIdeal` tries to make the property values more definite — using also the properties of `fts, R`. In particular, it completes the values according to the dependencies between `Prime, Primary, IsMaxIdeal`.

But it does not apply the primality test.

Example.
Build an ideal defined by a Gröbner basis `gs` in `R = E`$[x_1, \ldots, x_n]$,
`E` an Euclidean ring:

```
let {g = head gs;  dR = upLinSolvRing g Map.empty}
in
gensToIdeal gs [] [(IsGxBasis,Yes)] [] dR Map.empty     -- = (dI, iI)
```

This forms the ideal bundle `dI` starting from empty,
the ideal term `iI` — starting from `gs`.

`(IsGxBasis, Yes)` is an important information on `gs` — expensive to derive automatically.

Suppose now that

```
E = Q = Fraction Integer  (a field),   R = Q[x,y],
gs = [x^3 - 2, y^2 + 1].
```

Then, `R/I` is actually an extension of rational numbers `Q` with $\sqrt[3]{2}$ and $i = \sqrt{-1}$.

It is very useful in this case to to point out in the arguments that `I` is maximal:

```
let ... in gensToIdeal gs [] [(IsGxBasis,Yes)] [(IsMaxIdeal,Yes)] dR Map.empty
```

Then, `DoCon` would derive very simply that `R/I` is a field. In this example, the `IsMaxIdeal` value could be found from `gs`, but at some extra cost.

For more examples see `demotest/T_cubeext.hs, T_grbas2.hs`
— Cyclic Integer ring.

22 Module over a ring

Its items are exported from `RingModule`, implemented in `Ring*.hs`,
`Submodule, LinSolvModuleTerm` data — in module `Categs`.

Terminology with 'generators' and 'basis'

In mathematics, the notion of linear generator set ('generators') for a module over a ring specializes to the notions of a (a) basis of (sub)module, (b) basis of ideal.

The basis of a module is required to be *free* of linear relations, only then it is called a basis ([La] Chapter III §4).

But sometimes the algorithmic algebraists call a generator list a basis.

The word 'basis' in `DoCon` names
`gxBasis, moduloBasis, moduloBasisM, gxBasisM, IsGxBasis`
stands for 'generator list'.

22.1 LeftModule

```
class (Ring r, AddGroup a) => LeftModule r a
  where
  cMul           :: r -> a -> a                        -- module multiplication
  baseLeftModule :: (r, a) -> Domains2 r a -> (Domains2 r a, Submodule r a)
                    -- sa
```

This category has two parameters: a ring `r` and additive group `a`.

`baseLeftModule` has a sample argument `s :: (r, a)` which contains the samples for `r` and for `a`. For example,

`baseLeftModule (0,0) _` builds the module term for the module `Z` over `Z`,

`baseLeftModule (0, Vec [3,1]) _`
builds module term for the free module of vectors (of range 2) over `Z`.

Examples

1. `DoCon` puts `Vector R, UPol R, Pol R, EPol R, RPol R`
to be left modules over a ring `R`.

2. It also declares

```
instance Ring a => LeftModule a a where cMul = mul
                                      ...
```

— which means a ring is a module over itself.

3. It also *had* to declare

```
instance AddGroup a => LeftModule Z a where cMul n v = times v n
                                          ...
```

— “an additive group is a module over `Integer`”. But the current `Haskell` language cannot resolve this instance overlap with the instances from (1). We have to wait for an appropriate language extension (see Section 48).

22.2 Submodule

```
data Submodule r a = Submodule {moduleRank     :: InfUnn Z,
                                moduleGens     :: Maybe [a],
                                moduleProps    :: Properties_Submodule,
                                moduleGenProps :: Properties_SubmoduleGens,
                                moduleConstrs  :: [Construction_Submodule r a],
                                moduleOpers    :: Operations_Submodule r a
                               }
type Properties_Submodule     = [(Property_Submodule, PropValue)]
type Properties_SubmoduleGens = [(Property_SubmoduleGens, PropValue)]
type Operations_Submodule r a = [(OpName_Submodule, Operation_Submodule r a)]
data Property_Submodule       =
                         IsFreeModule | IsPrimeSubmodule | IsPrimarySubmodule
                         | IsMaxSubmodule | HasZeroDivModule | IsGradedModule
                         deriving (Eq, Ord, Enum, Show)
           --
           -- these properties generalize, as usual, the ring (or ideal) ones

data Property_SubmoduleGens = IsFreeModuleBasis | IsGxBasisM
                                                  deriving (Eq, Ord, Enum, Show)
data OpName_Submodule           = GradingM deriving (Eq, Ord, Enum, Show)
data Operation_Submodule r a    = GradingM' PPComp (a -> PPComp) (a -> [a])
data Construction_Submodule r a = ConsModuleDUMMY
```

The `Submodule` term is similar to the `Ideal` term.
Only `DoCon` does not have, so far, the residue by submodule.

<u>`Property`</u>`_Submodule`
generalizes the ideal properties. For example, (IsPrimeSubmodule, Yes) for a submodule N of a module M over R means: for any $a \in R$ the multiplication $(a*)$ on M/N is either zero homomorphism or an injective one. See ([La] Chapter VI §5).

<u>`IsFreeModuleBasis`</u> from `Property_SubmoduleGens`
means that the generators `gs` are linearly independent over a ring R. In this case M is isomorphic to $R \oplus \ldots \oplus R$ — direct sum of a finite set of copies of R.

22.3 Syzygy solvable module

```
class (LinSolvRing r, LeftModule r a) => LinSolvLModule r a
  where
  baseLinSolvLModule :: (r,a) -> Domains2 r a -> (Domains2 r a, LinSolvModuleTerm r a)
                                --
                                -- all these operations have sample argument
  canAssocM     :: r -> a -> a
  canInvM       :: r -> a -> r
  gxBasisM      :: r -> [a] -> ([a], [[r]])
  moduloBasisM :: r -> String -> [a] -> a -> (a, [r])
  syzygyGensM  :: r -> String -> [a] -> [[r]]
```

Examples.
DoCon declares

```
instance EuclideanRing a => LinSolvLModule a (Vector a) where...
instance EuclideanRing a => LinSolvLModule (Pol a) (Vector (Pol a)) where ...
```

The former defines the solvable submodule structure for the free module $R \oplus R \oplus \ldots \oplus R$ over an Euclidean ring R. It is done via the Gauss method for a linear system over an Euclidean ring. The latter instance uses the `EPol` representation for `Vector` over `Polynomial` and the Gröbner basis for e-polynomials. See [MoM].

`LinSolvLModule` category is very useful for dealing with the syzygies (linear relations) of polynomials — see the examples in `demotest/T_grbas2.hs`.

22.4 Syzygy solvable module term

```
data LinSolvModuleTerm r a =
     LinSolvModuleTerm {linSolvModuleProps :: Properties_LinSolvModule
                        -- this is all, so far
                       } deriving (Show)

type Properties_LinSolvModule = [(Property_LinSolvModule, PropValue)]
data Property_LinSolvModule =
     IsCanAssocModule | ModuloBasisDetaching_M | ModuloBasisCanonic_M
     | WithSyzygyGens_M | IsGxModule     deriving (Eq, Ord, Enum, Show)
```

`(IsCanAssocModule, Yes)` means that
(1) the ring is commutative,
(2) `canAssocM _ v` is a unique vector chosen in the set
$\{c * v \ : \ c$ an invertible factor$\}$,
(3) `canInvM _ v` is the corresponding factor.

Other properties generalize a ring (ideal) ones.

Auxiliary functions for `Submodule`

```
isGxModule:: LinSolvModuleTerm r a -> PropValue
isGxModule   (LinSolvModuleTerm ps) = lookupProp IsGxModule ps

isoModule :: (a -> b) -> (b -> a) -> Submodule r a -> Submodule r b
             -- f        fInv        mM
  -- Isomorphic submodule.
  -- Given the map f and its inverse ‘maps’ the Submodule description
  -- respectively.
```

23 Up-functions

A *base bundle* can be built from initial one (maybe, empty) by an up-function.

Up-function is a composition of several base-operations. See Section 3.5.8. Here we list all of them.

They are exported by the `SetGroup, RingModule` modules
and have a format

`up`*category* :: *category* `a => a -> Domains1 a -> Domains1 a`

category === `AddSemigroup | MulSemigroup | ... | Ring ...`,

An up-function differs from base-functions in that it also forms all the implied domains for the given sample `x` and puts them into new bundle.

```
type ADomDom a = a -> Domains1 a -> Domains1 a

upAddSemigroup :: AddSemigroup a => ADomDom a
upAddSemigroup a = fst . baseAddSemigroup a . fst . baseSet a

upAddGroup          :: (AddGroup          a) => ADomDom a
upMulSemigroup      :: (MulSemigroup      a) => ADomDom a
upMulGroup          :: (MulGroup          a) => ADomDom a
upRing              :: (Ring              a) => ADomDom a
upGCDRing           :: (GCDRing           a) => ADomDom a
upFactorizationRing :: (FactorizationRing a) => ADomDom a
upLinSolvRing       :: (LinSolvRing       a) => ADomDom a
upEucRing           :: (EuclideanRing     a) => ADomDom a
upField             :: (Field             a) => ADomDom a

upGCDLinSolvRing   :: (GCDRing a, LinSolvRing a) => ADomDom a
upEucFactrRing     :: (EuclideanRing a, FactorizationRing a) => ADomDom a
upFactrLinSolvRing :: (FactorizationRing a, LinSolvRing a) => ADomDom a

upAddGroup a = fst . baseAddGroup a . upAddSemigroup a
```

Other up-functions are defined similarly. The 'longest' one is

```
upEucRing = fst .  baseEucRing a .  fst .  baseLinSolvRing a .  upGCDRing a
```

And `upField = upEucFactrRing` — this is because `Field` has not the term of its own.

24 iso-functions

They serve for isomorphic copying of domain descriptions and bundles.

`isoOSet ... isoRing ... isoEucRingTerm ...`

are described each in the section on corresponding category. But there are also such functions for the bundles:

`isoDomain1, isoDomains1, isoDomain22, isoDomains22`

They are exported by `RingModule`. The idea is simple: look through the bundle and convert each `OSet` term by applying `isoOSet`, `Subsemigroup` term — by `isoSemigroup`, and so on.

```
isoDomain1 :: (a -> b) -> (b -> a) -> Domain1 a -> Domain1 b
isoDomain1    f           f'          dom       =  case  dom  of

             D1Set      t -> D1Set      (isoOSet            f f' t)
             D1Smg      t -> D1Smg      (isoSemigroup       f f' t)
             D1Group    t -> D1Group    (isoGroup           f f' t)
             D1Ring     t -> D1Ring     (isoRing            f f' t)
             D1GCDR     t -> D1GCDR     (isoGCDRingTerm     f f' t)
             D1FactrR   t -> D1FactrR   (isoFactrRingTerm   f f' t)
             D1LinSolvR t -> D1LinSolvR (isoLinSolvRingTerm f f' t)

isoDomains1 :: (a -> b) -> (b -> a) -> Domains1 a -> Domains1 b
isoDomains1    f           f'       =  mapFM (\_ dom -> isoDomain1 f f' dom)

isoDomain22 :: (a -> b) -> (b -> a) -> Domain2 c a -> Domain2 c b
isoDomain22    f           f'          dom         =  case  dom  of

                        D2Module   t -> D2Module   $ isoModule        f f' t
                        D2LinSolvM t -> D2LinSolvM $ isoLinSolvModule f f' t

isoDomains22 :: (a -> b) -> (b -> a) -> Domains2 c a -> Domains2 c b
isoDomains22    f           f'       = mapFM (\_ dom -> isoDomain22 f f' dom)
```

25 Domain Char

In DoCon, the only *ground* domains are `Char` and `Integer`. For `Char`, `DoCon` provides only with the `OrderedSet` instance. We recommend the advanced user to inspect its implementation in `auxil/Char_.hs`.

26 Domain Integer

`DoCon` calls it `type Z = Integer`. And `Int` is ignored.

The module `Z` exports the instances for the domain `Z` — in addition to the `Haskell` Prelude ones:

```
Set, OrderedSet, ..., OrderedAddGroup,
MulSemigroup, MulMonoid,
Ring, CommutativeRing, OrderedRing, GCDRing, FactorizationRing,
LinSolvRing, EuclideanRing.
```

26.1 On GCDRing Integer

The invertible elements in `Integer` are $\{1, -1\}$,
`canInv` yields 1 or -1, `canAssoc` acts respectively to `canInv`.

26.2 On LinSolvRing Integer

`gxBasis`
is presented by the extended Euclidean gcd algorithm:
`xs -> (g, qs)`, $\mathrm{g} = \mathrm{gcd}\,[\,x_1, \ldots, x_n\,] = q_1 * x_1 + \ldots + q_n * x_n \ldots,$ x_i `<- xs`, q_i `<- qs`

`moduloBasis` :
for `xs = [d]`, it is done via `y -> divRem 'c' y d`.
For a larger list, the composition with `eucGCDE` applies.

`syzygyGens`
is presented by linear system solution over an Euclidean ring: `solveLinear_euc`.

26.3 On EuclideanRing Integer

Do not confuse `divRem, quotEuc, remEuc` of `DoCon`
with `quotRem, div, mod` ... of the `Haskell` prelude.

`divRem 'c'` chooses a non-negative remainder for `Integer`, and this makes a canonical remainder map modulo ideal.

26.4 Bundle dZ

```
dZ :: Domains1 Integer
dZ = upEucFactrRing 0 Map.empty  -- puts to dZ all the known domain terms for Z
```

See Section 23.

26.5 Several useful functions for Integer

```
logInt ::  (Ord a, OrderedRing a) => a -> a -> Integer
```

`logInt b a` is the integer part of logarithm of `a` by the base `b`, `a, b > 1`.

Where it is correct? At least, for `Integer, Fraction Integer`.

Examples: `logInt 3 29 = logInt (3:/2) (29:/8) = 3`,
`logInt 29 3 = 0`.

Method:
repeat division while the quotient is greater than `b`.

```
orderModuloNatural :: Natural -> Integer -> Natural
                      -- r       a
```

Order of `a` in `Z/(r)  =  min [k > 0 | a^k = 1 (mod r)]`
— for `r > 1`, `r` mutually prime with `a`.

```
totient ::  Natural -> Natural
```

`totient n` is the number of the *totitive* natural numbers for `n`.

A number `k` is totitive for `n` iff `0 < k < n` and `gcd n k = 1`.

Restriction: `n > 1`.

Method: finds `gcd n k` for `k` in `[2 ..  (n-1)]`.

```
rootOfNatural :: Natural -> Natural -> Natural -> (Natural, Natural)
                 -- e       x          b          r        r^e
```

Integer part of the `e`-th degree root of a natural number `x` for `e > 0`.
`r = intPart (root_e x)`.
`b` is a bound for the search: the root is searched in the segment `[0, b]`.
By shoosing `b`, the client can save some cost. And anyway, `b = x` is correct.
Caution: setting `b < root_e x` leads to incorrect result.

Cost estimation: $O((log_2 b)^3 * e^2 * (log_2 e))$.

Method: bisection for the function `^e` — integer version, this is much cheaper than dealing with rational approximations.

```
squareRootOfNatural :: Natural -> Natural -> Natural
                    -- x          bound
```

Integer part of the *square* root of `x`, the root searched in `[0, bound]`.

Method: Newton's iteration, downwards starting from `bound`, with taking a certain integer part at each iteration.
It is a somewhat faster than `rootOfNatural`.
It suffices to set `bound = x`, but it has sense to set the smallest known upper approximation for the root.

```
minRootOfNatural :: Natural -> Maybe (Natural, Natural)
                 -- n                 e         r
```

This tests a natural `n > 1` for being a power of some degree `e > 1` of any `r > 1`.

If the needed `e, r` exist, then it returns `Just (e, r)` with the minimal possible `e`.

Otherwise, it returns `Nothing`.

Method: for each `e <- [2, log_2_n]`, apply `rootOfNatural e n bound`, where `bound = (previous root) + 1`.

Cost estimation: $O((log\ n)^5 * (log\ log\ n))$.

Examples:

```
2, 3 -> Nothing,       4  -> Just (2, 2),
16   -> Just (2, 4),   81 -> Just (3, 4).
```

27 Constructor List

Starting from a ground domain, we can build other domains with the domain constructors.

For the List constructor `[]` `DoCon` provides only the `Set` instance — in addition to the `Haskell` Prelude instances.

They are exported by `DoCon` module `DPrelude`.

`baseSet (xs :: [a])` is valid only for a non-empty `xs`.

We think that we do not need any algebraic instances for `List` beyond `Set`.

`Set [a]` and Set (Vector a) are very different.
The set for `[a]` contains all non-empty lists;
the vectors from the set for `(Vector a)` are of the same size. Thus

```
osetCard $ snd $ baseSet "b"        Map.empty  = Infinity,
osetCard $ snd $ baseSet (Vec "ab") Map.empty  = Fin (256^2)
```

28 Permutation

28.1 Preface

For the permutation constructor

```
newtype Permutation = Pm [Integer] deriving (Eq, Read)
```

the module `Permut` exports the instances up to `MulGroup, Num` with

`(*)` = permutation composition.

`(+)` = composing permutations on disjoint sets (like concatenation or map extension).

In `Pm xs` `xs` = $[x_1, \ldots, x_n]$ must be non-empty and free of repetitions.

And the domain of the sample `Pm xs` is `S(n)`.

`S(n)` is also viewed as the bijections on the set $\{x_1, \ldots, x_n\}$.

Examples

```
-- base set of  Pm [4,-1,2]  is  S(3):

(osetCard s, osetList s) where  s = snd $ baseSet (Pm [4,-1,2]) Map.empty
-->
   (Fin 6,  Just [Pm [-1,2,4], Pm [-1,4,2], Pm [2,-1,4],
                  Pm [2,4,-1], Pm [4,-1,2], Pm [4,2,-1]]
   )
                                                -- other computations
Pm [2,1] * Pm [2,1]           = Pm [1,2]
inv $ Pm [2,1]                = Pm [2,1]
Pm [2,1] + Pm [7,5,3]         = Pm [2,1,7,5,3]
permutCycles $ Pm [2,1,7,5,3] = [[2,1], [7,3], [5]]
```

See also `demotest/T_permut.hs`.

Computation method

The main tool is simply sorting of an integer pair list by the first component or by the second. For the large sets, it, probably, worths to add the implementation based on the binary search trees.

28.2 Definitions

```
newtype Permutation = Pm [Integer]  deriving (Eq, Read)
type    EPermut     = [(Integer, Integer)]

-- In  Pm xs   xs  must not contain repetitions.
-- EPermut     is a Permutation whose list  xs  is zipped with  xs
--             sorted increasingly.
```

```
toEPermut :: Permutation -> EPermut
toEPermut (Pm xs) =  zip (sort xs) xs

fromEPermut :: EPermut -> Permutation
fromEPermut = Pm . map snd

permutRepr :: Permutation -> [Integer]
permutRepr (Pm xs) =  xs

permutSign :: Permutation -> Char                -- '+'|'-'
permutSign =  snd . sortE compare . permutRepr

instance Show Permutation
  where
  showsPrec _ p =  ("(Pm "++) . shows (permutRepr p) . (')':)

instance Ord Permutation
  where
  compare p = lexListComp compare (permutRepr p) . permutRepr

applyPermut :: Ord a => [Z] -> [a] -> [a]
applyPermut             ks     xs  = map snd . sortBy (compBy fst) . zip ks
                                --
                                -- example:  [2,3,4,1] -> "abbc" -> "cabb"

applyTransp :: (Z, Z) -> [a] -> [a]
  -- Transpose the elements No i and j in list.  head xs  has No 1.
  -- Required: 1 <= i <= j <= length xs.
  -- Example:
  -- let xs = "abcde" in (applyTransp (2,2) xs, applyTransp (2,4) xs)
  --                                            = ("abcde", "adcbe")

applyTranspSeq :: (Ord a, Show a) => (a,a) -> [(a,b)] -> [(a,b)]
  --
  -- Transpose the entities of indices  i <= j  in the sequence.
  -- A sequence is a list of pairs  (k,x)  ordered increasingly by
  -- the index  k <- `a'.  Skipped index means ``zero'' component.
  -- Example:
  -- for  ps = [(2,'2'),  (4,'4'),  (6,'6'),  (8,'8')]
  --
  -- applyTranspSeq (4,4) ps == applyTranspSeq (3,7) ps == ps,
  -- map (flip applyTranspSeq ps) [(4,8),(4,7),(4,9)] ps
  --        ==
  --        [(2,'2'),   (4,'8'),    (6,'6'),          (8,'4')]
  --        [(2,'2'),               (6,'6'), (7,'4'), (8,'8')]
  --        [(2,'2'),               (6,'6'),          (8,'8'), (9,'4')]
```

```
transpsOfNeighb :: [a] -> [[a]]
  -- Transpositions of neighbour elements in xs = [x1..xn]  presents
  -- certain small generator list for S(n).
  -- For  xs = [x],  the result is put [[x]].
  -- For  Ord a,  x1 < x2 .. < xn,  the result really represents the
  -- transposition of neighbours.
  -- Otherwise, it still gives the generator list of cardinality n-1
  -- (for n > 1).

transpsOfNeighb [x] = [[x]]
transpsOfNeighb xs  = nbts xs
                        where
                        nbts [_]      = []
                        nbts (x:y:xs) = (y:x:xs):(map (x:) $ nbts (y:xs))

nextPermut :: [Z] -> Maybe [Z]  -- next permutation in lexicographic order

---------------------------------------------------------------------
instance Set Permutation
  where
  baseSet p dm =  ...
  compare_m p  = Just . compare p

                                 -- on input, permutation looks like a list
  fromExpr (Pm xs) e = case  fromExpr xs e  of

    ([ys],"") -> ([Pm ys], "")
    _         -> ([], "(fromExpr "++ (shows (Pm xs) " ")++ (shows e ")\n") )

invEPermut :: EPermut -> EPermut                        -- inversion
invEPermut =  map (\ (x,y) -> (y,x)) . sortBy (compBy snd)

addEPermut :: EPermut -> EPermut -> EPermut
                                    -- compose permutations on disjoint sets
addEPermut p = mergeBy (compBy fst) p

---------------------------------------------------------------------
instance MulSemigroup Permutation
  where
  baseMulSemigroup = ...
  unity_m          = Just . Pm . sort . permutRepr
  inv_m            = Just . fromEPermut . invEPermut . toEPermut

       -- mul p q = p*q  <--> \x-> p(q(x)   Required:  Set(p)=Set(q)
```

```
      --
  mul p q = Pm $ map fst $ sortBy (compBy snd) $ zip (permutRepr p) $ permutRepr r
                                                       where
                                                       Just r = inv_m q
instance MulMonoid Permutation
instance MulGroup Permutation  where  baseMulGroup p dm = ...

instance Num Permutation
  where
  (*) = mul

  -- (+) means composing permutations on disjoint sets
  --
  p+q = fromEPermut $ addEPermut (toEPermut p) (toEPermut q)
--------------------------------------------------------------------
ePermutCycles :: EPermut -> [EPermut]

  Decomposes  s  to cyclic permutations  [c(1) .. c(r)] :
  s = c(1) + ... + c(r),  length c(1) >= .. >= length c(r).

permutECycles :: Permutation -> [EPermut]
permutECycles = ePermutCycles . toEPermut

permutCycles :: Permutation -> [[Integer]]
```

29 Constructor Vector

29.1 Preface

In DoCon, `Vector` represents a direct product of domain $D \oplus \ldots \oplus D$ — n- times.

In the data `Vec xs` `xs` must be a non-empty list.

`Vector a` differs radically from `[a]` in that

the base domain of `(Vec xs)` *is parameterized by the length of* `xs`

— see (Section 3.4 DS).

The base set of a sample `v = Vec xs :: Vector a` consists of all the vectors `u` of size `vecSize v` with the components from the base set of a component of `v (= sample v = vecHead v)`.

So, the algebra of vectors refer to the vectors of the same length.

The items for `Vector` are exported by the modules
`VecMatr, RingModule, Pol`
and implemented in `Categs_.hs, Vec*.hs, Ring*.hs, Pol2_`. The instances

```
Show, Eq, Functor, Random, Dom, Set, OrderedSet, AddSemigroup ...
OrderedAddGroup, MulSemigroup ... MulGroup ... OrderedRing
```

are exported from `VecMatr` and are defined in an evident way: the operations perform component-wise.

Concerning `instance Random a => Random (Vector a) ...`,
see Section 30.

Further, the module `RingModule` defines

```
instance Ring a          => LeftModule     a (Vector a) where ...
instance EuclideanRing a => LinSolvLModule a (Vector a) where ...
```

The line with `LeftModule` has usual classic meaning.

And `LinSolvLModule` operations in this case are defined via the Gauss reduction to the staircase form over an Euclidean ring.

The module `Pol2_` also defines

```
instance EuclideanRing a => LinSolvLModule (UPol a) (Vector (UPol a)) where ...
instance EuclideanRing a => LinSolvLModule (Pol  a) (Vector (Pol  a)) where ...
```

Here the `LinSolvLModule` operations are defined via the Gröbner basis technique for a free module over $\mathtt{a[x_1,\ldots,x_n]}$ (see [MoM]).

The instance `... => LinSolvLModule (Pol a) (Vector (Pol a)`
overlaps with `... LinSolvLModule b (Vector b)`.

With this respect, `DoCon` relies on that `Haskell` chooses for the domains from the instance intersection the instance for the more special type expression. And here the type expression `(Pol a)` is more special.

This all makes a free module (`Vector R`) over an Euclidean ring `R` a module with solvable submodules;

the same is with `Vector` $\mathtt{R}[\mathtt{x}_1,\ldots,\mathtt{x}_n]$ over $\mathtt{R}[\mathtt{x}_1,\ldots,\mathtt{x}_n]$.

29.2 Usable functions for Vector

```
newtype Vector a = Vec [a]  deriving (Eq)     -- non-empty list boxed in Vec
vecRepr (Vec l) = l                         -- extract representation

vecSize :: Vector a -> Natural
vecSize =  genericLength . vecRepr

vecHead :: Vector a -> a
vecHead    v        =  case vecRepr v of x:_ -> x
                                         _   -> error "vecHead v:  empty v \n"
vecTail :: Vector a -> [a]
vecTail    v        =  case vecRepr v of _:xs -> xs
                                         _    -> error "vecTail v:  empty v\n"
scalProduct :: CommutativeRing a => [a] -> [a] -> a
scalProduct                         xs     ys  =

  case (xs ++ ys)
  of
  x: _ -> sum1 ((zeroS x) : (zipWith (*) xs ys))
  _    -> error ("\nscalProduct [] []:   " ++
                "at least one of the lists must be non-empty.\n"
               )

constVec :: Vector a -> b -> Vector b
constVec    v           b =  fmap (const b) v

allMaybesVec :: [Maybe a] -> Maybe (Vector a)
allMaybesVec              =  fmap Vec . allMaybes
```

29.3 Instances

```
instance Functor Vector  where  fmap f = Vec . map f . vecRepr
```

Here `Functor` is an `Haskell` Prelude constructor class.

For `Vector`, this instance enables the programmer to ‘map’ a function similarly as to the list: `map f (Vec xs)` means applying `f` component-wise.

```
instance Dom Vector
  where
  sample = vecHead
  dom _  = error "dom (Vec..):  dom  not defined for Vector\n"
```

Further instances for `Vector` are listed and explained earlier, in Section 29.1.

Example:

```
(v+v, v*v, fmap neg v)  where  v = Vec [1, 2, 3::Z]   =

   (Vec [2,4,6], Vec [1,4,9], Vec [-1,-2,-3])
```

30 Generating random values

For this, `DoCon` exploits the operation `randomR` of the `Haskell` library class `Random`.

Example:

```
instance Random a => Random (Vector a)
  where
              -- put a random vector 'between vl and vh' to have random
              -- components 'between' l(i) and h(i)  for each  i
              --
  randomR (vl,vh) g = (Vec $ reverse xs, g')
     where
     (xs,g') = foldl rnd ([],g) $ zip (vecRepr vl) (vecRepr vh)

     rnd (xs,g) (l,h) = (x:xs, g')  where  (x,g') = randomR (l,h) g
```

`randomR (l,h) g` returns a random value between `l` and `h` and a new state `g' :: StdGen`.

Repeating this operation yields a random sequence.

`StdGen` is an abstract type from the `Haskell` library `Random`.

The initial state can also be obtained standardly, for example, by

`mkStdGen n :: StdGen`, with any `n :: Int`.

See also the `Random UPol`, `Random Pol` instances.

<u>Remark</u>

What does this mean 'between l and h' for l, h from some non-trivial, non-ordered domain? For example, l, h may be from `Z/(4)`, or polynomials from `Z[x]`, or such.

The answer is that in `randomR (l, h)`, `l, h` of some domain R, `(l, h)` denotes only some subset in R, this subset is defined individually, by corresponding definition of opeartion `randomR` for the particular constructor.

This definition may not depend on the ordering on R.

For example, for `(l,h) = (-3*x^2 + 1, 3*x^2 + 4)` $\in$ `(Z[x], Z[x])`,

`DoCon` puts that "between `l` and `h`" are the polynomials `a*x^2 + b`, where `a <- [-3..3]`, `b <- [1..4]`.

Caution

DoCon also puts `randomR (l, h) == randomR (h, l)`,
for a certain reason. So that for a random value 'between' `l` and `h`, the 'direction' is immaterial.

But we failed so far to agree this convention with the Haskell standard. As soon as any practical problems arise from this for the users, DoCon can introduce its own category DRandom, and define simply its instances, referring when needed, to the standard Random.

Constructors supplied with Random instance

`Integer, Bool, Char` have the Haskell library instances of Random.

DoCon supplies with the instances of Random the constructors

`Vector, UPol, Pol, ResidueE, ResidueE, ResidueG.`

31 Matrix

31.1 Preface

A matrix is represented by the constructors `Matrix` and `SquareMatrix`.
Similarly as for Vector, `mt = Mt rows aD :: Matrix a`, `SqMt rows aD`
must contain a non-empty list `rows` of non-empty lists of the same length.

`mt` has the base set of all the matrices `n by m` over `aD`,
where `n = mtHeight mt`, `m = mtWidth mt`,
`aD` is the base domain of a matrix entity `(sample mt)`.

The base set for `SquareMatrix` is parameterized by one parameter
`n = mtHeight mt`.

But unlike with `Vector`, the constructors `Matrix, SquareMatrix` have the bundle field — see (Section 3.4 BF).

Example.
Form a matrix `m` 2 by 2 over `Z[x]` and find its cube: power:

```
let  x1 = cToUPol "x" dZ 1    -- unity of Z[x]
     dX = upGCDRing x1 Map.empty
     m' = mapmap (smParse x1) [["x", "x + 2"    ],
                               ["1", "x^2 - 2*x"] ]
     m  = Mt m' dX
in   m^3
```

The instances `Set ... AddGroup` for `Matrix, SquareMatrix` are similar to the `Vector` ones — operations defined component-wise.

```
instance CommutativeRing a => Num (Matrix a)       ...
instance CommutativeRing a => Num (SquareMatrix a) ...
```

define the operations `(+)`, `(*)`. `(*)` is the matrix multiplication.
For `SquareMatrix`, `(*)` satisfies the semigroup law. So, `DoCon` adds

```
CommutativeRing a => MulSemigroup (SquareMatrix a) ...
CommutativeRing a => Ring         (SquareMatrix a) ...
                                      -- non-commutative for n > 1
```

The classes `MatrixLike` and `MatrixSizes` (see below) are used to unify certain operation names (`mtRows`, `mtHeight`, `transp`, and such) for the data of `Matrix` and `SquareMatrix`.

The `Matrix` items are exported from `VecMatr`, implemented in the modules of `VecMatr`, `Ring0_`, `Matr*_`.

31.2 Usable items for matrices

```
data Matrix a = Mt [[a]] (Domains1 a)

-- Mt rows dm  must contain a non-empty list ‘rows’ of
--             non-empty lists of the Same length.

data SquareMatrix a = SqMt [[a]] (Domains1 a)

-- In  SqMt rows dm  it must hold
-- (length xs)==(length rows) > 0  for each  xs  from  rows

class MatrixLike m where                     -- for Matrix, SquareMatrix
                   mtRows :: m a -> [[a]]
                   mapMt  :: (a -> a) -> m a -> m a
                   transp :: m a -> m a

instance MatrixLike Matrix where mtRows (Mt rs _)  = rs
                                 transp (Mt rs d)  = Mt (transpose rs) d
                                 mapMt f (Mt rs d) = Mt (mapmap f  rs) d

instance MatrixLike SquareMatrix where
                                 mtRows (SqMt rs _)  = rs
                                 transp (SqMt rs d)  = SqMt (transpose rs) d
                                 mapMt f (SqMt rs d) = SqMt (mapmap f  rs) d

instance Eq a => Eq (Matrix a)       where  x == y =  mtRows x == mtRows y
instance Eq a => Eq (SquareMatrix a) where  x == y =  mtRows x == mtRows y

toSqMt :: Matrix a    -> SquareMatrix a
toSqMt (Mt rs dom) = SqMt rs dom             -- CAUTION: rs must be square

fromSqMt :: SquareMatrix a -> Matrix a
fromSqMt (SqMt rs d) =  Mt rs d
```

```
instance Dom Matrix  where  dom (Mt _ d) = d
                            sample       = matrHead

-- Similar items are defined for  SquareMatrix.

class MatrixSizes a where mtHeight :: a -> Natural
                          mtWidth  :: a -> Natural
                                            -- for [[a]], Matrix, SquareMatrix

instance MatrixSizes [[a]] where mtHeight = genericLength
                                 mtWidth []     = error "\nmtWidth []\n"
                                 mtWidth (r: _) = genericLength r

instance MatrixLike m => MatrixSizes (m a) where  mtHeight = mtHeight . mtRows
                                                  mtWidth  = mtWidth  . mtRows

mtHead :: [[a]] -> a
mtHead ((x:_): _) = x
mtHead _          = error "mtHead m: empty m\n"

matrHead :: MatrixLike m => m a -> a
matrHead =  mtHead . mtRows

mtTail :: MatrixLike m => m a -> [[a]]
mtTail mM =  case mtRows mM of  _: rs -> rs
                                _     -> error "\nmtTail <emptyMatrix>.\n"

constMt :: MatrixLike m => m a -> a -> m a
constMt mM a =  mapMt (const a) mM

rowMatrMul :: CommutativeRing a => [a] -> [[a]] -> [a]   -- multiply Row by Matrix

isZeroMt :: AddSemigroup a => [[a]] -> Bool
isZeroMt rows = case rows
                of
                (a: _): _ -> case zero_m a of Just z -> all (all (== z)) rows
                                              _      -> False
                _       -> error "\nisZeroMt m:  empty m.\n"

scalarMt :: [a] -> b -> b -> [[b]]
           -- xs   c    z
   -- Make  scalar matrix  NxN  from the given elements  c, z
   -- and the list  xs  of length N,  xs  serves as a counter.
   -- c  is placed on the main diagonal,  z in the rest of matrix.
   -- COST = O(n^2)
```

```
mainMtDiag :: [[a]] -> [a]    -- main matrix diagonal = [m(i,i) | i <- [1..]

isDiagMt, isStaircaseMt, isLowTriangMt :: AddGroup a => [[a]] -> Bool
  --
  -- ‘is diagonal’,  ‘is staircase’,
  -- ‘is lower-triangular’ matrix  (j > i ==> m(i,j) = 0)
```

Examples:

```
[[1,0,2,1,0],
 [0,0,1,1,1],
 [0,0,0,2,0]]   is staircase, not diagonal, not lower-triangular.

[[0,0,0,0],
 [2,1,0,0],
 [1,1,0,0]]     is lower-triangular, not staircase, not diagonal.
```

```
vandermondeMt ::  MulMonoid a => [a] -> [[a]]
```

presents a map to the Vandermonde matrix:

```
[a0,...,an] -> [[a0^n,     ... an^n    ]
                 [a0^(n-1), ... an^(n-1)]
                 ...
                 [1,        ... 1       ]
                ]
```

```
resultantMt ::  AddGroup a => [a] -> [a] -> [[a]]
```

is for resultant matrix. For the coefficient lists `xs = x:_`, `ys = y:_`
of dense polynomials `f = x*t^n + ...`, `g = y*t^m + ...`
from `a[t]`, `n = deg f`, `m = deg g`, `n, m > 0`, `x, y` non-zero,
build the resultant matrix `M` over `a` for `f, g`. So, `det M = resultant f g`.

Example:

```
[a3,0,a1,0] -> [b2,b1,b0] -> [[a3, 0 , a1, 0 , 0 ]
                              [0 , a3, 0 , a1, 0 ]
                              [b2, b1, b0, 0 , 0 ]
                              [0 , b2, b1, b0, 0 ]
                              [0 , 0 , b2, b1, b0]]
```

Method.

For `n = deg f = |xs|-1`, `m = deg g = |ys|-1`, zeroes prepend to `xs, ys` to make `xs0, ys0` both of length `n+m`.
`M = (xs0 shifted m times)++(ys0 shifted n times)`,
where shifting means the round left shift.

32 Linear algebra

The corresponding items are exported from the module `LinAlg` (to continue the items of the `VecMatr` module), and they are implemented in `Det_.hs, Stairc_.hs, Todiag_.hs, LinAlg.hs`.

32.1 Reduction of vector by subspace

```
reduceVec_euc :: EuclideanRing a => Char -> [[a]] -> [a] -> ([a], [a])
                                    --mode  us       v       rem  qs
```

A *staircase* vector list `us = [`u_1 `...`u_n`]` reduces the vector `v` as possible, by subtracting from it certain linear combination of u_i.

The quotient list `qs = [`q_1 `...`q_n`]` is produced such that

`v = `q_1`*`u_1` + ...+ `q_n`*`u_n` + rem`

'Reducing' means making zeroes in possibly many head positions. The tail of remainder is being reduced too.

Required: `v`, u_i must be of the same size.

`mode = 'c' | ...` is as in `EuclideanRing.divRem`

The function `reduceVec_euc` possesses the following properties.

- `isZero rem` $\Longleftrightarrow$ `v` depends linearly on the vectors `us`
- For a c-Euclidean ring, `(reduceVec_euc 'c' us)` is a canonical map by Submodule(`us`): `v==v' modulo us` $\Longleftrightarrow$ `rem(v) == rem(v')`

Method: since `us` is staircase, only one reduction pass has to perform.

32.2 Reduction to staircase matrix

```
toStairMatr_euc ::
EuclideanRing a => String -> [[a]] -> [[a]] -> ([[a]], [[a]], Char)
                -- mode      m        t0        s      t      sign
```

Gauss method for bringing a matrix `m` to the staircase form.

For `Field a`, it performs like usual Gauss method.

For not a field, it repeats the remainder division to obtain zeroes down in the current column ...
For an Euclidean ring, this leads, evidently, to a staircase matrix.

The transformations are applied parallelwise to the matrix `t`.

`t` is the *transformation matrix*, `height t = height m`.

If `t0 = unityMatrix` then it holds `t*m = s`,
where `t` is the protocol matrix for the Gauss reductions of `m`.

In the generic case, it holds `u*m = s,  u*t0 = t`
for the (invertible) protocol matrix `u`.

`t0 = []` is a synonym for the unity matrix of size `(height m)`.

If `m` is zero and `t0 = []`, then `(m, [])` is returned.

`mode == "" | "u"`

`"u"` means that `m` is already staircase, and the function must only try to reduce to zero the elements *super the main diagonal*. In this case, `(s', t)` are returned, where `s'` has the reduced (as possible) upper part and is diagonal if `m` is invertible.

`sign == '+' | '-'`

is the accumulated signum of the result row permutation.
It holds: `det(m)==det(s)` Or `det(m)== -det(s)`; `sign` shows one of these alternatives.

For example, the `det` function makes use of this signum result.

For `mode = "u"`, `sign` is always `'+'`.

The best case for performance is a finite field `a`.

The implementation comments are in `Stairc_.hs`.

Example
Find the staircase form `s` for a matrix `m` over `Integer` and test `t*m = s` for the corresponding transformation matrix.

```
let m = [[1,2,3,4,5,6],
         [5,0,6,7,1,0],
         [8,9,0,1,2,3]
        ]                              :: [[Integer]]
    (s, t, _) = toStairMatr_euc "" m []
in
(Mt t dZ)*(Mt m dZ) == (Mt s dZ)
```

32.3 Determinant

```
det ::  CommutativeRing a => [[a]] -> a
```

Determinant of a square matrix computed by the most generic method.
Method
It expands `det(M)` by the row (after moving ahead the rows which contain more of zeroes).
It costs $O(n!)$ in the worst case.

For the Euclidean case, see the Gauss method: `det_euc`:

```
det_euc :: EuclideanRing a => [[a]] -> a
```

Determinant of a square matrix over an Euclidean ring computed via the Gauss reduction to the staircase form — as in `toStairMatr_euc`.
The implementation comments are in `Det_.hs`

```
adjointMt ::  CommutativeRing a => [[a]] -> [[a]]
```

`mA = adjointMt mM` is the adjoint matrix for `mM`.
`mA(i,j) = coMinorDet(j,i)` is the cofactor in the inverse matrix formula.
For any square matrix `mM`, it holds `mM*(adjointMt mM) == (det mM)*UnityMatrix`.
Method
The cofactor determinants are found here by the generic method '`det`'
(for `CommutativeRing a`). The implementation is in `Det_.hs`

32.4 Reduction to diagonal form

```
toDiagMatr_euc ::
EuclideanRing a => [[a]] -> [[a]] -> [[a]] -> ([[a]], [[a]], [[a]])
                   -- m     t0       u0        d      t      u
```

Diagonal form `d` of a matrix `m` over an Euclidean ring obtained by the elementary transformations of the rows and columns.

`t, u` are the unimodular protocol matrices for the the row and column transformation respectively.

`t0 = []` or `u0 = []` is a shorthand for the unity matrix of appropriate size.

Let `h = matrixHeight(m)`, `wd = matrixWidth(m)`.
If `t0, u0` are the `(h by h) and (wd by wd)` unity matrices, then

`t * m * transp(u) = d`,

where the (lower) non-zero part of `d` is square and diagonal.
This means, it first applies the elementary transformation to the rows. If `m` is invertible over `a`, we always achieve a diagonal form by this. If it is not achieved, similar transformations are applied to the columns.

Sometimes the returned `u` occurs an unity matrix (that is the row transformations were sufficient). This holds when, for example, when `m` is invertible.

Remark: the determinant can only change its sign under the above transformations.

The implementation comments are in `Todiag_.hs`.

Example:
a program in the module `demotest/T_diagmatr.hs` applies diagonalization to matrix over $K[x]$, for a finite field K.

32.5 Linear system solution

```
diagMatrKernel ::  Ring a => [[a]] -> [[a]]
```

returns the kernel basis `ker` for a diagonal matrix `D` having no zero rows on diagonal.

In particular, it holds `height(D) <= width(D)`, `D*(transpose ker) = zeroMatrix`

A domain `a` must be a ring with unity and no zero divisors.
For the zero kernel the result is `[]`.

```
solveLinear_euc :: EuclideanRing a => [[a]] -> [a] -> ([a], [[a]])
                                   -- mM    v       p    ker
```

is the general solution of linear system `mM*(transp [x]) = (transp [v])`

`mM` is `n by m` matrix, `v` is a vector of size `n`.

`p` is a row for some partial solution, it is set `[]` if there is no such solution.

`ker` is the rows generating the `a` -module of solutions for the homogeneous system `mM * transp(x) = zeroColumn`.

For the zero kernel, `ker = []`.

Method

`m` converts to the diagonal form by the Gauss method for the rows and columns, the transformation matrices `t, u` are accumulated. Then, the diagonal system is solved, and the goal solution restores via `t, u`.

`IsField` is tested first for the domain `a` to separate easy case.

The implementation comments are in `LinAlg.hs`.

Example: see the `solveLinear_euc` application in Section 2.2.

```
solveLinearTriangular :: CommutativeRing a => [[a]] -> [a] -> Maybe [a]
                                           -- mA    row
```

solves a system `xRow x mA' = row` (1)
for an upper-triangular matrix `mA'`.

Size agreement:
`mA` is restricted to the main minor `mA'` of size `|row| by |row|`, the remaining part is removed, the solution is for `mA'`.

Hence, `|xRow| = size mA' = |row|`.

If `a` is free of zero divisors, `mA'` is upper triangular and has not zeroes on the main diagonal, then this function satisfies the property:

if (1) has solution then `solveLinearTriangular` returns `Just xs`, where `xs` is this (unique) solution,

otherwise it returns `Nothing`.

Method:

The usual method for solving a triangular system: find `x(1)`, then find `x(2)` via `x(1)`, and so on.

Cost = $O(|row|^2)$.

32.6 Other usable functions

```
rank_euc :: EuclideanRing a => [[a]] -> Natural
            -- rank of matrix via Gauss method - as in  toStairMatr_euc

maxMinor :: Natural-> Natural-> [[a]]-> [[a]]  -- delete i-th row and j-th column
maxMinor i j rows = delColumn j $ del_n_th i rows
```

```
inverseMatr_euc :: EuclideanRing a => [[a]] -> [[a]]
```

is inversion of a square matrix `mM` over an Euclidean ring.
If `mM` is invertible, the inverse matrix `iM` is returned, otherwise, returned is `[]`.

Method.

The Gauss reduction to the staircase form `(s,t)` is applied,
`t` the transformation matrix, `t0 = unityMatrix`.
This reduces the task to the inversion of a lower-triangular matrix.
The implementation comments are in `Stairc_.hs`.

```
linBasInList_euc :: EuclideanRing a => [[a]] -> ([Bool], [[a]])
```

For the matrix `M` = $[\mathtt{v}_1 \ldots \mathtt{v}_n]$, mark some $\mathtt{v}_i$ that constitute some maximal linearly independent subset `M1` in `M`.

In the returned list $[\mathtt{b}_1 \ldots \mathtt{b}_n]$, $\mathtt{b}_i$ = `True` means $\mathtt{v}_i \in$ `M1`.

Also it is returned the staircase form for `st` for `M1`.

The implementation comments are in `Stairc_.hs`.

33 Pair

33.1 Common approach

The instances for the pair constructor `(,)` are exported by the module `DPair` — in addition to the `Haskell` library instances.

The instances `Set ... AddGroup ... MulGroup ... LinSolvRing` are defined for the domain `(a,b)` under necessary condition instances for the types `a, b`.

Naturally, the operations on the pairs are defined component-wise.

Example:

```
let  p = (5,6)  :: (Z,Z)
in   (p*(3,3) - p,  p/p,  fst $ moduloBasis "" [(2,0),(0,3)] p)
-->
((10,12), (1,1), (1,0))
```

The base set for the sample `(x,y)` is the direct product `xBS` $\times$ `yBS` of the base sets for `x` and `y`.

The instances for `(a,b)` before `LinSolvRing` are evident.
The instance of `LinSolvRing (a,b)` bases on that any ideal in `(a,b)` is a direct sum of its projection ideals to `a` and `b`.

See Section 49.2, and maybe, `DPair*.hs` programs.

Also `DoCon` provides the following usable function for Pair:

```
maybePair :: Maybe a -> Maybe b -> Maybe (a,b)
maybePair    (Just x)   (Just y) = Just (x,y)
maybePair    _          _        = Nothing
```

33.2 Direct product of domain terms

Besides the mentioned above category instances, the module `DPair` exports the the functions that given the domain descriptions for `D1, D2`, produce the description for `D1` $\times$ `D2`.

```
directProduct_set :: OSet a -> OSet b -> OSet (a,b)

directProduct_semigroup :: Subsemigroup a -> Subsemigroup b -> Subsemigroup (a, b)
   --                      sH1               sH2
   -- directProduct_semigroup  yields the subsemigroup of the base
   -- semigroup H1 x H2 of (a,b). Basic operations are inherited from
   -- H1, H2.  Hence,  H1, H2  must have the same sort  AddOrMul.

directProduct_group ::
```

```
  (Set a, Set b) =>
  a -> OSet a -> Subgroup a -> b -> OSet b -> Subgroup b -> Subgroup (a,b)
--un1  sS1       sG1           un2  sS2       sG2
--
-- the groups must be of the same sort AddOrMul,
-- un1, un2  the unities of G1, G2 respectively.


directProduct_ring ::
(Ring a, Ring b) => a -> Subring a -> b -> Subring b -> Subring (a,b)
                  --zA   rA           zB   rB
                  -- zA, zB are the zeroes of A, B
```

34 Fraction

34.1 Common approach

DoCon uses the `Fraction` data instead of `Ratio` of the `Haskell` library.

The reasons for this are

- `Ratio` applies `divMod, sign, (<)` to cancel fractions, which is not universal, it is hardly applicable, say to polynomials over a field.
- `Fraction` prefers to apply slightly different algorithm for arithmetics. It cancels the intermediate elements by `gcd` as soon as possible.

Of course, whenever needed, the interaction between `Fraction` and `Ratio`, is simple.
For example, to exploit any function `g :: Ratio a -> Ratio a` for `Fraction a` one can set

```
f (n:/d) = let  h       = g (n%d) :: Ratio a
                (n',d') = (numerator h, denominator h)
           in   n':/d'
```

<u>More difference to `Ratio`</u>

`Ratio` of `Haskell` Prelude is an *abstract type*.
For example, you cannot match against it: `f (n%d) = ...`
And for the constructor (:/) of DoCon, matching `f (n:/d) = ...` is possible.

But this imposes additional care on the user of avoiding of non-canonical fractions.
For example, `(4:/2)^2, (2:/(-3))^2` may cause an incorrect result.

In DoCon, the `Fraction` constructor applies correctly only for the base ring with the attribute

`(WithGCD, Yes)`

This requires, in particular, `(Factorial, Yes)` and a correct `gcD` algorithm, though the factorization algorithm is not necessary. See Section 15.3.

Hence, `Fraction` yields a `Field`.

The operations take and return the fractions $n:/d$ in their canonical form. That is

$$d \neq 0, \quad (n:/d) == (n':/d') \iff (n,d) == (n',d')$$

— the algebraic equality is here the syntax one.

The mathematical correctness of such equality is achieved through the gcd cancellation and cancellation by `(canInv d)`.

Use `canFr` for the cancellation in extra cases.

The difficulties may arise for the subtle domains where `canInv` is hard to compute. At least, `Integer` and `Pol`-like constructors do not bring this difficulty.

<u>`Rational` via `Fraction Integer`</u>

`Haskell` Prelude puts `Rational = Ratio Integer`
In DoCon, this is replaced with `Fraction Integer`.

Example.
Form some polynomial `f` from `P = Z[x1,x2]` and compute `f^2 + 1/f` in `Fraction P`:

```
let
  p1      = cToPol (lexPPO 2) ["x1","x2"] dZ 1  -- unity polynomial
  [x1,x2] = varPs 1 p1
  [p2,p3] = map (fromi p1) [2,3]                -- integers in P
  f       = x1^2*x2^3 + p2*x1 + p3*x2
in
(f:/p1)^2 + (p1:/f)
```

The instances of

`Set ... OrderedAddGroup ... MulMonoid, Ring ... OrderedField`

are defined according to usual notion of a fraction ([La] Chapter II §3).
In particular, the instances of

`LinSolvRing, GCDRing, FactorizationRing, EuclideanRing`

for `Fraction` are valid but trivial, due to the trivial division relation in a field.

Further, any correct ordering instance for `compare_m` on `a` induces correct `compare_m` on `Fraction a` (agreed with operations) — see `IsOrderedRing` in Section 15.3 and theory in [La].

<u>`Fraction Integer`</u>

Special instances are defined for `Fraction Integer`, overlapping with the generic one.

This is arranged so, because `Fraction Integer` has more definite domain attributes than the generic `GCDRing a => Fraction a`.

The items for `Fraction` are exported by the module `Fraction`.

34.2 The main items for Fraction

```
infixl 7  :/
data Fraction a = a :/ a  deriving (Eq, Read)
                  -- deriving Eq refers to the canonic representation approach
num   (n :/ _) = n
denom (_ :/ d) = d

zeroFr, unityFr :: Ring a => a -> Fraction a

zeroFr  x = (zeroS x):/(unity x)
unityFr x = (unity x):/(unity x)

canFr :: GCDRing a => String -> a -> a -> Fraction a
  --
  -- This is for bringing the intermediate result to the canonical
  -- fraction.
  -- mode = "g"  means to cancel the pair by gcd,
  --       "i"                          by canonical invertible,
  --       ""                           by both.

instance Functor Fraction  where  fmap f (n:/d) = (f n):/(f d)
instance Dom     Fraction  where  sample= num
                                  dom _ = error "dom (n:/d)  not defined\n"
```

35 PolLike class

This constructor class unifies certain operations for

`UPol` — sparse univariate polynomial,
`Pol` — multivariate polynomial with dense power products,
`RPol` — 'recursive' form (the power products are sparse too),
`EPol` — sparse representation for vectors over polynomials,
`SymPol` — symmetric function (polynomial).

For example, the leading coefficient `lc f`, total degree `deg f` have the same denotation for all five models listed above.

Some of the below operations can be understood more definitely when observing their instances for `UPol, Pol, ...`

```
class Dom p => PolLike p
  where
  pIsConst  :: CommutativeRing a => p a -> Bool          -- "is constant"
  lpp       :: CommutativeRing a => p a -> PowerProduct  -- leading power product
  deg       :: CommutativeRing a => p a -> Z             -- total degree
  ldeg      :: CommutativeRing a => p a -> Z
                                  -- total degree of lpp (depends on ordering)
  lm        :: CommutativeRing a => p a -> Mon a         -- leading monomial
  pTail     :: CommutativeRing a => p a -> p a
  pFreeCoef :: CommutativeRing a => p a -> a             -- free coefficient
  pCoefs    :: p a -> [a]      -- coefficients listed in same order as monomials;
                               -- for RPol, the order is "depth first"

  pCoef     :: CommutativeRing a => p a -> [Z] -> a
                                         -- coefficient of given power product
  pVars     :: p a -> [PolVar]                         -- variable list
  pPPO      :: p a -> PPOrdTerm                        -- PP ordering description
  degInVar  :: CommutativeRing a =>  Z -> Z -> p a -> Z
                                -- for0 i    f
                         -- deg f  in variable No i; put it  for0  for zero f

  pMapCoef :: AddGroup a => Char -> (a -> a) -> p a -> p a
                         -- mode    f
                         -- map  f  to each coefficient.  mode = 'r' means
                         -- to detect (and delete) appeared zero monomials

  pMapPP   :: AddGroup a => ([Z] -> [Z]) -> p a -> p a
              --            f
              -- map f to each exponent. It does not reorder the
              -- monomials, nor sums similars. Examples:
              -- pMapPP (\ [i]   -> [i+2]  ) (x+1)   = x^3+x^2  in Z[x]
```

```
            -- pMapPP (\ [i,j] -> [i+j,j]) (x*y+1) = x^2*y + 1

pCDiv :: CommutativeRing a => p a -> a -> Maybe (p a) -- quotient by coeficient
varPs :: CommutativeRing a => a -> p a -> [p a]
                       -- Convert each variable from  f :: p a  multiplied
                       -- by the given non-zero coefficient to  p a.


pValue :: CommutativeRing a => p a -> [a] -> a
       --
       -- Value of "polynomial" at  [a1 .. an],  extra  ai  are cut.
       -- Example: for a[x], a[x,y]   x^2   -> [2]     -> 4
       --                             x^2+y -> [2,0]   -> 4
       --                             x^2+y -> [2,0,3] -> 4
       --                             x^2+y -> [2]     -> error...

pDeriv  :: CommutativeRing a => Multiindex Z -> p a -> p a
          -- Derivative by multiindex.
          -- Example: for  variables = [x1,x2,x3,x4],
          --            pDeriv [(2,3), (4,2)] === (d/dx2)^3*(d/dx4)^2

pDivRem :: CommutativeRing a => p a -> p a -> (p a, p a)
  --
  -- f -> g -> (quotient, remainder)
  -- For  a[x] and Field(a),  it has to satisfy the Euclidean division
  -- property.
  -- In any case, it has to continue the Euclidean-like reduction
  -- (applying  divide_m  to divide coefficients)
  -- while  lm(g)  divides  lm(currentRemainder).
  -- Example:
  -- 5*x^2+1, 2*x -> ((2/5)*x, 1      )   for  a = Rational
  --                 (0,       5*x^2+1)   for      Z

pFromVec :: CommutativeRing a => p a -> [a] -> p a
            --
            -- Convert (dense) vector to p. of the given sample.
            -- Example:  2*x -> [0,1,0,2,3,0,0] -> x^5 + 2*x^3 + 3*x^2
            -- So far, consider it ONLY for  UPol.

pToVec   :: CommutativeRing a => Z -> p a -> [a]
            -- List of the given length of p. coefficients,
            -- gaps between the power products filled with zeroes.
            -- So far, consider it ONLY for  UPol.
```

Remarks

`lpp f` is `Vec [deg f]` for `UPol` and `snd $ eLpp f` for `EPol`

`ldeg` is `totalDeg . lpp` for `Pol`, and
`totalDeg . snd . eLpp` for `EPol`

`lm` is extended `lmU` for `UPol` and
`case eLm f of (c, (_,p)) -> (c,p)` for `EPol`

`pPPO` is `(lexPPO 1)` for `UPol` and `pPPO . epolPol` for `EPol`

`pToVec n f = cs` is formed as follows.
let m be the number of monomials in the dense form of a polynomial `f`.
If $n > m$ then $n - m$ zeroes are prepended to the result,
otherwise, $m - n$ higher monomials are cut out.
Example: `7 (a*x^4 + b*x^2 + c)   -> [0,0,a,0,b,0,c]`
`7 (a*x^9 + b*x^5 + c*x) -> [0,b,0,0,0,c,0]`

`varPs` is widely usable. For example, for a polynomial `f` from `P = Z["x","y"]`
`varPs 1 f --> [x,y]`, where `x, y` are the elements of `P`. See also `varP`.

`deg, ldeg` may differ only in the multivariate case.
For example, for `f = x + y^2*z` represented as `Pol [<x>, <y^2*z>] ...`,
`ldeg f = 1, deg f = 3`.

`c = pCoef f js` has the following meaning.
For `UPol` : `js = [j]` and `c` = coefficient of degree `j` in `f`.
For `Pol` : `c` = coefficient of power product `Vec js`.
For `EPol` : `js = j:ks` and `c` = coefficient of `(j, Vec ks)`.
For `RPol, SymPol` it is undefined.

<u>Some polymorphic functions under PolLike context</u>

```
lc :: (PolLike p, Set a) => p a -> a            -- leading coefficient
lc  f = case  pCoefs f  of
                      a:_ -> a
                      _   -> error $ ("lc 0, \n0 <- R"++) $ shows (pVars f) $
                                   (",\nR = "++) $ showsDomOf (sample f) "\n"

pCont :: (GCDRing a, PolLike p) => p a -> a            -- gcd of coefficients
pCont =  gcD . pCoefs

pHeadVar :: (PolLike p, Set a) => p a -> PolVar
pHeadVar f = case pVars f of  v:_ -> v
                              _   -> error (...++"Empty variable list\n")
numOfPVars :: PolLike p => p a -> Z
numOfPVars = genericLength . pVars
```

```
varP :: (PolLike p, CommutativeRing a) => a -> p a -> p a
varP a = head . varPs a                         -- useful in univariate case

lc0 :: (PolLike p, AddSemigroup (p a), Set a) => a -> p a -> a
lc0 zr f = if isZero f then zr else lc f

deg0 :: (PolLike p, AddSemigroup (p a), CommutativeRing a)
        =>
        Char -> Z -> p a -> Z
deg0    mode    d    f   =  case (isZero f, mode) of (True, _  ) -> d
                                                     (_   , 'l') -> ldeg f
                                                     _           -> deg f
cPMul :: (PolLike p, Ring a) => a -> p a -> p a
cPMul a = pMapCoef 'r' (mul a)                      -- product by coefficient
```

The examples with these useful functions appear all through this manual and in `demotest/T_*.hs`.

Further, for `PolLike p`, `cPMul` already makes `(p a)` a module over `a` :

```
instance (Ring a, PolLike p, AddGroup (p a)) => LeftModule a (p a)
  where
  cMul                    = cPMul
  baseLeftModule (_,f) dm = ...
```

36 Univariate polynomial

36.1 Preface

```
type PolVar = String     -- polynomial "variable"
type UMon a = (a, Z)     -- univariate monomial


data UPol a = UPol [UMon a] a PolVar (Domains1 a)
```

This describes a sparsely represented univariate polynomial `UPol mons c v aD` :

`mons` is the list of monomials ordered decreasingly by `deg`, with zero monomials skipped,

`c` a sample coefficient, `aD` domain description for `a`.

Example: `UPol [(1,4),(-2,2),(3,0)] 0 "t" dZ`
represents the polynomial $t^4 - 2t^2 + 3$ from `Z[t]`.

The equality for `UPol` uses the above presumed conditions on representation: it compares only the monomial lists.

```
instance Dom UPol where  dom    (UPol _ _ _ d) = d
                         sample (UPol _ c _ _) = c
```

The polynomial constructors need several fields for auxiliary information. Therefore,

to build polynomials, apply `cToUPol, cToPol`*,*

and then, casting by a sample.

The casting by sample for `UPol` (besides the maps of `fromi, smParse`) is done via the operations `ct, ctr` related to the `Cast` instances: they cast to `UPol a` from `a`, monomial, monomial list over `a`.

`ct` casts "as it is", `ctr` — with filtering out zero coefficient monomials.

Example:

for `f = UPol _ 1 "x" d` $\in$ `P = Z[x]`,

`cast _ f 2` maps 2 to `P`,

`cast _ f (2,3)` maps monomial $2x^3$ to `P`,

`cast _ f [(2,3),(-1,1)]` maps given monomial list to `P`.

We hope, with the next `Haskell` version, `DoCon` would be able to cast from the *variable* too.

```
instance Ring a => Cast (UPol a) a
  where
  cast mode (UPol _ _ v d) a = case mode of 'r'-> cToUPol v d a
                                            _  -> UPol [(a,0)] a v d


instance AddGroup a => Cast (UPol a) (UMon a)      -- monomial to polynomial
```

```
  where
  cast mode (UPol _ c v d) (a,p) = UPol mons c v d
                     where
                     mons = if  mode=='r' && isZero a  then []  else [(a,p)]
```

```
instance AddGroup a => Cast (UPol a) [UMon a]
  where
  cast mode (UPol _ c v d) mons = UPol ms c v d
    where                                                -- order NOT checked
    ms = if mode /= 'r' then mons  else filter ((/= z) . fst) mons
    z  = zeroS c
```

The algebraic instances for `UPol a` $\longleftrightarrow$ `a[x]` are defined only for the Commutative ring `a` with unity:

`Set ... AddGroup ... MulMonoid, Ring ... LinSolvRing,`
`EuclideanRing, FactorizationRing,`
`LinSolvLModule (UPol a) (Vector (UPol a))`

— each of them is correct under the corresponding condition.

The items for `UPol` are exported from the module `Pol`,
implemented in `UPol*_.hs, Pol*_.hs, RPol*_.hs`.

36.2 PolLike UPol

This also illustrates the `PolLike` class meaning. It uses several auxiliary functions (`lmU, upolMons` ...) explained in the next section. Also note the use of `ct, ctr` below.

```
instance PolLike UPol
  where
  pIsConst f           = iszero_ f || (deg f)==0
  deg                  = snd . lmU      -- for UPol,
  ldeg                 = snd . lmU      --   deg = ldeg
  lpp      f           = Vec [deg f]    -- :: PowerProduct
  pVars (UPol _ _ v _) = [v]
  pPPO     _           = lexPPO 1
  pCoefs               = map fst . upolMons
  degInVar for0 i f    = case (upolMons f,i) of ([] , _) -> for0
                                                (m:_, 1) -> snd m
                                                _        -> 0
  varPs a f = [ct f (a, 1::Z)]
```

```
  pTail f = case upolMons f of _:ms -> ct f ms
                               _    -> error$ ("pTail 0  in  R"++)$ shows (pVars f)$
                                         (",\nR = "++) $ showsDomOf (sample f) "\n"
  pFreeCoef (UPol mons c _ _) = if null mons then zeroS c
                                else            case last mons of (a,0) -> a
                                                                  _     -> zeroS c
  pCoef f js = let {z = zeroS $ sample f;  vs = pVars f}
               in
               case js of [j] -> case  dropWhile ((>j) . snd) $ upolMons f
                                 of
                                   (a,i):_ -> if i==j then a else z
                                   _       -> z
                          _   -> error ...

  lm f = (a, Vec [j])  where  (a,j) = lmU f         -- lmU is usable

  pMapCoef mode f g = cast mode g [(f a, i) | (a,i) <- upolMons g]
  pMapPP f g        = ct g [(a, head $ f [n]) | (a,n) <- upolMons g]

  pCDiv f c = let (cs, exps) = unzip $ upolMons f
              in
              case allMaybes [divide_m a c | a <- cs] of
                                         Just qts -> Just $ ct f $ zip qts exps
                                         _        -> Nothing
  pValue f []    = error ...
  pValue f (a:_) = case unzip $ upolMons f of
                              ([],_ ) -> zeroS a
                              (cs,es) -> sum1$ zipWith (*) cs $ powersOfOne es a
  pDeriv [(1,n)] f = deriv_ n f
  pDeriv mInd    f = error ...

  pFromVec f coefs = ctr f $ reverse $ zip (reverse coefs) [0::Z ..]

  pToVec n f = case  (upolMons f, zeroS (sample f))  of

    (ms, z) -> dv n $ dropWhile ((>= n) . snd) ms
               where
               dv n []         = genericReplicate n z
               dv n ((a,j):ms) = (genericReplicate (n-j-1) z)++(a:(dv j ms))

  pDivRem (UPol monsF c _ _) g =  ...    -- see UPol*.hs

type Multiindex i = [(i,i)]
```

36.3 Usable items for UPol

```
upolMons :: UPol a -> [UMon a]
upolMons    (UPol ms _ _ _) = ms

instance Eq a => Eq (UPol a) where f==g = (upolMons f)==(upolMons g)

lmU :: Set a => UPol a -> UMon a                          -- leading monomial
lmU             f      =  case upolMons f of m:_ -> m
                                              _   -> error $ (..."lmU 0"...)

leastUPolMon :: Set a => UPol a -> UMon a
leastUPolMon f = case upolMons f of [] -> error $ (..."leastUPolMon 0"...)
                                    ms -> last ms

mUPolMul :: Ring a => UMon a -> UPol a -> UPol a      -- multiply by monomial


cToUPol :: Ring a => PolVar -> Domains1 a -> a -> UPol a
  --
  -- Coefficient --> polynomial.
  -- Apply it to create a  sample polynomial.
  -- See the examples all through this manual.
  --
cToUPol v aDom a = if a==(zeroS a) then  UPol []      a v aDom
                   else                  UPol [(a,0)] a v aDom



umonLcm :: GCDRing a => UMon a -> UMon a -> UMon a
  --
  -- Lcm  of monomials over a  gcd-ring.
  -- For the Field case, it is better to compute this "in place", as
  -- (unity a, ppLcm ...)
  --
umonLcm (a,p) (b,q) = case gcD [a,b] of g -> (a*(b/g), lcm p q)
--------------------------------------------------------------------
monicUPols_overFin :: CommutativeRing a => UPol a -> [[UPol a]]

  -- Given  f  from  a[x],  deg f > 0,  `a' a Finite ring,
  -- build the infinite listing --- partition  [gs(d), gs(d+1) ..]
  -- for the set  {g <- a[x] |  deg g >= d, lc g = 1},
  --
  -- were  d = deg f,   gs(n) = [g | deg g = n, lc g = 1]
  -- EXAMPLE:  a = Z/(3),
  --           2*x   -> [[x,x+1,x+2], [x^2,x^2+1..x^2+2x+2], [x^3..] .. ]
  --           2*x^2 -> [             [x^2,x^2+1..x^2+2x+2], [x^3..] .. ]
```

```
---------------------------------------------------------------------
upolPseudoRem :: CommutativeRing a => UPol a -> UPol a -> UPol a

  -- Pseudodivision in  R[x]  ([Kn], vol 2, section 4.6.1).
  -- For non-zero  f,g,  there exist  k,q,r  such that
  --   (lc(g)^k)*f = q*g + r,   k <= deg(f)-deg(g)+1,
  --   and either  r = 0  or  deg r < deg g.
  -- upolPseudoRem  returns only  r.
  -- It does not use the coefficient division, and it should be cheaper
  -- than   pDivRem (lc(g)^(n-m+1)*f) g

-----------------------------------------------------------------------
charMt :: CommutativeRing a => PolVar -> Matrix a -> Matrix (UPol a)
                               -- la     mM          charM
-- The characteristic matrix  mM' - la*E:
-- add (- la) to the main diagonal.   mM' is mM imbed to  a[la].

charPol :: CommutativeRing a => PolVar -> Matrix a -> UPol a
--
-- \ la mM ->  characteristic polynomial of mM in the variable  la
--
charPol la = det . mtRows . charMt la

-----------------------------------------------------------------------
resultant_1 :: CommutativeRing a => UPol a -> UPol a -> a
resultant_1                         f         g =

  -- the resultant computed in the generic and direct method
  if
    pIsConst f || pIsConst g  then
              error "... both positive degrees are required.\n"
  else
  let {n = succ $ deg f;  m = succ $ deg g}
  in  det $ resultantMt (pToVec n f) (pToVec m g)

-----------------------------------------------------------------------
resultant_1_euc :: EuclideanRing a => UPol a -> UPol a -> a
  --
  -- resultant of  f,g  from  a[x],  `a' an Euclidean ring,
  -- computed by a special method

-----------------------------------------------------------------------
discriminant_1 :: CommutativeRing a => UPol a -> a
--
-- Discr(f) = (Resultant(f, f')/a)*((-1)^(binomCoef n 2)) where
--                                                a = lc f, n = deg f.
```

```
-- n > 1 required.
-- This resultant is computed in the generic and direct way, no
-- optimization.

discriminant_1_euc :: EuclideanRing a => UPol a -> a
--
-- Here the resultant is computed by a more special method, with using
-- the Gauss elimination over an Euclidean ring.

matrixDiscriminant :: CommutativeRing a => Matrix a -> a
matrixDiscriminant =  discriminant_1 . charPol "lam"

--------------------------------------------------------------------
upolSubst :: CommutativeRing a => UPol a -> UPol a -> [UPol a] -> UPol a
                                  -- f      g         gPowers

  -- Substitute  g  for the variable into  f,  f,g <- R[x].
  -- The powers [g^2,g^3 ..] are either given in  gPowers
  -- or  gPowers = [],  and they are computed by the Horner scheme.
```

```
----------------------------------------------------------------------------
  upolInterpol :: CommutativeRing a => UPol a -> [(a,a)] -> UPol a
                                          -- smp    tab
```

Interpolate (rebuild) polynomial $y = y(x)$ of degree n, $x, y \in$ a, $y(x)$ given by a sample polynomial smp and by a table tab = [$(x_0, y_0), \dots, (x_n, y_n)$] in which x_i do not repeat.

Required: a must have unity.

Example:

for Z[x], upolInterpol _ [(0,1),(1,-2),(2,-1)] --> $2x^2 - 5x + 1$

Method:

Newton interpolation formula with the difference ratios:

$$p(x) = y_0 + (x - x_0) \cdot y[0,1] + \ \dots\ + (x - x_0) \cdot \dots \cdot (x - x_{n-1}) \cdot y[0, 1..n],$$

where $y[0, 1..k]$ is the difference ratio of order k:

$$y[0,1] = (y_1 - y_0)/(x_1 - x_0), \quad y[0,1,2] = (y[1,2] - y[0,1])/(x_2 - x_0), \ \dots$$

36.4 Random UPol

DoCon declares

```
instance (CommutativeRing a, Random a) => Random (UPol a)
```

```
where
                -- put a random polynomial "between l and h" to have random
                -- coefficients "between"  coef(i,l)  and  coef(i,h),
                -- for each  i <- [0 .. (max (deg l) (deg h)]
randomR (l,h) g =
           let  d      = succ $ maximum $ map (deg0 '_' 0) [l,h]
                [u,v]  = map (Vec . pToVec d) [l,h]
                (w,g') = randomR (u,v) g
           in   (pFromVec l $ vecRepr w, g')
```

Example of usage.
Make the list of random polynomials `f` $\in$ `Z[x]`, `deg f` $\leq$ `2`,
`f` having coefficients in the list `[-2 .. 6]` :

```
ps $ mkStdGen 0
  where
  ps g = f:(ps g')  where  (f,g') = randomR (l,h) g
  p1   = cToUPol "x" dZ 1
  l = ct p1 [(-2,2),(-2,1),(-2,0) :: UMon Z]    -- convert monomial list
  h = ct p1 [(6 ,2),(6 ,1),(6 ,0) :: UMon Z]    -- to polynomial
```

Further, to put, for example, the restriction
"coefficient c_1 of degree 1 is 0", the programmer has to skip the second monomial in the lists `l, h`;
and the restriction $c_1 = 9$ is obtained by setting the second monomial `(9,1)` in `l` and in `h`.

36.5 Advanced methods for univariate polynomial

36.5.1 GCDRing, FactorizationRing, LinSolvRing

These instances refer to the possibilities of
polynomial GCD, factorization, Gröbner basis,
and they are done as in the generic case of multivariate polynomial: see Section 38.5.

Example

For the rings `T = Z[t]`, `X = K[x]` and a field `K = Z/(p)`,
form some polynomials `f, g` from `T`, `h` from `X`, and find
`f'` = derivative of `f`, `gcd f f'`,
Gröbner basis of `[f,f',g]` over Z, `factor h` over K.

```
let  t1      = cToUPol "t" dZ (1 :: Z)       -- unity of T
     t       = varP 1 t1
     [t2,t3] = map (fromi t1) [2,3]
     f       = (t2*t + t1)^2 * (t - t1)
     g       = t2*t^2 + t2*t + t3
     f'      = pDeriv [(1,1)] f

     p  = 7 :: Z
     iI = eucIdeal "bef" p [] [] []          -- ideal (p) in Z
     k1 = Rse 1 iI dZ                        -- unity of K
     dK = upField k1 Map.empty                     -- domain description
     x1 = cToUPol "x" dK k1
     h  = smParse x1 "(x^2 + x + 3)*(x^14 + 2)"
in
(gcD [f,f'],  fst $ gxBasis [f,f',g],  factor h)
```

More examples can be found in `demotest/T_polArit_.hs, T_finfield.hs`.

36.5.2 Hensel lift

```
henselLift ::                              -- so far
  (EuclideanRing a, FactorizationRing a, Ring (ResidueE a))
  =>
  UPol a -> UPol a -> UPol a -> Maybe (UPol a) -> a -> a -> Z -> Z ->
  -- f      h1        g1               v1          p    p^n  n    m

                                                 (UPol a, UPol a, UPol a, a)
                                                  -- h'   g'      v'      p^m
```

Denote $F_k = a/(p^k)$, $r_k : a[x] \longrightarrow F_k[x]$ natural projection modulo p^k.

Given

- a square free `f` from $A[x]$,
- a prime `p` from A such that `p` does not divide `resultant f f'` (hence `deg` r_1`(f) = deg f`, `and` r_1`(f)` is square free),
- `h1, g1, v1, n` such that `f - h1*g1` $\in$ `(p^n)`, `v1 = (f - h1*g1)/(p^n)`, `0 < l = deg h1 < deg f`, `lc h1 = 1`,
- `m` $\geq$ `n`

find `(h', g', v', p^m)` such that

`f - h'*g'` $\in$ `(p^m)`, `v' = (f - h'*g')/(p^m)`,

`deg h' = deg h1`, `deg g' = deg g1`, `lc h' = 1`, `|h'|, |g'| < n·|p|` .

The algorithm cost is bounded by some degree of `|p|·(deg f)`.

36.5.3 Extension of finite field

Given a finite field k and degree $d > 1$, `extendFieldToDeg` builds an extension field F over k of dimension d.

DoCon does this by testing certain list of polynomials over k for primality. The cost is bounded with $O(d^3 \cdot |k|^{d+1})$. It is usable for small d.

```
extendFieldToDeg :: Field k => UPol k -> Domains1 (UPol k) -> Z ->
                               -- s      sDom                 d

                            (ResidueE (UPol k), Domains1 (ResidueE (UPol k)))
                             -- u               domF
```

`k` is given by the sample polynomial `s` over `k`.

`sDom` is the domain description for `s`, setting it with `Map.empty` will cause its forming by new with `upEucRing` function.

`u` is the unity of the reslt field `F = k[x]/(p)`, where `p` $\in$ `k[x]` is the found irreducible polynomial of sample `s`, `deg r = d`.

`domF` is the domain description (bundle) for the result field `F`.

This function is exported from the module `Pol`, implemented in `Pfact1_.hs`.

36.5.4 Determinant over $k[x]$ for a finite field k

For the large data, it is computed much cheaper by interpolation, than by the Gauss elimination. Sometimes, it requires to build first certain extension K of k (`extendFieldToDeg` applied), and then, interpolate $\det M$ over K, and project the result to k. Still, the cost is very low. See the module `demotest/T_detinterp.hs`.

```
det_upol_finField :: Field k => [ResidueE (UPol k)] -> [[UPol k]] -> UPol k
                                -- ext                 mM
```

Certain degree cost method to compute $\det M$ over a domain $k[x]$ for a finite field k. For $M = (f_{i,j} \ldots)$ of size m, $r_i = \max\{\deg f_{i,j} \ldots\}$, $r = \sum_i [r_i | \ldots]$, $\deg \det(M) \leq r$, the cost of $\det$ is bounded by $O(r^4 m^3 |k|^2)$.

This function is exported from the modle `Pol`, implemented in `Pfact1_.hs`.

37 Power product

37.1 Preface

The ordered additive group for the domain `PowerProduct = Vector Z` expresses the ordered multiplicative group of monomials of kind $x_1^{i_1} \dots x_n^{i_n}$, $i_k \geq 0$.

So, the arithmetic for the power products has to be defined, and several most usable admissible comparisons. The arithmetic (`OrderedAddGroup`) is induced by the `Vector` instances. It remains to implement other useful items.

They are exported by the `Pol` module, and implemented in `PP_.hs`.

37.2 Definitions

```
type PowerProduct = Vector Z

-- CAUTION:  Vec ns   can serve as a power product only if
--                    ns = [n(1)...n(k)]  is non-empty  and  n(i) >= 0

type PPComp = Comparison PowerProduct

isMonicPP :: PowerProduct -> Bool
isMonicPP =  (< 2) . genericLength . filter (/= 0) . vecRepr
                                -- corresponds to monomial  xi^ki,  ki >= 0

vecMax, ppLcm :: Ord a => Vector a -> Vector a -> Vector a
vecMax v = Vec . zipWith max (vecRepr v) . vecRepr
ppLcm = vecMax              -- Least Common Multiple of the power products

ppComplement :: PowerProduct -> PowerProduct -> PowerProduct
ppComplement    u              v            =  (ppLcm u v) - u

  -- Examples:  ppLcm (Vec [1,0,2]) (Vec [0,1,3]) -->  Vec [1,1,3]
  --
  --            let {p = Vec [1,0,1,1,1,0];  q = Vec [0,1,2,1,0,0]}
  --            in  (ppComplement p q, ppComplement q p)
  --            -->
  --            (Vec [0,1,1,0,0,0], Vec [1,0,0,0,1,0])

vecMutPrime :: AddMonoid a => Vector a -> Vector a -> Bool
vecMutPrime                   v        =
  case  zeroS $ vecHead v
  of
    z -> and . zipWith (\x y -> (x==z||y==z)) (vecRepr v) . vecRepr
```

```
ppMutPrime, ppDivides :: PowerProduct -> PowerProduct -> Bool

ppMutPrime = vecMutPrime          -- "power products are Mutually Prime"

ppDivides p q = all (>= 0) $ vecRepr (q-p)       -- "p divides q"
```

Some usable admissible comparisons for power products

Admissible pp-comparison `cp` means it is agreed with the multiplication by monomial, that is for the relation ((<) expressed by condition `cp u v == LT`,

(1) $zeroVector < v$ for any non-zero v,

(2) $u < v \;\Rightarrow\; u + w < v + w$ for any power products u, v, w.

Different cp-s define different representations for isomorphic polynomial rings and express various *gradings* on the polynomial algebra. Choosing power product comparison is important for the computational tasks with polynomials. `DoCon` pre-defines the following most usable orderings:

`lexComp, lexFromEnd, degLex, degRevLexppComp_blockwise`

Looking at their definitions, the user can easily define one's own ordering and transmit it to polynomials. Also keep in mind that for any admissible ordering `cp`

(1) `cp` can be defined by a *flag*, that is a matrix M over real numbers. This means that `cp p q` compares first $\mathtt{f}_1\mathtt{(p)}$, $\mathtt{f}_1\mathtt{(q)}$, where $\mathtt{f}_1$ is the linear functional defined by the first row of M, the rows $\mathtt{f}_1$, $\mathtt{f}_2$, ... are applied until the comparison is solved.

(2) For any finite set of the power products (and this is a common case for the computational tasks), M is equivalent to some integer matrix with non-negative elements.

```
lexComp, lexFromEnd, degLex, degRevLex :: PPComp

lexComp    v = lexListComp compare (vecRepr v) . vecRepr
lexFromEnd v = lexListComp compare (reverse $ vecRepr v) . reverse . vecRepr

degLex p@(Vec ns) q@(Vec ms) = case compare (sum ns) (sum ms) of EQ -> lexComp p q
                                                                  v  -> v
degRevLex  p@(Vec ns) q@(Vec ms) = case compare (sum ns) (sum ms) of
                                                               EQ -> lexFromEnd q p
                                                               v  -> v
  -- degRevLex  is not so trivial.  For example, for  n = 3
  -- it is presented by the matrix [u1,-e3,-e2] = [[1, 1, 1],
  --                                                [0, 0,-1],
  --                                                [0,-1, 0]
  --                                               ]
```

```
ppComp_blockwise :: Z -> PPComp -> PPComp -> PowerProduct -> PowerProduct -> CompValue
                 -- m    cp        cp'       p               q
  -- compare p q according by the direct sum of comparisons  cp, cp':
  -- first  cp  compares the vectors of the first  m  items from  p, q,
  -- then, if equal, the rests are compared by  cp'.
```

37.3 PP Ordering description

These items are exported by the `Pol` module, implemented in `UPol_.hs`.

```
type PPOId     = (String, Z)
type PPOrdTerm = (PPOId, PPComp, [[Z]])
```

The power product ordering description (term) `(id, cp, ws)` consists of the identifier `id`, comparison function `cp`, and a list `ws` of integer *weights*. `[]` for `ws` means the weights are not given.

For most purposes, it suffices to set this term, for example, like this:

```
(("dlex", 3), degLex, [])
```

`id` is set by the programmer (in addition to var list) in order to identify the domain parameter for polynomial. DoCon looks into `id` only when trying to find whether the two polynomials are under the same pp-ordering — this may be useful, for example, for the conversion between domains.

```
ppoId      = tuple31   -- extracting parts of PPOrdTerm
ppoComp    = tuple32   --
ppoWeights = tuple33   --

lexPPO :: Z -> PPOrdTerm                -- most usable ppo is  lexPPO n
lexPPO    i =  (("lex", i), lexComp, [])
```

Why do we need the pp ordering description to add to the comparison function? Because sometimes one needs the 'space' of all admissible orderings, to find there the ordering with certain property. In this case, it is natural to represent a pp ordering as say an integer matrix.

38 Multivariate Polynomial

For the polynomials, everything is exported from the `Pol` module. Most of implementation is in the subtree `source/pol/*`

38.1 Representation

```
type Mon a = (a, PowerProduct)              -- multivariate monomial

data Pol a = Pol [Mon a] a PPOrdTerm [PolVar] (Domains1 a)
                                                  -- (multivariate) polynomial
instance Dom Pol  where  dom    (Pol _ _ _ _ d) = d
                         sample (Pol _ c _ _ _) = c

instance Eq a => Eq (Pol a) where  f==g = (polMons f)==(polMons g)

-- extracting parts of (Pol ..)

polMons       :: Pol a -> [Mon a]
polPPOId      :: Pol a -> PPOId
polPPComp     :: Pol a -> PPComp
polPPOWeights :: Pol a -> [[Z]]

polMons (Pol ms _  _  _  _ ) = ms

polPPOId      = ppoId      . pPPO
polPPComp     = ppoComp    . pPPO
polPPOWeights = ppoWeights . pPPO
```

A polynomial `Pol mons c o vars aD`
contains the

- monomial list `mons` sorted decreasingly under the pp-ordering `o`, zero coefficient monomials skipped, each exponent is a vector of same size `n = length vars`,
- sample coefficient `c`,
- admissible pp ordering description `o`,
- finite list `vars` of variables (indeterminates),
- description `aD` of the coefficient domain.

— see (Section 3.4 DF), 3.5.10.

The equality of polynomials means only the equality of the `mons` lists. The domain of polynomials is parameterized by the

(1) coefficient domain `aD`, (2) number `n = length vars` of variables,

(3) pp ordering `o`.

For example, different `o` correspond to different membership functions for the polynomial set, because the monomial list is sorted by `o`.

Warning:
`DoCon` is not safe against mixing of polynomials with different domain parameters, or against mixing of exponents of different size in one polynomial.

Thus, evaluating `(Pol mons 0 o1 _ _) + (Pol mons 0 o2 _ _)` with different comparisons in `o1, o2` may yield incorrect result.

Naturally, the constructors may compose. For example, the types
`UPol (UPol a)` and `UPol (Pol (UPol a))`
represent (together with sample polynomials) the domains
$a[x][y]$ and $((a[x])[y_1, \dots, y_n])[z]$ respectively.

Examples

For $R = Z[x, y, z, u]$ and lexicographic pp ordering, $f = 2xz^2 + y^4$
is represented internally as

```
Pol [(2, Vec [1,0,2,0]), (1, Vec [0,4,0,0])]  0 (lexPPO 4) ["x","y","z","u"] dZ
```

Under the `degLex` comparison, it represents as

```
Pol [(1, Vec [0,4,0,0]), (2, Vec [1,0,2,0])]  0 (...) ["x","y","z","u"] dZ
```

38.2 How to build polynomials

Apply `cToPol` to obtain an initial sample `s` for a polynomial domain $P = a[x_1, \dots, x_n]$. Then, apply the casting maps

`fromi s,   ct s,  ctr s,   smParse s`

to obtain a polynomial from integer, coefficient or monomial(s), string respectively.

Apply also the map of variables `varPs 1 s -->` [x_1',. . . ,x_n'],
with x'_i are all the variables as polynomials from P, and then, combine x'_i in arithmetic expressions.

Example:
Form the polynomials $5/6$, $-2x^2y + 3y^4 + (1/2)x + y$, $7x + 8y$ in `Q[x,y]`,
`Q = Fraction Z`, with the degree-lexicographic ordering; also form a polynomial in `Q[x,y]` from the given monomial `m = (a,p)` :

```
f (a,p) = let  o          = (("degLex",2), degLex, [])
               uQ         = 1:/1 :: Fraction Z
               dQ         = upField uQ Map.empty
               p1         = cToPol o ["x","y"] dQ uQ
               [x,y]      = varPs uQ p1;
               [p2,p3,p4] = map (fromi p1) [2,3,4]
               f56        = 5:/6 :: Fraction Z
        in
        (ct p1 f56,              -p2*x^2*y + p3*y^4 + (p1/p2)*x + y,
         smParse p1 "7*x + 8*y",  ctr p1 (a,p)
        )
```

Remarks. Applying `smParse` may cost a couple of times more than the rest of this program.

`ctr p1 (a,p)` differs from `ct p1 (a,p)` in the case when `a = 0` in that `ctr` also tests for zero.

See the `Cast` instances for the constructors `Pol, UPol, RPol` and others.

Apply similar approach to the `UPol, RPol` constructors.

38.3 Usable items for polynomials

```
type PolPol a = Pol (Pol a)

instance AddGroup a => Cast (Pol a) (Mon a)           -- from monomial
  where
  cast mode (Pol _ c o v d) (a,p) = Pol mons c o v d
                         where
                         mons = if mode=='r' && isZero a then [] else [(a,p)]

instance AddGroup a => Cast (Pol a) [Mon a]           -- from monomial list
  where
  cast mode (Pol _ c o v d) mons = Pol ms c o v d
    where                                            -- order is NOT checked
    ms = if mode/='r' then mons  else filter ((/= z) . fst) mons
    z  = zeroS c

instance Ring a => Cast (Pol a) a                       -- from coefficient
  where
  cast mode (Pol _ _ o vs d) a = case (mode, isZero a, Vec $ map (const 0) vs)
                                 of
                                   ('r', True, _ ) -> Pol []       a o vs d
                                   (_  , _    , pp) -> Pol [(a,pp)] a o vs d
--------------------------------------------------------------------
instance PolLike Pol        -- see first class PolLike, instance PolLike UPol
  where
  pIsConst f = case polMons f of (_,p):_ -> all (==0) $ vecRepr p
```

```
                               _       -> True
  pPPO  (Pol _  _  o  _  _ ) = o
  pVars (Pol _  _  _  vs _ ) = vs

  lm  f   = case polMons f of m:_ -> m
                             _   -> error ...
  lpp f   = snd $ lm f
  pDeriv  = deriv_
  pCoefs  = map fst . polMons
  pTail f = case polMons f of _:ms -> ct f ms
                             _    -> error ...
  pFreeCoef (Pol mons c _ _ _) =  ... similar to UPol
  pCoef f js =           ...          -- coefficient of power product  Vec js

  ldeg f = case polMons f of (_,Vec js):_ -> sum1 js
                            _            -> error ...

  deg f = case  map (sum1 . vecRepr . snd) $ polMons f  of [] -> error ...
                                                          ds -> maximum ds
  degInVar for0 i f = case (i >= 0, polMons f) of

    (False, _ ) -> error (..."positive i needed \n")
    (_    , []) -> for0
    (_    , ms) -> maximum $ map (ith . vecRepr . snd) ms  where ith js = ...

  pMapCoef mode f g = cast mode g [(f a, pp) | (a,pp) <- polMons g]
  pMapPP f g     = ct g [(a, Vec $ f js) | (a, Vec js) <- polMons g]

  pCDiv f c = case  unzip $ polMons f  of ... -- similar to UPol
  varPs a f = [ct f (a, Vec js) | js <- scalarMt (pVars f) 1 (0::Z)]
    -- because
    -- variable's power product is a row in the unity matrix of size |vars|

  pDivRem     = ... -- similar to UPol - see comments to class PolLike
  pValue f cs = ... -- substitute  xi = ci  into  f

leastMon :: Set a => Pol a -> Mon a
leastMon f = case polMons f
             of
               m:ms -> last (m:ms)
               _    -> error $ ("leastMon 0  in  R"++) $ shows (pVars f) $
                               (",\nR = "++) $ showsDomOf (sample f) "\n"

reordPol :: PPOrdTerm -> Pol a -> Pol a        -- bring to given pp ordering
reordPol ppo (Pol ms c _ vars dom) =
            Pol (sortBy cmp ms) c ppo vars dom  where
```

```
                                              cmp (_,p) (_,q) = cp q p
                                              cp              = ppoComp ppo
fromUPol :: UPol a -> Pol a
fromUPol (UPol ms c v d) = Pol [(a,Vec [p]) | (a,p) <- ms] c (lexPPO 1) [v] d
  --
  -- to convert from univariate polynomial means to make a vector of size 1
  -- from each exponent, to set the variable list [v] and lexPPO ordering

toUPol :: Ring a => Pol a -> UPol a
  -- to convert to univariate polynomial means to remove the pp-ordering
  -- term,  to take the head variable only and the head of each power
  -- product,  to order obtained monomials by degree,
  -- to sum up the monomials of repeating degree

monMul :: Ring a =>  a -> Mon a -> Mon a -> [Mon a]
                   --zero
monMul z (a,p) (b,q) = case a*b of c -> if c==z then [] else [(c,p+q)]

mPolMul :: Ring a => Mon a -> Pol a -> Pol a
mPolMul              (a,p)    f     =  ctr f [(a*b,p+q) | (b,q) <- polMons f]

monLcm :: GCDRing a => Mon a -> Mon a -> Mon a
  -- Lcm of monomials over a gcd-ring.
  -- For the  field  case, it is better not to call monLcm,
  -- but to set directly  (unity a, ppLcm...)

monLcm (a,p) (b,q) = case gcD [a,b] of g -> (a*(b/g), ppLcm p q)
--------------------------------------------------------------------
cToPol :: Ring a => PPOrdTerm -> [PolVar] -> Domains1 a -> a -> Pol a
cToPol              ord          vars        aDom          a =
  -- Coefficient --> polynomial.
  -- Applying cToPol is the ONLY natural way to initiate a sample
  -- polynomial.
  case  Vec $ map (const 0) vars  -- power product for  x1^0*..*xn^0,  xi<- vars
  of
    p -> if a==(zeroS a) then Pol []      a ord vars aDom
         else                 Pol [(a,p)] a ord vars aDom

headVarPol :: CommutativeRing a => Domains1 (Pol a) -> Pol a -> UPol (Pol a)
                                   -- pDom             f        g
```

Bring polynomial to head variable: $a[x_1,x_2,\ldots,x_n] \longrightarrow PP' = P'[x1]$, $P' = a[x_2,\ldots,x_n]$, $n > 1$. This is `only` for the `lexComp` ordering on $a[x_1,x_2,\ldots,x_n]$, $a[x_2,\ldots,x_n]$.

New `PPOId` for $a[x_2,\ldots,x_n]$ is `("lex", n-1)`.

`pDom` is the domain description for P'.

How to prepare `pDom` ?

Starting from the empty, it may be, say `up`C `a Map.empty`, with C the strongest declared class possible in environment.

```
fromHeadVarPol ::  UPol (Pol a) -> Pol a
```

$(a[x_2,\ldots,x_n])[x1] \longrightarrow a[x_1,x_2,\ldots,x_n], \quad n>1.$
Inverse to `headVarPol`. It is for the `lexComp` ordering *only*.

New `PPOId` is `("lex", n)`.

```
toOverHeadVar :: CommutativeRing a => Domains1 (UPol a) -> Pol a -> Pol (UPol a)
                                      -- dX1
```

Bring polynomial to tail variables: $a[x_1,x_2,\ldots,x_n] \longrightarrow (a[x1])[x_2,\ldots,x_n]$.

$n>1$ is required, and only `lexComp` ordering is considered for the power products of $[x_1,x_2,\ldots,x_n]$, $[x_2,\ldots,x_n]$.

New `PPOId` for $[x_2,\ldots,x_n]$ is `("lex", n-1)`.

`dX1` is the domain description for $a[x1]$.

```
fromOverHeadVar ::  CommutativeRing a => Pol (UPol a) -> Pol a
```

$(a[y])[x_1,\ldots,x_n] \longrightarrow a[y,x_1,\ldots,x_n]$. This is inverse to `toOverHeadVar`.

Only `lexComp` ordering is considered for the power products.

```
data PPCoefRelationMode = HeadPPRelatesCoef | TailPPRelatesCoef
                                              deriving (Eq, Ord, Enum, Show)
```

This is the *mode* for the functions `toPolOverPol`, `fromPolOverPol` shown below.

```
toPolOverPol :: CommutativeRing a => PPCoefRelationMode ->      -- mode
                                     Natural            ->      -- n
                                     PPOrdTerm          ->      -- coefO
                                     PPOrdTerm          ->      -- ppO
                                     Pol a              ->      -- f
                                     Pol (Pol a)
```

For `f` from `a[xs  ys]`,

if `mode = HeadPPRelatesCoef` then it embeds `f` to `a[xs][ys]`,

otherwise, embeds to `a[ys][xs]`.

Here `n = length ys`,
`coefO, ppO` is the pp-ordering for the coef-part and the power-product part in `a[coefVars][pVars]` respectively.

The new domain bundles are supported as `upRing`.

```
fromPolOverPol ::
  CommutativeRing a => PPCoefRelationMode -> PPOrdTerm -> Pol (Pol a) -> Pol a
                       -- mode               ppo          f
```

If `mode = HeadPPRelatesCoef` then it embeds from `a[xs][ys]` to `a[xs ys]`, otherwise, it embeds to `a[ys xs]`.

`ppo` is the pp-ordering for the result.

```
polToHomogForms :: (AddGroup a, Eq b) => (PowerProduct -> b) -> Pol a -> [Pol a]
                                         -- weight
polToHomogForms w f =
          map (ct f) $ partitionN (\ (_,p) (_,q) -> (w p)==(w q)) $ polMons f
```

(non-ordered) list of homogeneous forms of polynomial over `a` with respect to `weight :: PowerProduct -> b`

```
addVarsPol :: Char -> PPOrdTerm -> [PolVar] -> Pol a -> Pol a
```

Embed from $a[x_1, \ldots, x_n]$ to $a[y_1, \ldots, y_m, x_1, \ldots, x_n]$ or to $a[x_1, \ldots, x_n y_1, \ldots, y_m]$ by prepending/appending the variable list `vars' =` $[y_1, \ldots, y_m]$ to `vars` and extending exponents with zeroes.

Caution: the new pp-order term `ord'` is required which must agree to the old one: `(cp' restricted to` $y_1 = \ldots = y_m = 0$`) == cp`, and the weights agreed too.

`mode = 'h'` means prepending `vars'` to head of `vars`,

any other value means appending to tail.

```
coefsToPol :: Ring a => Char -> Pol a -> [Z] -> [a] -> (Pol a, [a])
                        --mode  sample   degs   cs
```

Convert coefficient list to polynomial of given sample.

Coefficients zip with the power products from the cube
`[[`$k_1, \ldots, k_n$`] |` $0 \le k_i \le d_i$, $d_i \in$ `degs]` listed in the `lexComp` order.

Then, the zero coefficient monomials are filtered out.

`mode = 'l'` means the sample is under `lexComp`, in this case the final re-ordering is skipped.

```
polDegs ::   [Z] -> Pol a -> [Z]
          -- zdegs  f
```

Returns the degrees in each variable, `zdegs` for zero `f`.

Example:
for `f` $= x^2z + xz^4 + 1 \in Z[x, y, z]$, `polDegs [] f = [2,0,4]`

```
polPermuteVars ::  AddGroup a => [Z] -> Pol a -> Pol a
```

Substitution for polynomial variables $[x_1, \ldots, x_n]$ given by permutation at `[1 .. n]`.
Monomial list reorders, but the variable list remains.
Example: `[2,1]` $\longrightarrow$ $x^2y + xy^3$ $\longrightarrow$ $x^3y + xy^2$

```
polValueInCommRing ::
          (Ring a, CommutativeRing r) => Char -> (a -> r) -> [r] -> Pol a -> r
                                      -- mode    cMap        rs     f
```

This is value of `f` in the given commutative ring (a more generic variant of substitution in a polynomial):
map the coefficients in by `cMap` and return the result expression in `r` for the substitution `x(i) := r(i),  x(i) from  xs = pVars f,  r(i) from rs`.
It must hold `length xs` $\leq$ `length rs`, and the remainder of `rs` is ignored.
`mode = 'l'` means `f` has the `lexComp` ordering,
in this case the evaluations would be somewhat cheaper,
any other letter means generic case.
Method.
`f` converts recursively to $R[x_2, \ldots, x_n][x1]$, and so on, and the Horner scheme of substitution is applied by each x_i.

```
polSubst :: CommutativeRing a => Char -> Pol a -> [Pol a] -> [[Pol a]] -> Pol a
                              -- mode    f        gs         powerLists
```

(Use better `polValueInCommRing` !)
Substitute polynomials `gs = `$[g_1, \ldots, g_m]$ $\in R[x_1, \ldots, x_n]$
for the variables $x_1, \ldots, x_m$ in a polynomial `f` $\in R[x_1, \ldots, x_n]$.
The rest of x_i remain. If $m > n$ then the rest of g_i are ignored.
Method.
`f` converts recursively to $R[x_2, \ldots, x_n][x1]$, and so on, and the Horner scheme of substitution is applied by each x_i.
`mode = 'l'` means `f` has the `lexComp` ordering,
in this case the evaluations would be somewhat cheaper,
any other letter means generic case.
`powerLists` is either `[]` or $[g_1Powers, \ldots, g_kPowers]$, where $g_iPowers$ is either `[]` or the infinite list `[`g_i^2, g_i^3 `...]`.
`powersLists = []` means the powers are not listed at all, compute them by the Horner scheme.

$g_i Powers$ are ignored for $i > m$.

$k < m$ means `giPowers = []` for $i > m$.

$g_i Powers$ `= []` means again, to compute g_i^j by the Horner scheme.

```
sPol :: GCDRing a => Pol a -> Pol a -> (Pol a, Mon a, Mon a)
                                       -- sp  m1     m2
```

S-polynomial (the one related to Gröbner bases) for non-zero polynomials:
`sp = m1*f - m2*g`.

It also returns the corresponding complementary monomial factors `m1`, `m2`.

```
type PVecP a = (Pol a, [Pol a])
```

Polynomial-vector-polynomial `fv = (f,v)` is supplied with the polynomial pre-vector `v`.

`v` is often used to accumulate the coefficients f_i of some f — relatively to some initial basis gs : $f = \sum_1^m f_i g_i$.

This serves accumulating of the *quotient* (transformation) part in
the functions `moduloBasis, gBasis`.

```
mPVecPMul :: Ring a => Mon a -> PVecP a -> PVecP a
mPVecPMul              m        (f,gs)  =  (mPolMul m f, map (mPolMul m) gs)

sPVecP :: GCDRing a => PVecP a -> PVecP a -> (PVecP a,Mon a,Mon a)
sPVecP                 (f,us)     (g,vs) =   -- s-polynomial for  PVecP a
  let  (_,m1,m2) = sPol f g
       pdiff     = Pol_.sub_ (mPolMul m1 f) (mPolMul m2 g)

       (ps, qs)  = (map (mPolMul m1) us, map (mPolMul m2) vs)
       vdiff     = (zipWith Pol_.sub_ ps qs)
  in   ((pdiff,vdiff), m1, m2)
```

38.4 Random Pol

Similarly to instance of `Random UPol` (Section 36.4), `DoCon` declares

```
instance (CommutativeRing a, Random a) => Random (Pol a) where
                                                  randomR (l,h) g = ...
```

It puts a random polynomial `f` distributed uniformly in the "segment between polynomials `l` and `h`" to have random coefficients `coef(pp)` between `coef(pp,l)` and `coef(pp,h)` for all the power products `pp` such that `coef(pp,l)` $\neq$ `0` or `coef(pp,h)` $\neq$ `0`.

Example: the polynomials from `rands` $(3x^4y^3 + 2xy - 2,\ xy^2 - xy)$ `g`
may have the coefficients

`a <- [0..3]` for `pp = [4,3]`, `b <- [0..1]` for `[1,2]`,

`c <- [-1..2]` for `[1,1]`, `d <- [-2..0]` for `[0,0]`,

and `0` — for all other power products `pp`.

38.5 Advanced methods for Pol

The `GCDRing, FactorizationRing, LinSolvRing` instances for `Pol` refer to the functions of polynomial GCD, factorization, Gröbner basis, Their algorithms are set as follows.

38.5.1 Polynomial GCD

`DoCon` defines `instance GCDRing a => GCDRing (Pol a) where ...`
But its methods

`canInv, canAssoc` have sense when `a` possesses `(WithCanAssoc, Yes)`,

`gcD` has sense when `a` possesses `(IsGCDRing, Yes)`

— see Section 15.3.

`canInv, canAssoc` are defined via the (division by) `canInv(lc(f))`.

For gcd, the simplest method with pseudodivision is applied, so far: ([Kn] Volum 2, Section 4.6.1).

`headVarPol` maps `f, g` from $a[x_1, x_2, \ldots, x_n]$ to $a[x_2, \ldots, x_n][x1]$, `f, g` reduced to primitives, then, gcd for the primitive `f, g` $\in R[x]$ is found for a gcd-ring $R = a[x_2, \ldots, x_n]$ via the pseudodivision.

The implementation notes are in `Pgcd_.hs`.

38.5.2 Factorization in $k[x, y]$

DoCon can do it for any finite field k, presented as $k = F[t]/(g)$, g a generator over a prime field F. Such representation can always be found.

The residue ring here is represented by the `ResidueE` constructor.

For $k[x]$, the usual (generalized) Berlekamp's method is applied, with the cost bounded by $O((\deg f)^3)$.

For $k[x, y]$, the method is adopted from [CG]. See [Me3]. It allows the cost bound in deg_x, deg_y with the main part of $O((deg_x f)^4 * (deg_y f)^3)$.

See `demotest/T_pfactor.hs` for the examples of factorization.

38.5.3 Gröbner basis, syzygies

The corresponding items are exported by the modules
`RingModule (LinSolvRing(..)), GBasis`
and implemented in `PolNF_, GBasFld_, GBasEuc_, Polrel_`.

```
instance (EuclideanRing a) => LinSolvRing (Pol a) where ...
```

is defined as follows.

`gxBasis` is the (weak) reduced Gröbner basis for polynomials over an Euclidean ring `a`.

`syzygyGens` is the syzygy generators found via the Gröbner basis function.

`moduloBasis mode fs` is either `polNF _ fs` — Gröbner reduction by the Gröbner basis `fs` over an Euclidean ring, or composition of `gs = fst $ gBasis fs` and `polNF _ gs`.

`(Euclidean, Yes)` is sufficient for the necessary properties of the syzygy basis finding and the ideal inclusion solvability.

For the canonical reduction modulo the ideal (presented by some fixed basis), it is required `(DivRemCan, Yes)` for the base domain of `a`. That is the canonical remainder property for `remEuc 'c'`.
See (Section 19 DivRemCan), 18.1.

For example, DoCon supports `(DivRemCan, Yes)` for `Integer`, and $k[x]$ for a field k in a standard way.

{g} Gröbner basis

For `P = Euclidean a => Pol a, fs :: [P], gxBasis fs --> (gs,mt)` possesses the generic properties required by the `LinSolvModule` category and the following specific ones. `gs` is the reduced Gröbner basis `gBasis` for the polynomials `fs` over an Euclidean ring `a`.

In particular, `a` may be a field. When `a` is a field, `gBasis` builds the (strong) reduced Gröbner basis which is the unique representation of the *ideal*.

If `a` is not a field, returned is a *weak* reduced Gröbner basis. So far, this is arranged so only because it is simpler to obtain the canonical remainder by `gs` applying the `polNF` reduction with the `"c"` mode.

`gs` is ordered increasingly by lpp by the pp-comparison contained in f_i.

For a field `a`, and more generally, for `a` possessing `(WithCanAssoc, Yes)`, g_i are in the canonical associated form.

The matrix `mt` is also accumulated such that
`mt*(transpose [fs]) == transpose [gs]` (the requirement of LinSolvModule).

Zero elements of `fs` correspond to the zero columns in `mt`.

In the case of all `fs` being zeroes, the result is `([],[])`.

Method.
`isField` is tested first for the base ring of `a`. Then, for `(IsField, Yes)`, certain adaptation of the method from [GM] is implemented in `GBasFld_.hs` for the strong reduced Gröbner basis. See the comments there.

Otherwise, the weak Gröbner basis over a generic Euclidean ring is found. The method is adopted from [Mo]. See the comments in `GBasEuc_.hs`.

Reduction from `gxBasis` to real method

`gxBasis` for $f_i \in a[x_1, \ldots, x_n]$ allows also zeroes in the given list, and it corrects the transformation matrix respectively.

Also it calls the Gröbner basis for the (more generic) e-polynomials.

Examples: see `demotest/T_grbas1.hs, T_grbas2.hs`.

<u>{ig} isGBasis</u>

```
isGBasis :: EuclideanRing a => [Pol a] -> Bool
```

"`fs` is a Gröbner basis". For `a` not a field, it means a weak Gröbner basis.

Method.

Some of the s-polynomials `sPol` f_i f_j are formed and reduced with the Gröbner normal form `polNF` by `fs`. Optimizations are applied to test possibly smaller set of pairs — the same optimization as in `gBasis`.

See `isGBasis* in GBas_, GBasFld_.hs, GBasEuc_.hs` for the implementation comments.

<u>{s} Syzygy generators</u>

For `fs :: [P], P = Euclidean a => Pol a, syzygyGens mode fs --> [rels]` gives the list of linear relations (syzygies) on `fs` (of type `[P]`) that generates the module of all their syzygies.

For this polynomial case,

`mode = "g"` means that `fs` is a (weak reduced) Gröbner basis, in this case the evaluation would be more direct.

`""` means the generic case.

For the weak Gröbner basis `fs` over a *field* and `ecpTOP_weight` ordering [MoM] on the module P^n generated by the weights `lpp(`f_i`)`, $f_i \in$ `fs`, the result *is also a weak Gröbner basis* of the submodule [MoM].

(does this also hold for the Euclidean coefficients?)

Caution. We are not sure for the case when `a` is not a field: maybe, the method is correct, but we had not dealt with the proof.

Method

For the `"g"` mode, the needed generators `rels` are obtained as the quotient results of the `sPol` f_i f_j reduction to zero by `fs` with the Gröbner normal form `polNF`.

For other mode, `gBasis fs --> (gs,mt)` finds the Gröbner basis and the transformation matrix `mt`; `syzygyGens "g" gs --> relsG` evaluate;
then, `rels` is composed via `relsG` and `mt` in a certain simple way — see [Bu].

See `Polrel_.hs` for the implementation comments.

Examples: see `demotest/T_grbas2.hs` {n} Gröbner normal form

For `P = Euclidean a => Pol a,   fs ::  [P]`
a weak Gröbner basis `moduloBasis mode fs f --> (rem, qs)`
is the reduction of `f` by `Ideal(fs)` with the generic properties required by `LinSolvRing`.

In this particular `Pol` case, it applies the Gröbner normal form `polNF`.

But if `fs` is a Gröbner basis, then better ignore `polNF` and apply `moduloBasis mode` — with `mode = "g"` or `"cg"`.

If `fs` is not a Gröbner basis, then `moduloBasis _ fs` and `polNF _ fs` do very different things. The former finds the Gröbner basis `gs` and applies `polNF _ gs`.

While `polNF _ fs f` reduces `f` directly to normal form. And in this latter case, the `polNF` reduction does not necessarily detaches the ideal. `polNF` is applied repeatedly when the Gröbner basis is computed.

```
polNF :: EuclideanRing a => String -> [Pol a] -> Pol a -> (Pol a, [Pol a])
                            -- mode   gs         f        rem     qs
```

The following properties hold.

If `gs` is a weak Gröbner basis [Mo], then

`(fst $ polNF anyMode gs f) == 0` $\iff$ `f` $\in$ `I = Ideal(gs)`
and if also `a` is a c-Euclidean ring, then `fst . polNF "c" gs` is a canonical map modulo `I`.

`mode = tailMode++cMode`, the order is immaterial, also `"c"` is equivalent to `"cr"`, `"rc"`.

`tailMode` can be `""` or `"r"`.
`""` means to reduce the head monomials only, `"r"` — to reduce the tail too.

`cMode` is the mode for the coefficient reduction in moduloBasis; it is processed only when the current `(lm f)` is not the multiple of any `(lm g)`.
`(moduloBasis m bs a)` is applied for `bs = [lc g | (lpp g) divides (lpp f)]` with `m = cMode`.

About method.
For the current `monF = (a,ppF) = lm(f)`, find
`gsPPF = [g <- gs | (lpp g) divides ppF]`, `bs = [lc g | g <- gsPPF] ::  [a]`.

Then, b_i first try to divide `a` precisely. Hence, for `Field a`, it turns to usual strong reduction of `f` by `gs`.

Otherwise (if `a` is not a multiple of any b_i),
`(a',ds) = coefNF cMode bs a` reduces `a :: 'a'` to `a'` by `bs` accumulating the quotients `ds`.

And as the ring is Euclidean, `coefNF` acts like extended gcd,
applying `divRem _ a gcd(bs)` in the end.

Each d_i multiplies the corresponding power product `pi' = ppF-(lpp` g_i`)`.

The new head monomial for `f` is `(a',ppF)` for non-zero `a'`;
the new tail is `pTail(f)` $- \sum_(d_i p'_i pTail(g_i)...)$.

And so on, while `(lpp f)` or `(lc f)` can reduce. This is a weak reduction.

See `PolNF_.hs` for the implementation comments.

{ar} Algebraic relations for polynomials

```
algRelsPols :: EuclideanRing k =>
               [Pol k] -> [PolVar] -> PPOrdTerm -> [Pol k] -> [Pol k]
               -- hs      ys          oY            fs         rels
```

Ideal generators for the algebraic relations for the polynomials `fs` considered modulo `Ideal(hs)`, f_i, h_i are from $A = k[x_1, \ldots, x_n]$, k a field (*would this work for Euclidean k too?*).

Any `rel` $\in$ `rels` is from $B = k[y_1, \ldots, y_m]$;

y_i form the list `ys` of the variables corresponding bijectively to f_i, so that $rel(f_1, \ldots, f_m) = 0$ *in* A.

`rels` is the reduced Gröbner basis in B for the ideal of all such algebraic relations for `fs`.

`oY` is the power product ordering description chosen for B.

Method is taken from the paper [GTZ]:

1. Embed f_i, h_i to $C = k[x_1, \ldots, x_n, y_1, \ldots, y_m] = k[X, Y]$;

2. Let $p_i = f_i - y_i$ in C,
   C is viewed under the direct-sum power product comparison
   `(ppComp_blockwise |xs| cpX cpY)`
   (so, X-power-products > Y-power-products),
   `gs =` GröbnerBasis `(`$[p_1, \ldots, p_m]$`++hs)` in C;

3. `rels' = [g | g <- gs  and  g  does not depend on X]`

4. `rels = rels'` embedded in $k[Y]$.

Example 1:
`demotest/T_cubeext.hs` applies `algRelsPols hs ["y","x"] o [y',x']` to find certain quadratic relation over the discriminant field $k(d)$ between the two roots of a polynomial $x^3 + ax + b$, a, b from the field k.

Example 2.
Find the equation system in `x, y, z` for some curve in `K`^{3} given by parametric equations `x = x(t), y = y(t), z = z(t),  K = Fraction Z`.

```
type K = Fraction Z
type P = Pol K
eqs = algRelsPols [] zyx ord fs
      where
      k1  = 1:/1 :: K
      dK  = upField k1 Map.empty
      zyx = ["z", "y", "x"]
      ord = (("", 3), degLex, [])          -- pp ordering for z, y, x
      t1  = cToPol (lexPPO 1) ["t"] dK k1   -- 1 of T = K[t]
      p1  = cToPol ord        zyx   dK k1   -- 1 of P = K[z,y,x]
      dP  = upLinSolvRing p1 Map.empty
      t   = varP k1 t1
      fs  = [t^4, t^3, t + t^5]
            -- z  y    x         this is called the Abyankhar curve
```

Here `eqs` yields the Gröbner basis for all the equations in `z, x, y` for the above curve:

$[z^2 - yx + z, \quad y^3x + zy^2 - zx^2 + y^2,$
$y^4 - zyx + yx - z, \quad zy^2x + 2y^2x - x^3 + zy + y,$
$zy^3 + y^3 - yx^2 + zx\ ]$

{und} 'Under' power products

```
underPPs ::  [PowerProduct] -> (InfUnn Z, [PowerProduct])
```

A power product `p` is called an 'under' power product with respect to the power product list `pps` if none of `pps` divides `p`.

'Under' monomials are useful in computation in the residue rings of a polynomial ring. This is because for the coefficient field k, $P = k[x_1, \ldots, x_n]$, `gs` a Gröbner basis, `pps = map lpp gs`, under-s form a linear basis over k for the residue domain $P/Ideal(gs)$.

Trivial **Lemma:**
the set of 'under' power products is finite if and only if

for any x_i `<- vars`, $1 \geq i \leq n$, `pps` contains `p =` x_i^k, with some $k > 0$ — that is `p = Vec [0,...,0,k,0,...,0]`.

`underPPs pps` yields the list (and number) of 'under' power products for a given list of any power products `pps`.
It returns `(Infinity, [])` for the infinite list, `(Fin 0, [])` for the empty list.

Examples:

```
underPPs [Vec [0,3], Vec [1,2]]           --> (Infinity, [])
underPPs [Vec [0,3], Vec [1,2], Vec [2,0]] -->

            (Fin 5,  [Vec [0,0], Vec [0,1], Vec [0,2], Vec [1,0], Vec [1,1]])
```

Mind: the very list may be sometimes stupidly long, hard to print, and so on. For example, for `pps` = $[x^{10}, y^{10}, z^{10}, u^{10}]$, `underPPs pps` yields `(10000, pps')`, with `pps' = [[0,0,0,0], [0,0,0,1] ...]`.

But if you use only the *number*, like say in `let (n, _) = underPPs pps in` ..., then the evaluation is much cheaper.

39 Free module over Polynomial

39.1 Introduction

For any ring P, $M = P^m$ is a free module over P of rank m.

Further, for $P = a[x_1, \ldots, x_n]$, M may have the gradings induced from P, and hence, the Gröbner bases induced by these gradings [MoM].

We keep in mind that the Gröbner bases in M evaluate differently depending on the way the pp ordering (or grading) is continued from `cp` of P to `cpE` of M.

Different `cpE` make isomorphic copies of M, but the isomorphism may not preserve the grading.

39.1.1 Vector . Pol and VecPol

This additional algorithmic structure expresses as

```
instance EuclideanRing a => LinSolvLModule (Pol  a) (Vector (Pol a))
...                         LinSolvLModule (Pol  a) (EPol a)
...                         LinSolvLModule (UPol a) (Vector (UPol a))
```

For the `Vector . Pol` model of the free module M, `DoCon` puts the comparison (or grading) in M to be the so-called TOP (term over position) with the reverse position. This compares first the power products by `cp`, then, if equal, the reverse position numbers in the vectors.

But there exist other useful comparisons for M — see [MoM]. So, `DoCon` introduces also

```
data VecPol a = VP [Pol a] EPPOTerm
```

which expresses the domain M isomorphic to `Vector (Pol a)`, but provides a field for the extended pp ordering (epp ordering) `cpE`.

So, `VecPol`, with its `cpE` choice, is an appropriate, flex model for the free module M over $P = a[x_1, \ldots, x_n]$. And `Vector . Pol` is a special case of `VecPol`.

Further, to compute the Gröbner bases, `VecPol` uses the isomorphism to `EPol`.

39.1.2 EPol

is one more representation for `VecPol`: a linear combination of the extended power products $m_i \cdot e_i$, m_i a monomial, e_i canonical vector of i-th coordinate [MoM]. The admissible comparison on $m_i \cdot e_i$ are as above, for `VecPol`.

Example.

Let `V = EPol Z,   ord = (("dlex",2), degLex, [])`
and the extended comparison is `ecp = eAntiLex`:

```
eAntiLex (i,p) (j,q) = case degLex p q of EQ -> compare j i
                                          r  -> r
```

With this `ecp` and `eTerm = (ecp, "a", [], _)`, the vector `V =` $(5x+4, 5x^2y+4x, 4y)$ (with coordinate No-s 1, 2, 3) corresponds to e-pol =

```
(2, 5*x^2*y) + (1, 5*x) + (2, 4*x) + (3, 4*y) + (1,4)  <-->
EPol
    [(5, (2, Vec [2,1])), (5, (1, Vec [1,0])), (4, (2, Vec [1,0])),
     (4, (3, Vec [0,1])), (4, (1, Vec [0,0]))
    ]  eTerm  _
```

That is the initial list `fs ::  [Vector P]` converts in this way to `fe_s :: [EPol a].`

For `fe_s`, there are defined the functions

`polNF_e, gBasis_e, polRelGens_e`,

which generalize respectively the functions `polNF, gBasis, polRelGens`
to e-polynomials.

`VecPol` is supplied with the same instances as `EPol`, and applies the conversion to/from `EPol` for the gx-operations. The conversions

```
epolToVecPol :: ... Z -> EPol a -> Vector (Pol a)
vecPolToEPol :: ... EPPOTerm -> Vector (Pol a) -> EPol a
```

and the gx-module structure for `EPol a` bring the the gx-module structure to
`Vector $ Pol a,  VecPol a`.

Compute with `VecPol` or with `EPol` ?

`VecPol` looks better, more traditional, when parsed/unparsed. But the programmer has to decide in each case, whether this costs the conversion to/from `EPol` applied in each gx-operation.

Example:

`demotest/T_grbas2.hs` (function `abya`)
presents the computation of the generator lists `rels, check` (of different origin) for the syzygies in P^5 of the Abyankhar curve equations
(Section 38.5.3 Point {ar} Example 2). Then, it tests, via `gxBasisM`, that `rels`, `check` generate the same submodule.

39.2 E-polynomial

The e-polynomial items are exported by the module `Pol`,
and implemented in `pol/EPol*, Pol2_`.

39.2.1 Various items for EPol

```
type EPP = (Z, PowerProduct)       -- in the Extended power product  (i,pp)
                                   -- i = 1,2,... is the coordinate No
type EPPComp = Comparison EPP

ecpTOP_weights, ecpPOT_weights :: Bool -> [PowerProduct] -> PPComp -> EPPComp
                                -- mode   weights           cp
```

are the widely used EPP-comparisons induced by the pp-comparison `cp` and a list of weights w_i.

w_i `::` `PowerProduct` is for the coordinate No i.

TOP stands for the "term over position", POT — "position over term".
TOP means to compare first the pp parts of the given (i,p), (j,q) by `cp` (w_i+p) (w_j+q), then, if equal, compare the positions.

POT means to compare first the positions.

It is said this TOP/POT terminology is by W.Adams & P.Loustaunau.

`mode ::` `Bool` describes the comparison for the positions (integers),

`True` means `compare` — the straight order,

`False` — `flip compare` — reverse order.

Example:

```
ecpTOP_weights False [w1,w2,w3] degLex (3,p) (1,q) ==
                                     case degLex (w3+p) (w1+q) of EQ -> LT
                                                                  v  -> v
```

Some items for epp orderings:

```
ecpTOP0 :: Bool -> PPComp -> EPPComp
           -- Specialization of ecpTOP_weights to zero weights.
           -- Compares first pp by cp, then, if equal, loks into position No.

ecpTOP0 mode cp (i,p) (j,q) = case (cp p q, compare i j)
                              of
                                (EQ, v) -> if mode then v else antiComp v
                                (v , _) -> v

type EPPOTerm = (EPPComp, String, [PowerProduct], PPComp)
                -- ecp   mode    wts              cp
```

```
eppoECp     :: EPPOTerm -> EPPComp
eppoMode    :: EPPOTerm -> String
eppoWeights :: EPPOTerm -> [PowerProduct]
eppoCp      :: EPPOTerm -> PPComp

eppoECp     = tuple41
eppoMode    = tuple42
eppoWeights = tuple43
eppoCp      = tuple44
```

`eterm = (ecp, mode, wts, cp)` describes the epp ordering.

`ecp` the epp comparison function, is the main part.

`mode = 'a':_` means that `ecp` agrees with the pp-ordering `cp'` contained in the polynomial sample of e-polynomial:
`cp'` and `ecp` restricted to any fixed `i` define the same ordering.

`wts` is a list of weights, each related to its position in vector.
`wts = []` means the weights and `cp` are ignored.
Other value means that `ecp` is defined as the TOP or POT comparison respectively to `wts` and `cp`.

`mode = [aMode, tMode, dMode]`

`aMode = 'a'` means `ecp` agreed with the pp-ordering from the polynomial sample of e-polynomial.

`tMode = 't'` means TOP comparison, other letter means POT comparison.

`dMode = 'l'` means the less position No is considered as greater, other letter means the reverse direction.

`Examples.`
`(ecp, "", [], _, _)` means no extra description for `ecp` is given.
`(ecp, "atl", [w1,w2], degLex)` means the agreed TOP-degLex-(reverse-position) comparison `ecpTOP_weights False [w1,w2] degLex`

```
type EMon a = (a, EPP)
data EPol a = EPol [EMon a] EPPOTerm (Pol a)
```

`EPol` is the indexed monomial-wise representation for polynomial vector.

In `EPol emons eterm pol`

`pol` is a sample polynomial, `eterm = (ecp, _, _, _)`,

`ecp` is an admissible extended power product comparison.
See Sections 39.1, 39.1.2.

```
instance Eq a => Eq (EPol a) where  f==g = (epolMons f)==(epolMons g)
epolMons     :: EPol a -> [EMon a]
```

```
epolEPPOTerm :: EPol a -> EPPOTerm
epolPol      :: EPol a -> Pol a
epolECp      :: EPol a -> EPPComp

epolMons     (EPol ms _ _) = ms
epolEPPOTerm (EPol _  t _) = t
epolPol      (EPol _  _ p) = p
epolECp = eppoECp . epolEPPOTerm

epolPPCp = polPPComp . epolPol    -- the one contained in polynomial sample

eLm :: CommutativeRing a => EPol a -> EMon a  -- leading e-monomial
eLm f = case epolMons f of m:_ -> m
                           _   -> error $ ("eLm 0 \nin  "++) $ showsDomOf f "\n"
eLpp :: CommutativeRing a => EPol a -> EPP
eLpp =  snd . eLm
epolLCoord :: CommutativeRing a => EPol a -> Z    -- coordinate ("position")
epolLCoord =  fst . eLpp                          -- of leading monomial

leastEMon :: CommutativeRing a => EPol a -> EMon a
leastEMon f = case epolMons f
              of
                m:ms -> last (m:ms)
                _    -> error $ ("leastEMon 0  \nin  "++) $ showsDomOf f "\n"

zeroEPol :: EPPOTerm -> Pol a -> EPol a
zeroEPol    t           f     =  EPol [] t f

instance Dom EPol where  dom    = dom    . epolPol
                         sample = sample . epolPol

instance AddGroup a => Cast (EPol a) (EMon a)
  where
  cast mode (EPol _ t p) (a,e) = EPol mons t p
                       where
                       mons = if mode=='r' && isZero a  then []  else [(a,e)]

instance AddGroup a => Cast (EPol a) [EMon a]
  where
  cast mode (EPol _ cp p) mons = EPol ms cp p     -- order NOT checked
    where
    ms = if mode /= 'r' then mons  else filter ((/= z) . fst) mons
    z  = zeroS $ sample p

instance PolLike EPol
  where
```

```
  varPs        = error (..."varPs (EPol..)  is senseless\n")
  pFromVec _ _ = error (..."pFromVec (EPol..) _  is senseless\n")
  pToVec   _ _ = error (..."pToVec _ (EPol..)  is senseless\n")
  pIsConst     = all (isZero . snd . snd) . epolMons
  pVars        = pVars   . epolPol
  pCoefs       = map fst . epolMons
  lpp  = snd . eLpp           -- :: PowerProduct, in addition to eLpp
  pPPO = pPPO . epolPol       -- :: PPComp
  ldeg = sum . vecRepr . lpp

  deg f = case  map (sum . vecRepr . snd . snd) $ epolMons f  of
                                                    [] -> error ("deg 0"..)
                                                    ds -> maximum ds
  pTail f = case epolMons f of _:ms -> ct f ms
                               _    -> error ("pTail 0"...)
  pFreeCoef _    = error (..."pFreeCoef (EPol..)  is senseless\n")
  pCoef f (j:js) =  -- ... coefficient of  (j, Vec js)

  lm f = if null $ epolMons f then  error ("lm 0"...)
         else
           case eLm f of (c,(_,p)) -> (c,p)   -- :: Mon a, in addition to eLm

  degInVar for0 i f = ... maximum [deg_i mon | mon <- unExtendedMonomialsOf_f]
  pMapCoef mode f g = cast mode g [(f a, e) | (a,e) <- epolMons g]

  pMapPP f g = ct g [(a, (j, Vec ks)) | (a, (i, Vec js)) <- epolMons g,
                                        let j:ks = f (i:js)
                  ]
  pCDiv f c = ... similar to Pol case.

  pDeriv  mInd  f@(EPol emons _ g) =  ctr f $ map monDeriv emons
                                         ...differentiate each e-monomial...
  pValue  _ _= error (.."pValue  (EPol..)..to be defined in future")
  pDivRem _ _= error (.."pDivRem (EPol..)..to be defined in future")

reordEPol :: EPPOTerm -> EPol a -> EPol a      -- bring to given epp ordering
reordEPol t (EPol ms _ pol) = case  eppoECp t  of

          ecp -> EPol (sortBy cmp ms) t pol   where cmp (_,p) (_,q) = ecp q p

cToEMon :: [a] -> Z -> b -> EMon b
cToEMon    xs     i    b =  (b, (i, Vec $ map (const 0) xs))

cToEPol :: AddGroup a => EPol a -> Z -> a -> EPol a
                         --sample
cToEPol (EPol _ t f) i a = if a==(zeroS a) then  EPol [] t f
```

```
                          else                  EPol [cToEMon (pVars f) i a] t f
polToEPol :: Z -> EPPOTerm -> Pol a -> EPol a
                        -- embed polynomial to the e-polynomial of the given
                        -- constant coordinate No i and epp ordering term
                        --
polToEPol i o f = EPol [(c, (i,p)) | (c,p) <- polMons f] o f

                      -- the inverse operation is correct
                      -- * when the polynomial power products do not repeat *
                      --
epolToPol :: AddGroup a => EPol a -> Pol a
epolToPol                 (EPol ms _ f) = ct f [(a,p) | (a, (_,p)) <- ms]

instance CommutativeRing a => Show (EPol a)
  where
  showsPrec _ (EPol ms _ f) = case [(i, ct f (a,p)) | (a, (i,p)) <- ms] of

                        indexedPols -> ("(EPol "++) . shows indexedPols . (" )"++)
instance CommutativeRing a => Set (EPol a) where ...

-- ... The instances up to  AddGroup  are evident.
-- The addition is similar to ‘Pol’ one.

instance CommutativeRing a => Num (EPol a) where negate = neg
                                                 (+)    = add
                                                 (-)    = sub
                                                 -- NO (*)
                          -- EPol can be multiplied by Pol but not by EPol

instance CommutativeRing a => LeftModule (Pol a) (EPol a)
  where
  cMul = polEPolMul
  baseLeftModule (_, EPol _ _ pol) fdom = ...

instance EuclideanRing a => LinSolvLModule (Pol a) (EPol a)
  where
  canAssocM _ f = case (isZero f, inv $ canInv$ lc f) of
                      (True, _) -> f
                      (_   , c) -> if c==(unity c) then f  else  cPMul c f

  canInvM smp f = if isZero f then
                    error $ ("canInvM smp 0 \n"++) $ ("for  "++) $ showsDomOf f "\n"
                  else  ct smp $ canInv $ lc f

  gxBasisM     _                  = gBasis_e
  syzygyGensM  _                  = polRelGens_e
```

```
    moduloBasisM                    = ... via  polNF_e, gBasis_e
    baseLinSolvLModule (pol,_) fdom = ...

  emonMul :: Ring a =>  a -> Mon a -> EMon a -> [EMon a]
                        --zero
  emonMul z (a,p) (b, (j,q)) = case mul a b of c ->
                                        if c==z then [] else [(c, (j,p+q))]
  mEPolMul :: Ring a => Mon a -> EPol a -> EPol a
  mEPolMul (a,p) f =  ctr f [(a*b, (i, p+q)) | (b, (i,q)) <- epolMons f]

epolToVecPol :: CommutativeRing a =>   Z -> EPol a -> Vector (Pol a)
                                    -- n    f
```

Convert e-polynomial `f` to the polynomial vector `Vec` $[f_1, \dots, f_n]$ of the given size `n` $\geq$ maximal coordinate No in `f`.

It gathers the monomials of each coordinate to polynomial.

If `f` does not contain monomials of i-th coordinate (for $1 \leq i \leq n$), then $f_i = 0$.

Detail of method:
the gathered monomials are being sorted — unless the `eppoMode` in `f` starts with the 'agreed' mode `'a'`.

Example:

```
  let  f   = EPol [(5, (2,x^2*y)), (5, (1,x)), (4, (2,x)), (1, (4,y)),
                   (4, (1, x^0*y^0))
                  ] eo  pol
       pol = cToPol o ["x","y"] 0  :: Pol Z
       o   = (("",2), degLex, [])
       eo  = (eAntiLex, "a", [], _)
  in   epolToVector 6 f
                                      --> Vec [5*x+4, 5*x^2*y+4*x, 0, y, 0, 0]


  vecPolToEPol : CommutativeRing a => EPPOTerm -> Vector (Pol a) -> EPol a
                                      -- eo       Vec [f1 ..]
```

Inverse to `epolToVecPol`.
The monomials of each f_i extend with the coordinate number i, are sorted under `eo` (this is skipped if `eo` shows 'agreed'), and merged to the previously accumulated (eo-ordered) e-polynomial.

```
  sEPol :: GCDRing a => EPol a -> EPol a -> Maybe (EPol a, Mon a, Mon a)
```

`sEPol f g` differs from `sPol` in that it is in the `Maybe` format and returns
`Nothing` when the leading coordinates of `f`, and `g` differ,
`Just (s-epol, m1, m2)`, otherwise.

```
sEPol f g = if (epolLCoord f)/=(epolLCoord g) then  Nothing
            else
              let h         = epolPol f
                  (a,b)     = (lc f, lc g)
                  (_,m1,m2) = sPol (ct h (a, lpp f)) (ct h (b, lpp g))
              in  Just ((mEPolMul m1 f)+(mEPolMul m2 g), m1, m2)

type EPVecP a = (EPol a, [Pol a])                -- similar to PVecP

mEPVecPMul m (f,v) = (mEPolMul m f, map (mPolMul m) v)

sEPVecP :: GCDRing a => EPVecP a -> EPVecP a -> Maybe (EPVecP a, Mon a, Mon a)

sEPVecP (f,v1) (g,v2) = case sEPol f g of          -- similar to sPVecP

  Nothing          -> Nothing
  Just (_, m1, m2) ->
           let epdiff = (mEPolMul m1 f) - (mEPolMul m2 g)
               vdiff  = zipWith (\s t -> (mPolMul m1 s)-(mPolMul m2 t)) v1 v2
           in  Just ((epdiff, vdiff), m1, m2)
```

39.2.2 Gx-operations for EPol

For `EPol`, apply `gxBasisM, syzygyGensM, moduloBasisM`
for the similar purpose as `gxBasis, syzygyGens, moduloBasis`
are applied for `Pol` (see first Section 38.5.3).

Here the 'M' operations are the analogies for the ones for vectors represented by `EPol`. The implementation for this is in `pol/GBas*.hs, PolNF_.hs, Polrel_.hs`.

```
polNF_e :: EuclideanRing a => String -> [EPol a] -> EPol a -> (EPol a, [Pol a])
```

It differs from `polNF` in that the power products are extended and in that the condition "`(lpp` g_i`) divides (lpp f)`" is enforced by with adding of
"and `(eLpp` g_i`), (eLpp f)` have the same coordinate No".

`gxBasisM` for `EPol` is defined via `gBasis_e` - see [MoM] and the comments in `GBasFld_.hs GBasEuc_.hs`,

By the way, the `Pol` case is done via the generic `EPol` case. The main difference is that the s-e-polynomials are formed only for the items with the same leading coordinate.

Similar approach is for `syzygyGensM` for `EPol`.

Example:
`demotest/T_grbas2.hs` (function `abya`) contains the computation of the generator lists `rels, check` (of different origin) for the syzygies in P^5 of the Abyankhar curve equations (Section 38.5.3 Point {ar} Example 2).

Then, it tests, via `gxBasisM`, that `rels`, and `check` generate the same submodule.

<u>On submodule resolvent</u>

The gx-structure on P^n, P a polynomial ring, is very important. In particular, for the polynomials $[f_1, \ldots, f_n]$, the syzygies $[r_1, \ldots, r_k]$ are in P^n. Then it is possible to find the g-bases and syzygies $[s_1, \ldots, s_l]$ for $[r_1, \ldots, r_k]$, $s_i \in P^k$. And so on. This gives us the resolvent of an ideal (or a submodule) resolvent. Choosing specially the term ordering [MoM] and the theorem on the Taylor syzygies, one also can avoid the intermediate g-bases computation.

40 R-polynomial

40.1 Preface

'Recursively' represented polynomials, with their constructor `RPol`, are introduced to represent a totally sparse polynomial. For example, it is in-efficient to represent
$2x_1^3x_{100}^4 + 1 \in a[x_1, \ldots, x_{100}]$ via `Pol` — that is by
`[(2, [3,0,0,...,0,4]), (1, [0,0,...,0])]`.

It is better to set the r-polynomial with the variable multiindex `[i]` ranging in
`[1 .. 100]`.

`RPol` consists of the proper r-polynomial (`RPol'`) and the description of the variable set and ordering.

It is considered as a polynomial in the (finite) set of variables
$X = \{x[multiindex] \mid \ldots\}$, X represented as rectangular in the space of multiindice.

The instances `Set, ..., GCDRing` for `RPol` are defined almost same as for `Pol`. Naturally, `RPol` has the intance of `PolLike` class.

The items for `RPol` are exported by the module `Pol`,
implemented in `RPol_.hs, RPol0_.hs`.

40.2 Examples

Example 1

1. Put the r-variable set `vs = {x[i,j] | i <- [1..3], i <- [0..2]}`, ordered lex-increasingly;
2. Form `gr = x_1_1 + 2*x_3_0` of `R = Z[{vs}]` represented as `RPol`;
3. Convert `gr` to `g` of `P = Z[{vs}]` represented as `Pol`
4. Test that $\texttt{g}^2$ converted to `RPol` equals $\texttt{gr}^2$;
5. Find the number of variables set and list them.

```
import Pol ...
...
type P = Pol  Z   -- for  Z[{vs}]
type R = RPol Z   -- for  P  represented as RPol
f =
  let  vcp          = flip (lexListComp compare)
       ranges       = [(1,3), (0,2)]
       vt           = (vcp, "x", ranges)
       (vn, rvars) = (rvarsVolum ranges, rvarsOfRanges ranges)
       o            = lexPPO vn
       r1           = cToRPol vt dZ 1                -- 1 of R
       gr           = smParse r1 "x_1_1 + 2*x_3_0"
       g            = fromRPol o gr
```

```
      gp'          = toRPol 'l' vt rvars (g^2)
  in   (gr^2==gp', vn, rvars)
  -->
  (True, 9, [[1,0],[1,1],[1,2],[2,0],[2,1],[2,2],[3,0],[3,1],[3,2]])
```

Example 2

Here how DoCon exploits RPol for factoring in $k[x,y]$, k a finite field. To factor f from $k[t][x]$, DoCon builds the basis B of certain lattice over $k[t]$, mB being a triangular matrix over $k[t]$ (see the theory in [Me3]).

Then, it has to find a vector $b = [b_1 \ldots]$ in this lattice with $\deg b_i < bound$ for each i (the DoCon module Pfact2_ applies Pfact0_.smallInLattice_triang_ to do this).

This b is found by putting the linear equations for the unknown coefficients $u_{i,j}$ of polynomials u_i from $k[t]$: $b = u \times B$, $u = [u_1, \ldots, u_m]$, $m =$ height B.

It must hold $\deg(u \times B)_i < bound$ for all i.
And it is known a bound for $|u|$: $\deg u_i < boundU$,
under which the needed $u \times B$ is found — if such ever exists.
This condition $\deg(u \times B)_i < bound$ presents a linear system on unknown $u_{i,j} \in k$. The ring $U = k[u_{i,j} | \ldots]$ is built as represented by RPol, where the polynomials $u_i = \sum[u_{i,j}t^j | \ldots]$ are lifted to $U[t]$ and make the vector $uVec$ over $U[t]$.

$ub = uVec \times B$ is again of Vector $U[t]$, and the conditions $\deg ub_i < bound$ express as the equalities $c = 0$ for $c \in k[u_{i,j} \ldots]$ all the coefficients from $ub_i \in U[t]$ of monomials of degree $\geq bound$. This list of r-polynomials c is called eqs, and it forms a matrix, with the rows represented sparsely, as the linear r-polynomials. For example,
eqs = [$u_{1,2} + 3u_{3,3} + 2u_{3,8}$,
$4u_{1,2} + 3u_{2,1} + 1$,
$\ldots$]
Here $u_{i,j}$ are the unknowns. Then the Gauss method linRPolsToStaircase applies and converts it to the staircase form matrix eqs'. And eqs' is solved as usual, by substituting the values for the free variables starting from the last equation. Here the matrix eqs has the sparse rows — represented by the linear r-polynomials, each variable corresponds to the position in the row.

40.3 Definitions

Here follows a more precise description of RPol operations.

```
type RPolVar      = [Z]            -- r-pol variable is actually a multiindex
type RPolVarComp  = Comparison RPolVar
...
type RPolVarsTerm = (RPolVarComp, String, [(Z,Z)])
```

`(vcp, pref, ranges)` presents the variable ordering `vcp` and the variable set description for `[pref`$[i_1, \ldots, i_n]$ `| ...]`.

`pref` is the prefix to print variable,

`ranges =` $[(l_1, h_1), \ldots, (l_n, h_n)]$ contains the range (l_k, h_k) for each variable index component: $l_k \le i_k \le h_k$

```
rvarsTermCp :: RPolVarsTerm -> RPolVarComp
rvarsTermCp =  tuple31

rvarsTermPref :: RPolVarsTerm -> String
rvarsTermPref =  tuple32

rvarsTermRanges :: RPolVarsTerm -> [(Z,Z)]
rvarsTermRanges =  tuple33

data RPol a = RPol (RPol' a) a RPolVarsTerm (Domains1 a)
```

In `f = RPol rpol' a vterm aDom`

`a` is a sample coefficient,

`vterm = (cp, pref, rgs)` variable comparison and full variable set description,

`aDom` domain description for `a`,

`rpol' :: RPol' a` very representation.

`f` is viewed as the element of `a[XX]`,

`XX = [pref`$_{ind}$ `|` $ind \in$ `setDefinedBy-rgs]`,

`setDefinedBy-rgs = [`$[i_1, \ldots, i_n]$ `| for each` k $l_k \le i_k \le h_k$`]` .

Example:

`f = RPol rpol' 0 (cp, "y", [(0,2),(0,1)]) dZ` denotes some
`f` $\in$ $Z[y_{i,j} \mid i \in [0,2], j \in [0,1]]$, what `f` namely, this shows `rpol'`.

```
data RPol' a =   CRP a                                   -- constant r-pol
               | NRP RPolVar Z (RPol' a) (RPol' a)       -- non-constant
               deriving (Eq, Show, Read)
```

`CRP a` means a constant R-polynomial,

`NRP v n cf tl` means a non-constant R-polynomial $cf \cdot v^n + tl$, $n > 0$,
$0 \ne cf$ `:: RPol' a`, variables in cf are smaller than v,
tl `:: RPol' a`, variables in tl are not greater than v, `deg` v tl $< n$.

Example.

If $x_1 > x_2$ in a comparison `cp`,

$x1 \leftrightarrow [1]$, $x2 \leftrightarrow [2]$, `varsTerm = (cp, "x", [(1,2)])`,
then $5x_1^2 x_2^3 + x_1^2 + 6x_1$ of $Z[x_1, x_2]$ converts to
$x_1^2 \cdot (5x_2^3 + 1) + x_1 \cdot (6)$ $\longrightarrow$

```
 RPol (NRP [1] 2 (NRP [2] 3 (CRP 5) (CRP 1)) (NRP [1] 1 (CRP 6) (CRP 0)))
      0 varsTerm dZ
```

Furthger main definitions for `RPol` are as follows.

```
rpolRepr :: RPol a -> RPol' a
rpolRepr    (RPol f _ _ _) = f

rpolVTerm :: RPol a -> RPolVarsTerm
rpolVTerm    (RPol _ _ t _) = t

rpolVComp :: RPol a -> RPolVarComp
rpolVComp =  rvarsTermCp . rpolVTerm

rpolVPrefix :: RPol a -> String
rpolVPrefix = rvarsTermPref . rpolVTerm

rpolVRanges :: RPol a -> [(Z, Z)]
rpolVRanges = rvarsTermRanges . rpolVTerm

rp'HeadVar :: RPol' a -> Maybe RPolVar
rp'HeadVar (NRP v _ _ _) = Just v
rp'HeadVar _             = Nothing

rpolHeadVar :: RPol a -> Maybe RPolVar
rpolHeadVar = rp'HeadVar . rpolRepr

-- rp'Vars f, rpolVars f
-- yield only the variables  v(i,j)  on which  f  really depends.

rp'Vars :: Char -> RPol' a -> [RPolVar]
  --       mode    f
  -- List variables, first - into depth, then to the right,
  -- repetitions cancelled.
  -- mode = 'l'  means f is linear, in this case the computation is simpler.

rpolVars :: Char -> RPol a -> [RPolVar]
  --        mode    f
  -- The result is ordered by comparison from f.
  -- mode = 'l'  means f is linear - in this case the computation is simpler.
  --
rpolVars mode (RPol f _ t _) = case  (rp'Vars mode f, mode, rvarsTermCp t)
                               of
                                 (vs, 'o', cp) -> sortBy (flip cp) vs
                                 (vs, _  , _ ) -> vs
instance Dom RPol' where
                   dom _  = error ("no  dom  needed for RPol'" ...)
```

```
                   sample = sm  where  sm (CRP a)       = a
                                       sm (NRP _ _ c _) = sm c

instance Dom RPol where  sample (RPol _ a _ _) =  a
                         dom    (RPol _ _ _ d) =  d

instance Cast (RPol a) (RPol' a) where
                                 cast _ (RPol _ c t d) f = RPol f c t d
instance Cast (RPol' a) a where cast _ _ a = CRP a
instance Cast (RPol a) a  where
                          cast _ (RPol _ c t d) a = RPol (CRP a) c t d

rvarsVolum :: [(Z,Z)] -> Z
rvarsVolum    []      =  error "rvarsVolum []\n"
rvarsVolum    ranges  =  product [h-l+1 | (l,h) <- ranges]

rvarsOfRanges :: [(Z,Z)] -> [RPolVar]
  --
  -- All r-vars in the given ranges listed in the lexicographic-increasing order.
  -- Example:  rvarsOfRanges [(0,2),(3,4)] = [[0,3],[0,4],[1,3],[1,4],[2,3],[2,4]]
  -- CAUTION:  it may be very expensive, think before applying it.
--------------------------------------------------------------------
instance PolLike RPol'
  where
  pIsConst (CRP _) = True
  pIsConst _       = False

  pCoefs f = pc f []  where  pc (CRP a)         = (a:)
                             pc (NRP _ _ cf tl) = pc cf .pc tl
                                -- coefficients listed "first in depth"
  pTail (CRP a)       = CRP a
  pTail (NRP _ _ _ t) = t

  pFreeCoef (CRP a)        = a
  pFreeCoef (NRP _ _ _ tl) = pFreeCoef tl

  ldeg (CRP _      ) = 0                 -- deg in leading variable
  ldeg (NRP _ n _ _) = n                 --

  deg = dg  where  dg (CRP _)       = 0                     -- sum of degs
                   dg (NRP _ n c t) = max (n+(dg c)) (dg t)

  degInVar _ n f = case genericIndex (rp'Vars '_' f) (n-1) of

         v -> dg f  where
```

```
                  dg (CRP _)       = 0
                  dg (NRP u n c t) = if u==v then n  else max (dg c) (dg t)

  pMapCoef 'r' f g = fmap f g      -- example of programming
  pMapCoef _   f g = m g           -- with  RPol
    where
    z = zeroS $ head $ pCoefs g
    m (CRP a)        = CRP $ f a
    m (NRP v n c t) = case m c
                    of
                      CRP b -> if b==z then m t  else NRP v n (CRP b) (m t)
                      c'    -> NRP v n c' (m t)
  pMapPP f = m  where
              m (CRP a)       = CRP a
              m (NRP v n c t) = NRP v (head $ f [n]) (m c) (m t)
                -- CAUTION:
                -- applying bad f may yield incorrect r-polynomial
  pCDiv f a =  ...
  varPs a f = [NRP v 1 (CRP a) $ CRP $ zeroS a | v <- rp'Vars '_' f]

  -- lpp, lm, pDivRem  skipped
--------------------------------------------------------------------
instance Eq a => Eq (RPol a) where  f==g = (rpolRepr f)==(rpolRepr g)
instance Functor RPol'
  where
  fmap f (CRP a)       = CRP (f a)
  fmap f (NRP v n c t) = NRP v n (fmap f c) (fmap f t)
    --
    -- When this  map f :: RPol a -> RPol b  yields correct polynomial?
    -- When  f c  is non-zero for all the coefficient r-pols  c
    -- in the given r-pol g.
--------------------------------------------------------------------
instance PolLike RPol
  where
  pIsConst  = pIsConst  . rpolRepr
  ldeg      = ldeg      . rpolRepr
  deg       = deg       . rpolRepr
  pCoefs    = pCoefs    . rpolRepr
  pFreeCoef = pFreeCoef . rpolRepr
  pTail f   = ct f $ pTail $   rpolRepr f
  pCDiv f   = fmap (ct f) . pCDiv (rpolRepr f)

  pMapCoef mode f g = ct g $ pMapCoef mode f $ rpolRepr g
  pMapPP f g        = ct g $ pMapPP f        $ rpolRepr g

  degInVar for0 i = degInVar for0 i . rpolRepr
```

```
  varPs  a  f@(RPol g _ t _) =
                        let cp                              = rvarsTermCp t
                            cp' (NRP u _ _ _) (NRP v _ _ _) = cp u v
                        in  map (ct f) $ sortBy (flip cp') $ varPs a g

  pDivRem (RPol f a t _) g = case divrem_ (zeroS a) (rvarsTermCp t) f (rpolRepr g)
                             of
                               (q,r) -> (ct g q, ct g r)
  -- lpp, lm  skipped

cToRPol :: RPolVarsTerm -> Domains1 a -> a -> RPol a   -- coefficient to RPol
cToRPol    t               dm            a =  RPol (CRP a) a t dm

varToRPol' :: AddSemigroup a => a -> RPolVar -> RPol' a
varToRPol' a v = NRP v 1 (CRP a) $ CRP $ zeroS a
                                                -- variable to RPol', RPol
                                                --
varToRPol :: AddSemigroup a => RPol a -> a -> RPolVar -> RPol a
                            -- sample
varToRPol f a v = ct f $ varToRPol' a v

add_ :: AddSemigroup a => a -> RPolVarComp -> RPol' a -> RPol' a -> RPol' a
                        --zero
  -- Auxiliary for (+).  Sum of r'-polynomials.
  -- We give it as an example programming with of RPol:

add_ z cp f g = ad f g
  where
  ad (CRP a)         (CRP b)             = CRP $ add a b
  ad (NRP v n c t)   (CRP a)             = NRP v n c $ ad t (CRP a)
  ad (CRP a)         (NRP v n c t)       = NRP v n c $ ad t (CRP a)
  ad f@(NRP v n c t) g@(NRP v' n' c' t') =
    case
        (cp v v', compare n n', ad c c', ad t t')
    of
      (GT, _ , _ , _ ) -> NRP v  n  c  (ad t  g)
      (LT, _ , _ , _ ) -> NRP v' n' c' (ad t' f)
      (_ , GT, _ , _ ) -> NRP v  n  c  (ad t  g)
      (_ , LT, _ , _ ) -> NRP v' n' c' (ad t' f)
      (_ , _ , c1, t1) -> if c1==(CRP z) then t1  else NRP v n c1 t1
```

```
rHeadVarPol :: Set a => Domains1 (RPol a) -> RPol a -> UPol (RPol a)
                      -- rdom               r         rU
```

Convert non-constant r-polynomial `r` from R to univariate polynomial from $R[u]$, u the leading variable of `r`
(`rdom` can be set before, for example, via `up..Ring`).

Caution: coefficients of `rU` have the same variable set as in `r`.

```
rFromHeadVarPol :: AddSemigroup a => RPolVar -> UPol (RPol a) -> RPol a
                                   -- v
```

Inverse to `rHeadVarPol`.

The leading variable is given in argument — in order not to convert it from string nor to search in `ranges`.

```
fromRPol :: CommutativeRing a => PPOrdTerm -> RPol a -> Pol a
                               -- ord       rf
```

Convert r-polynomial `rf` to polynomial `f`, `f` is under the given pp ordering `ord`.

The variable set for `f` is described by `rpolVTerm rf` :

`vars` = [$prefix_{i'_1,\ldots,i'_n}$ | $[i_1,\ldots,i_n] \in ranges$], $i'_k =$ `show` i_k.

The order in `vars` is put so that last index in a multiindex $[i_1,\ldots,i_n]$ changes first.

Example:

if `rpolVTerm rf = (_, "u", [(0,2),(0,1)])`, then `fromRPol rf` yields `f` with the variable list `["u_0_0", "u_0_1", "u_1_0", "u_1_1", "u_2_0", "u_2_1"]`.

Caution: think before applying `fromRPol` !
This variable index expansion to the (dense) power product may be expensive.

```
toRPol' :: CommutativeRing a =>  Char -> [RPolVar] -> Pol a -> RPol' a
                              -- mode    rvars        f
```

Convert a polynomial `f` from $a[x_1,\ldots,x_n]$ to r-pol' `rf`, with respect to the given bijective variable correspondence `rvars =` $[v_1,\ldots,v_n] \longleftrightarrow [x_1,\ldots,x_n]$ `= pVars f`,
the variable order preserves.

`mode = 'l'` means the pp order in `f` is `lexComp`, in this case the computation is simpler, other letter adds the cost of initial reordering of `f`.

```
toRPol :: CommutativeRing a => Char -> RPolVarsTerm -> [RPolVar] -> Pol a -> RPol a
                             -- mode vt                rvars        f
```

Convert a polynomial `f` from $a[x_1,\ldots,x_n]$ to r-polynomial `rf` with respect to the given r-variable set description `vt` and the variable correspondence
`rvars =` $[v_1, ldots, v_n] \longleftrightarrow [x_1, ldots, x_n]$ `= pVars f`,

`rvars` is the subset of `set(vt)`.

Required: $v_1 > \ldots > v_n$ by the comparison from `vt`.

`mode = 'l'` means the pp order in `f` is `lexComp`, in this case the computation is more direct, other letter causes the initial reordering of `f`.

```
substValsInRPol :: CommutativeRing a => Char -> [(RPolVar,a)] -> RPol a -> RPol a
                                        --mode    pairs            f
```

Substitute $(v_1 = a_1), \ldots, (v_n = a_n)$ into r-polynomial `f`.

Required: $v_1 > \ldots > v_n$ in the comparison from `f`.

`mode = 'l'` means linear `f`, in this case the substitution is simpler.

Example: `subst 'l' [(y,2),(z,4)] (x + 3*y + 1) -->  x + 7`

```
instance CommutativeRing a => Show (RPol a)
  where
  showsPrec _ f = showsSPPol' zr unm ord pref . rpol'ToSPPol' $ rpolRepr f
                      where
                      (pref, zr) = (rpolVPrefix f, zeroS $ sample f)
                      unm        = Just $ unity zr
                      ord        = isOrderedRing $ snd $ baseRing zr $ dom f
               --
               -- rpol'ToSPPol'  converts to polynomial with sparse exponent
               -- (the one to be developed in future)
```

41 Constructor class Residue

It joins the residue ring and residue group constructors

ResidueE, ResidueG, ResidueI.

All the residue items are exported by the module `Residue`,
implemented in `residue/*.hs`

Residue model choice

For a residue ring `a/I`, choose the `ResidueE` model, if you know that `a` is Euclidean. This will make computations simpler and help to get more definite attributes for the residue.

```
class Residue r
  where
  resRepr   :: r a -> a                        -- extract representation
  resPIdeal :: r a -> PIRChinIdeal a   -- ideal the Euclidean residue is taken by
  resGDom :: r a -> (Subgroup a, Domains1 a)
                          -- - for the quotient group  a/gH.
                          -- It returns  (gH, dH),  gH a normal subgroup,
                          -- dH  domain bundle for gH (it contains gH too).

  resIDom :: r a -> (Ideal a, Domains1 a)      -- for generic residue ring
```

Each of the latter three operations either has to be skipped or to yield the error break when applied to the residue of improper kind.

Example: for the residue class `r` of 2 in `Z/iI`,

```
resRepr r --> 2,  resPIdeal r --> iI,    resGDom r, resIDom r --> error ...



----------------------------------------------- extracting parts of residue
resSubgroup :: Residue r => r a -> Subgroup a
resSubgroup = fst . resGDom

resSSDom :: Residue r => r a -> Domains1 a
resSSDom =  snd . resGDom

resIdeal :: Residue r => r a -> Ideal a
resIdeal =  fst . resIDom

resIIDom :: Residue r => r a -> Domains1 a
resIIDom =  snd . resIDom
-------------------------------------------------------------------
instance (Residue r, Show a) => Show (r a) where  showsPrec _ = shows . resRepr
instance (Residue r, Eq a) => Eq (r a) where  x==y = (resRepr x)==(resRepr y)
                           --
                           -- this relies on that  x  is canonically reduced
```

41.1 Preface to residue group, residue ring

Quotient group `a/H`
is represented by the constructor `ResidueG`. DoCon understands it only in the case of commutative additive group. And the subgroup description for `H` has to contain a canonical map by `H`. With this, the operations in `a/H` (type `ResidueG a`) are defined in an evident way. See Section 43.

The quotient group is almost dummy in DoCon. So far, it is provided only to show that a quotient group it is possible.

Generic residue ring `a/I`
is represented by `ResidueI`. It needs a gx-ring (`LinSolvRing`) `a` and the ideal description `iI`. `iI` has to contain a gx-basis `gs`. The operations in `a/I` are defined via the method like Gröbner basis. See (Section 3.12 Residue ring), 'Ideal', Sections 44, 49.3.

Special Euclidean residue ring `a/I`
is represented by `ResidueE`. It needs the special `PIRChinIdeal` representation for `I`. The access to the residue attributes are simpler than for `ResidueI`. This all brings more efficiency to this important special case. Also ResidueE has it special instances for `Z/I`, `k[x]/I`. See Section 42. The principle of computation with `ResidueE` is similar to the generic case. The Euclidean remainder is exploited similarly as the Gröbner normal form or `moduloBasis`. Extended GCD — as the Gröbner basis with the transformation accumulation.

42 Residue ring of Euclidean ring

42.1 Preface

The `ResidueE` items are exported from the module `Residue`,
implemented in `residue/ResEuc*_.hs, RsePol_.hs`.

The first example with `ResidueE` appears in this manual in
(Section 2.2 Fragment "arithmetics in R = Z/(b)"), the manual contains many other examples with `ResidueE`, as well as `demotest/T_*.hs`.

See first Section 21.2 on `PIRChinIdeal`, Section 41 on the class `Residue`.

The constructor for an Euclidean residue is

```
data ResidueE a = Rse a (PIRChinIdeal a) (Domains1 a)
```

`Rse x iI aD` denotes the residue in `a/I` for an Euclidean ring `a`.
Its components are as follows.
`aD` is the bundle describing the base domain `a`.

`x` is a representative for a residue class by `I`.
`I` is given by its term `iI`.
It contains the ideal base `b = pirCIBase iI`.

Required: `x` must be reduced canonically by `b` : `x == (remEuc 'c' x b)`.
Usually, `iI` is built by applying the function `eucIdeal`.

```
instance EuclideanRing a => Cast (ResidueE a) a
  where
  cast 'r' (Rse _ iI d) a = Rse (remEuc 'c' a $ pirCIBase iI) iI d
  cast _   (Rse _ iI d) a = Rse a iI d
```

This implemants the casting `a -> a/I`.
`mode = 'r'` forces canonical reduction by the ideal.
any other mode wraps the representative "as it is".

Use the `cast` operation (and `ct, ctr`) to obtain new residues by sample.

```
instance Residue ResidueE
  where
  resRepr   (Rse x _  _) = x
  resPIdeal (Rse _ iI _) = iI
  resGDom   _            = error (..."resGDom (Rse..)\n")
  resIDom   _            = error (..."resIDom (Rse..)\n")

instance Dom ResidueE where  dom (Rse _ _ d) = d
                             sample        = resRepr

isCorrectRse :: (EuclideanRing a) => ResidueE a -> Bool
isCorrectRse (Rse x i _) = case  pirCIBase i
                           of
                             b -> x == (remEuc 'c' x b)

ifCorrectRse x y z = if isCorrectRse x then z  else y
```

42.2 Main instances for `ResidueE`

They are `Set ... AddGroup ... MulMonoid, LinSolvRing,`
`GCDRing, FactorizationRing, EuclideanRing, Field`.

The last four of these instances are correct (and defined trivially) only for a prime ideal `I` (in this case `a/I` is a field).

To form a residue domain `a/I` with any of above instances, one needs

(1) a c-Euclidean base ring on `a` (see Section 19 DivRemCan, Section 49.3), `a` not a field,

(2) a non-zero and non-unity ideal `I`.

In DoCon, the condition (1) for `a` expresses as
`(Euclidean, Yes), (DivRemCan, Yes), (IsField, No)`.

Example of operation setting in `a/I` :

```
instance EuclideanRing a => AddSemigroup (ResidueE a)
  where
  zero_m r = Just $ ct  r $ zeroS $ resRepr r
  neg_m  r = Just $ ctr r $   neg $ resRepr r
  add    r = ctr r . add (resRepr r) . resRepr
  ...
```

Here, for example, `add` sums the representatives to `s` and casts `s` to residue (with reduction via the canonical Euclidean remainder).

In `MulSemigroup`, the inverse of `r` in `a/I` is found via the extended GCD for `resRepr r, pirCIBase iI`.

The attributes of `a/I` semigroups, ring, and so on, depend greatly on the attributes of `I`, mainly on the factorization of its base. For example, if `I` contains the factorization `ft = [(_,1)]`, then I is prime; `(_:_)` — not prime, `[]` — unknown primality; and so on.

See `residue/ResEuc*_.hs` for the implementation details.

<u>`Random ResidueE`</u>

`DoCon` declares

```
instance (EuclideanRing a, Random a) => Random (ResidueE a)
  where
  randomR (l,h) g = (ctr l a, g')  where
                                   (a,g') = randomR (resRepr l, resRepr h) g
```

This means to make a random *representative* and *project* it to the residue ring. See Section 30.

42.3 LinSolvRing instance for ResidueE

It has to implement `gxBasis, moduloBasis, syzygyGens` for `a/(b)`.

The specific here is that `gxBasis` in `a` and in `a/I` is either `[]` or `[g]`.

For `a/I`, `moduloBasis` does not depend on the mode: it does the canonic reduction and needs intermediate implicit `gxBasis` application.

`gxBasis` for such a residue ring is defined as follows:

```
gxBasis [] = ([],[])
gxBasis rs = let r            = head rs
                 b            = pirCIBase $ resPIdeal r
                 ([g], [row]) = gxBasis ((map resRepr rs)++[b])
```

```
            in
            if divides b g then ([],[])
            else                ([ctr r g], [map (ctr r) $ init row])
```

Here `gxBasis ((map resRepr rs)++[b])` is computed in an Euclidean `a`. It is done there by the Gauss reduction of the $n \times 1$ matrix to the staircase form — this is equivalent to the extended gcd.

`moduloBasis` for Euclidean residue:

```
moduloBasis _ [] r = (r, [])
moduloBasis _ rs r = let (b, xs)  = (pirCIBase $ resPIdeal r, map resRepr rs)
                         (rm, qs) = moduloBasis "c" (xs++[b]) $ resRepr r
                     in  (ctr r rm, map (ctr r) $ init qs)

syzygyGens _ rs = let r        = head rs
                      b        = pirCIBase $ resPIdeal r
                      zr       = zeroS b
                      canRem x = remEuc 'c' x b
                      rels'    = syzygyGens "" (b:(map resRepr rs))
                      rls'     = mapmap canRem rels'
                      rels''   = filter (not . all (== zr)) $ map tail rls'
                  in  mapmap (ct r) rels''
```

42.4 Specialization to Z, k[x]

`DoCon` also provides special definitions for some instances for
`ResidueE Z` (for `Z/I`), `Field k => ResidueE (UPol k)` (for `k[x]/I`).

They *overlap* with the generic instances. They help to obtain more definite domain attributes and sometimes define simpler operations. For example, the operation `(+)` in `k[x]/I` does not need the reduction by `I` after summing representatives.

43 Quotient group ResidueG

So far, this is only for a *commutative additive group*.

A group residue (element of a quotient group `a/H`) is represented as

```
Rsg x (gH,dH) dm  ::  ResidueG a
```

This expression has sense if

(1) a subgroup term `gH` contains a canonical map `canr` by `gH` :
`subgrCanonic gH --> Just canr`,

(2) `(canr x) == x`, that is `x` is reduced by the subgroup,

(3) `dm` is the base domain description (bundle) for `a`
and `dH` is such bundle for the subgroup `gH`.

The base set of the `ResidueG a` sample is parameterized by the subgroup term `gH` — see the `Set` instance below. Thus, the subgroups `gZ3 = (3*Z)` and `gZ5 = (4*Z)` in `Z` produce different base sets in `ResidueG Z` :

`r1 = Rsg 2 (gZ3,_) _` and `r2 = Rsg 2 (gZ4,_) _`,

have the same type `ResidueG Z`, but mathematically, belong to different domains.
And `DoCon` can discover that they have different base sets. For example,

```
let {s1 = snd $ baseSet r1 _;  s2 = snd $ baseSet r2 _}
in  (osetCard s1, osetCard s2)
```

yields `(3, 4)`.

`DoCon` is *not safe* against mixing the residues by different subgroups. Thus,
`(Rsg 2 (gZ3,_) _) + (Rsg 2 (gZ4,_) _)` may cause an incorrect result.

```
data ResidueG a = Rsg a (Subgroup a, Domains1 a) (Domains1 a)

isCorrectRsg :: AddGroup a => ResidueG a -> Bool
isCorrectRsg                  r@(Rsg x d _) =
  case subgrCanonic $ fst d
  of
    Just can -> x==(can x)
    Nothing  -> error $ ("isCorrectRsg r,"++) $
                        showsWithDom r "r" "" ('\n':messgCanMap)

messgCanMap = "\nCanonical map modulo subgroup not found\n"



instance Cast (ResidueG a) a
  where
  cast mode (Rsg _ gdom d) a = case (mode=='r', subgrCanonic $ fst gdom)
                               of
                                 (False, _      ) -> Rsg a      gdom d
                                 (_    , Just cn) -> Rsg (cn a) gdom d
                                 _                -> error (...++messgCanMap)

instance Dom ResidueG where  dom (Rsg _ _ d) = d
                             sample          = resRepr
instance Residue ResidueG
  where
  resRepr   (Rsg x _ _) = x
  resGDom   (Rsg _ d _) = d
  resIDom   _           = error (..."resIDom (Rsg..)  is senseless")
```

```
resPIdeal _           = error (..."resPIdeal (Rsg..)...        ")
```

Then, the instances `Set ... AddGroup` are defined for `A/H`, `A` a base additive group, `H` a non-trivial subgroup in `A`.

See the implementation notes in `residue/QuotGr_.hs`

The idea of computing in the quotient group `A/H` is simple. For example, for `x + y`,

(1) find `z = (resRepr x)+(resRepr y)` in `A`,

(2) reduce `z` with the canonical reduction extracted from `H`.

```
instance (AddGroup a, Random a) => Random (ResidueG a)
  where
  randomR (l,h) g = (ctr l a, g') where
                                  (a, g') = randomR (resRepr l, resRepr h) g
```

— make a random representative and project it to the quotient group.

44 Generic residue ring ResidueI

44.1 Preface

It is exported by the module `Residue`,
implemented in `residue/ResRing*, ResPol_.hs`.

For the residues `Rsi x (iI, d) dm :: ResidueI R`, `R = baseRing x _`, the instances up to `LinSolvRing` require

a gx-ring `R` (`(IsGxRing, Yes)`)

and the generators for `iI` possessing the property `(IsGxBasis, Yes)`.

And the instances of `GCDRing, FactorizationRing, EuclideanRing, Field` are correct (and trivial) only for `R/I` being a field, that is `iI` possessing `(IsMaxIdeal, Yes)`.

A generic residue data type is declared as

```
data ResidueI a = Rsi a (Ideal a, Domains1 a) (Domains1 a)
```

A residue element from `a/I` for a gx-ring `a` is

```
Rsi x (iI, iD) aD
```

Here

`aD` is a bundle describing the base domain `a`,

`x` is a representative for a residue class by `I`,

`iD` is a bundle of sub-domain of the ideal `I` in `a`,

`iI` is the description of the ideal `I`,

`iI` contains its gx-generators `gs = fromJust $ idealGens iI`.

Required:

`x` must be reduced canonically by `gs` : `x == (fst $ moduloBasis "cg" gs x)`

Usually, the ideal description `(iD,iI)` is built by appying the function `gensToIdeal`.

Here follows the sketch for the beginning of implementation.

```
...
instance Dom ResidueI where  dom (Rsi _ _ d) = d
                             sample        = resRepr


instance Residue ResidueI
  where
  resRepr   (Rsi x _ _) = x
  resIDom   (Rsi _ d _) = d
  resGDom   _           = error (.."resGDom (Rsi..)   is senseless")
  resPIdeal _           = error (.."resPIdeal (Rsi..) ...         ")
```

```
reduceCanG :: LinSolvRing a => [a] -> a -> a                -- LOCAL
reduceCanG                     gs  =  fst . moduloBasis "cg" gs

isCorrectRsi :: LinSolvRing a => ResidueI a -> Bool
isCorrectRsi                     (Rsi x (iI, _) _) =
  case (idealGens iI, idealGenProps iI)
  of
    (Nothing, _    ) -> False
    (Just gs, props) -> case lookup IsGxBasis props of
                                              Just Yes -> x==(reduceCanG gs x)
                                              _        -> False

instance LinSolvRing a => Cast (ResidueI a) a
  where
  cast mode (Rsi _ (iI, dg) d) a = case (mode=='r', idealGens iI) of

                (False, _      ) -> Rsi a (iI, dg) d
                (_    , Just gs) -> Rsi (reduceCanG gs a) (iI, dg) d
                _                -> error (..."cast _ (Rsi..)" ...++msgNoBas)

instance (LinSolvRing a, Random a) => Random (ResidueI a)
  where
  randomR (l,h) g = (ctr l a, g') where
                                  (a, g') = randomR (resRepr l, resRepr h) g
          -- make a random representative and project it to the residue ring
```

The operations for `ResidueI a` are defined similar as for `ResidueE a`, only the reduction by gx-basis is applied instead of Euclidean remainder.

The operation

`divide_m (Rsi x (iI,_) _) (Rsi y _ _),   x, y :: a,`

requires a non-trivial interaction between `gxBasis` and `moduloBasis` in `a`. Here `gxBasis` is used as the generalization of the extended gcd of Euclidean ring. See Section 49, and the module `residue/ResRing*_.hs` for details.

44.2 Specialization ResidueI . Pol

`DoCon` defines several special instances for the type

`EuclideanRing a => ResidueI (Pol a)`

which *overlap* with the generic instances for `LinSolvRing b => ResidueI b` and describe a residue ring `Q = a`$[x_1, \ldots, x_n]$`/I` in the case of a c-Euclidean factorization ring `a`.

This specialization concerns the instances `Set ... Ring`.

The attributes for `Q` are computed more definitely than in the generic case and — following the below principles.

Theoretic Prelude

Denotations.

`rC` denotes the coefficient ring.

`R = rC[`$x_1, \ldots, x_n$`]` polynomial ring over `rC`.

`I = Ideal(gs)` a non-trivial ideal in `R`, `gs` a reduced Gröbner basis [Mo].

And `DoCon` builds `gs` so that it contains not more than one constant polynomial, and if it contains such a constant `g0`, then `g0` is in the head: `gs = g0:_`.

If `gs` does not contain a constant, `DoCon` puts `g0 = 0`.

`Q = R/I` a polynomial residue ring — the goal.

`cI = Ideal(c0,rC)` restriction ideal of `I` to `rC`, `c0 = lc(g0)` is not invertible.

`rC' = rC/cI =isomorphic= rC/(c0)`,

Lemma:

`rC'` is a field $\iff$ (`rC` is a field & `c0 = 0`) Or `c0` is prime.

`rC'` embeds injectively and naturally into `Q`, and `Q` is naturally a module over `rC'`.

`canRed = (\f -> moduloBasis "cg" gs f)` is a canonical reduction modulo `I` (mind that `rC` is c-Euclidean), it is extracted from the *quotient group*.

'Under' power product is the one that is not multiple of any `lpp(g)`, `g` from `gs`.

UPP = set of all 'under' power products for `gs`.

Lemma.

UPP is finite if and only if

for each $x_i \in$ `vars` there exists `g` $\in$ `gs` such that `lpp g` $= x_i^k$, with some $k > 0$.

For `Q = rC[`$x_1, \ldots, x_n$`]/I`, the main `DoCon` effort is to find the following attributes.

(Card) Cardinality.

(FldIn) Whether `Q` contains a field inside.

(DimP) If `Q` does contain a field, than what is `DimOverPrimeField` for `Q`.

DoCon solves this boldly, as follows.

Case (0) `c0 = 0`, `rC` is not a field.

Example: $Z[x, y]/(x^2 - 1, y^2 - x)$

`Q` contains `rC = Z` and is an algebra over `rC`. `card(Q) = Infinity`, (DimP) not defined.

Case (g0NP) `c0` $\neq$ `0` and not prime.

Example: $Z[x, y]/(4, x^2 + 2, y^3 + 1)$

To solve the primality of `c0`, `DoCon` first sees `(Prime, _)` for `I`, then the factorization list — if the latter presents in `I`.

Then, if it is still not defined, applies `isPrime c0`.
This `isPrime` is the only item from `(FactorizationRing a)` which is really used.
Here `Q = rC'[`$x_1, \ldots, x_n$`]/I'`, `I' = Ideal(g's)`,
g_i' is g_i with the coefficients represented as the residues modulo `g0`.

```
card(Q) = if  infinite rC' Or infinite UPP  then  Infinity
          else                                    Nothing
DimOverPrimeField =
  if  rC' contains a prime field  And  infinite DimOverPrimeField(rC')
      then  Infinity
  else      Nothing
```

More advanced solutions remain for the future.

Case (g0P) `c0` is prime Or (`c0 = 0` & `rC` is a field)

Examples:
$Z[x]/(5, x^2+2)$ =isomorphic= $GF(5)[x]/(x^2+2)$ =isomorphic= $GF(25)$,
$Rational[x, y]/(x^2-2, y^2+3)$.
Here `rC'` is a field embedded into `Q`.
As before, `Q = rC'[`$x_1, \ldots, x_n$`]/I'`, `I'= Ideal(g's)`,
and here `g's` is a Gröbner basis over the field `rC'`.
`Q` is an algebra over the field `rC'`, UPP is its vector space basis,

`dim(Q, rC') = |UPP|`, `card(Q) = card(rC')^|UPP|`
`card(rC')` is determined by the `ResidueE` constructor applied to `rC/(c0)`.
`DimOverPrimeField(Q) = DimOverPrimeField(rC')*|UPP|`

44.3 Application examples for generic residue ring

44.3.1 Computing in algebraic extension of $i, \sqrt[3]{2}$

(1) Extend the rational number field `Q` to the field `K` generated by the square root `i` of `-1` and cubic root `r` of `2`;
(2) find `qt = 1/(i+r)` in `K`;
(3) form some matrix `M` over `K` and find its reverse `M'`;
(4) test `(i+r)*qt = 1`, `M*M' = unityMatrx`;
(5) print these results.

This is done mainly by
(1) introducing polynomials `i', r'` from `P = Q[i,r]`,

(2) setting the equations `eqs` for `i',r'` — square and cubic respectively,

(3) creating the residue `Rsi _ iI dP` belonging to `K = P/Ideal(eqs)` :

```
import qualified Data.Map as Map (empty)
import DPrelude (PropValue(..), Z, ct)
import Categs
import SetGroup (zeroS)
import RingModule
import VecMatr  (Matrix(..), scalarMt)
import LinAlg   (inverseMatr_euc     )
import Fraction (Fraction(..)        )
import Pol
import Residue

type Q = Fraction Z
type P = Pol Q         -- for  P = Q[i,r]

main =
  let q1            = 1:/1 :: Q
      dQ            = upField q1 Map.empty
      vars          = ["i", "r"]
      p1            = cToPol (lexPPO 2) vars dQ q1   -- 1        of P
      ([i',r'], p2) = (varPs q1 p1, fromi p1 2)      -- i, r, 2  in P
      dP            = upGCDLinSolvRing p1 Map.empty

      ---                                     forming description of ideal
      eqs      = [i'^2 + p1, r'^3 - p2]              -- equations for i, r
      (iD,iI)  = gensToIdeal eqs [] propsGen propsI dP Map.empty
      propsGen = [(IsGxBasis, Yes)]
      propsI   = [(IsMaxIdeal,Yes)]
      ---
      k1       = Rsi p1 (iI,iD) dP      -- 1     of K
      [i,r]    = map (ct k1) [i',r']    -- i, r  in K
      qt       = k1 / (i+r)
      qtTest   = (i+r)*qt == k1
      mM       = [[i+r,    i-r^3],
                  [r^2-k1, i*r  ]]
      mM'      = inverseMatr_euc mM
      unityMt  = Mt (scalarMt mM k1 $ zeroS k1) Map.empty
      mTest    = (Mt mM Map.empty)*(Mt mM' Map.empty) == unityMt
  in
  putStr $ concat [shows qt "\n\n",
                   shows (Mt mM' Map.empty) "\n\n",
                   shows (qtTest, mTest) "\n"]
```

The property value `(IsMaxIdeal, Yes)` is an important hint for DoCon; it would then understand easily that `K = P/I` is a field.

`propsGen = [(IsGxBasis,Yes)]`

is also necessary here, this tells to `DoCon` that the generators are ready to present canonical reduction by `I`. In our case, `[i'`^{2}` + p1, r'`^{3}` - p2]` is, evidently, a Gröbner basis, hence, it is correct to set `(IsGxBasis, Yes)`.

Without this information, we might have to apply the functions `isGBasis, gxBasis`.

44.3.2 Cyclic integers

The so-called cyclic integers were studied first in 19-th century by E.Kummer. We may view them simply as residues of integer polynomials, the elements of domain $CI = Z[x]/(g)$,

$$g = \sum_{i=0}^{p'} x^i, \quad p' = p - 1, \quad p \text{ a } prime.$$

As we see, $g = (x^p - 1)/(x - 1)$, and it is also known that g is irreducible.

Concerning the arithmetic in CI, the book ([Ed] Section 4.2) describes, for example, certain special division procedure.

As to DoCon, it simply applies the constructor composition

`ResidueI $ UPol Z`,

with its underlaying technique of Gröbner bases over an Euclidean ring. Under these circumstances, `gBasis` does not cost too much, it is almost like pseudodivision in $Z[x]$.

See the program in `demotest/T_grbas2.hs`, and `cyclInt` sub-function. This demonstration computes some *norms* and quotients in this integral domain CI.

44.3.3 Cardano cubic extension

See first Section 3.2. Then, `demotest/T_cubeext.hs`.
The latter presents the function `cubicExt` which forms a residue ring `B/I` by the ideal generated by Cardano radical expressions.

45 Symmetric function package

45.1 Introduction

The symmetric functions are implemented to the extent of the first seven paragraphs of the widely known book [Ma].

The items related to symmetric functions are exported from the modules
`Partition, AlgSymmF`,
implemented in `Partition.hs, AlgSymmF.hs`, `pol/symmfunc/*.hs`

First, a couple of the general notices concerning `DoCon` approach to the subject.

By a symmetric function we mean here something like a power series of the infinite set of variables $x_1, x_2, \ldots$ — see [Ma]. For example, the second elementary symmetric function e_2 is

$$\sum_{i,j \geq 1} x_i x_j$$

Restricting it to any finite list of variables, we obtain really a symmetric polynomial.

Since a symmetric polynomial contains usually many monomials that are the permutations of each other, `DoCon` introduces specially a Symmetric Polynomial (sym-polynomial) data, with the `SymPol` constructor).

This is done for the needs of efficiency and also for providing a finite representation for symmetric function. Thus, the mentioned above e_2 is represented as the sym-polynomial $1 \cdot m_{1,1}$ (single sym-monomial).

A sym-polynomial is a linear combination of *monomial symmetric functions* (MSF) m_λ — see [Ma]. It is some linear combination of *partitions* (Young diagrams).

Not only the elementary symmetric decomposition (to e_λ) is presented, but generally, the R-module isomorphisms are given for the most usable linear bases

$$m_\lambda,\ p_\lambda,\ s_\lambda,\ e_\lambda,\ h_\lambda$$

in the symmetric function algebra (*SymA*) over a commutative ring R, as it is shown in [Ma]. Here λ denotes a partition.

This requires various operations for the *partitions*, they may also occur useful by themself.

And the central point is auxiliary functions for computing of the Kostka numbers for the partitions and *irreducible characters* for the permutation group. To compute the Kostka numbers and the above characters we apply forming of the so-called *horizontal bands* and *skew hooks* ([Ma]). For these data we also provide certain operational minimum.

The partitions in the sym-polynomial are ordered decreasingly by the given *partition comparison function*; this latter is an attribute contained in a sym-polynomial data.

The conversion between `Pol a` and `SymPol a` depends on the comparison function for the power products and the one for the partitions.

45.2 Partition

Its items are exported from the module `Partition`,
implemented in `Partition.hs, pol/symmfunc/HookBand_.hs, Partit_.hs`.

45.2.1 Prelude

Following the denotations by [Ma], let

λ, μ denote the partitions of a natural number k,

λ' conjugation of partition λ,

m_λ symmetric monomial function (see above the example of $m_{1,1}$),

e_i i-th elementary symmetric function,

$e_\lambda = \prod_{i \in \lambda} e_i$.

And assume similar as e_i, e_λ denotations for other bases: s_i, s_λ, p_i, p_λ, ...

In the programs and program commentaries, we often use the following denotations.

$\lambda, \mu, \ldots$ (partitions) $\longleftrightarrow$ `la, mu, pt, pt1, p, q` ....

For example, `la = [6,5,5,2]`

e_i, $p_i, \ldots$ $\longleftrightarrow$ `e(i), p(i), e_i, p_i, ei, pi}`,

$m_\lambda, e_\lambda, \ldots$ $\longleftrightarrow$ `m[la], e[la]` ...,

Also we use here the following terminology
(as *we* translate it back from Russian translaion of [Ma]):

A part of a partition may be also called a *row* or a *line*.

Block of a diagram λ
is a rectangular diagram corresponding to some pair (i, m) denoting i^m in λ.

Skew diagram (shape)
is the difference $\lambda - \mu$ of the Young diagrams, for any μ inside λ.

Skew Hook (s-hook, SHook)
is a continuous skew diagram in which the neighbour rows overlap in exactly one box.

s-w-hook is an s-hook of weight w.

A it hook (i, j) in a diagram
is the box with the coordinates (i, j); it has *length = arm + leg*.

Horizontal band (h-band, HBand)
is a skew diagram that contains not more than one box in each column.

h-w-band is an h-band of weight w.

What is a *tableau of shape* λ and weight μ, see in [Ma].

We represent a skew hook as integer quartet `(w, b, r, b')`.
Its components are as follows.

`w` is the weight of hook,

`b` No of the block in λ where this hook starts,

`r` No of the row in this block at the end of which this hook starts,

`b'` No of the last block of the hook — it has to terminate at the last row of this block.

`b'` can be derived from `w, b, r,` but still let it be introduced.
Though in principle, a skew hook can be determined by a single point in a diagram.

An h-band `hb` in a diagram λ is represented as an integer list `[`$b_1, \ldots, b_n$`]`.

This means that for

$$\lambda = [(i_1, m_1), \ldots, (i_n, m_n)],$$

the so-called b-block of the b_k boxes is marked in the lowest row of the k-th block of rows of λ (this block corresponds to (i_k, m_k)) — for

$$1 \leq k \leq n, \quad \sum_{\ldots} b_k = w, \quad n = \mathit{length}\ \lambda = \mathit{length}\ hb, \quad hb \text{ may } \mathit{contain\ zeroes.}$$

An h-band consists of the b-blocks, each b-block is several continuous boxes in the end of the lowest row of the block of λ.

Neither two b-blocks have a common column.

The type

```
type Partition = [(Natural, Natural)]
```

is for a partition, the one known from combinatoric.
This is a representation for the Young (Ferrers) diagram.

We represent a partition as `[]` or as

$$\lambda = [(j_1, m_1), \ldots, (j_k, m_k)], \quad j_1 > \ldots > j_k > 0,$$

which is traditionally denoted in mathematics as $(j_1^{m_1} \ldots j_k^{m_k})$, m_i the multiplicity of j_i in a partition.

Below, we describe some functionality and implementation for the partitions.

```
type EPartition = [Z]   -- expanded partition - obtained from Partition by
                        -- expanding each (j,m) to jj...j  (m-times).

type PrttComp = Comparison Partition   -- SymPol a  uses  PrttComp Z  similarly as
                                       -- Pol a     uses  PPComp
isPrtt :: Partition -> Bool                            -- test "is partition"
isPrtt []                   = True
isPrtt [(i,m)]              = i > 0 && m > 0
isPrtt ((i,m):(j,k):pairs) = i > j && m > 0 && isPrtt ((j,k):pairs)

toEPrtt :: Partition -> EPartition
```

```
toEPrtt = concat . (map (\ (j,m) -> genericReplicate m j))

fromEPrtt :: EPartition -> Partition
fromEPrtt []     = []
fromEPrtt (j:js) = case span (==j) js
                   of
                     (js', ks) -> (j, 1+(genericLength js')): (fromEPrtt ks)

prttToPP :: Z -> Partition -> PowerProduct
                         -- Convert partition  la = [i1^m1 ... il^ml]
                         -- into power product of given length  n >= 1.
                         -- Example:  7 [5^2, 4, 2^3] --> Vec [0,3,0,1,2,0,0]
ppToPrtt :: PowerProduct -> Partition
ppToPrtt =  filter ((/=0) . snd) . reverse . zip [1 ..] . vecRepr

prttWeight :: Partition -> Z                                -- |pt|
prttWeight [] = 0
prttWeight pt = sum [j*m | (j,m) <- pt]

prttLength :: Partition -> Z          -- l(pt) = height of Young diagram
prttLength = sum . map snd            -- = number of "actual variables"

conjPrtt  :: Partition -> Partition            -- conjugated partition pt'
prttUnion :: Partition -> Partition -> Partition
                          -- repeated diagram lines
                          -- are copied, say  [3,2,1] [3*2] -> [3*3, 2, 1]
pLexComp :: PrttComp
             -- we call             pLexComp
             -- what is called the  inverse lexicographical ordering  in [Ma]

pLexComp' :: PrttComp          -- conjugated pLexComp comparison,
                               -- in Macdonald's book [Ma] it is denoted  Ln'
pLexComp' p q = pLexComp (conjPrtt q) (conjPrtt p)

prttLessEq_natural :: Partition -> Partition -> Bool
  --
  -- natural partial ordering on partitions:
  -- la <= mu  <=>  for each i > 0  sum(la,i) <= sum(mu,i),
  -- where  sum(la,i) = la(1)+..+la(i)
  -- if we represent the partition without multiplicities

minPrttOfWeight :: Z -> Partition    -- minimal partition of
minPrttOfWeight    0 =  []           -- the given weight
minPrttOfWeight    n =  [(1,n)]

prevPrttOfN_lex :: Partition -> Maybe Partition
```

```
  -- Partition of k = |pt|  previous to  pt in the pLexComp order.
  -- Returns  Nothing  for the minimal partition
  -- and the minimum is here either  [(1,k)]  or  [].

prttsOfW :: Partition -> [Partition]
                 -- pt -> [pt...minPt]  all partitions of  w = |pt|
                 -- starting from  pt  listed in pLexComp -decreasing order
                 --
prttsOfW pt = maybe [pt] ((pt:) . prttsOfW) (prevPrttOfN_lex pt)



--------------------------------------------------------------------
randomEPrtts ::  [Z] -> Z -> [EPartition]
               --rands  w    pts
```

— infinite list pts of the *random expanded partitions* of `w` produced out of the infinite list `rands` of the random integers `n`, such that `0` $\leq$ `n` $\leq$ `w`.

`rands` may be obtained by, say `Random.random (0,w) s`

Caution: we are not sure that these partitions are "very random".

```
showsPrtt :: Partition -> String -> String

type SHook = (Z, Z, Z, Z)

  -- Skew hook:  in  (w, hb, hr, hb')
  -- w   is the weight of the hook,
  -- hb        No of the starting block,
  -- hr        No of the row in this block where the hook starts,
  -- hb'       No of the last block of the hook.

sHookHeight :: Partition -> SHook -> Z   -- numberOfRowsOccupied - 1
sHookHeight    la           (_, b, r, b') =
                             case genericTake (b'-b+1) (genericDrop (b-1) la)
                             of
                               laOfHook -> (sum $ map snd laOfHook) - r

subtrSHook :: Partition -> SHook -> Partition   -- subtract hook from partition
firstSWHook ::  Z -> Partition -> Maybe SHook
              --w    la
              -- first skew hook  la-mu  of weight  w  in the diagram  la  -
              -- the one with the highest possible head - if there exists any

prevSWHook :: Partition -> SHook -> Maybe SHook
  --
```

```
  -- Previous  sw-hook  to the given  sw-hook.
  -- The ordering is so that the greater is the hook which head starts higher.
  -- prevSWHook la h  is obtained by taking the part  laT  of partition  la
  -- after the head row of the hook and applying  firstSWHook to laT ...
  -- We give it as Example of programming with partitions:

prevSWHook [] _            = Nothing
prevSWHook la (w, b, r, _) =
  let
    ((i,m):mu) = genericDrop (b-1) la
    laT        = if r==m then mu  else (i, m-r):mu
  in
  case firstSWHook w laT
  of
    Nothing               -> Nothing
    Just (_, b', r', b'') -> (case (r==m, b')
                               of
                                 (True, _) -> Just (w, b'+b  , r'  , b''+b  )
                                 (_   , 1) -> Just (w, b     , r'+r, b''+b-1)
                                 _         -> Just (w, b'+b-1, r'  , b''+b-1)
                             )
type HBand = [Z]
subtrHBand :: Partition -> HBand -> Partition         -- partition \ h-band

-------------------------------------------------------------------------------
maxHWBand :: Char -> Partition -> Z -> Maybe HBand
             --mode  la           w
```

— maximal h-w-band in a partition `la`.

`mode = 'l'` is so far the *only* valid value, and it means that the ‘lex’ ordering is used on the positions in a diagram `la` :

`(i,j) > (i',j') =  i < i' || (i==i' && j > j')`,

where `i` is the number of the block of rows, `j` number of the column.

Comparing bands means comparing lexicographically the sequences of their cell positions — starting from the maximal position.

We give the implementation as an example of programming bands:

```
maxHWBand mode la w =
  if
    mode /= 'l' then
    error $ ("maxHWBand "++) $ (mode:) $ (' ':) $ shows la $ (' ':) $ shows w $
                      " :\nmode = \'l\'  is the only possible so far\n"
  else  mb la w
    where
    mb la         0 = Just $ map (const 0) la
```

```
    mb []          _ = Nothing
    mb [(i,_)]     w = if i < w then Nothing else Just [w]
    mb ((i,_):la) w = let {(j, _) = head la;  d = i-j;  w' = min d w}
                      in
                      fmap (w':) (mb la (w-w'))

prevHWBand :: Char -> Partition -> HBand -> Maybe HBand
                                  -- Previous (to the given) h-w-band.
                                  -- Ordering, mode  are as in  maxHWBand.
```

45.2.2 Kostka numbers

For the partitions λ, μ of weight w, the Kostka number $K(\lambda, \mu)$ is the number of *tableaux of the shape* λ and weight μ ([Ma]).

From [Ma] we study the folloing:

1. The integer upper uni-triangular matrix $K(\lambda, \mu)$ presents the Z-module isomorphism from the basis $\{s_\lambda\}$ of the Schur functions to the basis $\{m_\mu\}$ of the symmetric monomial functions.

2. $K(\lambda, \lambda) = 1$, $K([n], \mu) = 1$,
if *not* $(\lambda \geq \mu)$ in the natural (partial) ordering, then $K(\lambda, \mu) = 0$,
$K(\lambda,\ [1^w]) =$ number of Standard tableaux $= w! \ / \ (\prod_{h \in hooks(\lambda)} length(h))$

Finding the Kostka numbers $K(\lambda, \mu)$ looks like a hard combinatoric task. In the generic case, the number of tableaux can be found by reversing of the expanded μ and forming of all the h-m_i-bands in λ, subtracting these bands and applying the recursion.

And this, in fact, is implemented in DoCon.

The only optimization is usage of a *table* (binary tree) to store the pairs (λ, v), $v = K(\lambda, i_{mu})$ for some previously encountered λ and the initials i_μ of the weight μ.

This accelerates considerably the current and future computations of the Kostka numbers — even if one gives `Map.empty` in the argument. This is because the tableaux often repeat in the above process of the $K(\lambda, \mu)$ computation.

The tables are supported by the `Map.Map` item from the `data` library (a balanced binary tree programmed in `Haskell`). It preserves the functionality.

Maybe, there exists more efficient way to compute Kostka numbers, the one basing on some law for the remainder tableaux repetitions. So far, we do not know of such a method.

So, the format is

```
kostkaNumber table la mu --> (newTable, value)
```

```
kostkaNumber :: Map.Map Partition Z -> Partition -> Partition ->
                -- table              la           mu
                                                   (FiniteMap Partition Z, Z)
                                                    -- newTable            value
```

Number of tableaux of the shape `la` and weight `mu`, — for `|la| = |mu|`.

`table = tab(ro)` is a `Map.Map` (binary table) serving to store the pairs `(la, v)`, `v = K(la, i_mu),` for some previously encountered `la` and the initials `i_mu` of weight `mu`.

`i_mu` are not stored, for they are defined uniquely by `la`.

Each time the argument `la, mu` is non-trivial and not contained in the table, the value is found by the below method and added to table. Accumulating this table accelerates the current and future computation of the Kostka numbers that use the previously accumulated table — even if `Map.empty` is initiated in the argument.

But the table needs memory. The programmer can prevent further growth of the table by giving to the next `kostkaNumber` invocation the argument `table = Map.empty`.

See the examples for this presented by the functions `kostkaColumn, kostkaTMinor`.

<u>Repeated computation with Kostka numbers</u>

The below two function also illustrate an application of `kostkaNumber`.

The `kostkaColumn` script introduces the local variable `tab` for the table, initiating it with the empty value. `kostkaColumn` spends more space in order to increase the performance.

And `kostkaTMinor` 'maps' `kostkaColumn` to the list of partitions, it does not carry the table between the `kostkaColumn` invocations.

Accumulating `tab` all through the matrix `K(la,mu)` causes too much memory expense.

The same approach applies to the *irreducible character* matrix (see below).

```
kostkaColumn :: Partition -> [Partition] -> (Map.Map Partition Z, [Z])
                -- mu        la_s            tab                   col
```

yields the list `col = [K(la, mu) |  la <- la_s]` of the Kostka numbers, `la, mu` the partitions of the same positive weight, and the table of the accumulated values `K(la~, i_mu)` produced by the repeated application of `(tab', v) = kostkaNumber tab _ _`

In particular, setting `la_s = allPartitions(w)` we obtain the whole column of `mu` of the Kostka matrix.

```
kostkaColumn mu la_s =
  if
     null mu then  error $ showString "kostkaColumn [] la_s \n\n" $ msg "\n"
```

```
    else             (tab, reverse col)
      where
      (tab, col)          = foldl addKNum (Map.empty,[]) la_s
      addKNum (tab,vs) la = case kostkaNumber tab la mu of (tab',v) -> (tab',v:vs)
      msg  = ("length la_s = "++) . shows ... . msg'
      msg' = ... ("head la_s = "++) . shows la  ...

  kostkaTMinor :: [Partition]-> [Partition]-> [[Z]]
  kostkaTMinor la_s mu_s = [snd $ kostkaColumn la mu_s | la <- la_s]
```

— transposed minor of the matrix of the Kostka numbers.

`la_s, mu_s` must be partitions of same positive weight `w`. They define respectively the rows and the columns of transposed minor.

In particular, setting `la_s = mu_s = all_partitions(w)` yields the whole transposed Kostka matrix.

Certain **property:**
it is known [Ma] that `K` is strongly upper uni-triangular and has 1 -s in the first row. So, the result is strongly lower uni-triangular.

Remark.
In practical computation with the symmetric functions, there often appear quite large Kostka matrices.

And we are lucky that the columns of a Kostka matrix are computed independently. The positions in a Kostka matrix are indexed by the partitions. This gives the effect of a sparse matrix represented as the map of type

```
[Partition] -> [Partition] -> [[Z]]
```

The same is with the character matrix (see below).

```
hookLengths ::  Partition -> [[Z]]
```

The matrix `{h(i,j)}`, `h(i,j)` is the length of the hook of the point `(i,j)` in the given Young diagram `la`.

The result (a list of lists of integers) has the form of *expanded partition* for `la` (and the lists may differ in length), each `h(i,j)` being put in the corresponding cell of the diagram.

```
numOfStandardTableaux ::  Partition -> Z
```

Number of standard tableaux $=$ $weight(\lambda)!\ /\ (\prod_{h \in hooks(\lambda)} length(h))$

The program tries to keep the intermediate products possibly small.

45.2.3 Irreducible characters for $S(n)$

The group representation theory ([La], Chapter 18) asserts the following.

1. A character χ of a group G is a map $g \mid\to Trace(F(g))$, where F is any representation of G by the linear operators on some vector space V.

2. The character values $\chi(g)(h)$ depend only on the conjugacy classes of g, h in G. For the permutation group $S(n)$, these implies that $\chi(g)(h)$ is defined by the cyclic types of the permutations g, h — and these types can be represented as the partitions of n. So, we write $\chi(\lambda)(\mu)$, λ, μ the partitions of n.

3. Irreducible representations and their characters are important.
For a finite group, the irreducible characters form an orthogonal basis.

For our program, this means that the matrix C of irreducible character values has the reciprocally orthogonal rows — in usual meaning of the scalar product of rows.

In the case of symmetric functions, the group is $S(n)$, and the character matrix C presents the isomorphism from the basis $\{p_\lambda\}$ to $\{s_\lambda\}$ — see [Ma].

$C(\lambda, \rho)$ can be computed by the Murnaghan-Nakayama rule [Ma], and `DoCon` does this

(like with the Kostka numbers, there might exist more efficient way to compute these character values).

`DoCon` achieves certain optimization by using of the intermediate binary table — similarly as with the Kostka numbers.

```
permGroupCharValue :: Map.Map Partition Z -> Partition -> Partition ->
                      -- table               la           ro

                                           ( FiniteMap Partition Z,  Z )
                                             -- newTable           chi(la,ro)
```

Irreducible character value for the permutation group $S(n)$.
It is known [Ma] that any such character `chi` is defined by some partition `la` of n, and `chi` can be expressed as certain determinant of the unit characters of the groups S`(la(i))`.

`table = tab(ro)` is a `Map.Map` (binary table — made functionally) serving to store the pairs `(mu, chi(mu, i_ro))` for some previously encountered `mu` and the initials `i_ro` of `ro`.

`i_ro` are not stored, for they are defined uniquely by `mu`.

Even if `table = Map.empty`, it still accumulates so that it accelerates on average the computation of current and further `chi(la, ro)` (for the same ro), see, for example, the function `permGroupCharMatrix`.

Method:
Murnaghan-Nakayama rule + storing/searching in table.

The tabulation applies for the similar reason as in `kostkaNumber`.

```
permGroupCharColumn :: Partition -> [Partition] -> (Map.Map Partition Z, [Z])
                       -- ro        la_s           tab                 col
```

This yields

(1) the list `col = [cha(la,ro) | la <- la_s]` of the character values obtained from `permGroupCharValue`, `la, ro` the partitions of the same positive weight,

(2) the table of accumulated values `cha(la~,i_ro)` produced by the intermediate applications of `(tab',v) = permGroupCharValue tab _ _`

In particular, setting `la_s` = all partitions, we obtain the whole column of irreducible character matrix for $S(w)$.

```
permGroupCharTMinor :: [Partition] -> [Partition] -> [[Z]]
permGroupCharTMinor ro_s la_s = [snd $ permGroupCharColumn ro la_s | ro<- ro_s]
```

Transposed minor `tC` of the matrix of values of irreducible characters `cha(w)(la,ro)` for the permutation group $S(w)$.

`ro_s, la_s` must be partitions of same positive weight. They define respectively the rows and columns of transposed minor.

In particular, setting `ro_s = la_s = all_partitions(w)` yields the whole transposed character matrix `tC`.

One of the tests: `tC * (transp tC)` must be diagonal; this is due to orthogonality of irreducible characters.

45.3 Sym-polynomial

45.3.1 Preface

Its items are exported from the module `AlgSymmF`,
implemented in `AlgSymmF.hs, pol/symmfunc/Sympol_.hs`

A sym-polynomial is a linear combination of the monomial sym-functions $m(\lambda)$, that is a linear combination of partitions. DoCon represents it similarly as a polynomial over R, but with the following differences:

(1) it contains partitions instead of the power products,

(2) it contains `pcp :: PrttComp` instead of `PPComp` (see Section 45.2.1),

(3) it does not contain the variable list.

The sym-monomials in `SymPol` must be ordered decreasingly by `pcp`.

For a *polynomial* data, the variables (algebraic indeterminates) indicate the *domain*. For example, `f` may belong to $Z[x]$ or to $Z[x,y]$, depending on the list `vars = pVars f`.

A sym-polynomial `f` does not refer to the variables, because `DoCon` presumes `f` to relate to the infinite variable list $x_1, x_2, \ldots$ (may be, renamed). Say $p_2 = x_1^2 + x_2^2 + \ldots$ can be represented as `SymPol [1*m[2]] 0 pLexComp _`

The operations with the sym-polynomials require a

coefficient domain R being a commutative ring with unity.

Here are the descriptions of some main items related to the sym-polynomials.

Note that `SymPol` is one more model for the `class PolLike`.

45.3.2 Initial definitions

```
type SymMon a = (a, Partition)   -- like  Mon a, Pol a,
                                 -- only with partition instead of power product
data SymPol a = SymPol [SymMon a] a PrttComp (Domains1 a)

instance Dom SymPol where  sample (SymPol _ a _ _) = a
                           dom    (SymPol _ _ _ d) = d

symPolMons     :: SymPol a -> [SymMon a]
symPolPrttComp :: SymPol a -> PrttComp

symPolMons     (SymPol m _ _  _) = m
symPolPrttComp (SymPol _ _ cp _) = cp

symLm :: CommutativeRing a => SymPol a -> SymMon a    -- leading sym-monomial
symLm f = case symPolMons f of m:_ -> m
                               _   -> error$ ("symLm 0 \nin"++)$ showsDomOf f "\n"
symLdPrtt :: CommutativeRing a => SymPol a -> Partition
symLdPrtt =  snd . symLm

instance Eq a => Eq (SymPol a) where  f==g = (symPolMons f)==(symPolMons g)

instance AddGroup a => Cast (SymPol a) (SymMon a)  --sym-monomial to sym-polynomial
  where
  cast mode (SymPol _ c cp dm) (a,p) = SymPol mons c cp dm
                      where
                      mons = if mode=='r' && isZero a then []  else [(a,p)]

instance AddGroup a => Cast (SymPol a) [SymMon a]       -- from sym-mon list
  where
  cast mode (SymPol _ c cp dm) mons = SymPol ms c cp dm
    where                                                -- order NOT checked
    ms = if mode /= 'r' then mons  else filter ((/= z) . fst) mons
    z  = zeroS c
```

```
instance Ring a => Cast (SymPol a) a           -- from coefficient
  where
  cast mode (SymPol _ _ cp dm) a = case mode of 'r' -> cToSymPol cp dm a
                                                _   -> SymPol [(a,[])] a cp dm
--------------------------------------------------------------------
instance PolLike SymPol
  where
  pIsConst f = case symPolMons f of (_, p): _ -> null p
                                    _         -> True
  pCoefs  = map fst . symPolMons

  pTail f = case symPolMons f of
                      _: ms -> ct f ms
                      _     -> error$ ("pTail 0  \nin"++) $ showsDomOf f "\n"
  pFreeCoef (SymPol mons a _ _) = case (last mons, zeroS a)
                                  of
                                  ((a, p) ,z) -> if null p then a else z

  ldeg f = case symPolMons f of (_, p): _ -> prttWeight p
                                _         -> error ("ldeg 0" ...)

  deg f = case map (prttWeight . snd) $ symPolMons f of  [] -> error ("deg 0"...)
                                                         ds -> maximum ds
  pCDiv f c         = ... similar to Pol
  pMapCoef mode f g = cast mode g [(f a, p) | (a, p) <- symPolMons g]

  -- Skipped:
  -- pMapPP, pPPO, pVars, lm, lpp, pDeriv, degInVar, varPs, pDivRem, pValue
--------------------------------------------------------------------
cToSymMon :: a -> SymMon a    -- correctness condition:  c /= 0
cToSymMon    a =  (a, [])

cToSymPol :: AddGroup a => PrttComp -> Domains1 a -> a -> SymPol a
cToSymPol cp dm a = if
                      isZero a then  SymPol []            a cp dm
                    else             SymPol [cToSymMon a] a cp dm

instance Show a => Show (SymPol a)
  where
  showsPrec _ f = ("(SymPol "++).
                 (foldr (\mon f-> showsMon mon .f) (" )  "++) $ symPolMons f)
                   where
                   showsMon (c,la) = shows c . ('*':) . showsPrtt la . (' ':)

reordSymPol :: PrttComp -> SymPol a -> SymPol a      -- bring to new ordering
reordSymPol    cp (SymPol mons c _ dm) =
```

```
                          SymPol (reverse $ sortBy (compBy snd) mons) c cp dm

monToSymMon :: Mon a -> SymMon a
monToSymMon (a, Vec js) = (a, gather $ reverse $ sort$ filter (/= 0) $ js)
  where
  gather []     = []
  gather (j:js) = case span (==j) js
                  of
                  (js', js'') -> (j, 1 + (genericLength js')) : (gather js'')
```

```
--------------------------------------------------------------------
  symPolHomogForms :: AddGroup a => SymPol a -> [SymPol a]
```

The list `hs` of the homogeneous parts of `f`.
`hs(f)` is empty for zero `f`,
otherwise, `h(1)` is the homogeneous form of `lm f`, `h(2)` of `lm (f-h(1)`, and so on.

45.3.3 Main instances

The algebraic category instances are defined naturally for `SymPol` — from `Set` up to `AddGroup`.

For the instance `...Num SymPol ...`, the operations `negate, (+), (-)` are defined, and `(*)` is skipped, as we do not know so far how to define it.

45.3.4 Conversion Pol - SymPol

```
toSymPol :: Eq a => PrttComp -> Pol a -> Maybe (SymPol a)
                    -- pcp      f               symF
```

Given a polynomial `f`, symmetric under the permutations of its variables, partition comparison `pcp`,
produce the sym-polynomial `symF` by collecting each monomial orbit into corresponding sym-monomial.

Yields `Just symF` for symmetric polynomial, `f`, otherwise, yields `Nothing`

```
symmetrizePol :: CommutativeRing a => PrttComp -> Pol a -> Maybe (SymPol a)
symmetrizePol pcp f =
                   case fromi (sample f) $ factorial $ genericLength $ polVars f
                   of
                     nFactorial -> pCDiv (symmSumPol pcp f) nFactorial
```

(see `symmSumPol`, as somehow more generic)

converts polynomial `f` to symmetric form polynomial under the given partition ordering `pcp`.

It sums up the symmetric orbit and divides by $n!$, $n = length(vars)$.

Required: $n!$ must not be zero in a ring `a`.
Also if the above quotient by $n!$ does not exist, the result is `Nothing`.

Examples.
(1) For a field `a` with `char > n`, the result is of kind `Just sf` ;
(2) For `a = Z/4`, `n = 3`, symmetric `f`, it is `Just sf`,
for non-symmetric `f`, it may be `Nothing`.

```
symmSumPol :: CommutativeRing a => PrttComp -> Pol a -> SymPol a
                                   -- pcp      f        symF
```

`symF = n!*(symmetrizePol f),  n = length vars`,
`symF` is under the given partition ordering `pcp`.

Method.
Convert the power products `pp` to partitions `[i,j,...]`. Gather the pp-s of same orbit, that is of same partition. Each `orbit(i,j,...)` sum is `c(i,j,...)*m[i,j,...]`, with appropriate coefficient `c(i,j,...)` = stabilizator order of pp `(i,j,...)` in the group $S(n)$.

```
fromSymPol :: CommutativeRing a => Pol a -> Domains1 (Pol a) -> SymPol a -> Pol a
                                   -- smp   dP                  f
```

Expand sym-polynomial to polynomial of the given sample `smp`.

Method:
`f` converts to $h(e_1, e_2, \ldots)$, e_i the elementary symmetrics;
then, the expressions $e_i(vars)$ are substituted in h.
Here it is set $e_i = 0$ for $i > n = |vars|$.

Caution: this may be very expensive, think before applying it.

45.3.5 Example

Form the elementary symmetric polynomials $[e_1, \ldots, e_n]$ in `P =` $Z[x_1, \ldots, x_n]$, $n = 4$, and find the sym-pol form of $e_2 + e_3$.

```
import DPrelude (Z)
...
main = let  vars                = ["x1","x2","x3","x4"]
            samplePol           = cToPol (lexPPO 4) vars dZ 1  :: Pol Z
            dP                  = upRing samplePol Map.empty
            elems@(_:e2:e3:_) = elemSymPols samplePol dP
       in
       fromMaybe (error "f is not symmetric\n") $ toSymPol pLexComp (e2+e3)
```

`show` has to print the result as `"(SymPol 1*[1*3] 1*[1*2] )"`
— here the expanded partitions are `[1,1,1], [1,1]`.

45.4 Symmetric bases transformations

They are exported from the module AlgSymmF,
implemented in AlgSymmF, pol/symmfunc/SymmFn*_.

45.4.1 Preface

A symmetric functions is often considered as something that may be decomposed into the polynomial of elementary symmetric functions. But it is known a more systematic approach to the transformations of such kind — see [Ma]. Following this book, DoCon implements the transformations that are the R-module isomorphisms

```
M(u,v)
```

for the commutative coefficient ring R of the symmetric function algebra, where

```
u, v =  'e', 'h', 'm', 'p', 's'
```

For example, M(m,e) presents the decomposition to the elementary symmetrics.

The DoCon view of the subject is as follows. The generic transformation format is

```
to_<v>      msgMode basisId tab   f ->  (tab, h)          (T1)

to_<v>_pol  msgMode basisId tab o f ->  (tab, h)          (T2)
```

basisId = u: t, u, <v> = 'e', 'h', 'm', 'p', 's' are the basis names.

'e' means the elementary symmetrics e_i, e_λ,

'h' full homogeneous functions h_i, h_λ,

'm' symmetric monomials m_λ,

'p' power sums p_i, p_λ,

's' Schur functions s_i, s_λ.

(T1), (T2) mean the symmetric function basis transformation from <u> to <v>.

The format (T2) differ from (T1) in that it converts the result to the *polynomial* — in the given pp-ordering o and variables [<v>$_1$,<v>$_2$,...,<v>$_n$], $n = max(1, weight(f))$.

msgMode :: SymmDecMessageMode specifies how to issue intermediate messages. Because as some sym-monomials take long to decompose to another basis, it may be convenient to issue intermediate messages about which partition is being currently decomposed.

Coefficient ring:

to_p requires a field of zero characteristic,
others — any commutative ring R with unity.

`basisId`
is a string `u: t` of one or two letters,

`u <- ['e','h','m','p','s']` is the name of basis,

`t` is either `[]` or `"n"`, we call it a proper mode.

So far, `"n"` may be used *only* in the case of `to_e(_pol) msgMode "mn"`.

`t = "" | "n"` chooses between the two computation methods.
We shall return to this subject later.

`tab`
can be set as `Map.empty`.
Those who do not want to care for `tab`, may set the program like this:

```
let (_, h) = to_s msgMode "m" Map.empty f  in  what_is_needed h
```

(and for `msgMode`, the simplest choice is `NoSymmDecMessages`).

This table `tab` (see `'type SymFTransTab = ...'`) contains the pairs

```
(w, (pts, tC, tK)),
```

`w` the integer weight,

`pts` partitions of `w` listed decreasingly by `pLexComp`,

`tC` transposed irreducible character ptp-matrix for $S(n)$,

`tK` transposed Kostka ptp-matrix — see `SymFTransTab`.

These matrices are for the weight `w`. And they are the ptp-matrices, that is the tables of type

```
Map.Map Partition [Integer],
```

the row being looked up by the partition index.

`tab` accumulates automatically and is returned in the result.
This is arranged so because `DoCon` commonly uses the matrices C, K for the above transformation, the elements of C, K are expensive to to evaluate and are often re-used. Each transformation uses and updates only certain part of `tab`. Thus, `m_to_p` ignores the table at all.

Again, each matrix from `tab` is 'lazy', it unfolds its rows only when needed. It depends on the sequence of transformations and their data which parts of `tab` will actually work.

Example: `to_e _ "m" Map.empty f --> (tab, h)`

means the `M(m,e)` transformation (from 'm' to 'e').

`to_<v> msgMode (u: t)` means the conversion of $f = \sum_{\lambda \dots} c_\lambda u_\lambda$ to $h = \sum_{\mu \dots} a_\mu v_\mu$, λ, μ the partitions [Ma].

For $\lambda = [\lambda_1, \dots, \lambda_l]$,
$\texttt{<u>}_\lambda =_{def}$ symmetric monomial m_λ, if $\texttt{<u>} = \texttt{'m'}$, otherwise, it is $\prod_{i=1}^{l} \texttt{<u>}_{\lambda_i}$

For example, $p_{5,3,3,1} = p_5 p_3 p_3 p_1 = p_5 p_3^2 p_1, \quad m_{5,3,3,1} =$ symmetric orbit of $x_5 x_3^2 x_1$

Ordering:
the sym-monomials in f may be ordered arbitrarily.
But the ones of h are in the pLexComp ordering only — apply reordSymPol if needed.

Another example: to_e msgMode "mn" Map.empty f

In basisId, t = "n" means to convert from <u> to <v>, applying the method 'via-p'. It avoids the tables and matrices K, C (in some examples this may save cost).

About the to_e conversion from 'm' :
it performs as the composition 'm' --mToS-> 's' --sToE-> 'e'.
— from 'm' to the Schur basis, and then, to 'e'.

Here the step mToS can perform in two ways. If basisId = "m", it converts by the inverse Kostka operator.

If basisId = "mn", it avoids the Kostka operator and converts first to 'p' by the Newton-Serret formulae, and converts further to 's' via the irreducible character matrix tC.

t = "" may be particularly efficient when K or C is not large and there are many repeating monomial weights in f.

Variable list

The result polynomial of to_<v>_pol needs the variable list.
DoCon sets it to be [<v>$_1$,…,<v>$_n$], $n = max(1,\ weight f)$,
they can be renamed easily if necessary. This is set so because it would suffice this many variables to print the result. For example,

```
show $ snd $ to_e_pol msgMode "m" tab o (SymPol [m[3]] _ _ _)
```

will look like "e1^3 - 3*e1*e2 + 3*e3" when printed; here $n = 3$.

45.4.2 Summary

```
to_<v>      msgMode basisId tab   f ->  (tab, h)          (T1)

to_<v>_pol  msgMode basisId tab o f ->  (tab, h)          (T2)
```

mean to transform a symmetric function `f` from the basis `<u>` to `<v>` over a commutative coefficient ring.

`basisId = u:  t,` u, `<v>` = `'e', 'h', 'm', 'p', 's'` are the basis names.

`t` is either `[]` or `"n"`,

`"n"` may be used so far only in `to_e(pol) "mn"`

The table `tab` (initiate it with `Map.empty`) contains the pairs `(w, (pts, tC, tK))`,
`w` the weight, `pts` the partitions of `w`,
`tC, tK` the ptp-matrices for the irreducible characters and Kostka numbers respectively.

`to_p(_pol)` requires a field of zero characteristic.
`(T1)` converts to the sym-polynomial.
`(T2)` — to the polynomial in the given pp ordering `o` and variables
$[\texttt{<v>}_1, \texttt{<v>}_2, \ldots, \texttt{<v>}_n]$, $n = \max(1,\ weight(f))$.

```
type SymmDecBasisId = String

data SymmDecMessageMode =  DoSymmDecMessages Integer | NoSymmDecMessages
                           deriving (Eq, Show)
```

Here `DoSymmDecMessages w` means to issue a short message about each current sym-monomial being processed — if its partition weights not less than `w`.

```
to_e, to_h, to_m, to_s ::

  CommutativeRing a
  =>
  SymmDecMessageMode -> SymmDecBasisId -> SymFTransTab -> SymPol a ->
                                                    (SymFTransTab, SymPol a)

to_p ::
  Field k   -- REQUIRED is  char(k) = 0
  =>
  SymmDecMessageMode -> SymmDecBasisId -> SymFTransTab -> SymPol k ->
                                                    (SymFTransTab, SymPol k)
```

```
to_e_pol, to_h_pol, to_m_pol, to_s_pol ::

  CommutativeRing a
  =>
  SymmDecMessageMode -> SymmDecBasisId ->
  SymFTransTab       -> PPOrdTerm      -> SymPol a -> (SymFTransTab, Pol a)

to_p_pol ::

  Field k   -- REQUIRED is  char(k) = 0
  =>
  SymmDecMessageMode -> SymmDecBasisId ->
  SymFTransTab       -> PPOrdTerm      -> SymPol k -> (SymFTransTab, Pol k)
```

The four `to_<v>_pol` functions differ from `to_<v>` in that they return *polynomial* — in the given pp ordering `o`, and variables [`<v>`$_1$,`<v>`$_2$,...,`<v>`$_n$], $n = \max(1, \deg f)$.

Method.

`to_<v>_pol` consists mostly of `to_<v>`.
Only in the end, it is applied `toDensePP_in_symPol o vars`, which converts each partition λ from `symPol` into the power product of the length n by `prttToPP`.

Then the polynomial is reordered by `o`.

Method for transformations

See the preface in the modle `AlgSymmF.hs`,
comments in `pol/symmfunc/SymmFn*_.hs`.

45.4.3 Other items

```
type PrttParamMatrix a = Map.Map Partition [a]
                    --
                    -- a partition-parameterized matrix over ‘a’ (ptp-matrix)
                    -- is a table of pairs (Partition,Row)

type SymFTransTab = Map.Map Z ([Partition], PrttParamMatrix Z, PrttParamMatrix Z)
                              -- pts       tC                 tK
                                                            -- see Preface
ptpMatrRows :: PrttParamMatrix a -> [[a]]
ptpMatrRows tab =  map snd $ sortBy gtLex $ fmToList tab
                                            where
                                            gtLex (p,_) (q,_) = pLexComp q p

transpPtP :: PrttParamMatrix a -> PrttParamMatrix a
transpPtP tab = listToFM $ zip pts $ transpose rows
                      where
                      (pts,rows)        = unzip $ sortBy gtLex $ fmToList tab
```

```
gtLex (p,_) (q,_) = pLexComp q p
```

The construction of $p_i(x_1,\ldots,x_n)$, $e_i(x_1,\ldots,x_n)$ is *not* used in the decomposition. They serve for testing and other needs.

```
elemSymPols :: CommutativeRing a => Pol a -> Domains1 (Pol a) -> [Pol a]
                                    -- f      dP
```

Elementary symmetric polynomials $[e_1,\ldots,e_n]$, e_i from $P = R[x_1,\ldots,x_n]$,
built from given sample polynomial `f`; `dP` is the description for P.

Example: `elemSymPols f $ upRing f Map.empty`

builds certain small necessary part of description for P, and then, $[e_1,\ldots,e_n]$.

Method:

`let g =` $\prod_{i=1}^{n}(y+x_i)$ `(of` $P[y]$`) in coefficients $ pTail g`

```
hPowerSums :: CommutativeRing a => Pol a -> [Pol a]
```

Homogeneous power sums $[p_1,p_2,\ldots]$, $p_i = \prod_{k=1}^{n} x_k^i$,
built from given sample polynomial.

```
h'to_p_coef :: Partition -> Z
h'to_p_coef    []         = 1
h'to_p_coef    ((i,m):la) = (factorial m)*(i^m)*(h'to_p_coef la)
h'to_p_coef    ((i,m):la) = (factorial m)*(i^m)*(h'to_p_coef la)
```

Coefficient z_λ of partition λ in the formula $h_n = \sum_{|\lambda|=n} p_\lambda/z_\lambda$
expressing the full homogeneous function as a linear combination of p_λ over rational numbers — see ([Ma] 1.2 formula (2.14')).
Namely, `h'to_p_coef` $[(i_1,m_1),\ldots,(i_l,m_l)] = \prod_{1\le k\le l} i_k^{m_k} m_k!$
This is more direct and nice than iterating the Newton formula.
And it is also used in the $e_n \longrightarrow p_\lambda$ decomposition.

```
intListToSymPol :: Ring a =>
                   Char -> SymPol a -> Partition -> [Partition] -> [Z] -> SymPol a
                -- mode    smp         bound        allPts         row
```

converts integer list `row` to sym-polynomial under `pLexComp` ordering, considering `row` as numeration of partitions.

`smp` is the sample sym-polynomial (maybe, not in `pLexComp`).

`row` is an integer list representing a dense homogeneous sym-polynomial over `Integer` of positive weight w: `f =` $\sum_{\lambda\in allPts} i(\lambda)\lambda$ — in `pLexComp` ordering.

`allPts` is the full list of partitions of w ordered decreasingly by `pLexComp`.

`f` converts to sym-pol `g` over `a`, mostly, by filtering out zero monomials.

`mode = 'a'` means to run through all the partitions ignoring 'bound',
`'u'` means the row is zero beyond the segment `[maximal, bound]`,
`'l'` means the row is zero beyond the segment `[bound, minimal]`.

This is all used for the integer vectors `row` who are computed 'lazily'.

45.5 Examples on symmetric transformation

45.5.1 Finding discriminant polynomial

Problem:
for the polynomial scheme $f = x^n + c_1 x^{n-1} + c_2 x^{n-2} + c_n$,
derive the algebraic condition on c_i for f to have a multiple root.

Usually, the algebra guides solve this task as follows. The searched condition expresses as

$$0 = d(x_1, \ldots, x_n) = \Big(\prod_{1 \leq i < j \leq n} (x_i - x_j) \Big)^2,$$

where x_i denote the roots of f in some extension field.
$d \in Z[x_1, \ldots, x_n]$ is symmetric, hence it decomposes to $d = h(e_1, \ldots)$, $h \in Z[e_1, \ldots]$, e_i the elementary symmetrics. And $e_i = c_i$ or $-c_i$ because of the Viete's formula for $c_i(x_1, \ldots, x_n)$. Hence, decomposing d to $h(e_i)$ with `to_e_pol` gives the needed coefficients of the discriminant polynomial. Let us program this.

n is given. $d = discr(n) = d(x_1, \ldots, x_n)$ computes as the element of $Z[x_1, \ldots, x_n]$, converts to the sym-pol form `discrS` and decomposes to `h` $\in Z[e_1, \ldots, e_w]$ in elementary symmetrics by `to_e_pol`. The result is `(discrS, h)`.

```
--------------------------------------------------------------------------------
import qualified Data.Map as Map (empty)
import DPrelude  (product1)
import SetGroup  (sub)
import Z         (dZ)
import Pol       (PolLike(..), Pol(..),  lexPPO, cToPol)
import Partition (pLexComp)
import AlgSymmF  (SymPol(..), SymmDecMessageMode(..), SymmDecBasisId,
                  toSymPol, to_e_pol
                 )

discrimToE :: SymmDecBasisId -> Integer -> (SymPol Integer, Pol Integer)
discrimToE    basisId           n       =
```

```
let pcp  = pLexComp
    vars = map (('x':) . show) [1..n]      -- ["x1"..."x<n>"]
    o    = lexPPO n
    unP  = cToPol o vars dZ 1              -- 1 of P = Z[x1...xn]
    listDiscr fs = (product1 $ diffs fs)^2
                           where
                           diffs [_]    = []
                           diffs (f:fs) = (map (sub f) fs) ++ (diffs fs)
                           --
                           -- example: [a,b,c] -> (a-b)^2*(a-c)^2*(b-c)^2

    xPols       = varPs 1 unP          -- x_i  as polynomials
    discr       = listDiscr xPols
    Just discrS = toSymPol pcp discr
    w           = deg discrS
    messageMode = NoSymmDecMessages
in
(discrS,  snd $ to_e_pol messageMode basisId Map.empty (lexPPO w) discrS)

main = let n           = 2                        -- edit this
           basisId     = "mn"                     -- alternative: "m"
           (discrS, h) =  discrimToE basisId n
       in
       putStr $ concat ["discrS =\n", shows discrS "\n\n",
                        "h =\n",      shows h "\n"
                       ]
```

Now, run it for `n = 2, 3, 4, 5`. The result size seems to grow exponentially in `n`.

In `DoCon-2.11`, the `"mn"` way occurs in this example `n` times faster than `"m"` — for `n = 2, 3, 4, 5`.

```
n = 2.  discrS = SymPol [(1, [(2,1)]), (-2, [(1,2)])] _ _ _#
                                            -- does not depend on  vars
h = e1^2 - 4*e2
--------------------------------------------------------------------
n = 3.  discrS = SymPol [(1, [(4,1),(2,1)]), (-2, [(4,1),(1,2)] ),
                         (-2,[(3,2)]),      (2, [(3,1),(2,1),(1,1)]),
                         (-6,[(2,3)])
                        ] _ _ _
h = -4*e1^3*e3 + e1^2*e2^2 - 8*e1^2*e4 + 18*e1*e2*e3 - 4*e2^3
      + 24*e2*e4 - 27*e3^2
--------------------------------------------------------------------
n = 4.  discrS = SymPol [(1,  [(6,1),(4,1),(2,1)]),
                         (-2, [(6,1),(4,1),(1,2)]) ... (24, [(3,4)])
                        ] _ _ _
h = 24*e1^4*e3*e5 - 27*e1^4*e4^2 - 8*e1^3*e2^2*e5 +
```

```
    ...
    + 200*e2*e5^2 - 27*e3^4 + 216*e3^2*e6 - 480*e3*e4*e5 + 256*e4^3
--------------------------------------------------------------------
(SymPol 1*[8,6,4,2] -2*[8,6,4,1*2] -2*[8,6,3*2] 2*[8,6,3,2,1]
  -6*[8,6,2*3] -2*[8,5*2,2] 4*[8,5*2,1*2] 2*[8,5,4,3] -2*[8,5,4,2,1]
  ...
  -12*[5*3,3,2] 12*[5*2,4*2,2] 12*[5*2,4,3*2] -24*[5,4*3,3] 120*[4*5]
)
h = -192*e1^5*e3*e5*e7 + 200*e1^5*e3*e6^2 + 216*e1^5*e4^2*e7 -
    480*e1^5*e4*e5*e6 + 256*e1^5*e5^3 + 72*e1^4*e2^2*e5*e7 -
    75*e1^4*e2^2*e6^2 - 168*e1^4*e2*e3*e4*e7 ...
    ...
    + 5600*e4^2*e5*e7 + 2880*e4^2*e6^2 - 9000*e4*e5^2*e6 + 3125*e5^4
```

45.5.2 Other examples

`demotest/T_symfunc.hs` contains the examples with the Kostka and permutation character matrices, and others.

46 Non-commutative polynomials

DoCon provides some support for a *f*ree associative algebra over a commutative ring (**FAA**). That is for non-commutative polynomials over a commutative ring.

For example, for the non-commutative polynomials in `x, y` over `Integer`

```
f = 2*x*3*y = 6*x*y;   g = 3*y*2*x = 6*y*x;   f == g = False
```

The corresponding items are re-exported from the module `Pol` (together with the items for commutative polynomials), and they are defined in the modules

`FreeMonoid, FAAO_, FAANF_`

The aim was some Gröbner basis analogue for FAA: although such an algorithm may not terminate in non-commutative case, it still can be useful.

But we stopped, so far, at defining some arithmetics for FAA and reduction to the normal form (`polNF` analogue).

46.1 FreeMonoid

The power products for non-commutative polynomials form a free (non-commutative) monoid.

See the comments below:

```
-----------------------------------------------------------------
module FreeMonoid
  (
   FreeMonoid(..), FreeMOrdTerm,
   freeMN, freeMRepr,  freeMOId, freeMOComp, freeMWeightLexComp,
   freeMGCD
   --
   -- , instance Cast FreeMonoid [(Z,Z)],
   --   instances  Show .. MulMonoid    for  FreeMonoid
  )
where
...


data FreeMonoid = FreeM (Maybe Z) [(Z,Z)]  deriving (Show, Eq)

-- Free monoid generated by anonymous generators.
--
-- (FreeM (Just n) _)  means the generator set   {No 1, No 2...No n}
-- (FreeM Nothing  _)  means infinite generator set:  {No 1 ...}
--
-- Example:
-- (non-commutative) monomial  x1*x3^2*x6^5  <- R<x1..x11>
-- can be represented as       FreeM (Just 11) [(1,1),(3,2),(6,5)]
```

```
freeMN :: FreeMonoid -> Maybe Z
freeMN  (FreeM mn _) =  mn

freeMRepr :: FreeMonoid -> [(Z,Z)]
freeMRepr  (FreeM _ ps) =  ps

instance Cast FreeMonoid [(Z,Z)]
  where
  cast _ (FreeM nm _) ps = FreeM nm ps            -- without check

--------------------------------------------------------------------
instance Set FreeMonoid
  where
  showsDomOf (FreeM mn _) = case mn of
                                Just n -> ("FreeMonoid_of(g_1,...,g_"++) . shows n
                                _      -> ("FreeMonoid_of[g_1,g_2,...]"++)

  baseSet (FreeM mn _) dm =  dummy ....
  compare_m _ _ =
          error "compare_m  is not defined for FreeMonoid, so far\n"
  fromExpr _ =
          error "fromExpr  is not defined for FreeMonoid, so far\n"


--------------------------------------------------------------------
instance MulSemigroup FreeMonoid
  where
  baseMulSemigroup _ =
               error "baseMulSemigroup (FreeM..):   dummy, so far\n"

  unity_m f = Just $ ct f ([] :: [(Z,Z)])

  mul (FreeM mn ps) (FreeM mn' ps') =  ...
    --
    -- multiplication of non-commutative power products

  inv_m f = if f == (unity f) then Just f else Nothing

  power_m = SetGroup.powerbin        -- binary method for powering

  root _ (FreeM _ _) = error "root n (FreeM..):   skipped\n"

  divide_m f g = fmap (ct f) $ dv (freeMRepr f) (freeMRepr g)
          --
          -- q = f/g  means that  q*g = f :  g is a suffix word of f
          -- Examples:  (x*y, y) -> Just x;  (y*x, y) -> Nothing
```

```
  divide_m2 f g = (divide_m f g, divR f g, biDiv f g)
       where
       divR f g = fmap (ct f) $ divRRepr (freeMRepr f) (freeMRepr g)
...


biDiv :: FreeMonoid -> FreeMonoid -> Maybe (FreeMonoid, FreeMonoid)
  --
  -- LOCAL.
  -- biDiv f g = Just (l,r),  if f= l*g*r  (any such pair returned),
  --             Nothing,     if there does not exist such pair.
...


---------------------------------------------------------------------
instance MulMonoid FreeMonoid

freeMGCD :: FreeMonoid -> FreeMonoid -> (FreeMonoid, FreeMonoid)

  -- freeMGCD f g = (gcdl f g, gcdr f g),
  -- where
  -- gcdl  is the greatest left  factor in  f  which is a right
  --       factor in  g,
  -- gcdr  is the greatest right factor in  f  which is a left
  --       factor in  g.


-- Grading, comparison on FreeMonoid  ------------------------------

type FreeMOrdTerm = (PPOId, Comparison FreeMonoid)

freeMOId   :: FreeMOrdTerm -> PPOId
freeMOComp :: FreeMOrdTerm -> Comparison FreeMonoid
freeMOId   = fst
freeMOComp = snd


-- usable comparisons ----------------------------------------------

freeMWeightLexComp ::
                  (Z -> Z) -> FreeMonoid -> FreeMonoid -> CompValue
                  -- weight
  -- Compare non-commutative pp by  totalWeight  first, then,
  -- if equal, compare lexicographically by  xi.
  -- totalWeight  is defined as below by the given map
```

```
  --                                                \x -> weight x

freeMWeightLexComp weight pp pp' =
  (case
       (compare (totalWeight ps) (totalWeight qs), lcomp ps qs)
   of
     (EQ, v) -> v
     (v,  _) -> v
  )
  where
  (ps, qs)    = (freeMRepr pp, freeMRepr pp')
  totalWeight = sum . map (\ (x,e) -> (weight x)*e)

  lcomp []         []           = EQ
  lcomp []         _            = LT
  lcomp _          []           = GT
  lcomp ((x,e):ps) ((x',e'):qs) = case (compare x x', compare e e')
                                  of
                                    (LT, _ ) -> GT
                                    (GT, _ ) -> LT
                                    (_,  EQ) -> lcomp ps qs
                                    (_,  v ) -> v
```

46.2 Free associative algebra: arithmetics

```
---------------------------------------------------------------------
module FAA0_
  (
   module FreeMonoid,

   FAA(..), FAAMon, FAAVarDescr,
   faaMons, faaFreeMOrd, faaVarDescr, faaN, faaVMaps, faaFreeMOId,
   faaFreeMOComp, faaLM, faaLeastMon, faaLPP, reordFAA ,
   faaMonMul, faaMonFAAMul, cToFAA, faaToHomogForms

   -- instances for FAA :
   --   Dom, Eq, Show,  Cast (FAA a) (FAAMon a),
   --                   Cast (FAA a) [FAAMon a], Cast (FAA a) a,
   --   PolLike FAA,    Set .. Ring, Fractional
  )
where
...

---------------------------------------------------------------------
type FAAMon a = (a, FreeMonoid)                        -- FAA monomial
```

```
type FAAVarDescr = (Maybe Z, (Z -> Maybe PolVar, PolVar -> Maybe Z))
  --               mn       toStr              fromStr
  --
  -- mn       is as in FreeMonoid.
  -- variable indices range in   iRange = [1 .. upp],
  -- upp = n  (case mn = Just n)  or  infinity  (case mn = Nothing).
  -- toStr    shows variable as string, it is defined on iRange  and
  --          produces  Just str  for some (showable) indices.
  -- fromStr  is the reverse to  toStr,  it produces  Just index
  --          for a variable name which corresponds to some index
  --          in iRange.


data FAA a = FAA [FAAMon a] a FreeMOrdTerm FAAVarDescr (Domains1 a)
--
-- (element of) Free associative algebra
-- - non-commutative polynomial.
-- The monomials are ordered similar as in polynomials.
-- The below  Show  and  Set  instances for FAA allow to print a
-- non-commutative polynomial displaying it is a given variable
-- system  and  reading it from expression in this variable system.


instance Dom FAA where  dom    (FAA _ _ _ _ d) = d
                        sample (FAA _ c _ _ _) = c

faaMons         :: FAA a -> [FAAMon a]
faaFreeMOrd     :: FAA a -> FreeMOrdTerm
faaVarDescr     :: FAA a -> FAAVarDescr
faaN            :: FAA a -> Maybe Z
faaVMaps        :: FAA a -> (Z -> Maybe PolVar, PolVar -> Maybe Z)
faaFreeMOId     :: FAA a -> PPOId
faaFreeMOComp   :: FAA a -> Comparison FreeMonoid

faaMons     (FAA ms _ _ _  _) = ms
faaFreeMOrd (FAA _  _ o _  _) = o
faaVarDescr (FAA _  _ _ vd _) = vd
faaN                          = fst . faaVarDescr
faaVMaps                      = snd . faaVarDescr

faaFreeMOId   = freeMOId   . faaFreeMOrd
faaFreeMOComp = freeMOComp . faaFreeMOrd

faaLM, faaLeastMon :: (Set a) => FAA a -> FAAMon a

faaLM f = case faaMons f of  m:_ -> m
```

```
                           _   -> error ...

faaLeastMon f = case faaMons f of  m:ms -> last (m:ms)
                                   _    -> error ...

faaLPP :: Set a => FAA a -> FreeMonoid
faaLPP             f     =  case faaMons f of (_,p):_ -> p
                                              _       -> error ...

instance (Eq a) => Eq (FAA a) where  f==g = (faaMons f)==(faaMons g)

reordFAA :: FreeMOrdTerm -> FAA a -> FAA a   -- bring to given ordering
                                             --
reordFAA o (FAA ms c _ vd dom) = FAA (sortBy cmp ms) c o vd dom
                                      where
                                      cmp (_,p) (_,q) = cp q p
                                      cp              = freeMOComp o


--------------------------------------------------------------------
instance (AddGroup a) => Cast (FAA a) (FAAMon a)
  where
  cast mode (FAA _ c o vd d) (a,p) = FAA mons c o vd d
             where
             mons = if  mode=='r' && isZero a  then []  else [(a,p)]

instance (AddGroup a) => Cast (FAA a) [FAAMon a]
  where
  cast mode (FAA _ c o vd d) mons =  FAA ms c o vd d
    where                                       -- order NOT checked
    ms = if  mode /= 'r'  then  mons
         else                   filter ((/= z) . fst) mons
    z  = zeroS c


instance (AddGroup a) => Cast (FAA a) [(a, [(Z,Z)])]
  where
  cast mode f preMons = cast mode f
                        [(a, FreeM (faaN f) ps) | (a,ps) <- preMons]

instance (Ring a) => Cast (FAA a) a
  where
  cast mode (FAA _ _ o vd d) a =  case (mode, isZero a) of

                ('r', True) -> FAA []                       a o vd d
                _           -> FAA [(a, FreeM (fst vd) [])] a o vd d
```

```
----------------------------------------------------------------------
instance PolLike FAA
  where
  pPPO     _   = error "pPPO (FAA _):   use  faaNCPPO  instead\n"
  lm       _   = error "lm (FAA _):   use  faaLM  instead\n"
  lpp      _   = error "lpp (FAA _):  use  faaLPP  instead\n"
  pCoef    _ _ = error "pCoef  is not defined for  FAA R\n"
  pFromVec _ _ = error "pFromVec  is not defined for  FAA R\n"
  pToVec   _ _ = error "pToVec  is not defined for  FAA R\n"
  pDeriv   _   = error ("pDeriv (FAA _):   derivative skipped, so "
                        ++"far, for non-commutative polynomial\n"
                    )
  pMapPP  _ _ = error "pMapPP f (FAA _):  not defined\n"
  pDivRem _ _ = error "pDivRem  is not defined so far for  FAA R\n"
  pValue  _ _ = error "pValue   is not defined so far for  FAA R\n"

  pIsConst f = case faaMons f of (_, p):_ -> p == unity p
                                 _        -> True

  pVars f = catMaybes $ map toStr js
                        where
                        (mn, (toStr,_)) = faaVarDescr f
                        js           = case mn of Just n -> [1 .. n]
                                                  _      -> [1 ..  ]
  pCoefs  = map fst . faaMons

  pTail f = case faaMons f of _:ms -> ct f ms
                              _    -> error ...

  pFreeCoef (FAA mons c _ _ _) =
                let  { z = zeroS c;  (a,p) = last mons }
                in
                if  null mons  then  z
                else                 if p == (unity p) then a else z

  ldeg f = case faaMons f of (_,p):_ -> sum $ map snd $ freeMRepr p
                             _       -> error ...

  deg f = case  map (sum . map snd . freeMRepr . snd) $ faaMons f
          of
            d:ds -> maximum (d:ds)
            _    -> error ...

  degInVar for0 i f =
    --
```

```
  -- Put degree of i-th variable in monomial to be the sum of
  -- degrees of its occurences.
  -- Example:  if   faaMons f = [(1,1),(2,2),(5,3),(2,6)]
  --           then  degInVar _ 2 f = 8
  (case
       (i >= 0, faaMons f)
   of
    (False, _ ) -> error $ msg "\n\nPositive i needed\n"
    (_,     []) -> for0
    (_,     ms) -> maximum $ map (degInMon i . freeMRepr . snd) ms
  )
  where
  degInMon i = sum . map snd . filter ((== i) . fst)

  msg = ("degInVar for0 i f,  \ni = "++) . shows i .
        ("\nf <- FAA R,  R = "++) . showsDomOf (sample f)


pCDiv f c =  ...

pMapCoef mode f g = cast mode g [(f a, pp) | (a,pp) <- faaMons g]

varPs a f = [ctr f [(a, FreeM mn [(i,1)])] | i <- range]
                             where
                             mn    = faaN f
                             range = case mn of Just n -> [1 .. n]
                                                _      -> [1 ..  ]

--------------------------------------------------------------------
faaMonMul :: (Ring a) =>  a -> FAAMon a -> FAAMon a -> [FAAMon a]
                        --zero
                                                   -- product of monomials
faaMonMul z (a,p) (b,q) = let c = a*b in  if  c==z  then  []
                                          else        [(c, mul p q)]



faaMonFAAMul :: (Ring a) => FAAMon a -> FAA a -> (FAA a, FAA a)
  --
  -- multiply FAA element f by FAA monomial m forming the pair
  -- (m*f, f*m)
  --
faaMonFAAMul (a,p) f = (ctr f [(a*b, mul p q) | (b,q) <- faaMons f],
                        ctr f [(b*a, mul q p) | (b,q) <- faaMons f]
                       )

                                               -- coefficient to FAA
```

```
cToFAA :: (Ring a) =>
          FreeMOrdTerm -> FAAVarDescr -> Domains1 a -> a -> FAA a
cToFAA    ord             varDescr       dom           a =
                                         FAA mons a ord varDescr dom
      where
      mons = if isZero a then [] else [(a, FreeM (fst varDescr) [])]


faaToHomogForms ::
         (AddGroup a, Eq b) => (FreeMonoid -> b) -> FAA a -> [FAA a]
                              -- weight map
faaToHomogForms w f =
  map
     (ct f) $ partitionN (\ (_,p) (_,q) -> (w p)==(w q)) $ faaMons f
       --
       -- (non-ordered) list of homogeneous forms of non-commutative
       -- polynomial over `a'  with respect to
       -- weight :: PowerProduct -> b


instance (Ring a) => Show (FAA a)
  where
  showsPrec _ (FAA mons c _ varDescr dom) =  ...
  --
  -- If  a  is and Ordered ring, then the mode `ord'  is set which
  -- writes ..- m  instead of ..+(-m) for the negative coefficient
  -- monomials.
  -- If  a  has unity  then unity coefficient images  are skipped.


---------------------------------------------------------------------
instance (CommutativeRing a) => Set (FAA a)
  where
  compare_m    = compareTrivially
  fromExpr     = fromexpr_
  showsDomOf f = ("FAA("++) . shows (faaN f) . (',':) .
                 showsDomOf (sample f) . (')':)

  baseSet _ _ = error "baseSet (FAA..):  dummy, so far \n"


---------------------------------------------------------------------
instance (CommutativeRing a) => AddSemigroup (FAA a)
  where
  add       = add_
  zero_m f  = Just $ ctr f $ zeroS $ sample f
```

```
  neg_m     = Just . neg_
  times_m f = Just . (times_ times f)

  baseAddSemigroup _ _ = error "baseAddSemigroup (FAA..):  dummy, so far\n"

instance (CommutativeRing a) => AddMonoid (FAA a)

instance (CommutativeRing a) => AddGroup (FAA a)
  where
  baseAddGroup _ _ = error "baseAddGroup (FAA..):  dummy, so far\n"


instance (CommutativeRing a) => MulSemigroup (FAA a)
  where
  unity_m f = fmap (ct f) $ unity_m $ sample f

  mul f g = case faaMons f of
                 []  -> zeroS f
                 m:_ -> (fst $ faaMonFAAMul m g) + (pTail f)*g

  inv_m f = if  isZero f || not (pIsConst f)  then  Nothing
            else                          fmap (ct f) $ inv_m $ lc f

  divide_m f g =
    let
       zeroP = zeroS f
    in
    case (f == zeroP, g == zeroP)
    of
      (True, _   ) -> Just zeroP
      (_   , True) -> Nothing
      _            -> let (q,r) = pDivRem f g
                      in  if isZero r then Just q  else Nothing

  divide_m2 _ _ = error "divide_m2  is not defined for ..=> FAA a  so far\n"
  root _ _      = error "root  is not defined for ..=> FAA a  so far\n"

  -- power  is the default

  baseMulSemigroup _ _ = error "baseMulSemigroup (FAA..):  dummy, so far\n"


instance (CommutativeRing a, MulMonoid a) => MulMonoid (FAA a)

instance (CommutativeRing a) => Num (FAA a)
  where
```

```
  negate = neg
  (+)    = add
  (-)    = sub
  (*)    = mul
  signum _      = error "signum  is not defined for ..=> FAA a  ... "
  abs    _      = error ("abs  is not defined for ..=> FAA a  ..."
  fromInteger _ = error "fromInteger  to (FAA _):  use  fromi, fromi_m\n"

instance (CommutativeRing a) => Fractional (FAA a)
  where
  (/) = divide
  fromRational _ = error ("fromRational  to (FAA _):  \n"++
                          "use  fromi, fromi_m  combined with  divide_m\n"
                         )

instance (CommutativeRing a) => Ring (FAA a)
  where
  fromi_m f    = fmap (ctr f) . fromi_m (sample f)
  baseRing _ _ = error "baseRing (FAA..):  dummy, so far\n"
```

46.3 Free associative algebra: reduction

```
--------------------------------------------------------------------
module FAANF_ (faaNF)

-- Reduction to Groebner Normal Form of non-commutative polynomial
-- over a Commutative Ring.
-- Its main difference from  polNF  is in that it bases on different
-- kind of division on power products (in free monoid).
...

faaNF :: (EuclideanRing a) => String -> [FAA a] -> FAA a ->
                              -- mode   gs         f

 ((FAA a,[FAA a]), (FAA a,[FAA a]), (FAA a,[[(FAAMon a,FAAMon a)]]))
 -- lrem lqs        rrem  rqs        biRem biQs

  -- Non-commutative analogue for  polNF.
  -- mode   is as in polNF.
  --
  -- (lrem,lqs)   is the reduction result for  LeftIdeal(gs):
  --                                                 f - q1*g1 -...,
  -- (rrem,rqs)   -- for RightIdeal(gs):             f - g1*q1 -...,
  --
  -- biRem        -- remainder for DoubleSidedIdeal(gs):
  --                                      biRem = f - m1*g1*m1' -...
```

```
--   For  gs = [g_1..g_k]
--   each qq_i <- biQs is  [(m_i_1,m_i_1')..(m_i_l(i),m_i_l(i)')],
--   and it represents a linear combination
--           comb_i = m_i_1*g_i*m_i_1' +..+ m_i_l(i)*g_i*m_i_l(i)'
--   to substract from f.
--
-- To understand the below program, read first the commutative
-- case: polNF.
```

47 Parsing. More details

See Section 3.14.
The items for parsing are exported from the module `DPrelude`,
implemented in `parse/Iparse_.hs, parse/OpTab_.hs`

`(fromExpr sample e)` interprets algebraically the expression `e` according to `sample`. This `e` is a data organized into a *tree*, and it often has to be parsed from the `String`. This parsing is performed by the `infixParse` function described below. The techique of the below infix operation parsing bases on the well known finite automaton principle.

Certain new details, like the arbitrary left-hand and right-hand arities and priorities, were taken from the design of the `FLAC` functional programming system where it was developed by A.P.Nemytykh and S.V.Chmutov.

```
  data Expression a =  L a | E (Expression a) [Expression a] [Expression a]
                       deriving (Eq, Show, Read)
```

`a` is a type of lexeme, `L` a tag of a lexeme expression,
`E` a tag of a non-lexeme expression.

`(E op ls rs)` denotes an expression which is an application of an *operation expression* `op`; the left-hand argument list for `op` is `ls`, the right-hand argument list is `rs`.

Usually, `op` is a lexeme. For example `(L "++")`.

Example:
if `a = String`, then `"(1+ 22)*- 3a "` may parse to the expression

```
                 (E  (L "*")  [ (E (L "+") [L "1"] [L "22"]) ]
                              [ (E (L "-") []      [L "3a"]) ]
                 )
```

```
--------------------------------------------------------------------
lexLots :: String -> [Expression String]

  -- Break string to the list of lexemes - according to standard
  -- {\tt Haskell} set of delimiters.
  -- Example:  (lexLots  " {2+ 33}*3 " )  ->
  --           [ L "{", L "2", L "+", L "33", L "}", L "*", L "3" ]

lexLots xs = case lex xs of [("", _  )] -> []
                            [(lx, xs')] -> (L lx): (lexLots xs')
                            _           -> []



-- Infix Operation Table  ----------------------------------------------

type OpDescr = (Natural, Natural, Integer, Integer)
               -- (left arity, right arity, left precedence, right precedence)
```

```
type OpGroupDescr a = (a, (Maybe OpDescr, Maybe OpDescr, Maybe OpDescr))
                          -- 0r_class    l0_class       lr_class
```

`a` is a type of *operation name*. For example, `"+."` may be an operation name — if `a = String`.

Each operation name corresponds to some *group* of the three possible operations of class
`(0,r)` — prefix operation of arity `r`, say `-x`,
`(l,0)` — postfix operation of arity `l`, say `n!`,
`(l,r)` — general infix operation of arities `l, r`, say, `x+y`.

`ij_class = Nothing` means that the given operation name cannot denote an operation of class `(i,j)`.

Example 1:
`("+", (Just (0,1,210,190), Just (1,1,100,100), Nothing) )`
describes the group `"+"` which may denote either a prefix operation of arity 1, or a mixed operation of arities 1, 1.

Example 2:
`(":", (Nothing, Just (1,1,60,50), Nothing))`
contained in the standard `DoCon` table describes the binary operation symbol which usually denote the `List` constructor.
Similar is the pair constructor `","`. Thus, `"(1:2-3:nil, a+1)"` would parse to

```
(,  [:  [1]  [: [- [2] [3]] [nil]]  ]
    [, [+ [a] [1]] ]
)                        -- we omit here the double quotes and ().
```

Study the module `OpTab_.hs` to get a more definite idea of how to set the standard infix operations and how to define the new ones.

```
type OpTable a = [OpGroupDescr a]       -- this contains all the operations
                                        -- that are processed by 'infix'
getOp :: Eq a => OpTable a -> a -> Maybe (OpGroupDescr a)
                                            -- get operation group from table
getOp xs x = case dropWhile ((/= x) . fst) xs of []    -> Nothing
                                                 (y:_) -> Just y
type ParenTable a = [(a,a)]
```

A *parenthesis* may be any lexeme from `a`.

`ParenTable` lists all the allowed parentheses pairs.

`parenTable` is the standard `DoCon` parenthesis list: see `OpTab_.hs`

So, `lookup x parenTab` returs either `Nothing` or
`Just oppositeTo_x_parenthesisLexeme`

```
infixParse :: (Eq a, Read a, Show a) =>
              ParenTable a-> OpTable a-> [Expression a]-> ([Expression a], String)
              -- pT          oT          xs
```

Parsed is a list `xs` of expressions some of which may be lexemes, say `(L "x1")`.

Parenthesis or an infix operation sign must be a lexeme, say `(L "{"), (L "+")`. The lexeme expression list can be parsed by `lexLots <string>`.

The infix parser "sets the parentheses" in `xs` according to the tables `parenTable`, `opTable`.

It returns `(expression_list, message)`. Here
`message == ""` when the parse succeeds. In this case `expression_list == [e]`.
Otherwise, `message` is non-empty and tells what is wrong in the input syntax.

Examples

```
Let  pT = parenTable;  opT = opTable.

(infixParse pT opT (lexLots  " x-  ab*20" ))  -->

               ( [E  (L "-")  [L "x"]  [E (L "*") [L "ab"] [L "20"]] ],  "" )

(infixParse pT opT (lexLots  "0--2" ))   -->
                   ( [E  (L "-")   [L "0"]  [E (L "-") [] [L "2"]]  ],  "" )

(infixParse pT opT (lexLots  "* 2"           )) -->  ([], "wrong arity for *")
(infixParse pT opT (lexLots  "- 2)"          )) -->  ([], "no left ( for )"  )
(infixParse pT opT (lexLots  " (x+1)^2:nil " )) -->
               ( [E  (L ":")
                        [ E  (L "^")   [E (L "+") [L "x"] [L "1"]]  [L "2"] ]
                     [L "nil"]
                 ],
                 ""
               )
```

See `parse/princip.txt` for the illustration of the method.

48 Language extension proposal

We think, the following language features will make `Haskell` fit better the needs of programming mathematics:

- `(dtp)` dependent types,
- `(der)` more 'deriving' abilities,
- `(overl)` extended polymorphism for values and instance overlap,
- `(dc)` automatic conversion between types (domains),
- `(recat)` reorganising the `Haskell` library algebraic categories,
- `(es)` equational simplifier annotations.

48.1 Dependent types

The example of computing in $Z/(m)$, [Me2] sets a question of the `Haskell` type system fitness for programming mathematics. A domain depending on a value parameter cannot be expressed as an `Haskell` type. So, `DoCon` applies the SA approach. But it looks that the dependent types language extension should be the most adequate approach. See, for example, the `Aldor` [Al] and `Cayenne` [Au] languages.

48.2 More 'deriving' abilities

Imagine the program with the instances

```
instance <C1 a> => D1 (T a) where ...
instance <C2 a> => D2 (T a) where ...
instance <C3 a> => D3 (T a) where ...
instance <C4 a> => D4 (T a) where ...
```

for the type constructor `T`. And for another type

```
data D a = D a                                          (1)
```

we define explicitly the functions, the reciprocally inverse bijections:

```
b ::  D a -> T a,   b' ::  T a -> D a,   b .  b' = b' .  b = id
```

Then, it should be clear what we might mean by declaring

```
data D a = D a  derivingBy(T, b, b', D1, D2, D3)          (1')
instance ... => D4 (D a) where ...
```

This means that $\mathtt{D}_1$, $\mathtt{D}_2$, $\mathtt{D}_3$ are 'copied' from `(T a)` to `(D a)` according to the maps `b, b'`, while `D4 (D a)` is defined specially.

That is each operation `op` of the class $\mathtt{D}_i$ is defined for `(D a)` by mapping necessary arguments to `(T a)`, applying `op` and mapping the result (if necessary) back to `(D a)`.

Examples.

1. Binary operation `op :: T a -> T a -> T a` induces `op` for `D a`:
`op (D x) (D y) = b' (op (b (D x)) (b (D y)))`

2. The `Ring` operation of integer image `fromi :: T a -> Integer -> T a` induces `fromi` for `D a` : `fromi (D x) n = b' (fromi (b (D x)) n)`

3. The set cardinality `card :: T a -> Integer` might induce `card (D x) = card (b (D x))`

Maybe, the investigation [HJ] will have its practical result in the language development and help to solve the above problem. The implementation by [GH] announces some tool for this.

48.3 Extended polymorphism for values and instance overlap

Consider the example of the matrix determinant programming. We would like to program it like this:

```
det :: CommutativeRing a => [[a]] -> a
det                         mM    =    ... expand_by_row method

det :: Field a => [[a]] -> a
det               mM    =    ... Gauss method
```

The former method is more generic, the latter method is more efficient — while both should be called `det` in the interface. Because `det` is a classic name in mathematics. This is a common situation.

`Haskell-2010` rejects this as the double definition for `det`. Therefore `DoCon` provides the functions `det_euc, det` of the corresponding types. And this is a bad style: the same maps (values) have better to be called the same. Neither would help making `det` an operation of a class, say `WithDet`. Because `Haskell-2-pre` does not resolve such kind of the instance overlaps — with the different type contexts.

Instead, this is how it should be: since `CommutativeRing a` is a superclass of `Field a`, the compiler has to substitute the latter definition of `det` everywhere in the program where it derives `Field a`, and the former definition — in other places, where only `CommutativeRing a` was checked.

For the more complex overlaps, the programmer might set `{instanceOrder n}` in some of the function type declarations. `instanceOrder` is the suggested reserved word. This means the *ad hoc* resolution. When the data belongs to several types for different definitions of a function `f`, the compiler chooses the definition of `f` with the smaller `n :: Integer`. Skipping `instanceOrder` has to set the default value for `n`.

Instance overlaps

Similarly, the extended instance overlaps can be resolved — which is as important as for ordinary values. For example, the definitions

```
instance Ring a     => LeftModule a       a where  cMul = (*)
instance AddGroup a => LeftModule Integer a where  cMul = flip times
```

look quite natural mathematically. But these instances overlap at `a = Integer`, and `Haskell-2-pre` cannot resolve this overlap. This has to be improved in the same manner as above with `det`.

48.4 Automatic conversion between types (domains)

Consider the task of computation of

```
a = 1 + 1:%2
f = (2*x + 3*y)*(4*x^2 + 5*y^2)
g = (1:%2) * x,
```

where `x` is the integer polynomial in variable `"x"` (belongs to $Z[x]$),

`y` polynomial in `"y"` with coefficients from $Z[x]$.

We also presume that the program has to prepare once some 'environment' values before starting to compute many expressions like `a, f, g`.

According to mathematical common sense, the domains of these values have to be respectively

```
Q  = Fraction Z,
PP = (Z[x])[y]        (UPol (UPol Z)    ),
QX = Q[x]             (UPol (Fraction Z))
```

With `Haskell-2-pre`, the problem is here in the need of explicit and implicit value conversion between the domains. In DoCon, the `cast` operation does half of this job. First, it has to generalize to the `Convertible a b` class, to be able to force several constructor levels. But this possibility is reduced by the restriction on overlapping instances — see Section 48.3.

Second, the implicit conversion is needed too. For example, the above expression `(1:%2) * x` might be converted automatically by the compiler to `(ctr x (1:/2)) * x`, and only then — compiled as usual. Though, this has to occur only in a program scope marked with some appropriate reserved word. Here `ctr` is the casting by sample (Section 3.4 CS).

48.5 Reorganising the Haskell algebraic categories

Haskell-2010 declares the algebraic categories like

Num, Fractional, Integral with the operations +, *, / ...

And their definitions do not agree with what a mathematician would expect. Though DoCon coexists peacefully with these classes, with the Haskell-2010 Prelude, it is still better to reformulate the standard Haskell algebraic categories as it is suggested in [Me2].

48.6 Equational simplifier annotations

The compiler developers are considering the optimizations like `map f . map g --> map (f . g)` to be performed at the compilation stage. Similarly, many optimizations are possible, like say

```
forall x :: Ring a => a   x - x --> 0,    2*x+2*y --> 2*(x+y),
```

and such. The programmer may set the rewrite rules — or better to call them 'equations', like

```
 {equations
  { map f .map g === map (f.g) }

  {x,y,z :: RingWithUnity a => a, (defined x,y,z) ==>

   "df"      (defined  zero, unity, neg x, x*y),
   "assoc+"  x+(y+z)   === (x+y)+z
   ...
   "distr"   x*(y+z)   === x*y + x*z
  }
}
```

The compiler applies them as rewrite rules to algebraic expressions [BL]. It depends on the programmer-defined *strategy*, which equations to apply first, and whether it applies the equation from left to right, or from right to left.

Example: changing the term ordering for +, -, * would cause the distributivity equation applied differently, say `(x+y)*(x-y) --> x*x - y*y` or the reverse.

The main part of the *strategy* is the ordering (<') defined on the expressions. We hope, the methods from [BL, Md] would help to develop the needed equational simplifier for Haskell.

The proposed simplification has to take place at the compile-time, apply only to the parts inside the { eqScope ...} marks and may cost differently (for the compilation process), depending on the strategy defined by the programmer.

The programs avoiding the word 'equations' would remain with the old Haskell treating of expressions.

Question 1:
the programmer hardly ever writes directly `x-x`. How can it appear?

After defining, say `f x y = ...`, the program often applies `f` differently:

`f b b`, `map (f x) [y1,y2,y3]`, `f . f . g`, and so on.
Evidently, the expressions like `x - x` often appear this way, and they should be optimized automatically.

Question 2:
may the above equations simplify, for example, `1/0 - 1/0 --> 0` ?
If yes, then this is incorrect. Because this changes the program into one not precisely equivalent. This is the so-called 'undefined' or 'bottom' problem in programming.

The matter is that the condition `'defined x ==>'` is set in the 'equations'. This condition would not let, for example, `x - x :: Integer` to simplify to `0`.
But, maybe, to `if x==0 then 0 else 0`.

Proving 'defined' for the variable occurrences, the compiler could apply correctly the simplifier to various parts of a program.

49 On gx-rings. Some theory

See first Section 18.

In this section the word 'ring' means an

algorithmic Noetherian commutative ring with unity.

A gx-ring is a ring with solvable linear equations (notion from 20-th century computer algebra), and with certain additional properties for canonical remainders.

This describes the situation in which `DoCon` is able to define the canonical form and arithmetic algorithms for the residue ring.

The Euclidean ring structure and Gröbner basis, with its related functions [Bu], are imporatant special cases of the gx-ring, gx-basis notion.

The property names ruling gx-operations are

```
IsGxRing, IsGxBasis,
ModuloBasisDetaching, ModuloBasisCanonic,   WithSyzygyGens
```

And we have to explain here what does it mean `IsGxRing`.

Below `[A]` denotes a set of finite sequences over the set `A` — this corresponds to the type `[a]` in an `Haskell` program.

Consider a ring `A` supplied with the operations

(gxSig):

```
gxBasis    : [A] -> [A] x Matrix(A),
modBasisG : [A] x A -> A x [A],
syzGens    : [A] -> [[A]]
```

And introduce the denotation for projections of the first two of above operations:

$gxBasis_g$ `= fst . gxBasis : [A] -> [A]` — gx-basis,
$gxBasis_m$ `= snd . gxBasis : [A] -> Matrix[A]` — transformation matrix,
$modBasisG_r$ `= fst . modBasisG : [A] -> A -> A`
— normal form, remainder by basis,
$modBasisG_q$ `= snd . modBasisG : [A] -> A -> [A]`
— 'quotient' for normal form.

Definition. A map $f : A \longrightarrow A$ is canonical modulo ideal I in A when $f(0) = 0$ and for each x, y from A there hold

$f(x) - x \in I$, $(x - y \in I \Longleftrightarrow f(x) = f(y))$

That is f chooses a unique representative in each class $x + I$ and chooses 0 from $0 + I$.

Below, we denote with `xs, ys, gs`, $\ldots$, xs, ys, gs, $\ldots$ the lists — elements of a domain `[A]`.

Definition. For a ring A with the operations **(gxSig)**, gs is called a gx-basis (of $I = Ideal(gs)$)
if it does not contain zeroes and $(modBasisG_r\ gs)$ is a canonical map modulo $Ideal(gs)$.

Definition. An ideal basis xs for I is minimal (by inclusion), if
$Ideal(xs \setminus \{x\}) \neq Ideal(xs)$ for any x from xs.

Let us also put that empty basis presents zero ideal.

Definition.
A ring `A` with algorithmically given operations **(gxSig)** is called a gx-ring if the following conditions are satisfied:

(GxS) for each `xs` `(syzGens xs)` is some generator list
for the `A`-module of linear relations beween `xs`

(GxM) for each xs $(gxBasis_g\ xs)$ is a gx-basis

(GxGT) for each xs $M = gxBasis_m\ xs$ is the transformation matrix over `A`
for $(gxBasis_g\ xs)$: $M * xs^{\rightarrow} == (gxBasis_g\ xs)^{\rightarrow}$,
where the upper index arrow means the vector -column made from the given list.

(GxMQ) for each gx-basis gs $q = modBasisG_q\ gs\ a = [q_1, \ldots, q_n]$
is the quotient vector of a by gs, that is

$$a = g_1q_1 + \ldots + g_nq_n + (modBasisG_r\ gs\ a)$$

In the `DoCon` program, the counterparts for the operatins **(gxSig)** are

```
gxBasis             :: [a]    -> ([a], [[a]])
(moduloBasis "cg") :: [a]    -> a   -> (a, [a])
syzygyGens          :: String -> [a] -> [[a]]
```

Remarks.

1. `DoCon` defines $(modBasis_r\ xs)$ so that $Ideal(xs)$ maps to zero, and other elements to possibly 'small' elements.

2. $gxBasis_g\ xs$ is not necessarily a minimal basis.
For example, `[2, 2]` is a gx-basis in `Integer`.

3. Usually, the ideal has many minimal gx-bases. And it often occurs that $Ideal(xs) = Ideal(ys)$ and $(gxBasis_g\ xs) \neq (gxBasis_g\ ys)$. Different gx-bases in ideal I may present different canonical projections $(modBasisG_r\ gs)$ modulo I.

4. For any list `xs`, not necessarily a gx-basis, the composition of
`gs = gxBasis xs` and `(modBasisG gs)` gives a canonical map modulo `Ideal(xs)`. We

call the corresponding maps

`modBasis`, $modBasis_r$, $modBasis_q$;

their g-counterparts are `modBasisG`, $modBasisG_r$, $modBasisG_q$.

5. `DoCon` uses the property `IsGxBasis` related to the above definition of gx-basis for the needs of optimization. If the description of an ideal `I` contains `(IsGxBasis, Yes)` in the attributes of generators, then the reduction modulo I may skip applying `gxBasis` without breaking the result correctness.

The also concerns, for example, finding syzygis for the polynomials over a field.

6. `DoCon` does not provide a function to detect "is a gx-basis".
Sometimes, `DoCon` defines its value automatically, and often it has to be set by the programmer. And when it is not set to `Yes`, this may cause extra `gxBasis` computation.

7. The conditions (GxS) — (GxMQ) are much weaker than the Gröbner basis structure conditions, dnd they have few in common with grading. Why do we need them?
They serve for defining in an universal way the arithmetics of residue domains for the gx-rings like (1) c-Euclidean ring, (2) polynomail ring over a c-Euclidean ring,
(3) direct sum of gx-rings, (4) maybe, some other constructs (see below).

Examples of gx-rings

c-Euclidean ring.

Definition. A c-Euclidean ring is an Euclidean ring with the property of the canonical division remainder (see Section 19, `DivRemCan` property).

For a c-Euclidean ring, `DoCon` puts

`syzGens =` algorithm `solveLinear` for general linear system over an Euclidean ring.

`gxBasis xs = ([g], [qs])`,
where `(g,qs) = gcdE xs` is the extended GCD method.

`modBasisG [b] a = (r, [q])`, where `(q,r) = divRem 'c' a b`
— see above the c-Euclidean ring definition.

And it occurs here that for any non-zero `b` `[b]` is a gx-basis.

Proof: an evident consequence of a c-Euclidean ring definition.

Polynomial ring $P = k[x_1, \ldots, x_n]$ **over a field** k**.**

`DoCon` defines a gx-ring structure for P by putting

`gxBasis =` extended Gröbner basis algorithm ([Bu] extGB),

`modBasisG =` Gröbner normal form algorithm, with complete reduction of the tail and with accumulation of quotient vector ([Bu] Section 2), `syzGens =` syzygy generator list algorithm for polynomials ([Bu] Section 7), via finding of Gröbner basis and reducing of S-polynomials.

[Bu] contains certain theory from which it follows easily that the above definition makes P a gx-ring.

Furher, for many composed domains `DoCon` defines automatically the gx-operations by construction. This is done as follows.

49.1 Polynomials over a c-Euclidean ring

For an Euclidean ring A, $P = A[x_1, \ldots, x_n]$ is supplied with gx-operations via the weak reduced Gröbner basis method [Mo].

49.2 Direct sum

The Gröbner basis notion for the ring $C = A \oplus B$ is senseless. Neither C is Euclidean. Nevertheless, C is naturally a gx-ring if A and B are. Any ideal I in C is represented uniquely as the direct sum of ideals: $I = I_1 \oplus I_2$.

Therefore, the gx-operations are defined for C in an evident way, which description we skip.

49.3 Residue ring

`DoCon` defines a residue ring $B = A/I$ only for a gx-ring A, and ideal I given by its gx-basis `gs`. Applying `gxBasis` in A always reduces the task to this case.

We have first to define the equality, canonical representation and arithmetic in B.

`DoCon` denotes by `Rsi x iI dA` the residue element modulo I. But for this section, let us denote it

`Rs x gs`,

`gs` a gx-basis of I.

It is essential that `DoCon` requires `x` to be canonically reduced modulo `gs` *ab initio*. It is the same as if we replaced `Rs x gs` with `Rs` ($modBasisG_r$ `gs x`) `gs` each time before computing anything with residues.

Hence the equality in B is 'derived': `(Rs x gs)==(Rs y _) = x==y`

Arithmetic:

```
(Rs x gs)+(Rs y _) = Rs (modBasisG_r gs (x+y)) gs,
(Rs x gs)*(Rs y _) = Rs (modBasisG_r gs (x*y)) gs,
```

and so on.

Consider now the division.

What is division

A quotient of division is *any* solution of a linear equation $a \cdot x = b$. It may not have a solution, may have a unique solution (say in a field case), or many solutions.

And DoCon provides *any* solution, if such exists. Together with other gx-operations, this provides a convenient complete solution for the division.

Let us consider the inversion in `B = A/I` (division is similar):

```
inv_m (Rs a gs) = let (r, q0:_) = modBasis (a:gs) 1
                  in
                  if r==0 then  Just $ Rs (modBasisG_r gs q0) gs
                  else          Nothing
```

Proof of its correctness:

`(Rs a gs)` is invertible $\Leftrightarrow$ $1 \in$ `I = Ideal(gxBasis`$_g$ `(a:gs))`. Due to gx-ring laws, this is equivalent to `(modBasisG`$_r$ `(gxBasis`$_g$ `(a:gs)) 1) == 0`, and this latter is equivalent to the condition `r == 0` in the considered program. Hence, according to the `modBasis` definition, `1 = q0 · a` $+ \sum_{i=1}^{n} q_i \cdot g_i$, where `[q0,`$q_1, \ldots, q_n$`]` is the quotient vector returned by `modBasis (a:gs) 1`.

Therefore, `q0` is the inverse of `a` in `A/I`.

The gx-operations in a residue ring base on that the inclusion and equality of ideals J in $B = A/I$ is isomorphic to ones of the set of ideals I' in A that contain I.

On computation cost for residue ring

Division and gx-operations in A/I may be expensive, for they may cause extra `gxBasis` computation. Such is the nature of a residue ring. But for the 'simple' ideals, the operations in A/I are cheap. Typical example: for the rational numbers field $\mathbb{Q}$, DoCon represents its extension $B = \mathbb{Q}(\sqrt[3]{2},\ \sqrt{-3})$ as the residue ring $\mathbb{Q}[x,y]/Ideal(x^3 - 2,\ y^2 + 3)$, and this leads to quite simple computations in B. For example the inversion in B costs about as much as inversion of a matrix of size 6 over $\mathbb{Q}$.

49.4 Ring description transformations

The class of gx-rings extends wider that it appears from previous sections. For examle, let $A = (Q \oplus Q)[x_1, \ldots, x_n]$ be a polynomial ring over the pairs of rationals. We cannot directly use the Gröbner bases for A because $Q \oplus Q$ has zero divisors, and DoCon cannot the find a Gröbner basis for such coefficient ring.

But consider the computable isomorphism

$$(B \oplus C)[x_1, \ldots, x_n] \longleftrightarrow B[x_1, \ldots, x_n] \oplus C[x_1, \ldots, x_n]$$

In our example, B, C, are fields, so, $B[x_1, \ldots, x_n]$, $C[x_1, \ldots, x_n]$ are gx-rings according to the lemma for the polynomial constructor. Hence, $B[x_1, \ldots, x_n] \oplus C[x_1, \ldots, x_n]$ is supplied with gx-operations according the direct sum rule. Similar approach is valid for the constructions

(CC):

$$B[x_1, \ldots, x_n][y_1, \ldots, y_m], \quad (B/I)[x_1, \ldots, x_n],$$
$$(C \oplus B)/I, \qquad (B/I)/J,$$
$$\ldots$$

— let us call them g-constructions.

<u>Constructor isomorphisms and `Haskell` instances</u>

Concerning such transformations, DoCon implements only one kind of them: the isomorphism $a[x_1, \ldots, x_n][y] \longleftrightarrow a[y, x_1, \ldots, x_n]$ is applied for defining the operations `gxBasis`, `moduloBasis` for the domain $a[x_1, \ldots, x_n][y]$. The corresponding instance is

```
instance (LinSolvRing (Pol a), CommutativeRing a) => LinSolvRing (UPol (Pol a))
  where ...
  -- see  Pol3_.hs.
```

This is nice that `Haskell` allows such recursion on the instances. Note this '`LinSolvRing (Pol a)`' in the context. It allows `a` to specialize further to `a = b[`$z_1, \ldots, z_m$`]`. For an appropriate domain `b`, say Euclidean, this whole instance would work too.

For example, '`LinSolvRing a`' in the context is too weak, because `gxBasis` for `(Pol a)` is applied. And '`EuclideanRing a`' is too strong: in this case the recursion would not work, for example, for `a = Z[u,v]`.

This `DoCon` experiment with the domain $a[x_1, \ldots, x_n][y]$ shows that the `Haskell` instances fit to implement all the g-constructions in a recursive manner.

49.5 gx operations in programs

49.5.1 moduloBasis

The 'theoretical' operation `modBasis` introduced above corresponds to the `DoCon` operation from the category `LinSolvRing`:

```
moduloBasis :: String -> [a] -> a -> (a, [a])
               --mode    basis   x    rem quot


mode = cMode++gMode,  cMode = "" | "c",   gMode = "" | "g"
```

`cMode = "c"` means the canonic reduction.

`""` means only the reduction in which the zero modulo I is detached.

Example.

For the polynomials from $Q[x, y]$ and any monomial ordering with $y > x$, the map `fst . moduloBasis "g"` $[x]$ remains the polynomials $y + x$, $y + 2x$ unchanged, while `fst . moduloBasis "cg"` $[x]$ maps them to y.

`gMode = "g"` means a gx-basis `basis`.

Any other value means that nothing is known about `basis`. In this case. it is applied the composition of `gxBasis` and `moduloBasis (_:'g')`.

49.5.2 syzygyGens

The theoretical `syzGens` introduced above corresponds to the `DoCon` operation from the category `LinSolvRing`:

```
syzygyGens :: String -> [a] -> [[a]]
              --mode    fs
```

`mode = ""` means the generic case.

`"g"` :

(1) for a polynomial ring $A = R[x_1, \ldots, x_n]$ over an Euclidean ring R, `mode = "g"` means that `fs` is a (weak) Gröbner basis (the evaluation will be simpler),

(2) for $A = B \oplus C$, `mode = "g"` means `mode = "g"` for `syzygyGens` for both `map fst fs` and `map snd fs`.

50 DoCon module export lists

When the module reexports something from other module, it is marked in the commentary. For example,

```
module Pol ( f, g,
             u, v  -- from Pol_
           )
```

means that Pol defines f, g and exports them, and reexports u, v defined in Pol_.

Further, in Haskell, exporting any data constructor causes automatically the export of its related instances. DoCon sets this instance export as the commentary.

The following DoCon 'open' module export list informs the user where the needed items have to be imported from.

```
module DExport
            -- Joint export of DoCon and some libraries of GHC.
            -- In *extra* case, 'import DExport' imports everything.

  (module AlgSymmF,     module Categs,      module Z,
   module DPair,        module DPrelude,    module Fraction,
   module GBasis,       module LinAlg,      module Partition,
   module Permut,       module Pol,         module Residue,
   module RingModule,   module SetGroup,    module VecMatr,

   module Prelude, module List, module Data.Map, module Ratio,
   module Random
  )
------------------------------------------------------------------------
module DPrelude                                         -- DoCon Prelude

  (sublists, listsOverList, smParse,

   -- from Prelude_:
   Cast(..), ct, ctr,
   PropValue(..), InfUnn(..), MMaybe, CompValue, Comparison,
   Z, toZ, fromZ,
   tuple31, tuple32, tuple33,  tuple41, tuple42, tuple43, tuple44,
   tuple51, tuple52, tuple53, tuple54, tuple55,
   zipRem, allMaybes, mapmap, mapfmap, fmapmap, fmapfmap,
   boolToPropV, propVToBool, not3, and3, or3,  compBy,  delBy,
   takeAsMuch, dropAsMuch, separate, pairNeighbours,
   removeFromAssocList, addToAssocList_C, addListToAssocList_C,
   propVOverList, mbPropV, lookupProp, updateProps, addUnknowns,
   foldlStrict, foldl1Strict,
   antiComp,  minBy, maxBy, minimum, maximum, minAhead, maxAhead,
```

```
   -- from Iparse_:
   Expression(..), OpDescr, OpTable, ParenTable, lexLots, infixParse,

   parenTable, opTable,   -- from OpTab_

   module Char_,
   module List_,  -- cubeList_lex

   -- from Common_:
   partitionN, eqListsAsSets, del_n_th, halve, mulSign, invSign,
   evenL, factorial, binomCoefs, isOrderedBy, mergeBy, mergeE, sort,
   sortBy, sortE,  sum1, product1, alteredSum, lexListComp,
   minPartial, maxPartial,

   -- from Set_:
   less_m, lessEq_m, greater_m, greaterEq_m, incomparable,
   showsWithDom
  )
---------------------------------------------------------------------
module Categs                                -- domain descriptions

  (Dom(..),   Domain1(..), Domain2(..), Domains1, Domains2,
   CategoryName(..),  Factorization,
   Ideal(..), Properties_Ideal, Properties_IdealGen,
   Property_Ideal(..), Property_IdealGen(..),
   Construction_Ideal(..), Operations_Ideal, OpName_Ideal(..),
   Operation_Ideal(..),
   Submodule(..), Properties_Submodule, Properties_SubmoduleGens,
   Operations_Submodule, Property_Submodule(..),
   Property_SubmoduleGens(..), OpName_Submodule(..),
   Operation_Submodule(..), Construction_Submodule(..),
   LinSolvModuleTerm(..), Properties_LinSolvModule,
   Property_LinSolvModule(..),

   -- from Categs_:
   Vector(..), vecRepr, PowerProduct, PPComp,
   AddOrMul(..),
   OSet(..), Properties_OSet, Property_OSet(..),
   Construction_OSet(..), Operations_OSet, OpName_OSet(..),
   Operation_OSet(..),
   Subsemigroup(..), Properties_Subsemigroup,
   Property_Subsemigroup(..), Construction_Subsemigroup(..),
   Operations_Subsemigroup, OpName_Subsemigroup(..),
   Operation_Subsemigroup(..),
   Subgroup(..), Properties_Subgroup, Property_Subgroup(..),
```

```
   Construction_Subgroup(..), Operations_Subgroup,
   OpName_Subgroup(..), Operation_Subgroup(..),

   Subring(..), Properties_Subring, Property_Subring(..),
   Construction_Subring(..), Operations_Subring,
   OpName_Subring(..), Operation_Subring(..),

   GCDRingTerm(..), Properties_GCDRing, Property_GCDRing(..),

   FactrRingTerm(..), Properties_FactrRing,
   Property_FactrRing(..),
   LinSolvRingTerm(..), Properties_LinSolvRing,
   Property_LinSolvRing(..),
   EucRingTerm(..), Properties_EucRing, Property_EucRing(..)
  )
-------------------------------------------------------------------------------
module SetGroup             -- Set, Semigroup ... Group   categories
  (
   AddGroup(..), OrderedAddGroup(..),
   isOrderedGroup, absValue, trivialSubgroup, isoGroup,
   isoConstruction_Subgroup,
   MulSemigroup(..), MulMonoid(..), OrderedMulSemigroup(..),
   OrderedMulMonoid(..), MulGroup(..), OrderedMulGroup(..),
   upAddGroup, upMulSemigroup, upMulGroup,  unity, inv, divide,
   invertible, divides, power, powerbin,
   unfactor, isPrimeFactrz, isPrimaryFactrz, isSquareFreeFactrz,
   factrzDif, eqFactrz, gatherFactrz, rootOfNatural, squareRootOfNatural,
   minRootOfNatural,
   -- instances for Integer:
   -- MulSemigroup, MulMonoid, AddGroup, OrderedAddGroup,

   -- from Set_
   Set(..), OrderedSet(..), compareTrivially, isFiniteSet,
   isBaseSet, intervalFromSet, card, ofFiniteSet, isoOSet,
   props_set_full_trivOrd, listToSubset,

   -- from Semigr_
   AddSemigroup(..),  OrderedAddSemigroup(..), AddMonoid(..),
   OrderedAddMonoid(..),
   upAddSemigroup, isGroup, isCommutativeSmg, isoSemigroup,
   trivialSubsemigroup, isoConstruction_Subsemigroup,
   zeroS, isZero, neg, sub, times
   -- ,instances for Integer:  Set .. OrderedAddMonoid
  )
```

```
module RingModule

  -- Ring..GCDRing..Field, LeftModule, LinSolvLModule  categories

  (LeftModule(..), LinSolvLModule(..),
   isGxModule, isoModule, isoLinSolvModule,isoDomain22,isoDomains22,
   numOfNPrimesOverFin,
     -- instance Ring a          => LeftModule a a,
     -- instance Ring a          => LeftModule a (Vector a),
     -- instance EuclideanRing a => LinSolvLModule a (Vector a),

   -- from Ring0_:
     Ring(..), CommutativeRing(..), OrderedRing(..), GCDRing(..),
   FactorizationRing(..), LinSolvRing(..), EuclideanRing(..),
   Field(..), RealField(..), OrderedField(..),
     isFactorizOfPrime, isFactorizOfPrimary,
     property_Subring_list, fromi, char, props_Subring_zero,
   zeroSubring, dimOverPrimeField, isField, isPrimeIfField,
     isOrderedRing, rankFromIdeal, isMaxIdeal, isPrimeIdeal,
     isPrimaryIdeal,
   genFactorizationsFromIdeal, zeroIdeal, unityIdeal,
     isGCDRing, isRingWithFactor, isGxRing, isEucRing, isCEucRing,
     upRing, upGCDRing, upFactorizationRing, upLinSolvRing,
   upEucRing, upGCDLinSolvRing, upFactrLinSolvRing,
   upEucFactrRing, upField,

   -- from Ring_:
   eucGCDE, powersOfOne, logInt, diffRatios,
   isoRing, isoGCDRingTerm, isoFactrRingTerm,
   isoLinSolvRingTerm, isoEucRingTerm,  PIRChinIdeal(..), eucIdeal,
   isoDomain1, isoDomains1,

   -- from Ring1_:
   quotEuc, remEuc, multiplicity, isPowerOfNonInv,
   gxBasisInEuc, moduloBasisInEuc, syzygyGensInEuc,
   moduloBasis_test, gxBasis_test, syzygyGens_test, gcd_test
  )
-------------------------------------------------------------------
module Z                                 -- items for  Z = Integer

  (dZ,

   -- from SetGroup
   rootOfNatural, minRootOfNatural
   -- , instances for Integer:
   -- Set, OrderedSet, AddSemigroup, OrderedAddSemigroup, AddMonoid,
```

```
   -- OrderedAddMonoid, MulSemigroup, MulMonoid, AddGroup,
   -- OrderedAddGroup

   -- from Ring_
   -- instances for Integer:
   -- Fractional, Ring, CommutativeRing, OrderedRing,

   -- from Ring1_
   -- instances for Integer:  LinSolvRing, EuclideanRing
  )
--------------------------------------------------------------------
module DPair
             -- category instances for Direct Product of two domains
  (
   -- instances for (,):
   -- Set,AddSemigroup,AddMonoid,AddGroup,MulSemigroup,MulMonoid,
   -- MulGroup,Ring,CommutativeRing,LinSolvRing

   module DPair_
   -- directProduct_set, directProduct_semigroup,
   -- directProduct_group, directProduct_ring
  )
--------------------------------------------------------------------
module Fraction

  (Fraction(..),                          -- from Ring0_
   num, denom, zeroFr, unityFr, canFr      -- from Fract_

   -- ,instances for Fraction a:
   -- Functor, Dom, ... OrderedSet ... RealField, OrderedField
   -- --- some of them imported from Fract_, Ring0_
   --
   -- Specializations for some instances for  Fraction Z.
  )
------------------------------------------------------------------------
module VecMatr
            -- Vector and Matrix operations and their category instances

  (Vector(..), vecRepr,  {- Eq Vector, -}                -- from Categs
   {- class -} MatrixLike(..),  Matrix(..),              -- from Ring0_

   -- from Vec0_
   allMaybesVec, vecSize, vecHead, vecTail, constVec, scalProduct,
   -- instance Functor

   -- from Vec1_
```

```
   vecBaseSet, vecBaseAddSemigroup, vecBaseAddGroup,
   vecBaseMulSemigroup, vecBaseMulGroup, vecBaseRing,

   module Vec_,
   -- instances for Vector:
   -- Show,Set,OrderedSet,AddSemigroup,AddMonoid,OrderedAddSemigroup,
   -- OrderedAddMonoid,AddGroup,OrderedAddGroup,MulSemigroup,
   -- OrderedMulSemigroup,MulMonoid,MulGroup,Ring,CommutativeRing,
   -- OrderedRing,Num,Fractional,

   -- from Matr0_:
   {- class -} MatrixSizes(..),  SquareMatrix(..),
   toSqMt, fromSqMt, mtHead, matrHead, isZeroMt, mtTail, constMt, rowMatrMul,
   scalarMt, mainMtDiag, isDiagMt, isStaircaseMt, isLowTriangMt,
   vandermondeMt, resultantMt,

   -- from Matr1_:
   matrBaseSet, matrBaseAddSemigroup, matrBaseAddGroup,
   sqMatrBaseSet, sqMatrBaseAddSemigroup, sqMatrBaseAddGroup,
   sqMatrBaseMulSemigroup, sqMatrBaseRing
  )
--------------------------------------------------------------------
module Permut                                  -- permutation group

  (Permutation(..), EPermut, toEPermut, fromEPermut, permutRepr,
   isPermut, permutSign, isEvenPermut, applyPermut, invEPermut,
   addEPermut, ePermutCycles, permutECycles, permutCycles,
   transpsOfNeighb, allPermuts, nextPermut, test_allPermuts,
   gensToSemigroupList
   -- , instances for  Permutation:
   -- Eq, Show, Ord, Num, Set, MulSemigroup, MulMonoid, MulGroup
  )
--------------------------------------------------------------------
module LinAlg                                     -- linear algebra

  (diagMatrKernel, solveLinearTriangular, solveLinear_euc,
   test_solveLinear_euc,

   -- from Stairc_:
   reduceVec_euc, toStairMatr_euc, rank_euc, inverseMatr_euc,
   linBasInList_euc, test_toStairMatr_euc, test_inverseMatr_euc,

   det, det_euc, maxMinor, delColumn,      -- from Det_
   toDiagMatr_euc, test_toDiagMatr_euc     -- from Todiag_
  )
--------------------------------------------------------------------
```

```
module Pol                                       -- polynomial items
  (
   UMon, PowerProduct, PPComp,   -- from  Categs

   -- from PP_
   isMonicPP, ppLcm, ppComplement, vecMax, ppMutPrime, ppDivides,
   lexComp, lexFromEnd, degLex, degRevLex, ppComp_blockwise,

   -- from UPol_
   PolLike(..), PolVar, Mon, UPol(..), PPOrdTerm, PPOId,
   Multiindex, ppoId, ppoComp, ppoWeights, lexPPO, varP, deg0, lc0,
   pHeadVar, lc, cPMul, pCont, numOfPVars, upolMons, lmU, umonMul,
   mUPolMul, umonLcm, leastUPolMon, cToUPol, upolPseudoRem,
   monicUPols_overFin,
   -- instance (PolLike p...) => LeftModule a (p a),
   -- instances for UPol:        Dom, Eq, Cast, PolLike,

   -- from UPol0_
   charMt, charPol, resultant_1_euc, resultant_1, resultant_1_euc,
   discriminant_1, discriminant_1_euc, matrixDiscriminant,
   upolSubst, upolInterpol,
   -- instances for UPol:
   --  Show, DShow, Set, AddSemigroup, AddMonoid, AddGroup, MulSemigroup,
   --  MulMonoid, Num, Fractional, Ring, CommutativeRing

   -- from Pol_:
   Pol(..), PolPol, polMons, cToPol, reordPol, leastMon, monMul,
   mPolMul, monLcm, headVarPol, fromHeadVarPol, polToHomogForms,
   addVarsPol, toUPol,fromUPol, coefsToPol, polDegs, polPermuteVars,
   -- instances for Pol:  Show, Eq, Dom, Cast, PolLike,

   PVecP, sPol, mPVecPMul, sPVecP,                   -- from  Pol__
   RPolVar, showsRPolVar, SPProduct, SPMon, SPPol', showsSPPol',

   polSubst,                                         -- from  Pol1_
   -- instances for Pol:  Show, Set .. CommutativeRing,

   module Pgcd_,   -- GCDRing instances for UPol, Pol

   module Pol2_,
   -- toOverHeadVar, fromOverHeadVar,
   -- LinSolvRing, EuclideanRing  instances for  UPol,
   -- LinSolvRing                                Pol,
   -- instance..=> LinSolvLModule (Pol  a) (EPol a),
   --              LinSolvLModule (Pol  a) (Vector (Pol  a)),
   --              LinSolvLModule (UPol a) (Vector (UPol a)),
```

```
-- from Pol3_:
VecPol(..), vpRepr, vpEPPOTerm, vpECp, vpToV,
-- instances for VecPol up to LinSolvLModule (Pol a) (VecPol a),

-- from RPol_:
RPol(..), RPol'(..), RPolVarComp, RPolVarsTerm,
rpolRepr, rpolVComp, rpolVTerm, rvarsTermCp, rvarsTermPref,
rvarsTermRanges, rpolVPrefix, rpolVRanges, rvarsVolum,
showsRVarsTerm, rvarsOfRanges, rp'HeadVar, rpolHeadVar,
rp'Vars, rpolVars, cToRPol, varToRPol', varToRPol, rHeadVarPol,
rFromHeadVarPol, toRPol, toRPol', fromRPol, substValsInRPol,
-- instances for RPol', RPol:  Show, Eq, Functor, Dom, PolLike,

-- from RPol0_
-- instances for RPol:
-- Set, AddSemigroup, AddMonoid, AddGroup, MulSemigroup,
-- MulMonoid, Num, Fractional, Ring, CommutativeRing, GCDRing,

vecLDeg, henselLift, testFactorUPol_finField,     -- from Pfact0_

extendFieldToDeg, det_upol_finField, resultant_1_upol_finField,
                                                  -- from Pfact1_

module Pfact__,
-- RseUPol, RseUPolRse, toFromCanFinField, factorUPol_finField,
-- instance of  FactorizationRing  for  k[x],  k  a finite field

module Pfact3_,
              -- FactorizationRing instances for  k[x][y], k[x,y]

-- from EPol_:
EPP, EPPComp, EPPOTerm, EMon, EPol(..),
eppoECp, eppoMode, eppoWeights, eppoCp,
epolMons, epolPol, epolEPPOTerm, epolECp, epolPPCp, eLm, eLpp,
epolLCoord, leastEMon, reordEPol, cToEMon, cToEPol, zeroEPol,
polToEPol, epolToPol, ecpTOP_weights, ecpPOT_weights, ecpTOP0,
EPVecP, emonMul, mEPolMul, polEPolMul, epolToVecPol,
vecPolToEPol, sEPol, mEPVecPMul, sEPVecP,
-- instances for EPol:
--             Dom, Cast, PolLike, Show, Eq, Set .. AddGroup, Num

-- from RdLatP_:
reduceLattice_UPolField, reduceLattice_UPolField_special,

-- from FAA0_               items for non-commutative polynomials
```

```
   FreeMonoid(..), FreeMOrdTerm,
   freeMN, freeMRepr,  freeMOId, freeMOComp, freeMWeightLexComp,
   -- instances for FreeMonoid:
   --                 Cast FreeMonoid [(Z,Z)],  Show .. MulMonoid,
   --
   FAA(..), FAAMon, FAAVarDescr,
   faaMons, faaFreeMOrd, faaVarDescr, faaN, faaVMaps, faaFreeMOId,
   faaFreeMOComp, faaLM, faaLeastMon, faaLPP, reordFAA,
   faaMonMul, faaMonFAAMul, cToFAA, faaToHomogForms,
   -- instances for FAA :
   --   Dom, Eq, Show,  Cast (FAA a) (FAAMon a),
   --                   Cast (FAA a) [FAAMon a], Cast (FAA a) a,
   --   PolLike FAA,    Set .. Ring, Fractional,

   faaNF, faaNF_test    -- from FAANF_
  )
---------------------------------------------------------------------
module Residue                       -- quotient group, residue ring

  (ResidueE(..),    -- from Categs

   module QuotGr_,
   -- ResidueG(..), isCorrectRsg,
   -- instances  Show, Eq, Cast, Residue, Dom, Set .. AddGroup

   gensToIdeal,   -- from IdealSyz_

   module ResEuc0_,
   -- class Residue(..),  ResidueE(..),
   -- resSubgroup, resSSDom, resIdeal, resIIDom, isCorrectRse,
   -- ifCorrectRse,
   -- instances for Residue class:          Show, Eq,
   --           for constructor  ResidueE:  Dom, Cast, Residue

   module ResEuc_,
   -- instances for  ResidueE:
   -- Set,AddSemigroup,AddMonoid,AddGroup,MulSemigroup,MulMonoid,
   -- Ring,CommutativeRing,LinSolvRing,GCDRing, FactorizationRing,
   -- EuclideanRing, Field,
   -- specialization for  ResidueE Z  for the instances
   --   Set, AddSemigroup, AddGroup, MulSemigroup, Ring,

   module RsePol_,
   -- specialization of  ResidueE  to  Field k => ResidueE (UPol k):
   -- instances up to  Ring
```

```
   module ResRing_,
   -- ResidueI(..), isCorrectRsi,
   -- instances for  .. => ResidueI a:
   --                    Dom, Residue, Cast, Set .. Num, Fractional,

   module ResRing__,   -- continuation:  instances  Ring .. Field,

   module ResPol_
   -- instances Set,AddSemigroup,Ring  for  ..=> ResidueI (Pol a)
  )
---------------------------------------------------------------------
module GBasis                     -- items related to Gr\"obner basis

  (polNF, polNF_v, polNF_e, polNF_ev, test_polNF, underPPs,
                                                 -- from PolNF_
   gBasis, gBasis_e, isGBasis, isGBasis_e,          -- from GBas_
   polRelGens, polRelGens_e, algRelsPols            -- from Polrel_
  )
---------------------------------------------------------------------
module Partition
       -- items related to Partitions (Young diagrams), bands, hooks

  (isPrtt, subtrSHook_test, kostkaNumber, kostkaColumn,
   kostkaTMinor, hookLengths, numOfStandardTableaux,
   permGroupCharValue, permGroupCharColumn, permGroupCharTMinor,

   -- from Partit_:
   Partition, EPartition, PrttComp,
   toEPrtt, fromEPrtt, prttToPP, ppToPrtt, prttWeight, prttLength,
   conjPrtt, prttUnion, pLexComp, pLexComp', prttLessEq_natural,
   minPrttOfWeight, prevPrttOfN_lex, prttsOfW, randomEPrtts,
   showsPrtt,

   -- from HookBand_:
   SHook, HBand, sHookHeight, subtrSHook, firstSWHook, prevSWHook,
   subtrHBand, maxHWBand, prevHWBand
  )
---------------------------------------------------------------------
module AlgSymmF                -- symmetric function transformations

  (toSymPol, symmSumPol, symmetrizePol, fromSymPol,
   to_e, to_e_pol,  to_h, to_h_pol,  to_m, to_m_pol,
   to_p, to_p_pol,  to_s, to_s_pol,

   -- from Sympol_
   SymPol(..), SymMon, symPolMons, symPolPrttComp, symLm, symLdPrtt,
```

```
 cToSymPol, reordSymPol, monToSymMon, symPolHomogForms,
 -- , instances for SymPol:
 -- Show, Eq, Dom, Cast, PolLike, Set .. AddGroup, Num,

 -- from SymmFn_
 PrttParamMatrix(..), SymFTransTab(..), ptpMatrRows, transpPtP,
 h'to_p_coef, elemSymPols, hPowerSums, toDensePP_in_symPol,
 fromDensePP_in_pol, intListToSymPol
)
```

51 Performance comparison

We differ between the two ways to compare the performance: “fixed algorithm”, and “non-fixed algorithm”. The former is to program the same (as possible) algorithm for the given task in two programming systems and see the timing for several example data.

The latter means that the algorithm programmed in our system (`DoCon- Haskell`) is known, and the algorithm implemented in another system is not known. Such comparison also may be useful.

We describe here only the latter, non-fixed algorithm comparison to the popular CA programs

`Axiom, MuPAD`

— see [Je, Mu]. The below benchmarks were prepared and run in

March 20 - April 10, 2002

on the same machine `laudomia4` of the CA centre

MEDICIS: Unité Mixte de Service, CNRS/Polytechnique
`<http//www.medicis.polytechnique.fr>`

The machine parameters are Intel (i-688, Pentium II, 400 MHz),
and it was run under the Linux operation system.

The program versions are:

- `Axiom-2.2` (= `Axiom`),
- `MuPAD-1.4.2` (= `MuPAD`),
- `DoCon`
  running under ghc-5.02.2 [GH], the library compiled with the optimization key `-O`, the head test program either interpreted or compiled with -Onot

<u>Memory space:</u>

In the below tests, `DoCon` was given
50 Mbyte for factoring in $GF(p)[x, y]$, 24 Mbyte for other tasks
(running option `+RTS -MXXm -HXXm -RTS` with `XX = 50, 24`).

The author had not studied yet how to specify the memory bound in `Axiom` and `MuPAD` systems. But the process mesurements by means of the Unix `top` command show that on these examples

`Axiom` uses less that 45 Mb, and sometimes uses at least 30 Mb, `MuPAD` uses less than 12 Mb.

Though, in most of the below examples, all the three programs can do the task in smaller memory space, maybe, with additional expencies for the "garbage collection".

Generally, `MuPAD` looks to win in these examples about 3 times in space in comparison to `DoCon` and about 2 times in comparison to `Axiom`.

The comparison was done only for the `speed`. And we keep in mind that the space expenses always cause the corresponding time expenses (at least, proportional).

<u>On compared tools</u>

All the three programs are "high level": have categories and domain constructors, and so on.

`DoCon` is written in `Haskell`, is functional and 'lazy'.
`Axiom` is more close to `DoCon` than `MuPAD`.
`Axiom` and `MuPAD` are the strict evaluators.
`DoCon` is more functional than `Axiom`.
`MuPAD` is not functional at all.

So, the below test gives certain impression of the performance ability of a 'lazy' and functional programming tool. It is measured at the complex enough practical applications.

Though, the set of examples is rather small.

Detail of `Axiom` timing: we use the commands

```
... )set message time on; ... Running:  time axiom;  )r a.input  ...
```

But tabled here is the timing shown by interpreter-dialogue (it is smaller). The lines from 'Running:' are for those who do not remember how to run Axiom; and they provide some additional check: include the time for the library loading, etc.

<u>Task choice</u>

We choose the real-world computer algebra tasks, and the popular ones. They are programmed in `Haskell` for the needs of mathematical practice, *not* chosen to promote any particular language or implementation.

On these tasks a pure functional, 'lazy' tool `Haskell`, — in its ghc-5.02.2 implementation, — occurs efficient enough in comparison to `Axiom`, `MuPAD`.

The tasks are

- powering `(^)` in `Z[x,y,z], Rational[x,y]`
  — polynomial arithmetic: `+, *`
- gcd in $Z[x,y,z]$
- Gröbner basis in $Rational[x_1,\ldots,x_n]$

* factoring in `(Z/(p))[x]` — involves polynomial arithmetics, gcd, large linear system solution over `Z/(p)` ...
* factoring in `(Z/(p))[x,y]` — involves (in DoCon) factoring in $GF(p^m)[x]$, large sparse linear system solution over `Z/(p)` and other things.

We only run the executable programs of `Axiom` and `MuPAD` and do not know how the algorithms applied and how the above tasks are implemented in these systems.

On programs

We specify the complete programs for the tests, together with the instructions of how to prepare them for running and how to run them.

`Haskell -DoCon` programs are more lengthy. This is mainly due to that they include additional correctness tests (made so that they do not bring essential cost overhead).
and also because they are designed to use in interactive interpreter mode as well as by running the compiled executable from the command line.

We do not write such expanded programs in `Axiom` and `MuPAD` because we do not know these systems in enough detail.

51.1 Powering polynomial

Z[z,y,x]

Find `f^n`, `f = 2*y*z + z + y*x + 3*y + 1` from `Z[z,y,x]`.
DoCon can represent this `f` as in

- `Z[x][y][z] = UUU` — `UPol (UPol (UPol Z))`,
- and as in `Z[z,y,x]` — `Pol Z`.

In MuPAD, the representation `poly( 2*y*z + ... [z,y,x] )` is, probably, close internally to `UUU` (?).

Timing [sec]

```
ghc-5.02.2 + DoCon make executable a.out:  read n, show $ t n, time a.out
The programs are specified below.

                DoCon          Axiom          MuPAD
             UUU   Z[z,y,x]    (UUU)?         (UUU)?

f^n, n = 18|   1.1   1.5       1.3            0.9
         21|   2.8   3.2       2.4            1.7
         24|   7.5   8.7       6.8            2.8
         27|  17    22         8.3            4.5
         30|  34    47        12              6.9
         33|  51    71        35             10
         36|  63    86        43             14

Control sums:  snd $ t n =
Z[z,y,x]
n = 18| (3344457783112,(Vec [5,95,-10] ))
    21| (-9279064030550640,(Vec [5,76,511] ))
UUU
    18| (9,1256648)
    21| (11,914941444)
    36| (18,3046219662585200)
```

All the three programs (DoCon with `Z[z,y,x]` model) do not change essentially the performance with the change of variable order: `< 10%`
(for DoCon — `UUU`, we had not tried other orders).

Q[x,y], Q = Fraction Z

```
f^n,  f = x^2 + (2/3)*x*y + (3/4)*y
ghc-5.02.2 + DoCon make  a.out:  read n, show $ t n, time a.out ...
Timing [sec]:
             DoCon           Axiom          MuPAD
        Q[y][x] Q[x,y]      Q[x,y]?
n = 30|   0.8    0.9         1.0              0.6
    40|   2.3    2.4         3.5              1.6
    50|   6.9    7.2         9.1              3.5
    60|  16     17          17                6.6
    70|  24     26          38               11
    80|  42     45          71               18
Control sums:
for Q[x,y]
n = 30| (-1243073261628119203,      (Vec [16,112]))
    40| (1431303549982259961049279, (Vec [40,20] ))
    80| (513301774737...2015210929, (Vec [80,40] ))
for Q[y][x]
n = 30| (15,2723214048358064267)
    40| (20,1826685434290034100820683)
    80| (40,796862960953900814670346839749160221437753216182 3)
```

The cost here does not change essentially with the `x <-> y` swap — in both programs.

Conclusion: It looks like

- **(1)** For the polynomials within a very large size (say $(2*y*z+z+y*x+3*y+1)^{20}$) `DoCon` arithmetic `(+, *)` is almost as fast as of `Axiom`, and `MuPAD`.
- **(2)** `MuPAD` arithmetic seems to have somewhat better asymptotic.

(1) reflects a good quality of the GHC `Haskell` compiler and possible 'lazy' functional system efficiency.
(2) may be caused by a simplest naive algorithm of DoCon for the polynomial product (no special clever methods applied).

And everything in `DoCon` (except arithmetic of long integers) is written in `Haskell`, the exponents consist of the arbitrary-size integers, the polynomial data contains any pp-ordering term and any variable list, the program works over any commutative ring ...

Details

Integer model:
both MuPAD and DoCon use the arbitrary size representation of Integer.

On laziness
We must separate the cost of f^n computation itself from the cost of the result output.

In MuPAD the output is suppressed by the ':' postfix: `p := f^n:`
In Axiom it is done by appending ';'.
In the 'lazy' system, we cannot do this. So, we provide the 'equivalent' program for the comparison, the one composed with the 'forcing' function force. force should have a small output and bring small extra cost respectively to f^n and should force the evaluation of all the parts of f^n. We choose

```
force r = show (sumOfExponents r, sumOfCoefficients r)
```

In R[z][y][x], R[x][y] model, we mean by 'coefficients' the leaf coefficients from R. And for R = Q, sum the numerators and denominators separately.

Programs in detail

Powering in Q[x,y]

```
In MuPAD:
---------
  n  := ...:
  f  := poly( x^2 + (2/3)*x*y + (3/4)*y, [x,y] ):
  t0 := time();  p := f^n:  time() - t0;
    Runing:
          mupad;  read("m");  quit

In Axiom
---------
  P := MPOLY([x,y], Fraction Integer)
  )set message time on
  f :P := x^2 + (2/3)*x*y + (3/4)*y;
  n := 10;
  f^n;
     Running:  time axiom;
               )r a.input
               )q
               y
```

```
In DoCon, Q[x,y] model:
-----------------------
  import qualified Data.Map as Map (empty)
  import DExport
  t n =                                      -- benchmark: snd (t n)
    let q1 = 1:/1  :: Fraction Z
        dQ = upField q1 Map.empty
        p1 = cToPol (lexPPO 2) ["x","y"] dQ q1
        f  = smParse p1 " x^2 + (2:/3)*x*y + (3:/4)*y "
        fp = f^n
    in
    (fp, force fp)
         where                  -- extra thing needed to compare to MuPAD
         force f = (fc cs, alteredSum es)
                                      where  (cs,es) = unzip $ polMons f
         fc cs = sum $ map alteredSum [map num cs, map denom cs]

  Running:  ghci -package docon +RTS -M..m -RTS  Foo
            ...
            Foo> :set +s
            Foo> snd (t n)


In DoCon, Q[x][y] model:
------------------------
  module Foo where
  import qualified Data.Map as Map (empty)
  import DExport
    t n =                                     -- benchmark:  snd $ t n
    let q1 = 1:/1  :: Fraction Z
        dQ = upField q1 Map.empty
        x1 = cToUPol "x" dQ q1
        dX = upRing x1 Map.empty
        y1 = cToUPol "y" dX x1
        f  = smParse y1 " x^2 + (2:/3)*x*y + (3:/4)*y "
        fp = f^n
    in
    (fp, force fp)
      where                                -- extra thing to meet MuPAD
      force f = (alteredSum $ exps f, m2 f)
      exps    = map snd . upolMons
      m2      = alteredSum . map m1 . pCoefs
      m1 f    = (alteredSum ns)+(alteredSum ds)
                where
                (ns,ds) = (map num $ pCoefs f, map denom $ pCoefs f)
```

Powering in Z[z,y,x]

```
In Axiom:
---------
  P := MPOLY([z,y,x],Integer)
  )set message time on
  f :P := 2*y*z + z + y*x + 3*y + 1;
  n := ...;
  f^n;
  Running:  axiom;  )read a.input   )q y
```

```
In MuPAD:
------
  n  := ...:
  f  := poly( 2*y*z + z + y*x + 3*y + 1, [z,y,x]):
  t0 := time();
  p := f^n:  time() - t0;  quit
  Running:                       mupad;  read("m")
```

```
In DoCon, Z[z,y,x] model:  (a bit simpler than for Q[x,y])
-------------------------
import DExport
main = interact (\s -> shows (snd $ t $ read s) "\n")
t n =                                    -- benchmark: snd (t n)
  let p1 = cToPol (lexPPO 3) ["z","y","x"] dZ 1
      f  = smParse p1 " 2*y*z + z + y*x + 3*y + 1 "
      fp = f^n
  in
  (fp, force fp)
    where                          -- extra thing needed to compare to  MuPAD
    force f = (alteredSum cs, alteredSum es)
                              where  (cs,es) = unzip $ polMons f
Example of running:
  ghci -package docon +RTS -M..m -RTS  Main
  ...
  Main> :set +s
  Main> main
  2                -- and press  Enter, Ctrl-d  to complete input
```

```
DoCon, Z[x][y][z] model:
------------------------
module Foo where
import qualified Data.Map as Map (empty)
import DExport
t n = let x1 = cToUPol "x" dZ 1 :: UPol Z       -- benchmark: snd (t n)
          dX = upRing x1 Map.empty
          y1 = cToUPol "y" dX x1
          dY = upRing y1 Map.empty
          z1 = cToUPol "z" dY y1
          f  = smParse z1 " 2*y*z + z + y*x + 3*y + 1 "
          fp = f^n
      in
      (fp, force fp)
        where                                 -- extra thing to meet MuPAD
        force f = (alteredSum $ exps f, m3 f)
        exps    = map snd . upolMons
        m3      = alteredSum . map m2 . pCoefs
        m2      = alteredSum . map m1 . pCoefs
        m1      = alteredSum . pCoefs
```

51.2 GCD in Z[z,y,x]

In Axiom:

```
P := MPOLY([z,y,x],Integer)
)set message time on
d :P   := (2*z + 3*y + 4*x)*(z*y*x^2 + y^3 + 1)
n      := ...;
m      := ...;
dp     := d^n;
f1 : P := z + y + x;
f2 : P := z - y + x + 2;
g      := gcd (dp*f1^m, dp*f2^m);
                                  Running:  time  axiom
                                            )r a.input
                                            )q
```

In MuPAD:

```
t0   := time();                             -- file  ./m
n    := ...:
m    := ...:
vars := [z,y,x]:
d    := poly( (2*z + 3*y + 4*x)*(z*y*x^2 + y^3 + 1), vars ):
dp   := d^n:
f1   := poly( z + y + x,     vars ):
f2   := poly( z - y + x + 2, vars ):
g    := gcd (dp*f1^m, dp*f2^m);
time() - t0;
                      Running:   time mupad < m
```

In DoCon:

```
import DExport
main = interact process  where  process s = case read s :: (Z,Z) of
                                                 (n,m) -> shows (f n m) "\n"
f n m = let p1      = cToPol (lexPPO 3) ["z","y","x"] dZ 1  :: Pol Z
            d       = smParse p1 "(2*z + 3*y + 4*x)*(z*y*x^2 + y^3 + 1)"
            dp      = d^n
            [f1,f2] = map (smParse p1) ["z+y+x", "z-y+x+2"]
        in  gcD [dp*f1^m, dp*f2^m]

Example of running:  ghci -package docon +RTS -M..m -RTS  Main
                     ...
                     Main> :set +s
                     Main> main
                     (2,2)        -- press  Enter, Ctrl-d  to complete input
```

Timing [sec]:

```
ghc-5.02.2 + DoCon make \ a.out: \ read n, show $ t n,  time a.out

          DoCon       Axiom             MuPAD
                                    Z[z,y,x]  Z[y,x,z]
(n,m)
(2, 8)|   1.0          5.0            3.0        23
(2,12)|   3.8         15              6.7        -
(2,16)|   9.9         30             15          -
(2,20)|  21           63             31         186
(2,24)|  42           93             63         372
(2,28)|  78          165            129          -
```

```
---------------------------------------------------------------------------
Further comparing versions of DoCon and GHC
-------------------------------------------

June 2005.  600 Mhz machine.  -M24m for memory.

main = putStr $ shows (f n m) "\n"

Making:  ghc --make -O ... Main
Runing:  ./a.out +RTS -M24m -RTS

               DoCon-2.06,    DoCon-2.09
               ghc-5.02.3     ghc-6.4.1-pre-June-14-2005

Code size [byte]:                          ghc-July-12
  a.out  |     8.833.439      7.113.969          7.143.078
  Main.o |        11.104         13.560             13.560

Time [sec]
  (n,m)
  -------------------------------------
  (2,24)|      31              33              32
  (2,28)|      58              62              60

Minimal -M size [Mb]
  (2,24)|       4               4
  (2,28)|       5               5               5

The time does not change with the memory increase above 24 Mb.
---------------------------------------------------------------------------
```

It is measured for all the computation, including the evaluation of `d^n*f1^m, d^n*f2^m`.

This is because in the 'lazy' system it takes extra considerations to separate the cost of this preliminary `d^n*f1^m, d^n*f2^m`.

`DoCon` applies here the plain `Z[z,y,x]` model, though its GCD method still gets to the intermediate domain of `Z[z][y]` ... The initial permutations on `[z,y,x]` could make difference for the computation cost. But for this example, it is less than `15%`.

`MuPAD` is not so lucky: some of these permutations increase the cost 7 times.

<u>Conclusion</u>

In this example `DoCon` looks like a winner. And `MuPAD` slows down considerably with the unlucky variable permutation.

<u>Comments</u>

Either `MuPAD` has to gain at other data for `gcd`, or something is wrong with its GCD algorithm for $R[x_1, \ldots, x_n]$. For, the very polynomial arithmetics in `DoCon` looks as fast as of `MuPAD` (Section 51.1) — for the polynomials of the size like in this example. And `DoCon` applies the old simplest GCD method, known from the popular book [Kn] by D.Knuth (Volum 2, Section 4.6.1).

51.3 Gröbner basis in $Q[x_1, \ldots, x_n]$

We denote `Q = Fraction Z`.

The programs are asked to find the *reduced Gröbner basis* for a list `fs` of polynomials under the same power product comparison. The following examples were tried.

51.3.1 'Consistency'

Problem. Derive the consistency condition for the given `n` generic monic polynomial equations `fs`, each in variable `x` over the domain of rationals `Q = Fraction Z`:
the condition(s) on the parameter coefficients $a_{i,j}$, ... of the polynomials from `fs` for `fs` to have a common root in the complex numbers (resultant(s)).

Solution: consider `fs` as polynomials in `x`, $\mathtt{a}_{i,j}$ and set any pp-ordering in which `x` is greater than any power product free of `x`, find the Gröbner basis *gs* for `fs`. The set
`gs0 = [g <- gs | deg_x g = 0]`.
presents the neccessary conditions for the consistency. Theay are also sufficient for the consistency of each pair $(\mathtt{f}_i,\ \mathtt{f}_j)$.

I do not know so far whether they are sufficient in general.

But our business here is only to set several parameters $a_{i,j}$ in `fs` and the pp-ordering, and see the cost at which the constant-in-x polynomials appear.

Case n = 3:

```
fs = [x^2 + x*a1 + a0,  x^2 + x*b1 + b0,  x^2 + x*c1 + c0]
```

for the lexicographic pp-ordering and the variables listed in decreasing order are `[x,a1,b1,c1,a0,b0,c0]`. The found Gröbner basis contains 9 polynomials:

```
b1^2*c0 - b1*c1*b0 - b1*c1*c0 + c1^2*b0 + b0^2 - 2*b0*c0 + c0^2,
a1*b0 - a1*c0 - b1*a0 + b1*c0 + c1*a0 - c1*b0,
a1*b1*c0 - a1*c1*c0 - b1*c1*a0 + c1^2*a0 + a0*b0 - a0*c0 - b0*c0 + c0^2,
a1^2*c0 - a1*c1*a0 - a1*c1*c0 + c1^2*a0 + a0^2 - 2*a0*c0 + c0^2,
x*b0 - x*c0 - b1*c0 + c1*b0,
x*a0 - x*c0 - a1*c0 + c1*a0,
x*b1 - x*c1 + b0 - c0,
x*a1 - x*c1 + a0 - c0,
x^2 + x*c1 + c0
```

4 of them do not depend on `x`.

The programs are as follows:

In DoCon:

```
---------
conds = (pps, gs)
        where
        q1   = 1:/1  :: Fraction Z
        dQ   = upField q1 Map.empty
        vars = ["x","a1","b1","c1","a0","b0","c0"]
        ppo  = lexPPO 7
        p1   = cToPol ppo vars dQ q1
        fs   = map (smParse p1) ["x^2 + x*a1 + a0",
                                 "x^2 + x*b1 + b0",
                                 "x^2 + x*c1 + c0"]
        gs  = fst $ gxBasis fs
        pps = map (vecRepr . lpp) gs       -- for control

  Running:  ghci -package docon Main +RTS -M..m -RTS
            Main> :set +s
            Main> let (ps, gs) = conds
            ...
            Main> gs
            ...
            Main> ps
```

In Axiom:

```
---------
vars := [x,a1,b1,c1,a0,b0,c0];
P    := MPOLY(vars, Fraction Integer)
)set message time on
fs : List P := [x^2 + x*a1 + a0, x^2 + x*b1 + b0, x^2 + x*c1 + c0];
gs := groebner fs;
ms := [leadingMonomial g  for g in gs];
)spool res
output (# gs); output ms; output gs;
)spool off
  -- It writes the result to file  ./res
  -- (# gs), ms  are for control
  -- But we do not know so far how to specify the term ordering in
  -- Axiom, do not know in what ordering this Groebner basis is found.
```

```
In MuPAD:
---------
vars := [x,a1,b1,c1,a0,b0,c0]:
cp   := LexOrder:
grb  := func( groebner::gbasis(hs,cp), hs ):
lm   := func( lmonomial(h,cp), h ):
fs   := [poly(x^2 + x*a1 + a0, vars),   -- ** enter this as 1 line
         poly(x^2 + x*b1 + b0, vars),   -- ** when running
         poly(x^2 + x*c1 + c0, vars)    -- ** MuPAD
        ]:                              -- **
t0 := time():  gs := grb(fs):  time() - t0;
ms := map(gs,lm):
write(Text,"res" ,ms):       -- see the files  ./res,
write(Text,"res1",gs):       --                ./res1
quit
```

But this task is too fast to compute. Let us consider

Case n = 4, degree = 3: The programs are as follows:

```
In DoCon:
---------
conds = (pps, gs)
        where
        q1   = 1:/1  :: Fraction Z
        dQ   = upField q1 Map.empty
        vars = ["x","c2","d2","a1","b1","c1","d1","a0","b0","c0","d0"]
        ppo  = (("", 11), cp, [])
        cp (Vec (j: js)) (Vec (j': j's)) =
                                   case compare j j'
                                   of
                                   EQ -> degLex (Vec js) (Vec j's)
                                   v  -> v
        p1   = cToPol ppo vars dQ q1
        fs   = map (smParse p1) ["x^3          + x*a1 + a0",
                                 "x^3          + x*b1 + b0",
                                 "x^3 + x^2*c2 + x*c1 + c0",
                                 "x^3 + x^2*d2 + x*d1 + d0"]
        gs  = fst $ gxBasis fs
        pps = map (vecRepr . lpp) gs       -- for control
```

We skip the parameters `a2, b2` because the task is too expensive. To compute it under `lexPPO 11` also looks too expensive (in `DoCon`). So, `DoCon` sets the `degLex` ordering for the tail power product.

Then, the Gröbner basis contains 65 polynomials; 29 of them do not depend on `x`:

```
d2*a0^2 - ... - b1^2*a0 + b1^2*d0 + b1*d1*a0 - b1*d1*b0,
c2*a0*b0 - ... 0 + b1*c1*b0 - b1*c1*d0 - b1*d1*b0 + b1*d1*c0,
c2*a0^2 - ...  + b1*c1*b0 - 2*b1*c1*d0 - 2*b1*d1*b0 + 2*b1*d1*c0,
c2*a1*b0 - ... + d2*a1*c0 + d2*b1*a0 - d2*b1*c0 - d2*c1*a0 + d2*c1*b0,
a1^3*b0 - ...  - b1^3*a0 - a0^3 + 3*a0^2*b0 - 3*a0*b0^2 + b0^3,
...
x*b1^3*d0^2 - 2*x*b1^2*d1*b0*d0 - ... - 2*d1^2*b0^2*d0,
...
x^2*c0 - x^2*d0 - x*c2*d0 + x*d2*c0 - c1*d0 + d1*c0,
x^2*b0 - x^2*d0 + x*d2*b0 - b1*d0 + d1*b0,
x^2*a0 - x^2*d0 + x*d2*b0 - a1*b0 + b1*a0 - b1*d0 + d1*b0,
x^2*c1 - x^2*d1 - x*c2*d1 + x*d2*c1 + x*c0 - x*d0 - c2*d0 + d2*c0,
x^2*b1 - x^2*d1 + x*d2*b1 + x*b0 - x*d0 + d2*b0,
x^2*d2 - x*b1 + x*d1 - b0 + d0,
x^2*c2 - x*b1 + x*c1 - b0 + c0,

x^3 + x*b1 + b0

In MuPAD:         (it succeeds with  LexOrder  too)
---------
vars := [x,c2,d2,a1,b1,c1,d1,a0,b0,c0,d0]:
cp   := LexOrder:
grb  := func( groebner::gbasis(hs,cp), hs ):
lm   := func( lmonomial(h,cp), h ):
fs   := [poly(x^3 + x*a1 + a0,          vars),   -- ** enter as One line
         poly(x^3 + x*b1 + b0,          vars),
         poly(x^3 + x^2*c2 + x*c1 + c0, vars),
         poly(x^3 + x^2*d2 + x*d1 + d0, vars)
        ]:
t0 := time():  gs := grb(fs):  time() - t0;
ms := map(gs,lm):
write(Text,"res" ,ms):
write(Text,"res1",gs):
quit

In Axiom:  as earlier, only change  vars, fs  respectively.
---------
```

51.3.2 AL — Arnborg-Lazard system

is

```
[x^2*y*z + x*y^2*z + x*y*z^2 + x*y*z + x*y + x*z + y*z,
 x^2*y^2*z + x^2*y*z + x*y^2*z^2 + x*y*z + x + y*z + z,
 x^2*y^2*z^2 + x^2*y^2*z + x*y^2*z + x*y*z + x*z + z + 1
]
```

in `Q[x,y,z]`, pp-ordering = `degLex` (DegreeOrder in `MuPAD`).

This example is said to come from certain paper by Faugere, Gianni, Lazard, Mora of 1989.

The found Gröbner basis contains 15 of polynomials, their leading power products are

```
[0,0,4], [0,1,3], [0,2,2], [0,3,1], [0,4,0],
[1,0,3], [1,1,2], [1,2,1], [1,3,0], [2,0,2],
[2,1,1], [2,2,0], [3,0,1], [3,1,0], [4,0,0]
```

Programs
are like in 'consist' test, only
`vars = ["x","y","z"],  ppo = (("degl",3), degLex, [])`
(in `MuPAD`, `ppo = DegreeOrder`).

51.3.3 Cyclic roots

Do not confuse this example with the one of the roots of equation $x^n - 1$.

It is said, this example comes from the works by G.Bjoerck:

`fs = [`
$x_1 + x_2 + \ldots + x_n$,
$x_1x_2 + x_2x_3 + \ldots + x_{n-1}x_n + x_nx_1$,
$\ldots$
$x_1x_2..x_{n-1} + x_2x_3..x_n + \ldots + x_{n-1}x_n..x_{n-3} + x_nx_1..x_{n-2}$,
$x_1x_2..x_n - 1$ `],`

```
gs             = Groebner basis( fs ),
variables      = [x_1 .. x_n],
pp-comparison  = degRevLex   (DegInvLexOrder in MuPAD)
```

For `n = 2, 3, 5, 6` it appears that
`I = Ideal(gs) (= Ideal(fs))` has the zero dimension.
That is for each $i \in [1..n]$ there exists $g \in gs$ such that $lpp(g) = {x_i}^{n_i}$, $n_i > 0$.

Some control:
for `n = 4`, `gs` contains 7 polynomials, and their leading power products are

```
[[1,0,0,0],[0,2,0,0],[0,1,2,0],[0,1,1,2], [0,1,0,4],[0,0,3,2],[0,0,2,4]]
```

; for `n = 5`, `gs` contains 20 polynomials, and their leading power products are

```
[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,1,2,0,0],[0,0,0,4,0],
 [0,0,1,3,0],[0,1,0,3,0],[0,0,2,2,0],[0,1,1,2,0],[0,1,1,1,2],
 [0,1,0,0,5],[0,0,1,2,3],[0,1,0,2,3],[0,0,1,1,5],[0,0,2,0,5],
 [0,0,0,3,4],[0,0,0,0,8],[0,0,0,1,7],[0,0,1,0,7],[0,0,0,2,6]
]
```

<u>DoCon program</u>

The below function `gcRoots`

takes `n` and builds the needed equations `eqs = eqs(n)` in variables $[x_1, \ldots, x_n]$;

finds their Gröbner basis `gs` and its leading power products `ps`;

returns `(eqsM, ps, gsM)`, where `eqsM, gsM` are `eqs` and `gs` respectively converted to the matrices for the need of nice printing.

`eqsM, ps` are used for the control.

Run this function, for example, in Interpreter:

```
ghci -pacage docon M.hs  +RTS -M... -RTS
...
M> :set +s
M> let (eqs, ps, gsm) = gcRoots n    -- put n = 4,5,...
...
M> eqs                             -- control
...
M> gs                              -- time consuming part
...
M> ps                              -- control
...
```

```
import qualified Data.Map as Map (empty)
import DExport
type K = Fraction Z
gcRoots :: Z -> ([Pol K], [[Z]], [Pol K])
gcRoots n =  (eqs, pps, gs)
  where
  gs   = fst $ gxBasis eqs
  pps  = map (vecRepr . lpp) gs        -- for control
  eqs  = fn: (eqs' [vpols] vpols 1)
         where
         unK   = 1:/1 :: Fraction Z
         dK    = upField unK Map.empty
         vars  = map (('x':) . show) [1 .. n]       -- ["x1".."xn"]
         o     = (("degRevLex", n), degRevLex, [])
         unp   = cToPol o vars dK unK
         vpols = varPs unK unp             -- [x1..xn] as polynomials
         fn    = (product1 vpols) - unp    -- x1*..+xn - 1

         eqs' (monpols: mpss) vps k =
                    if k == (n-1) then  map sum1 (monpols: mpss)
                    else
                      let rotate (x: xs) = xs ++ [x]
                          vps'           = rotate vps
                          mps            = zipWith (*) monpols vps'
                      in  eqs' (mps: monpols: mpss) vps' (k+1)
```

MuPAD program

Caution: you may need to remove all line breaks in each of the below polynomial expressions:

```
vars := [x1,x2,x3,x4,x5,x6]:
cp   := DegInvLexOrder:
grb  := func( groebner::gbasis(hs,cp), hs ):
lm   := func( lmonomial(h,cp), h ):

f1 := poly( x1*x2*x3*x4*x5*x6 - 1, vars ):
f2 := poly( x1*x2*x3*x4*x5 + x1*x2*x3*x4*x6 + x1*x2*x3*x5*x6
            + x1*x2*x4*x5*x6 + x1*x3*x4*x5*x6 + x2*x3*x4*x5*x6,
            vars
          ):
f3 := poly( x1*x2*x3*x4 + x2*x3*x4*x5 + x1*x2*x3*x6 + x1*x2*x5*x6
            + x1*x4*x5*x6 + x3*x4*x5*x6,
            vars
          ):
f4 := poly( x1*x2*x3 + x2*x3*x4 + x3*x4*x5 + x1*x2*x6 + x1*x5*x6 +
            x4*x5*x6,
            vars
          ):
f5 := poly( x1*x2 + x2*x3 + x3*x4 + x4*x5 + x1*x6 + x5*x6, vars):
f6 := poly( x1 + x2 + x3 + x4 + x5 + x6, vars ):
fs := [f1,f2,f3,f4,f5,f6]:

t0 := time():
gs := grb(fs):
t  := time() - t0;

ms := map(gs,lm):
write(Text,"res" ,ms):
write(Text,"res1",gs):
quit
```

Usage: put this program to file `m` and command

`time  mupad < m`

This will
form the equations for `n = 6`, find their Gröbner basis `gs`,
print `gs` to file `res`, print leading monomials of `gs` to file `res1`,
print the timing.

Axiom program

Caution: you may need to remove all line breaks in each of the below polynomial expressions:

```
vars := [x1,x2,x3,x4,x5,x6];
P    := MPOLY(vars, Fraction Integer)
)set message time on
f1 :P := x1*x2*x3*x4*x5*x6 - 1;
f2 :P := x1*x2*x3*x4*x5 + x1*x2*x3*x4*x6 + x1*x2*x3*x5*x6
          + x1*x2*x4*x5*x6 + x1*x3*x4*x5*x6 + x2*x3*x4*x5*x6;

f3 :P := x1*x2*x3*x4 + x2*x3*x4*x5 + x1*x2*x3*x6 + x1*x2*x5*x6
          + x1*x4*x5*x6 + x3*x4*x5*x6;

f4 :P := x1*x2*x3 + x2*x3*x4 + x3*x4*x5 + x1*x2*x6 + x1*x5*x6 + x4*x5*x6;
f5 :P := x1*x2 + x2*x3 + x3*x4 + x4*x5 + x1*x6 + x5*x6;
f6 :P := x1 + x2 + x3 + x4 + x5 + x6;

fs : List P := [f1,f2,f3,f4,f5,f6];
gs := groebner fs;   ms := [leadingMonomial g  for g in gs];
)spool res
output (# gs); output ms; output gs;
)spool off
           Running:   time axiom
                      )r a.input
                      )q
```

51.3.4 Timing [sec]:

```
                           DoCon                  Axiom       MuPAD

'Consistency' n = 3 |      0.3                     0.2         0.4
              n = 4 |    174 (special order)      34          17
AL                  |      2.2                    82, 329      1.9
Cyclic roots (5)    |      4.1                     -           -
             (6)    | > 1000 (interrupted,        > 1800       80
                               50Mb used)
Note:
Axiom computes the Groebner basis in unknown to us term ordering.
The numbers  82, 329  in AL example correspond to the lists
[x,y,z], [z,y,x].
```

Comments

Generally, the Gröbner basis task is rather expensive. In unlucky cases the cost of finding Gröbner basis for $F = [f_1, \ldots, f_k]$ in $R[x_1, \ldots, x_n]$ may grow as about

$$O((d \cdot k)^{2^n}), \quad d = max_{f \in F} totalDegree(f).$$

Hence any algorithmic method has to "drop into a hole" of many hours of computation on some example of small enough size. Very naturally, the Gröbner basis algorithms in `DoCon, Axiom, MuPAD` contain several optimization rules of the mathematical nature. But most probably, they have different rule sets, the author of this manual does not know the precise algorithms applied in `Axiom` and `MuPAD`. The purpose of these optimizations is to prevent "dropping into a hole" in possibly many cases.

Concerning the nature of `Axiom` falls, we cannot say anything definite, because do not know so far how to control the term ordering.

The last example is unlucky for the `DoCon` optimization. We think, this is not due to the programming system.

Probably, `DoCon` has to optimize its Gröbner basis strategy.

May this `DoCon` loss in CyclicRoots(6) be due to some bug? Hardly so. Because the cases of `n = 4, 5` are computed correct, as well as several other examples, not mentioned in this manual.

51.4 Factoring in $GF(p)[x]$

```
Kn   factor (t^8 +t^6 +10*t^4 +10*t^3 +8*t^2 +2*t +8)   in  (Z/(13))[t]
--
  = [(t^3 +8*t^2 +4*t +12, 1), (t^4 +2*t^3 +3*t^2 +4*t +6, 1), (t+3, 1)]

Z3-81  factor (t^81 - t)  over Z/(3) -->
-----                       t*(t+1)*(t+2)                    -- 1
                            *(t^2+1)*(t^2+2*t+2)*(t^2+t+2)   -- 2
                            *(t^4+2*t+2)*(t^4+t+2) *         -- 4
                            ... further, all of  deg = 4
                            *(t^4+t^3+t^2+2*t+2)*(t^4+t^3+t^2+t+1)

Z3-36-1  factor (t^36 + t + 1)  over Z/(3) -->
-------     (t+2)
           *(t^3 +2*t^2 +2*t +2)
           *(t^13 +t^11 +2*t^10 +t^8 +t^7 +t^6 +2*t^5 +t^3 +t^2 +2)
           *(t^19 +2*t^18 +t^16 +2*t^15 +2*t^14 +t^10 +2*t^8 +2*t^7
             +t^6 +2*t^3 +2*t^2 +2*t +2
            )

Z3-72-1    factor (t^72 + t + 1)  over Z/(3) -->
-------                       [p(1), p(2), p(3), p(4), p(5), p(6)],
                              deg p(i) = [1, 4, 4, 18, 22, 23]

Z3-144-1   factor (t^144 + t + 1)  over Z/(3)  -->
--------                                [p(1), p(2), p(3), p(4)],
                                        deg p(i) = 1, 22, 29, 92.

Program in DoCon:
-----------------
module Foo where
import qualified Data.Map as Map (empty)
import DExport
t p str =                                -- example:  t 3 "t^81 - t"
  let
      iI = eucIdeal "be" p [] [] [(p,1)]   -- start preparing K = Z/(p)
      r0 = Rse 0 iI dZ                     -- 0 in Z/I
      dK = upField r0 Map.empty
      p1 = cToUPol "t" dK r0               -- <- K[t]
      f  = smParse p1 str
  in  writeFile "res" $ show $ factor f
Running:
         run ghci interpeter, see the result in file  ./res
```

```
In Axiom:
--------    P := UnivariatePolynomial(t, PrimeField ...)
            )set message time on
            f :P := ...
            factor(f)
            Running:     time  axiom  ‘)r a.input   )q y’

In MuPAD:   q := ...:
---------   f := poly( ..., [t], IntMod(q) ):
            t := time():
            factor(f);    time() - t;  quit
  Running:
  write the above program to file ./m and command  time mupad < m
  To see the result in the file  ./t  apply   tee | time mupad > t
  and command there  read("m"); Control-d
```

Timing [sec]:

```
            DoCon    Axiom     MuPAD
Kn       |   0.1      0.2       0.2
Z3-81    |   0.5      0.5       0.6
Z3-144-1|   1.2      1.9       2.9
Z3-288-1|   3.4     20         16
Z3-432-1|  12       55        114
Z3-576-1|  39       83        618
```

Conclusion

This table shows that `DoCon` and `Axiom` computation seem to have the same cost order in `n` on this `Z3-n` task, and `MuPAD` seems to have somewhat higher cost order.

51.5 Factoring in $GF(p)[x,y]$

Example to start with: factoring in `(Z/(5))[x,y]`

```
E4-1   factor (f*h),  f = x^4 + y*(y*x^2 + 1)      -- irreducibles,
----                  h = x^4 + y*(y*x + y^2 + 2)  -- n = 4

E4-2   factor (f*h),  f = x^4 + y*(y^2*x^2 + 1)    -- irreducibles
----                  h = x^4 + y*(y*x + y^2 + 2)  --

Program in DoCon:
-----------------
import qualified Data.Map as Map (empty)
import DExport
main = interact process           -- it reads, factors, tests and prints out
  where
  process s = case  tf (read s :: (Z, [String], String, String))
              of
              (b, ft) -> shows b "\n" ++ (shows ft "\n")

type P = Pol (ResidueE Z)    -- (Z/(q))[x,y]

tf :: (Z, [String], String, String) -> (Bool, Factorization P)
tf    (q, vars,     fStr,   gStr  ) =
                     -- Read two polynomials, multiply them,
                     -- factor the product over  Z/(q),
                     -- multiply the factors by 'unfactor',
                     -- return factors and test for their product.
  let                -- Example:   tf (5, ["x","y"], "x+y", "x-y")
     iI = eucIdeal "bef" q [] [] []
     r0 = Rse 0 iI dZ
     dK = upField r0 Map.empty
     p1 = cToPol (lexPPO 2) vars dK r0         -- unity polynomial
     h  = canAssoc ((smParse p1 fStr)*(smParse p1 gStr))
     ft = factor h
  in ((canAssoc $ unfactor ft) == h,  ft)
Usage:
  ghc $doconCpOpt --make Main
  time a.out +RTS -M... -RTS  < d > r
                                   -- reads from file ./d writes to ./r
  Example contents of ./d :  (5,
                              ["x","y"],     -- try swapping x,y
                              "(x+y)^2*x",
                              "x^3 - y + 3"
                             )
```

```
in Axiom:  P := MPOLY([x,y], PrimeField 5)
--------   )set message time on
           f :P := ... ;
           h :P := ... ;  e := expand(f*h);  factor(e)

in MuPAD:  q    := ...:
---------  vars := ...:
           f    := poly( ..., vars, IntMod(q)):
           h    := poly( ..., vars, IntMod(q)):
           t    := time():
           e := expand(f*h):  factor(e);  time() - t;  quit

--------------------------------------------------------------
Timing:     Example E4-1    Example E4-2
-------    [x,y]  [y,x]

DoCon      0.8    0.6        2.4  2.4
Axiom      1.2    2.2        3.3  2.6
MuPAD      1.8    1.7       20    1.8
```

RandRand family

Given, `n`, `m`, form random non-constant monic polynomials $f_1 \dots f_m$ from $(Z/(p))[x,y]$, $p = 5$, $\deg_v f_i \leq n$ for $v = x,\ y$ (f_i often occur irreducible); find $factor(f_i * f_j)$ for $1 \leq i < j \leq m$.

Each pair (f_i, f_j) defines two examples: with the variable orders $[x, y]$ and $[y, x]$. Thus, for f_1, f_2, f_3, f_4, we have to factor 12 polynomials.

To avoid displaying too large expressions, we sieve each obtained random polynomials by deleting some monomials. For example, for `n = 8, 9, 10`, there remains each fourth or each fifth monomial.

The obtained poynomials are displayed below, so that the reader can see the precise test.

Below, the average and maximal time are shown for $n = 3, 8, 9, 10$.

n = 3

```
f1 = x^3*y^3 + 2*x^3*y + 3*x^3 + 3*x^2*y^3 + 3*x^2*y^2 + 4*x^2*y
      + 3*x^2 + 4*x*y^3 + x*y^2 + x*y + 2*x + y^2 + 4*y + 4

f2 = x^3*y^3 + 3*x^3*y^2 + 3*x^3 + 2*x^2*y^3 + 3*x^2*y^2 + 4*x^2*y
      + 3*x*y^3 + 2*x*y^2 + 4*x*y + x + 3*y^3 + y^2 + y + 4

f3 = x^3*y^3 + 4*x^3*y^2 + 3*x^3*y + 2*x^2*y^3 + 4*x^2*y + 4*x^2
      + x*y^3 + 2*x*y^2 + 4*x*y + 3*x + 4*y^2 + 4
```

For the pairs `(f1,f2), (f1,f3), (f2,f3)`, set the `./d` file contents and run the program as shown above. For example, for `(f1,f2)`, put

```
(5,
 ["x","y"],
 "x^3*y^3 + 2*x^3*y + 3*x^3 + 3*x^2*y^3 + 3*x^2*y^2 + 4*x^2*y
   + 3*x^2 + 4*x*y^3 + x*y^2 + x*y + 2*x + y^2 + 4*y + 4 ",

 "x^3*y^3 + 3*x^3*y^2 + 3*x^3 + 2*x^2*y^3 + 3*x^2*y^2 + 4*x^2*y
  + 3*x*y^3 + 2*x*y^2 + 4*x*y + x + 3*y^3 + y^2 + y + 4 "
)
```

For `n = 8,9,10`, the pairs

`(f1,f2), (f1,f3), (f1,f4),  (f2,f3), (f2,f4), (f3,f4)`

are tested.

```
Timing [sec]:   DoCon  Axiom  MuPAD
n=3 (f1,f2)       1.0    1.7    2.0
    (f1,f3)       0.5    1.6    2.2
    (f2,f3)       1.2    1.7    2.3
```

n = 8

```
f1 =
  x^8*y^8 + 3*x^8*y^4 + 3*x^8*y + 4*x^8 + 2*x^7*y^6 + 4*x^7*y^2
   + x^7*y + 2*x^6*y^6 + 3*x^6*y^2 + 4*x^6 + x^5*y^6 + 2*x^5*y^3
   + 4*x^4*y^8 + 2*x^4*y^4 + x^4*y + x^3*y^5 + 2*x^3*y^2 + 4*x^2*y^8
   + 2*x^2*y^7 + 2*x^2*y^3 + 3*x^2*y^2 + 2*x*y^8 + 4*x*y^5 + 2*x*y^2
   + x*y + y^5 + 2*y + 2
f2 =
  x^8*y^6 + x^8*y^5 + 2*x^8*y + 3*x^7*y^7 + 4*x^7*y^3 + x^7
   + x^6*y^8 + x^6*y^6 + 2*x^6*y^2 + x^5*y^7 + x^5*y^3 + x^4*y^7
   + x^4*y^5 + 2*x^4*y + 4*x^3*y^3 + 4*x^3 + 4*x^2*y^5 + 4*x^2
   + 4*x*y^8 + 4*x*y^6 + 3*x*y^2 + y^3 + 4*y + 4
f3 =
  x^8*y^7 + 2*x^8*y^4 + x^7*y^8 + 3*x^7*y^7 + x^7*y^4 + 3*x^6*y^8
   + 3*x^6*y^6 + 2*x^6*y^3 + x^6 + 4*x^5*y^6 + x^5*y^3 + 2*x^5
   + 3*x^3*y^4 + 4*x^3*y + 4*x^2*y^6 + 4*x^2*y^2 + x*y^7 + x*y^3
   + 3*x*y^2 + y^3 + 4*y + 4
f4 =
  x^8*y^8 + 3*x^8*y^5 + 4*x^7*y^8 + 2*x^7*y^3 + 2*x^6*y^8
   + 4*x^6*y^4 + 3*x^6*y^3 + 4*x^5*y^8 + 2*x^5*y^4 + x^4*y^8
   + 2*x^4*y^5 + 4*x^4 + 3*x^3*y^6 + 2*x^3*y^3 + 3*x^3 + 4*x^2*y^6
   + 2*x^2*y^5 + 2*x^2*y^2 + 2*x*y^8 + x*y^5 + 2*x*y + 4*y^7 + 2*y^6
   + 3*y^2 + y + 1
```

```
Timing [sec]:
n = 8       DoCon          Axiom        MuPAD
         [x,y]  [y,x]

(f1,f2)     24      9 |    174   176 |    26     30
(f1,f3)     11     33 |    166   160 |    23     28
(f1,f4)    242    108 |    166   175 |    36     23
(f2,f3)    245     25 |    153   146 |   252     32
(f2,f4)    107    103 |    350   164 |   138     27
(f3,f4)     61      8 |    297   151 |    33     26
```

```
------------------------------------------------------------------------
Further comparing versions of DoCon and GHC
-------------------------------------------

June 2005.  600 Mhz machine.   -M60m  for memory.

Parsing, factoring and testing of   f1 * f4    (for n = 8).

main = putStr (shows (tf (5, ["x", "y"], f1, f4)) "\n")

Making:  ghc --make -O $doconCpOpt  Main
Runing:  ./a.out +RTS -M60 -RTS

                      DoCon-2.06,     DoCon-2.09
                      ghc-5.02.3      ghc-6.4.1-June-14    July-12

Code size [byte]
  a.out  |            8.974.226        7.193.088          7.215.984
  Main.o |              202.288          124.188            119.704

Time [sec]
  [x,y] |             290              313                  308
  [y,x] |                              145                  144

Minimal -M size [Mb]
  [x,y] |               9                9                    9

The time does not change with the memory increase above 60 Mb.
------------------------------------------------------------------------
```

n = 9

```
f1 =
  x^9*y^9 + 2*x^9*y^7 + 3*x^9*y^6 + 3*x^9*y^2 + 4*x^9*y + x^9
   + x^8*y^9 + 4*x^8*y^5 + 4*x^8*y^4 + 2*x^7*y^4 + 4*x^7*y^3
   + x^7*y^2 + 4*x^6*y^8 + 2*x^6*y^7 + 3*x^6*y + 3*x^6 + 3*x^5*y^2
   + x^5*y + 4*x^5 + 4*x^4*y^5 + 2*x^4*y^4 + 4*x^3*y^7 + 2*x^3*y^6
   + 3*x^3*y^5 + x^2*y^9 + 2*x^2*y^6 + 2*x^2 + 2*x*y^9 + x*y^6
   + x*y^5 + 2*x + 2*y^9 + 3*y^8 + 4*y^7 + y^3 + y^2 + y + 1
f2 =
  x^9*y^8 + x^9*y^7 + 2*x^9*y^5 + x^9 + 3*x^8*y^8 + x^8*y^3 + x^8*y
   + x^8 + x^7*y^9 + 4*x^6*y^8 + 3*x^6*y^7 + 4*x^6*y + 3*x^6
   + 4*x^5*y^8 + x^5*y + x^5 + x^4*y + x^4 + 4*x^3*y^9 + 2*x^3*y^3
   + x^3*y + x^3 + x^2*y^5 + x^2*y^4 + 4*x*y^6 + 3*x*y^5 + 2*x*y^4
   + 2*x*y^3 + 4*y^8 + y^7 + 2*y^6 + 2*y^2 + 3*y
f3 =
  x^9*y^8 + 2*x^9*y^7 + x^9*y^6 + 2*x^9*y + x^8*y^9 + 3*x^8*y^4
   + 3*x^8*y^2 + 4*x^7*y^4 + 4*x^7*y^3 + 3*x^6*y^6 + x^6*y^5
   + 2*x^6*y^3 + 2*x^5*y^7 + 4*x^5*y^6 + x^4*y^7 + 4*x^4*y^6 + 2*x^4
   + 3*x^3*y^8 + 3*x^3*y^7 + x^2*y^9 + 3*x^2*y^7 + 3*x*y^7 + 2*x*y^4
   + x*y^3 + x*y^2 + 4*y^8 + 3*y^7 + 3*y^6 + 3*y^2 + 2*y + 1
f4 =
  x^9*y^9 + 2*x^9*y^8 + 2*x^9*y^7 + x^9*y^3 + 4*x^9*y^2 + 3*x^9
   + 2*x^8*y^9 + 4*x^8*y^7 + x^8*y^2 + 4*x^8*y + 3*x^7*y^6
   + 4*x^7*y^5 + 2*x^7*y^4 + x^7 + 2*x^6*y^9 + 2*x^6*y^8 + 3*x^6
   + 2*x^5*y^8 + 3*x^5*y^7 + 2*x^5 + 3*x^4*y^9 + 4*x^4*y^2
   + 3*x^4*y + 2*x^4 + 2*x^3*y^4 + 4*x^3*y^3 + 4*x^2*y^5 + 3*x^2*y^4
   + 4*x^2*y^3 + 3*x^2*y^2 + 4*x*y^6 + 4*x*y^5 + 4*y^9 + 3*y^8
   + 4*y^7 + 2*y^4 + y^3 + 2
```

Timing [sec]:

n = 9	DoCon		Axiom		MuPAD	
	[x,y]	[y,x]				
(f1,f2)	16	170	514	787	42	2851
(f1,f3)	439	49	371	720	41	2846
(f1,f4)	120	98	396	386	40	41
(f2,f3)	4	374	629	551	2517	2527
(f2,f4)	611	37	630	737	20907	2581
(f3,f4)	61	177	369	759	45	50

n = 10

```
f1 =
 x^10*y^10 + 2*x^10*y^8 + 3*x^10*y^3 + 4*x^9*y^7 + 2*x^9*y + 3*x^9
   + 4*x^8*y^10 + 3*x^8*y^8 + x^8*y^2 + 2*x^8 + 3*x^7*y^10
   + 3*x^6*y^7 + x^6*y^6 + 3*x^5*y^9 + x^5*y^8 + x^4*y^9 + 3*x^4*y^8
   + 4*x^4 + 4*x^3*y^2 + x^3 + 4*x^2*y^3 + x^2*y^2 + x^2*y + x^2
   + x + 3*y^6 + y^4 + y^2 + y + 1
f2 =
  x^10*y^9 + 3*x^10*y^7 + x^9*y^8 + 3*x^9*y^6 + 2*x^8*y^3
   + 3*x^7*y^10 + x^7*y^7 + 4*x^7 + 4*x^6*y + 4*x^6 + 3*x^5
   + x^4*y^6 + 4*x^4*y^5 + 3*x^3*y^6 + 4*x^3*y^5 + 2*x^2*y^8
   + 2*x^2*y^7 + x*y^10 + 4*x*y^9 + 4*x + 4*y^10 + 2*y^8 + 3*y^3
   + 4*y^2 + 3*y
f3 =
  x^10*y^10 + 2*x^10*y^9 + 2*x^10*y^8 + x^9*y^7 + x^9*y^5 + x^9*y^4
   + 4*x^9*y^2 + 4*x^8 + x^7*y^5 + 3*x^7*y^4 + 2*x^6*y^2 + x^6*y
   + x^6 + 3*x^5*y^3 + 3*x^5*y^2 + x^4 + 2*x^3*y^9 + x^3*y^8 + 4*x^3
   + 3*x^2*y^10 + x^2*y^2 + 3*x*y^2 + 2*x*y + 3*y^6 + 3*y^3 + 4*y^2
   + 3*y + 4
f4 =
  x^10*y^10 + 4*x^10*y^9 + 4*x^10*y^8 + 4*x^10*y^2 + x^10*y
   + 3*x^9*y^10 + 2*x^9*y^5 + 2*x^9*y^3 + 3*x^9 + 3*x^8*y^5
   + x^8*y^4 + 3*x^8*y^3 + 4*x^7*y^5 + 2*x^7*y^4 + 2*x^6*y^3
   + 2*x^6*y^2 + x^6 + 3*x^5 + x^4*y^6 + 3*x^4*y^5 + x^3*y^4
   + 2*x^3*y^2 + x^2*y^9 + 3*x^2 + 4*x*y^10 + 4*x + 4*y^10 + 2*y^9
   + 4*y^3 + 3*y^2 + y + 1

Timing [sec]:
n = 10         DoCon        Axiom       MuPAD
            [x,y] [y,x]
(f1,f2)     313   1422 |  566   578 |   541   542 |
(f1,f3)    1310   1397 |  585   876 |  4968   109 |
(f1,f4)     451   1373 |  619   622 |    89    72 |
(f2,f3)      30    105 |  537   557 |    64    72 |
(f2,f4)    1387     16 |  590   602 |   102    62 |
(f3,f4)      56   1370 |  620  1141 |    74    93 |
```

Summarized statistic

```
March 30 - April 10, 2002, Medicis Center, laudomia4, Linux

Timing [sec]:
-------------
    Average time | maximal time

DoCon      0.9         1.2               n =  3
Axiom      1.7         1.7
MuPAD      2.2         2.3
----------------------------------------------
DoCon     81         245                n =  8
Axiom    190         350
MuPAD     56         252
----------------------------------------------
DoCon    180         611                n =  9
Axiom    571         787
MuPAD   2874       20907
----------------------------------------------
DoCon    769        1422                n = 10
Axiom    658        1141
MuPAD    566        4968
```

Conclusions

Of course, without knowing the very factoring methods in `Axiom`, `MuPAD`, these numbers do not mean much.
The impression is as follows.
The programs look equally fast on average. Because n = 10 implies very large intermediate computations with the polynomials of degree 20 both in x and in y. And any essential difference in the order of the algorithm is likely to cause a great difference in the cost for such large data.
`Axiom` looks the most stable.
`MuPAD` falls into a hole of 6 hours in one of the examples for n = 9. This is somewhat suspicious.

Some details about `DoCon` :
it applies the method by [Me3] adopted from [CG] (see also [Le]),
whith the proved upper bound main part of $O(\deg_x^4 \cdot deg_y^3)$.

The Hensel lift takes from 0.15 (for large `n`) up to 0.60 of the cost (ghc-4.08 profiling).
The rest is spent by finding of a small vector in certain lattice over $GF(p)[x]$.

The square-freeing repeats the GCD operation in $GF(p)[x, y]$; it cost is made small due to applying a special Chinese-remainder method `upolGCD_Chinese`.

51.6 Conclusions on whole testing

Algorithms and programming tools

We believe, in the above examples, as everywhere in practice, the method details (*mathematics*) cost more than the difference in the programming systems — for any reasonably designed programming tools.

Looking at the above examples, we think that `Haskell` seems efficient enough not to loose the performance gain, when it comes from the algorithm.

A good benchmark needs much more effort, maybe, one year of studying `Axiom`, `MuPAD` programming and designing examples.

Algorithm quality

- Polynomial arithmetic (`+, *, ^`)
  looks almost as fast in `DoCon` as in `Axiom`, `MuPAD` — for not very large data.
  Asymptotically, `DoCon` looses a little to `Axiom` and looses considerably to `MuPAD`. Probably, this is due to insufficient optimization in algorithm.
- On half of the Gröbner basis tasks, `DoCon` looses greatly in performance to `MuPAD`. No doubt, this is due to the insufficient optimization in the algorithm applied in `DoCon`.
- In the example serie for multivariate polynomial GCD,
  and for factoring in $GF(p)[x]$
  (and maybe, some implied linear algebra),
  `DoCon` wins a little relatively to `Axiom`
  and wins considerably relatively to `MuPAD`.
- In factoring in $GF(p)[x,y]$,
  the programs `Axiom, DoCon, MuPAD` show approximately equal performance, even for the large degrees.
  But `MuPAD` falls strangely into a 6 hours 'hole' on one of the suggested examples.

Remark on linear systems

In factoring in $GF(p)[x]$, `DoCon` relies on the large dense linear system solution over $Z/(p)$, and $GF(p)[x,y]$ relies on the large *sparse* linear system solution over $Z/(p)$. In both cases `DoCon` represents a matrix as a list of rows, a row — as a list of matrix entities or as a list of indexed entities. `DoCon` avoids arrays.

52 Literature references

References

[Al] S. M. Watt et al. *Aldor Compiler User Guide.*
IBM Thomas J. Watson Research Center. `<http://www.aldor.org>`

[Au] L. Augustsson. *Cayenne — a language with dependent types.*
In Proceedings of International Conference on Functional Programming (ICFP'98). ACM Press, September 1998.

[Bu] B. Buchberger.
Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.
CAMP. Publ. No.83–29.0 November 1983

[BL] B. Buchberger, R. Loos. *Algebraic Simplification.*
In "Computer Algebra. Symbolic and Algebraic Computation",
edited by B.Buchberger et al., Springer Verlag, 1982,1983

[CG] A. L. Chistov, D. Yu. Grigoryev.
Polynomial-time factoring of the multivariable polynomials over a global field.
Preprint E-5-82 of the Leningrad department of Steklov Mathematical Institute LOMI, USSR, 1982.
There also exists the LOMI preprint in Russian of 1984 containing the same algorithm description.

[Da] J. H. Davenport, B. M. Trager.
Scratchpad's View of Algebra I: Basic Commutative Algebra.
Lecture Notes in Computer Science, Vol. 429, pp.40–54 (1990).

[Ed] H. M. Edwards.
Fermat's last theorem. A genetic introduction to algebraic number theory. Springer-Verlag, 1977

[FH] A. J. Field, P. G. Harrison *Functional programming.*
Addison-Wesley, 1988.

[Fo] J. Fokker. *Explaining algebraic theory with functional programs.*
Proceedings of FPLE'95, LNCS 1022, pp. 139-158.
`<http://www.cs.ruu.nl/people/jeroen>`

[GH] *Glasgow Haskell Compiler.* `<http://www.haskell.org/ghc>`.

[GM] R. Gebauer, H. M. Moeller. *On Installation of Buchberger's Algorithm.* Journal of Symbolic Computation (1988), Vol. 6.

[GTZ] P. Gianni, B. Trager, G. Zacharias. *Gröbner Bases and Primary Decomposition of Polynomial Ideals.* Journal of Symbolic Computation (1988), Vol.6, pp.149-167

[Ha] *Haskell 2010: A Non-strict, Purely Functional Language.* Report of 201. `<http://www.haskell.org>`

[HJ] R. Hinze, S. P. Jones. *Derivable type classes.* In Proceedings of the Haskell Workshop, Montreal, 2000. `<http://www.dcs.gla.ac.uk/~simonpj>`

[HFP] P. Hudak, J. Fasel, J. Peterson. *A gentle introduction to Haskell.* Technical report, YALEU/DCS/RR-901, Yale University. 1996. `<http://haskell.org/tutorial>`

[Je] R. D. Jenks, R. S. Sutor, et al. *Axiom, the Scientific Computation System.* Springer-Verlag, New York–Heidelberg–Berlin (1992).

[JJM] S. P. Jones, M. Jones, E. Meijer. *Type classes: an exploration of the design space.* 1997. In Proceedings of the Haskell Workshop. `<http://www.dcs.gla.ac.uk/~simonpj/>`

[Jo] S. L. P. Jones et al. *Implementation of Functional Programming Languages.* Prentice Hall. 1987.

[Ka] J. Karczmarczuk. *Functional Programming and Mathematical Objects.* Proceedings of FPLE'95, Nijmegen 1995; Springer LNCS 1022.

[Kn] D. E. Knuth. *The Art of Computer Programming.* Volums 1,2,3. Addison-Wesley, 1973

[La] S. Lang. *Algebra.* Addison-Wesley, 1965.

[Le] A. K. Lenstra. *Factoring Multivariate Polynomials over Finite Fields.* Journal of Computer and System Sciences, Volum 30 (1985), pp.235-248.

[LLL] A. K. Lenstra, H. W. Jr. Lenstra, L. Lovasz.
Factoring polynomials with rational coefficients.
Math. Ann. Volum 261 (1982), pp.515-534.

[Ma] I. G. Macdonald. *Symmetric Functions and Hall Polynomials.*
Clarendon Press, Oxford, 1979.

[Md] M. Clavel & al.
Maude: Specification and Programming in Rewriting Logic.
<http://maude.csl.sri.com>

[Me1] S. D. Meshveliani.
The Algebraic Constructor CAC: Computing in Construction-defined Domains.
Lecture Notes in Computer Science, 722, 1993.

[Me2] S. D. Mechveliani. *BAL basic algebra library for Haskell.*
Paper manuscript (haskellInCA*) and BAL program source. Pereslavl-Zalessky, 2000-2001. <http://www.botik.ru/pub/local/Mechveliani/basAlgPropos/>

[Me3] S. D. Mechveliani.
Cost bound for LLL-Grigoryev method for factoring in GF(q)[x,y].
To appear (in Russian) in Fundamental and Applied Mathematics, Math.dep. of Moscow State University <http://www.math.msu.su/~fmp/>.
The file with English version will be available as
<http://www.botik.ru/pub/local/Mechveliani/otherPapers/ovfEng*>

[Me4] S. D. Mechveliani.
Computer algebra with Haskell: applying functional-categorial-'lazy' programming .
In Proceedings of International Workshop CAAP-2001, Dubna, Russia.

[Mi] M. Mignotte. *Mathématiques pour le caclul formel.*
Presses Universitaires de France, 1989.
English translation: *Mathematics for computer algebra.*
Springer Verlag, 1992.

[Mo] H. M. Mœller. *On the Construction of Gröbner Bases Using Syzygies.*
Journal of Symbolic Computation (1988), Vol. 6.

[MoM] H. M. Mœller, F. Mora.
New Constructive Methods in Classical Ideal Theory.
Journal of Algebra, v 100, pp.138-178 (1986)

[Mu] *MuPAD the Multi Processing Algebra Data Tool*
<http://www.mupad.de>, <ftp.mupad.de>,
<http://math-www.uni-paderborn.de/MuPAD/>

[Zi] R. Zippel. *The Weyl Computer Algebra Substrate.*
Lecture Notes in Computer Science, vol.722, pp.303-318 (1993)

53 Cross reference and help

The items are listed below in the alphabetical order.
Usually, an item is provided with reference, type description, short commentary.

`absValue` 14.3 `:: AddGroup a => a -> a`
Absolute value. Correct for `(IsOrderedGroup, Yes)`

`add` 12.2 addition operation of the `AddSemigroup` category

`addEPermut` 28.2 `:: EPermut -> EPermut -> EPermut`
composes e-permutations on disjoint sets

`addListToAssocList_C` 8 repeated `addToAssocList_C`
`:: Eq a => (b -> b -> b) -> [(a,b)] -> [(a,b)] -> [(a,b)]`

`AddGroup` 14.1 category of additive commutative groups

```
class AddMonoid a => AddGroup a where
    baseAddGroup ::  a -> Domains1 a -> (Domains1 a, Subgroup a)
```

`AddMonoid` 13 `class AddSemigroup a => AddMonoid a`
category of additive monoid (`zeroS x` gives zero)

`AddSemigroup` 12.2 a category

```
class Set a => AddSemigroup a where
          baseAddSemigroup, add, zero_m, neg_m, sub_m, times_m
```

`ToAssocList_C` 8
`addToAssocList_C :: Eq a => (b->b->b) -> [(a,b)] -> a -> b -> [(a,b)]`
combines with the previous binding

`addUnknowns` 8 `:: Eq a => [(a,PropValue)] -> [a] -> [(a,PropValue)]`
adds `(nm, Unknown)` for any `nm` skipped in association list

`addVarsPol` 38.3, 38.3
`:: Char -> PPOrdTerm -> [PolVar] -> Pol a -> Pol a`
$a[x_1,\ldots,x_n] \to a[y_1,\ldots,y_m,x_1,\ldots,x_n]$ or to $a[x_1,\ldots,x_n,y_1,\ldots,y_m]$

`adjointMt` 32.3 `:: CommutativeRing a => [[a]] -> [[a]]`
the adjoint matrix `A` for a square matrix `M`,
so that `M*A = (det M)*UnityMatrix`.

`ADomDom` 23 `type ADomDom a = a -> Domains1 a -> Domains1 a`

`algRelsPols` (38.5.3 {ar}) `:: EuclideanRing k =>`
`[Pol k]-> [PolVar]-> PPOrdTerm-> [Pol k]-> [Pol k]`

`AlgSymmF` 50, 45 `DoCon` module exporting items for Symmetric functions

`allMaybes` 8 `:: [Maybe a] -> Maybe [a]`

`allMaybesVec` 29.2

`(fmap Vec . allMaybes) :: [Maybe a] -> Maybe (Vector a)`

`allPermuts` 28.2 `:: [Integer] -> [Permutation]`
builds the full permutation list (in any order) given a list ordered *decreasingly*

`alteredSum` 8 `:: Num a => [a] -> a`
$[x_1, \ldots, x_n] \longrightarrow \sum_{i=1}^{n}(-1)^{i-1}x_i$ — for non-empty list

`and3` 8 `:: PropValue -> PropValue -> PropValue`

`antiComp` 8 `:: CompValue-> CompValue,   LT-> GT; GT-> LT; EQ-> EQ`

`applyPermut` 28.2 `:: Ord a => [Integer] -> [a] -> [a]`
`\ks -> map snd . sortBy (compBy fst) . zip ks`
example: `[2,3,4,1] "abbc" --> "cabb"`

`applyTransp` 28.2 `:: (Natural, Natural) -> [a] -> [a]`
Transpose elements No i, j, $1 \leq i \leq j \leq$ `length xs`

`applyTranspSeq` 28.2 `:: (Ord a, Show a) => (a,a) -> [(a,b)] -> [(a,b)]`
Transpose entities indexed by $i \leq j$ in a sequence of pairs …

`baseAddGroup` 14.1 operation from `AddGroup` category
`:: a -> Domains1 a -> (Domains1 a,Subgroup a)`

`baseAddSemigroup` 12.2, (3.4 BO)
`:: a -> Domains1 a -> (Domains1 a, Subsemigroup a)`
operation from `AddSemigroup` category

`baseEucRing` 19 operation from `EuclideanRing`
`:: a -> Domains1 a -> (Domains1 a, EucRingTerm a)`

`baseFactrRing` 17 operation from `FactorizationRing`
`:: a -> Domains1 a -> (Domains1 a, FactrRingTerm a)`

`baseGCDRing` 16 operation from `GCDRing` category
`baseGCDRing :: a -> Domains1 a -> (Domains1 a, GCDRingTerm a)`

`baseLinSolvLModule` 22.3, 22.4
`:: (r, a) -> Domains2 r a -> (Domains2 r a, LinSolvModuleTerm r a)`
operation from `LinSolvLModule r a` category

`baseLinSolvRing` 18 operation from `LinSolvRing` category
`:: a -> Domains1 a -> (Domains1 a, LinSolvRingTerm a)`

`baseMulSemigroup` 12.4, (3.4 BO) operation from `MulSemigroup` category
`:: a -> Domains1 a -> (Domains1 a, Subsemigroup a)`

`baseRing` 15, (3.4 BO) operation from `Ring` category
`:: a -> Domains1 a -> (Domains1 a, Subring a)`

`baseSet` 10.1, (3.4 DS) operation from `Set` category
`:: a -> Domains1 a -> (Domains1 a, OSet a)`
description of subset related to sample element

base-operation (3.4 BO) — such as `baseSet, baseAddGroup`, …

`binomCoefs` 8 `:: Natural -> [Natural]`

binomial coefficients: $n \to [C(n,k), \ldots, C(n,0)]$, $k \leq n/2+1$

`boolToPropV` 8 `(\b -> if b then Yes else No) :: Bool -> PropValue}`

`bundle` (3.4 MD) (domain, many-domain)

a finite collection of domain terms stored each under its Key,
Key = `CategoryName = Set | AddGroup| Ring |` ...

`canAssoc` 16 `:: a -> a`

operation from `GCDRing` category: canonical associated element

`canAssocM` 22.3, 22.4 `:: r -> a -> a`

Operation from `LinSolvLModule r a` category: canonical associated vector in `a`. First argument is a sample for `r`

`canFr` 34.2 bring to canonical fraction

`:: GCDRing a => String -> a -> a -> Fraction a`

`mode = "g"` means to cancel by gcd, "i", "" mean ...

`canInv` 16 `:: a -> a`

operation from `GCDRing` category: canonical invertible factor

`canInvM` 22.3, 22.4 `:: r -> a -> r`

Operation from `LinSolvLModule r a` category:

canonical invertible factor for the vector from `a`

`Cast` (3.4 CT), 8

category for mapping from domain of `b` to domain of `a`

`class Cast a b where cast :: Char -> a -> b -> a`

`category` (3.4 CT, IN), 3.3, a notion 'domain'.

In DoCon, it is a class of domains defined by `Haskell` class declarations, presumed conditions and sample element

CategoryName 9.1 `data CategoryName =`

`Set | AddSemigroup | AddGroup | MulSemigroup | MulGroup | Ring |`
`LinSolvRing | GCDRing | FactorizationRing | EuclideanRing |` ...

`Categs` 50 `DoCon` module exporting various `Domain` data descriptions

`char` 15.4 `:: Ring a => a -> Maybe Natural`

characteristic of a ring defined by the given sample

`Char` 25 Haskell type and Domain in DoCon.

`DoCon` supplies it with the `OrderedSet` instance

`charMt` 36.3 `:: CommutativeRing a => PolVar -> Matrix a -> Matrix (UPol a)`

`\ la mM -> the characteristic matrix (mM' - la*E)`,
— add `(- la)` to the main diagonal. `mM'` is `mM` imbed to `a[la]`.

`charPol` 36.3 `:: CommutativeRing a => PolVar -> Matrix a -> UPol a`

`\ la mM -> characteristic polynomial of mM in the variable la`

`coefsToPol` 38.3 coefficient list to polynomial
`:: Ring a => Char -> Pol a -> [Z] -> [a] -> (Pol a, [a])`
the polynomial domain is given by a sample

`CommutativeRing` 16 a category
`class Ring a => CommutativeRing a`
presumed: `(Commutative, Yes)`

`compare_m` 10 `:: a -> a -> Maybe CompValue`
operation from `Set` category for partial ordering
on the base set. Its simplest definition is `compareTrivially`

`Comparison` 8 `type Comparison a = a -> a -> CompValue`

`compBy` 8 `:: Ord b => (a -> b) -> Comparison a`
`\ f x y -> compare (f x) (f y)`

`compose` 8 `:: [a -> a] -> a -> a,` `compose = foldr (.) id`

`CompValue` 8 `type CompValue = Ordering`
a name `DoCon` uses for Haskell's `Ordering (LT | EQ | GT)`

`conjPrtt` 45.2.1 `:: Partition -> Partition` conjugated partition

`constMt` 31.2 `:: mapMt (const a) mM`
replaces all entities in a matrix with a given value

`Construction_Ideal` 21.3
`newtype Construction_Ideal a = GenFactorizations [Factorization a]`

`constructor` (3.4 DS) `DoCon` uses the data type
constructors (`Pol, Fraction, ResidueE ...`) to build domains
from other domains. And *instances* are defined for constructors.

`constVec` 29 `:: Vector a -> b -> Vector b`
Constant vector. Example: `(Vec [1,2,3]) 'c' -> Vec "ccc"`

`cPMul` 35 product of a polynomial class thing by coefficient
`(\a-> pMapCoef 'r' (mul a)) :: (PolLike p, Ring a) => a -> p a -> p a`

`ct` see `class Cast` in 8, (3.4 CS), 3.5.4
`(cast '_') :: Cast a b => a -> b -> a`

`cToEMon` 39.2.1 `:: [a] -> Z -> b -> EMon b`
makes e-monomial from coefficient, position No and number of variables

`cToEPol` 39.2.1 coefficient to e-polynomial
`AddGroup a => EPol a -> Z -> a -> EPol a`
parameters taken from e-polynomial sample

`cToPol`   38.3   coefficient to polynomial

`:: Ring a => PPOrdTerm -> [PolVar] -> Domains1 a -> a -> Pol a`

a natural way to produce some polynomial

`cToRPol`   40.3   coefficient to r-polynomial

`:: RPolVarsTerm -> Domains1 a -> a -> RPol a`

usable way to initiate some r-polynomial

`cToSymMon`   45.3.2   `(\c-> (c, [])) :: a -> SymMon a`

for `c` $\neq 0$, this yields sym-monomial

`cToSymPol`   45.3.2   similar to `cToPol`

`:: AddGroup a => PrttComp -> Domains1 a -> a -> SymPol a`

`cToUPol`   36.3   coefficient to univariate polynomial

`:: Ring a => PolVar -> Domains1 a -> a -> UPol a`

apply it to produce some univariate polynomial

`ctr`   see `class Cast` in refsec-dprel, (3.4 CS), 3.5.4

`(cast 'r') :: Cast a b => a -> b -> a`

`cubeList_lex`   8   lists in lex-increasing order all vectors $[a_1, \ldots, a_n]$

over `a` in the cube defined by bounds:

`:: (Shows a, Ord a, Enum a) => [(a,a)] -> [[a]]`

`deg`   35   `:: CommutativeRing a => p a -> Z`

Total degree. Operation from constructor class `PolLike`

`deg0`   35   `deg` generalized to "if zero, return given value"

`:: (PolLike p, AddSemigroup (p a), CommutativeRing a) =>`
`Char -> Z -> p a -> Z`

`degInVar`   35   operation from constructor class `PolLike`

`:: CommutativeRing a => Z -> Z -> p a -> Z`

deg f in variable No i, `for0` for zero f

`degLex`   37.2   `:: PPComp`   degree-lexicographic pp comparison

`degRevLex`   37.2   `:: PPComp`

pp comparison: first by total deg, then, by `lexFromEnd`

`delBy`   8   `:: (a -> Bool) -> [a] -> [a]`

deletes first element satisfying given predicate

`del_n_th`   8   `:: Z -> [a] -> [a]`   removes element No n from list

`denom`   34.2   `(_ :/ d) -> d`   denominator of fraction

`:: Ring a => a -> Mon a -> EMon a -> [EMon a]`
`\zero (a,p) (b,(j,q)) -> case mul a b of c -> if c==zero ...`

`EPartition` see 45.2.1, `toEPrtt`
`type EPartition = [Z]` — expanded partitions

`EPermut` 28.2 `type EPermut = [(Z,Z)]`
Extended form of permutation. It is made from
`Pm xs :: Permutation` by `zip (sort xs) xs`

`ePermutCycles` 28 `:: EPermut -> [EPermut]`
Decomposes extended permutation s to cycles $[c_1, \ldots, c_r]$,
length $c_1 \geq \ldots \geq$ length c_r

`EPol` see 39.2.1, `EMon`, 39.1, 39.1.2
`data EPol a = EPol ![EMon a] !EPPOTerm !(Pol a)`
indexed monomial-wise representation for polynomial vector

`epolECp` 39.2.1 `:: EPol a -> EPPComp`
`(eppoECp . epolEPPOTerm)` extracts comparison for e-power-products

`epolEPPOTerm` 39.2.1 `:: EPol a -> EPPOTerm` extracts EPPOTerm from e-pol

`epolLCoord` 39.2.1 `:: CommutativeRing a => EPol a -> Z`
leading e-monomial coordinate of non-zero e-polynomial

`epolMons` 39.2.1 `:: EPol a -> [EMon a]` extracts e-monomial list

`epolPol` 39.2.1 `:: EPol a -> Pol a` extracts polynomial sample

`epolPPCp` 39.2.1 `(polPPComp . epolPol) :: EPol a -> PPComp`
extracts comparison for polynomial power products

`epolToVecPol` 39.2.1 `:: CommutativeRing a => Z -> EPol a -> Vector (Pol a)`
Converts e-polynomial to polynomial vector `Vec [...]` of given size

`EPP` 39.2.1 `type EPP = (Z, PowerProduct)`
Extended power product (i, pp) represents the part of a
polynomial vector: monomial $1 \cdot pp$ related to coordinate No i

`EPPComp` 39.1 `type EPPComp = Comparison EPP`
used similarly as `PPComp`

`eppoCp` 39.1 `tuple44 :: EPPOTerm -> PPComp`
extracts pp comparison

`eppoECp` 39.1 `tuple41 :: EPPOTerm -> EPPComp` extracts epp comparison

`eppoMode` 39.1 `tuple42 :: EPPOTerm -> String` extracts mode

`eppoWeights` 39.1 `tuple43 :: EPPOTerm -> [PowerProduct]`
extracts weights

`EPPOTerm` 39.1 term describing epp ordering.
`type EPPOTerm = (EPPComp, String, [PowerProduct], PPComp)`
epp comparison function, mode, list of weights, ...

`EPVecP` 39.1 `type EPVecP a = (EPol a, [Pol a])`
used similarly as `PVecP`

`Eq` (3.4 BSE) the Haskell library class for the equality
operation `(==)`. `DoCon` defines `(==)` as algebraic
equality for each domain (often, not as syntactic idenity)

`eqListsAsSets` 8 `:: Eq a => [a] -> [a] -> Bool`
`\xs ys -> all ('elem' xs) ys && all ('elem' ys) xs`

`eqFactrz` 17.1 `:: Eq a => Factorization a -> Factorization a -> Bool`
"Equivalent factorizations". Order of factors is immaterial

`eucGCDE` 19.1 `:: EuclideanRing a => [a] -> (a, [a])`
extended GCD for a list: `\xs -> (d,qs), d =` $\gcd[x_1, \ldots, x_n] = \sum_{i=1}^{n} q_i x_i$

`eucIdeal` see `PIRChinIdeal`, 21.2 builds ideal description.
`:: (FactorizationRing a, EuclideanRing a) =>`
`String -> a -> [a] -> [a] -> Factorization a -> PIRChinIdeal a`

`Euclidean` 19 `:: Property_EucRing`
(Euclidean,Yes) means `eucNorm, (divRem anyMode)`
are correct algorithms for Euclidean ring structure

`EuclideanRing` 19 a category
`class (GCDRing a, LinSolvRing a) => EuclideanRing a where`
`{eucNorm :: a -> Z; divRem ... baseEucRing ...}`

`eucNorm` 19 `:: a -> Z` a norm, operation of `EuclideanRing` category

`EucRingTerm` 19 description of Euclidean ring attributes:
`data EucRingTerm a =`
`EucRingTerm {eucRingProps :: !Properties_EucRing} deriving(Show)`

`evenL` 8 `:: [a] -> Char`

'+' means that a given list has even length, '-' — odd length

`Expression` 3.14, 47
intermediate data between `String` and domain element.
Elements are parsed from expressions. Say `"(1+22)*3a"`
may parse by `(infixParse . lexLots)` to `E (L "*") ...`

`extendFieldToDeg` 36.5.3 extends finite field k to field of given dimension over k

```
:: Field k => UPol k -> Domains1 (UPol k) -> Z ->
            (ResidueE (UPol k), Domains1 (ResidueE (UPol k)))
```

`FAA` 46 Free associative algebra
(of non-commutative polynomials)

```
data FAA a = FAA [FAAMon a] a FreeMOrdTerm FAAVarDescr (Domains1 a)
```

`FAAMon` 46.2

```
type FAAMon a = (a, FreeMonoid)    -- FAA monomial
```

`faaNF` 46.3

```
:: (EuclideanRing a) => String -> [FAA a] -> FAA a -> ...
```

reduction to Groebner normal form of non-commutative polynomial over a Commutative Ring.

`FAAVarDescr` 46.2

```
type FAAVarDescr = (Maybe Z, (Z-> Maybe PolVar, PolVar-> Maybe Z))
```

`factor` 17 `:: a -> Factorization a`
factoring to primes: operation from `FactorizationRing` category

`factorial` 8 `{\0 -> 1; \n -> product [1..n]} :: Z -> Z`

`Factorial` 15.3 `:: Property_Subring` "is a unique-factorization ring"

`Factorization` 17 `type Factorization a = [(a,Z)]` Example:
`8*49*3 = 2^3*7^2*3` expresses as `[(2,3),(7,2),(3,1)] :: Factorization Z`

`FactorizationRing` 17 category of rings with factorization algorithm:

```
class GCDRing a => FactorizationRing a where
                     isPrime...factor...primes...baseFactrRing
```

`FactrRingTerm` 17 `data FactrRingTerm a = ...`
factorization ring attribute description

`factrzDif` 17.1 difference of factorizations, factors order immaterial.
`:: Eq a => Factorization a -> Factorization a -> Maybe (Factorization a)`
Example: `[(a,1),(b,2)] [(b,1)] --> Just [(a,1),(b,1)]`

`Field` see 20, notion `IsField`. A category:

`GCDRing` 16 category of rings with gcd algorithm:
`class (CommutativeRing a, MulMonoid a) => GCDRing a where`
`baseGCDRing...canAssoc...canInv...gcD...lcM...hasSquare...toSquareFree`

`GCDRingTerm` see 16, `Property_GCDRing` gcd-ring description term:
`data GCDRingTerm a = GCDRingTerm {gcdRingProps :: ...} ...`

`genFactorizationsFromIdeal` 21.3.1 `:: Ideal a -> Maybe [Factorization a]`
extract factorization list from first `GenFactorizations` construction

`gensToIdeal` 21.3.2 makes ideal from generators
`:: LinSolvRing a => [a] -> [Factorization a] -> Properties_IdealGen ->`
`Properties_Ideal -> Domains1 a -> Domains1 a -> (Domains1 a, Ideal a)`

`getOp` 47 get operation group from table (for parsing)
`:: Eq a => OpTable a -> a -> Maybe (OpGroupDescr a)`

`greaterEq_m` 8 `(x==y || greater_m x y) :: Set a => a -> a -> Bool`

`greater_m` 8 `:: Set a => a -> a -> Bool`
`case compare_m x y of {Just GT -> True, _-> False}`

`gxBasis` 18.1 `:: [a] -> ([a], [[a]])` generalization of Gröbner basis,
operation from `LinSolvRing` category,

`gxBasisM` 22.3, 22.4, 49 `:: r -> [a] -> ([a], [[r]])`
operation from `LinSolvLModule r a` category

`halve` 8 `:: [a] -> ([a], [a])` breaks list by the middle
Example: `[1,2,3,4,5] --> ([1,2,3],[4,5])`

`HasNilp` 15.3 `:: Property_Subring` "ring has a nilpotent"

`HasZeroDiv` 15.3 `:: Property_Subring` "ring has zero divisor"

`headVarPol` 38.3 brings polynomial to head variable
`:: CommutativeRing a => Domains1 (Pol a) -> Pol a -> UPol (Pol a)`
`a`$[x_1, x_2, \ldots, x_n] \longrightarrow$ `a`$[x_2, \ldots, x_n][x_1]$

`HBand` 45.2.1 `type HBand = [Z]` horizontal band (of combinatorics):
a skew diagram containing not more than one box in each column

`henselLift` 36.5.2
`:: (EuclideanRing a, FactorizationRing a, Ring (ResidueE a) =>`
`UPol a -> UPol a -> UPol a -> Maybe (UPol a) -> a -> a -> Z -> Z ->`

`(UPol a, UPol a, UPol a, a)`

`hookLengths` 45.2.2 `:: Partition -> [[Z]]`
a matrix, each `h(i,j)` is the length of the hook of the point `(i,j)` in a given Young diagram.

`hPowerSums` 45.4.3 `:: CommutativeRing a => Pol a -> [Pol a]`
homogeneous power sums $p_i = \sum_{k=1}^{n} x_k^i$
built from the given sample polynomial

`h'to_p_coef` 45.4.3 `:: Partition -> Z`
coefficient z_λ in expression $h(n) = sum_{|\lambda|=n}(1/z_\lambda)p_\lambda$
of the full homogeneous function h_n

`Ideal` 21.3 description of ideal in a ring:
`data Ideal a = Ideal {idealGens :: !(Maybe [a]), ...}`

`idealConstrs` 21.3 `:: ![Construction_Ideal a]`
part of description of ideal. So far, it may contain only
`(GenFactorizations fts)` — for a factorial ring `a`

`idealGenProps` 21.3 `:: !Properties_IdealGen`
part of description of ideal: properties of generator list

`idealGens` 21.3 `:: !(Maybe [a])`
Part of description of ideal: generators. `Just gs` means
a finite generator list, `Nothing` — such list not found

`idealOpers` 21.3 `:: !(Operations_ideal a)` part of description of ideal

`idealProps` 21.3 `:: !Properties_Ideal` part of description of ideal

`incomparable` 8 `(compare_m x y == Nothing) :: Set a => a -> a -> Bool`

`infixParse` (see `lexLots, infixParse` in 47) parses expression list:
```
:: (Eq a, Read a, Show a)=> ParenTable a-> OpTable a-> [Expression a]->
                                                ([Expression a], String)
```

`InfUnn` 8
```
data InfUnn a = Fin !a | Infinity | UnknownV
                                deriving (Eq, Show, Read)
```
domain `a` extended with the values `Infinity, Unknown`

`Integer` 26 `type Z = Integer` ring of integers,
supplied with the instances of `EuclideanRing, OrderedRing`, and some others

`intListToSymPol` 45.4.3
```
:: Ring a => Char-> SymPol a-> Partition-> [Partition] -> [Z] -> SymPol a
```

`:: Partition -> [Partition] -> (Map.Map Partition Z, [Z])`

`kostkaNumber` 45.2.2 mumber of tableaux of shape λ ...
`:: FiniteMap Partition Z -> Partition -> Partition ->`
`(Map.Map Partition Z, Z)`

`kostkaTMinor` 45.2.2 `:: [Partition] -> [Partition] -> [[Z]]`
transposed minor of matrix of Kostka numbers

`lc` 35 `:: (PolLike p, Set a) => p a -> a`
`\f -> = case pCoefs f of {a:_ -> a; _ -> error "lc 0 ..."}`
leading coefficient for non-zero polynomial-like thing

`lc0` 35 `:: (PolLike p, AddSemigroup (p a), Set a) => a -> p a -> a`
`\zr f -> if isZero f then zr else lc f`

`lcM` 16 `:: [a] -> a`
operation from `GCDRing` category: lcm for a list

`ldeg` 35 `:: CommutativeRing a => p a -> Z`
operation from constructor class `PolLike`:
total degree of `lpp` (depends on monomial ordering)

`leastEMon` 39.2.1 `:: CommutativeRing a => EPol a -> EMon a`
`\f -> case epolMons f of {[] -> error...; ms -> last ms}`

`leastMon` 38.3 `:: Set a => Pol a -> Mon a`
last monomial of a non-zero polynomial
`\f-> case polMons f of {[] -> error ...; ms -> last ms}`

`leastUPolMon` 36.3 `:: Set a => UPol a -> UMon a`
`\f -> case upolMons f of {[] -> error...; ms-> last ms}`

`LeftModule` 22.1 category Left Module over a Ring
`class (Ring r, AddGroup a) => LeftModule r a where`
`{cMul :: r -> a -> a, baseLeftModule :: ...}`

`essEq_m` 8 `(x==y || less_m x y) :: Set a => a -> a -> Bool`

`less_m` 8 `:: Set a => a -> a -> Bool`
`case compare_m x y of {Just LT -> True; _-> False}`

`lexComp` 37.2 `:: PPComp` lexicographic pp comparison

`lexFromEnd` 37 `:: PPComp`
compares power products lexicographically-from-end

`lexListComp` 8 `:: (a -> b -> CompValue) -> [a] -> [b] -> CompValue`

`mainMtDiag` 31.2 `:: [[a]] -> [a]` main diagonal of matrix

`Makefile` see 2.2, file `install.txt`.

`mapfmap` 8 `(\f -> map (fmap f)) :: Functor c => (a->b) -> [c a] -> [c b]`

`mapmap` 8 `map (map f) :: (a -> b) -> [[a]] -> [[b]]`

`mapMt` 31.2 `:: (a -> a) -> m a -> m a`,
an operation of the costructor class `MatrixLike`: map to each element in a matrix.

`matrHead` 31.2 `mtHead . mtRows`

`Matrix` 31.2 `data Matrix a = Mt [[a]] (Domains1 a)`
`Mt rows dm` must contain non-empty list of non-empty lists of same length

`MatrixLike` 31.2 `class MatrixLike m where mtRows .. mapMt .. transp`
This is for `m := Matrix, SquareMatrix`.

`MatrixSizes` 31.2 for `[[a]], Matrix, SquareMatrix`,

```
class MatrixSizes a where mtHeight :: a -> Natural
                          mtWidth  :: a -> Natural
```

`matrixDiscriminant` 36.3 `:: CommutativeRing a => Matrix a -> a`

```
matrixDiscriminant = discriminant_1 . charPol "lam"
```

— discriminant of a characteristic polynomial for a given square matrix (computed by a generic and direct method).

`maxAhead` 8 `:: Comparison a -> [a] -> [a]`
put ahead maximum without changing the order of the rest

`maxBy` 8 `:: Comparison a -> [a] -> a` maximum by given comparison

`maxHWBand` 45.2.1 `:: Char -> Partition -> Z -> Maybe HBand`
maximal h-w-band in a given partition

`maxMinor` 32.6 `:: Z -> Z -> [[a]] -> [[a]]`
pre-matrix obtained by deleting i-th row and j-th column
`\i j rows -> delColumn j (del_n_th i rows)`

`maxPartial` 8 like `minPartial`, only for maximum

`maybePair` 33.1 `:: Maybe a -> Maybe b -> Maybe (a,b)`
`{(Just x) (Just y) -> Just (x,y); _ _ -> Nothing}`

`mbPropV` 8 `:: Maybe PropValue -> PropValue`
`{Just v -> v; _-> Unknown}`

`membership` 10.2 `:: !(Char -> a -> Bool)` part of description of a subset S: algorithm for solving $x \in S$

`mEPolMul` 39.2.1 `:: Ring a => Mon a -> EPol a -> EPol a`
product of monomial by e-polynomial

`mEPVecPMul` 39.2.1 `\m (f,v) -> (mEPolMul m f, map (mPolMul m) v)`

`mergeBy` 8 `:: Comparison a -> [a] -> [a] -> [a]`
merge lists ordered by given comparison

`mergeE` 8 `:: Comparison a -> [a] -> [a] -> ([a], Char)`
extended merge: permutation sign `'+' | '-'` also accumulates

`minAhead` 8 `:: Comparison a -> [a] -> [a]`
puts ahead minimum without changing the order of the rest

`minBy` 8 `:: Comparison a -> [a] -> a` minimum by given comparison

`minPartial` 8 minimum by Partial ordering:
`:: Eq a => (a -> a -> Maybe CompValue) -> [a] -> Maybe a`
Returns either `Nothing` or `Just m` ($m \in xs$, $m \leq x$ for all $x \in xs$)

`minPrttOfWeight` 45.2.1 `{\0 -> []; \n -> [(1,n)]} :: Z -> Partition`
minimal partition for the given weight

`minRootOfNatural` 26.5

```
:: Natural -> Maybe (Natural, Natural)
   -- n              e        r
```

For `n > 1` finds `e` and `r` such that `n = r^e`,
and returns `Just (e, r)` with the minimal possible `e > 1`, if such exists.
It such does not exist, it returs `Nothing`.

`MMaybe` 8 `type MMaybe a = Maybe (Maybe a)`

`module` see `LeftModule`

`moduleConstrs` 22.2 `:: ![Construction_Submodule r a]`
data field in Submodule description

`moduleGenProps` 22.2 `:: !Properties_SubmoduleGens`
data field in submodule description: properties of generator list

`moduleGens` 22.2 `:: !(Maybe [a])` data field in `Submodule`
description for `sM :: Submodule r a`: generator list

`moduleOpers` 22.2 `:: !(Operations_Submodule r a)`

`mtRows` 31.2 `:: m a -> [[a]],` an operation of the class `MatrixLike`.

`mtTail` 31.2 `\ m -> case mtRows m of {_: rs -> rs, _ -> error...}`

`mtWidth` 31.2 an operation of the class `MatrixSizes(..)`

`mul` 12.4 `:: a -> a -> a`
multiplication operation from `MulSemigroup` category

`MulGroup` 14.1 category Multiplicative Group:
```
class MulMonoid a => MulGroup a where
      baseMulGroup :: a -> Domains1 a -> (Domains1 a, Subgroup a)
```

`MulMonoid` 13 category Multiplicative Monoid
`class MulSemigroup a => MulMonoid a`
Presumed is that `(unity _)` yields the true unity element

`MulSemigroup` 12.4 categogy Multiplicative Semigroup:
```
class Set a => MulSemigroup a where
    baseMulSemigroup...mul...unity_m...inv_m...divide_m...power_m...
```

`mulSign` 8 `( x y -> if x==y then '+' else '-') :: Char -> Char -> Char`

`Multiindex` 36.2 `type Multiindex i = [(i,i)]`

`multiplicity` 15.4 `:: CommutativeRing a => a -> a -> (Z,a)`
$(m,\ y/(x^m))$, $m =$ multiplicity of x in y in a factorial ring.
$x,\ y$ must be non-zero, x not invertible.

`mUPolMul` 36.3 `:: Ring a => UMon a -> UPol a -> UPol a` product by monomial

`neg` 12.3 `:: AddSemigroup a => a -> a`
$x \to -x$, composition of `neg_m` and `error`

`neg_m` 12.2 `:: a -> Maybe a`
operation of `AddSemigroup` category: opposite element

`nextPermut` 28.2 `:: [Z] -> Maybe [Z]`
next permutation by lexicographic order

`not3` 8 `:: PropValue -> PropValue`
`Yes -> No, No -> Yes, Unknown -> Unknown`

`num` 34.2 `(\ (n :/ _) -> n) :: Fraction a -> a` numerator of fraction

`numOfPVars` 38.3 `(genericLength . pVars) :: PolLike p => p a -> Z` `numOfStandardTableaux` 45.2.2, [Ma] `:: Partition -> Natural` number of standard tableaux

$= \mathit{weight}(\lambda)! \;/\; (\prod_{h \in \mathit{hooks}(\lambda)} \mathit{length}(h))$

The program tries to keep intermediate products possibly small.

`ofFiniteSet` 10.4 `:: Set a => a -> PropValue` "sample defines a finite base set"

```
\x -> case baseSet x Map.empty of (_,s) -> isFiniteSet s
```

`OpDescr` 47 `type OpDescr = (Z,Z,Z,Z)`

```
(left arity, right arity, left precedence, right precedence)
```

`operation` (3.4 CT) `Haskell` class operation.

DoCon also calls it a *category* operation.

Examples: `add, +, *, baseRing, fromExpr`

`Operations_Ideal` 21.3

```
type Operations_Ideal a = [(OpName_Ideal, Operation_Ideal a)]
```

`Operations_Submodule` 22.2 `type Operations_Submodule r a =`

```
[(OpName_Submodule, Operation_Submodule r a)]
```

`Operations_Subring` 15.2 part of `Subring` term

```
type Operations_Subring a = [(OpName_Subring, Operation_Subring a)]
```

`Operation_Ideal` 21.3

```
newtype Operation_Ideal a = IdealRank' (InfUnn Z) deriving (Eq, Show)
```

`Operation_Submodule` 22.2 `data Operation_Submodule r a =`

```
GradingM' !PPComp !(a -> PPComp) !(a -> [a])
```

`Operation_Subring` 15.2 some ring (field) attributes

```
data Operation_Subring a = WithPrimeField'
              {frobenius...dimOverPrime...primeFieldToZ ...}
```

`OpGroupDescr` 47 fixity & arities of operation

```
type OpGroupDescr a = (a, (Maybe OpDescr, Maybe OpDescr, Maybe OpDescr))
```

(0,r) — prefix & arity r (like $-x$), (l,0) — postfix ($n!$), (l,r) — infix ($x+y$)

`OpName_Ideal` 21.3

```
data OpName_Ideal = IdealRank deriving (Eq, Ord, Enum, Show)
```

`OpName_Submodule` 22.2

```
data OpName_Submodule = GradingM deriving (Eq, Ord, Enum, Show)
```

`OpName_Subring` 15.2

```
data OpName_Subring = WithPrimeField deriving (Eq,Ord,Enum,Show)
```

`OpTable` 47 `type OpTable a = [OpGroupDescr a]` to describe formats and arities of operations processed by `infixParse`

`OrderedAddGroup` 14.1 category Ordered Additive Group:

`class (AddGroup a, OrderedAddMonoid a) => OrderedAddGroup a`

Presumed: base `AddGroup` possesses `(IsOrderedSubgroup,Yes)`

`OrderedAddMonoid` 13 category Ordered Additive Monoid:

`class (OrderedAddSemigroup a, AddMonoid a) => OrderedAddMonoid a`

`OrderedAddSemigroup` 12.2 category Ordered Additive Semigroup:

`class (OrderedSet a, AddSemigroup a) => OrderedAddSemigroup a`

Presumed: base `AddSemigroup` possesses `(IsOrderedSubsemigroup,Yes)`

`OrderedField` 20 category Ordered Field:

`class (RealField a, OrderedRing a) => OrderedField a`

`OrderedMulGroup` 14.1 category Ordered Multiplicative Group:

`class (OrderedMulMonoid a,MulGroup a) => OrderedMulGroup a`

Presumed: base `MulGroup` is as an ordered group

`OrderedMulMonoid` 13 category Ordered Multiplicative Monoid:

`class (OrderedMulSemigroup a, MulMonoid a) => OrderedMulMonoid a`

`OrderedMulSemigroup` 13 category Ordered Multiplicative Semigroup:

`class MulSemigroup a => OrderedMulSemigroup a`

Presumed: `bS = base MulSemigroup` is an ordered subsemigroup

`OrderedRing` 16

`class (CommutativeRing a, OrderedAddGroup a) => OrderedRing a`

Presumed: base ring possesses `(IsOrderedRing, Yes)`

`OrderedSet` 11 `class (Ord a, Set a) => OrderedSet a`

Presumed: on base set, `compare_m` possesses `(OrderIsTotal,Yes)`
and agrees with `compare`: `(compare_m x y)==(Just (compare x y))`

`OrderIsArtin` 10.2 `:: Property_OSet`

"subset contains no infinite increasing sequence by `compare_m`"

`OrderIsNoether` 10.2 `:: Property_OSet`

"`compare_m` is Noetherian on this subset"

`OrderIsTotal` 10.2 `:: Property_OSet`

"`compare_m` never returns Nothing on this subset"

`OrderIsTrivial` 10.2 `:: Property_OSet` “`compare_m` coincides with `compareTrivially` on this subset”

`orderModuloNatural` 26.5

```
:: Natural -> Integer -> Natural
   -- r       a
```

`min [k > 0 | a^k = 1 (mod r)]` — for `r > 1`, `r` mutually prime with `a`.

`or3` 8 `:: PropValue -> PropValue -> PropValue`
`\Yes _ -> Yes, \_ Yes -> Yes, \No No -> No, \_ _ -> Unknown`

`OSet` 10.2 constructor for subset description

`osetBounds` 10.2 `:: !(MMaybe a, MMaybe a, MMaybe a, MMaybe a)`
data field in subset description

`osetCard` 10.2 `:: !(InfUnn Z)`
data field in subset description: cardinality

`osetConstrs` 10.2 `:: ![Construction_OSet a]` data field in subset term

`osetList` 10.2 `:: Maybe [a]`
data field in subset description: finite listing.
`Nothing` means the listing is unknown.

`osetPointed` 10.2 `:: !(MMaybe a)` data field in subset X :
chosen element in X. `Just (Just e)` means `e` chosen from X,
`Just Nothing` — X is empty, `Nothing` — `DoCon` cannot find element in X

`osetProps` 10.2 `:: Properties_OSet` data field in subset term

`osetSample` 10.2 `:: !a` data field in subset term: sample data for type

`Pair` 33 functor `(,)` of direct product.
for `a, b` ...`DoCon` provides `(a,b)` with the instances
of `Set, AddSemigroup, ..., LinSolvRing`

`pairNeighbours` 8 `:: [a] -> [(a,a)]`
Example: `[1,2,3,4,5] -> [(1,2),(3,4)]`

`ParenTable` see 47, `OpTab_.hs` `type ParenTable a = [(a,a)]`
describes all the allowed parentheses pairs for parsing.
A parenthesis may be any lexeme from `a`.

`parse` 3.14, 47
`smParse e` parses from string to domain element by a
sample `e`, `smParse` composes with `infixParse, fromExpr`

`Partition` 45.2.1 `type Partition = [(Z,Z)]`
A partition known from combinatorics. It is either `[]` or
$\lambda = [(j_1, m_1), \ldots, (j_k, m_k)], \quad j_1 > \ldots > j_k > 0, \quad (\text{or } (j_1^{m_1} \ldots j_k^{m_k}))$

`Partition module` see 50, 45.2 `DoCon` module exporting items for partitions

`partitionN` 8 `:: (a -> a -> Bool) -> [a] -> [[a]]`
breaks list into groups by the given equivalence relation.
For the equivalent items being *neighbours* apply better `groupBy`

`pCDiv` 35 `:: CommutativeRing a => p a -> a -> Maybe (p a)`
operation from `PolLike` : divide a pol-like thing by coefficient

`pCoef` 35 `:: CommutativeRing a => p a -> [Z] -> a`
operation of `PolLike` class: coefficient of given power product.
`c = pCoef f js` For `UPol`, `js = [j]` ...

`pCoefs` 35 `:: p a -> [a]` operation from `PolLike`:
coefficients listed in same order as monomials;
for `RPol`, the order is "depth first"

`pCont` 35 operation from `PolLike` class:
`(gcD . pCoefs) :: (GCDRing a, PolLike p) => p a -> a` content of pol-like data

`pDeriv` 35 `:: CommutativeRing a => Multiindex Z -> p a -> p a`
Operation from `PolLike` constructor class: derivative by multiindex.
Example: for $[x_1, x_2, x_3, x_4]$, `pDeriv [(2,3),(4,2)]` $\leftrightarrow$ $(d/dx_2)^3(d/dx_4)^2$

`pDivRem` 35 `:: CommutativeRing a => p a -> p a -> (p a, p a)`
Operation from `PolLike` constructor class: division with remainder.
For $k[x]$, k a field, it is Euclidean division.

`permGroupCharColumn` 45.2.3
`:: Partition -> [Partition] -> (Map.Map Partition Z, [Z])`
column of the character values obtained by `permGroupCharValue`

`permGroupCharTMinor` 45.2.3 `:: [Partition] -> [Partition] -> [[Z]]`
transposed minor of matrix of Irreducible Character Values $cha(w)(\lambda, \rho)$
for permutation group $S(w)$. See also `permGroupCharColumn`

`permGroupCharValue` 45.2.3 Irreducible character values for $S(n)$.
`:: Map.Map Partition Z -> Partition -> Partition -> (Map.Map Partition Z, Z)`

`Permut` see 50, 45

DoCon module exporting items for symmetric functions (sym-polynomials)

`Permutation` 28.1 `newtype Permutation = Pm [Z] deriving(Eq,Read)`
In `Pm xs` `xs` must be non-empty and free of repetitions.
The base domain of such sample is $S(length(n))$.

`permutCycles` 28.2 `:: Permutation -> [[Z]]` decomposes permutation to cycles
Example: `Pm [2,1,7,5,3] -> [[2,1],[7,3],[5]]`

`permutRepr` 28.2 `(\ Pm xs -> xs) :: Permutation -> [Z]`

`permutSign` 28.2 `:: Permutation -> Char` permutation sign: `'+'` or `'-'`

`pFreeCoef` 35 `:: CommutativeRing a => p a -> a`
operation from `PolLike` constructor class: free coefficient

`pFromVec` 35 `:: CommutativeRing a => p a -> [a] -> p a`
operation from `PolLike` constructor class (so far, only for `UPol`):
convert (dense) vector to pol-like data of the given sample.

`pHeadVar` 35 `:: (PolLike p, Set a) => p a -> PolVar`
`\f -> case pVars f of {v:_ -> v; _ -> error...}`

`PIR` 15.3 `:: Property_Subring` "is a principal ideal ring"

`PIRChinIdeal` (see 21.2, `eucIdeal`) Special ideal representation,
for a principle ideal ring, suitable for Chinese remainder method:
`...PIRChinIdeal {pirCIBase...pirCICover...pirCIOrtIdemps...}`

`pirCIBase` 21.2 `:: !a`
data field in ideal `iI :: PIRChinIdeal a` : base for `iI`

`pirCICover` 21.2 `:: ![a]`
data field in ideal `iI :: PIRChinIdeal a`. contains `bs`
such that `iI` = $\cap_{i\ldots} b_i$, $(b_i) + (b_j) = (1) \ldots$

`pirCIFactz` 21.2 `:: !(Factorization a)` factorization of a base:
a data field in ideal `iI :: PIRChinIdeal a`

`pirCIOrtIdemps` 21.2 `:: ![a]` data field in `iI :: PIRChinIdeal a`
Lagrange orthogonal idempotents e_i, $1 = \sum_{i=1}^{n} e_i$,
$e_i \in (b_i)$, $[b_1, \ldots, b_n]$ is decomposition of ideal base

`pIsConst` 35 `:: CommutativeRing a => p a -> Bool`
"is a constant pol-like thing": operation from `PolLike` constructor class

`pLexComp` 45.2.1 `:: PrttComp`
inverse lexicographical ordering on partitions

polSubst 38.3 Substitute polynomials for variables

`:: CommutativeRing a => Char -> Pol a -> [Pol a] -> [[Pol a]] -> Pol a`

`polToEPol` 39.2.1 `:: Z -> EPPOTerm -> Pol a -> EPol a`

embed pol to e-pol of given constant coordinate No i and epp ordering term:

`\ i o f -> EPol [(c, (i,p)) | (c,p) <- polMons f] o f`

`polToHomogForms` 38.3 homogeneous forms of polynomial

`:: (AddGroup a, Eq b) => (PowerProduct -> b) -> Pol a -> [Pol a]`

`PolVar` 36.1 type PolVar = String type of polynomial variable (indeterminate)

`polynomial` 38

`DoCon` represents a polynomial in `UPol, Pol, RPol` forms,
joining these kinds into the `PolLike` constructor class

`power` 12.4 `:: MulSemigroup a => a -> Z -> a`

`(^n)` as composed `power_m` and `error`

`PowerProduct` 37 `type PowerProduct = Vector Z`

Power product for polynomial. `DoCon` uses that its instance of an ordered additive group. Admissible comparison functions are related to it.

`power_m` 12.4 `:: a -> Z -> Maybe a`

operation `(^n)` from `MulSemigroup` category. Examples:

`(2::Z) 3 -> Just 8, 2 (-3) -> Nothing, (1:/2) (-3) -> Just (8:/1)`

`PPCoefRelationMode` 38.3

```
data PPCoefRelationMode = HeadPPRelatesCoef | TailPPRelatesCoef
                                                  deriving (...)
```

`PPComp` 37, 37.2 `type PPComp = Comparison PowerProduct`

Power product comparison. A part of `PPOrdTerm`,
and `PPOrdTerm` is a part of polynomial data (`Pol`)

`ppComplement` 37.2

`(\ u v -> (ppLcm u v)-u) :: PowerProduct -> PowerProduct -> PowerProduct`

`ppComp_blockwise` 37.2 compares power products
by the direct sum of two given comparisons:

`:: Z -> PPComp -> PPComp -> PowerProduct -> PowerProduct -> CompValue`

`ppDivides` 37.2 `:: PowerProduct -> PowerProduct -> Bool`

`\p q -> all (>= 0) (vecRepr (q-p))` "p divides q"

`ppLcm` 37.2 `:: Ord a => Vector a -> Vector a -> Vector a`

Lcm of power products. Example: `(Vec [1,0,2]) (Vec [0,1,3]) -> Vec [1,1,3]`

`ppMutPrime` 37.2 `::` `PowerProduct -> PowerProduct -> Bool`
“power products are mutually prime”

`ppoComp` 37.3 `= tuple32 ::` `PPOrdTerm -> PPComp` extracts pp comparison

`ppoId` 37.3 `= tuple31 ::` `PPOrdTerm -> PPOId` extracts identifier

`PPOId` 37.3 `type PPOId = (String, Z)`

`ppoWeights` 37.3 `= tuple33 ::` `PPOrdTerm -> [[Z]]` extract weights

`pPPO` 35 `::` `p a -> PPOrdTerm`
operation from `PolLike` constructor class: pp ordering description

`PPOrdTerm` 37.3 `type PPOrdTerm = (PPOId, PPComp, [[Z]])`
The power product ordering description (term) `(id, cp, ws)`
consists of identifier, pp comparison function, list of weights

`ppToPrtt` 45.2.1 `::` `PowerProduct -> Partition` power product to partition:
`filter ((/=0) . snd) . reverse . zip [1..] . vecRepr`

`prevHWBand` 45.2.1 `::` `Char -> Partition -> HBand -> Maybe HBand`
Previous (to the given) h-w-band. Ordering and mode are as in `maxHWBand`

`prevSWHook` 45.2.1 `::` `Partition -> SHook -> Maybe SHook`
Previous sw-hook to the given one. The ordering is so that
the greater is the hook which head starts higher.

`prevPrttOfN_lex` 45.2.1 `::` `Partition -> Maybe Partition`
Partition of weight `k = |pt|` previous to `pt` in
`pLexComp` order, `Nothing` for `[(1,k)]` or `[]`.

`primeFieldToRational` 15.2 `::` `a -> Fraction Z` Part of `WithPrimeField`
term of `Operation_Subring`. For characteristic $= 0$,
it must present the isomorphism Prime field $\to$ Rational numbers

`primeFieldToZ` 15.2 `::` `a -> Z` Part of `WithPrimeField`
term of `Operation_Subring`. For characteristic $= p > 0$, the
the restriction of `primeFieldToZ` to prime field must be inverse to `fromi` on `[0..p-1]`

`primitiveOverPrime` 15.2
`= (powers, mp, toPol) ::` `([a], [UMon a], a -> [UMon a])`
Part of `WithPrimeField` term of `Operation_Subring`.

`Primary` 21.3 `:: Property_Ideal` “is a primary ideal”

`Prime` 21.3 `:: Property_Ideal` “is a prime ideal”

`primes` 17 `::` `a -> [a]` operation from `FactorizationRing`:

infinite list of primes, free of repetitions,
built from given sample element

`product1` 8 `:: Num a => [a] -> a`
Product of a non-empty list of elements.
For a non-empty list, use `product1` rather than `product`

`Properties_EucRing` 19
`type Properties_EucRing = [(Property_EucRing, PropValue)]`

`Properties_FactrRing` 17
`type Properties_FactrRing = [(Property_FactrRing, PropValue)]`

`Properties_GCDRing` 16
`type Properties_GCDRing = [(Property_GCDRing, PropValue)]`

`Properties_LinSolvModule` 22.4
`type Properties_LinSolvModule = [(Property_LinSolvModule, PropValue)]`

`Properties_LinSolvRing` 18.4
`type Properties_LinSolvRing = [(Property_LinSolvRing, PropValue)]`

`Properties_OSet` 10.2 `type Properties_OSet = [(Property_OSet, PropValue)]`

`Properties_Submodule` 22.2
`type Properties_Submodule = [(Property_Submodule, PropValue)]`

`Properties_SubmoduleGens` 22.2
`type Properties_SubmoduleGens = [(Property_SubmoduleGens, PropValue)]`

`property` 3.5.13 An attribute stored in the
property list of domain description together with its value.
`DoCon` relates to each category a finite list of most valuable properties.

`Property_EucRing` 19
```
data Property_EucRing = Euclidean | DivRemCan | DivRemMin
                        deriving (Eq, Ord, Enum, Show)
```

`Property_FactrRing` 17
```
data Property_FactrRing = WithIsPrime | WithFactor | WithPrimeList
                          deriving (Eq, Ord, Enum, Show)
```

`Property_GCDRing` 16 `data Property_GCDRing =`
`WithCanAssoc | WithGCG deriving (Eq,Ord,Enum,Show)`

`Property_Ideal` 21.3 `data Property_Ideal =`
`IsMaxIdeal | Prime | Primary  deriving (Eq,Ord,Enum,Show)`

`prttLength` 45.2.1 `(sum . map snd) :: Partition -> Z`
$l(\lambda)$ = height of Young diagram = number of "actual variables"

`prttLessEq_natural` 45.2.1 `:: Partition -> Partition -> Bool`
natural partial ordering on partitions: $\lambda \leq \mu \iff$
for each $i > 0$ $sum_{j=1}^{i}\lambda_j \;\leq\; sum_{j=1}^{i}\mu_j$ (in expanded form)

`PrttParamMatrix` 45.4.3 `type PrttParamMatrix a = Map.Map Partition [a]`
partition-parameterized matrix over `a` (ptp-matrix)
is a table of pairs `(Partition, Row)`

`prttsOfW` 45.2.1 `:: Partition -> [Partition]`
all partitions of the same weight, starting from `pt`, listed
in `pLexComp` -decreasing order. `(pt -> [pt,...,minPt])`

`prttToPP` 45.2.1 `:: Z -> Partition -> PowerProduct`
converts partition to power product of given $length \geq l$
Example: $7\ [5^2, 4, 2^3] \rightarrow$ `Vec [0,3,0,1,2,0,0]`

`prttUnion` 45.2.1 `:: Partition -> Partition -> Partition` union,
with repeated diagram lines copied. Example: `[3,2,1] [3*2] -> [3*3,2,1]`

`prttWeight` 45.2.1 `:: Partition -> Z` weight of partition

`pTail` 35 `:: CommutativeRing a => p a -> p a`
operation from `PolLike` constructor class:
tail of a non-zero pol-like thing

`pToVec` 35 `:: CommutativeRing a => Z -> p a -> [a]`
Operation from `PolLike` constructor class (so far, only for `UPol`):
list of given length of coefficients of pol-like data (0 for gaps)

`ptpMatrRows` 45.4.3 `:: PrttParamMatrix a -> [[a]]`

`pValue` 35 `:: CommutativeRing a => p a -> [a] -> a`
operation from `PolLike` constructor class:
value of pol-like thing at $x_i = a_i$, extra a_i discarded

`pVars` 35 `:: p a -> [PolVar]`
operation from `PolLike` constructor class: list of variables

`PVecP` 38.3 `type PVecP a = (Pol a, [Pol a])`
Polynomial-vector-polynomial `(f,v)` is supplied with the polynomial
list `v`. `v` usually accumulates transformation coefficients ...

`quotEuc` 19 `:: EuclideanRing a => Char -> a -> a -> a`
`\ mode x -> fst . divRem mode x`

`randomEPrtts` 45.2.1 `:: [Z] -> Z -> [EPartition]`
infinite list of random expanded partitions of given weight w
produced out of infinite list of random integers $0 \leq n \leq w$

`rankFromIdeal` 21.3.1 `:: Ideal a -> InfUnn Z`
`\ iI -> case lookup IdealRank (idealOpers iI) of`
`{Just (IdealRank' v) -> v; _ -> UnknownV}`

`rank_euc` 32.6 `:: EuclideanRing a => [[a]] -> Z`
rank of matrix (Gauss method)

`read` 47

`RealField` see 20, notion `IsRealField`
`class Field a => RealField a` presumed: `(IsRealField,Yes)`

`reduceVec_euc` 32.1 reduces vector by vector list `us`:
`:: EuclideanRing a => Char -> [[a]] -> [a] -> ([a], [a])`
`us` is staircase. Several $a_i u_i$ subtract to make zeroes ...

`remEuc` 19 `:: EuclideanRing a => Char -> a -> a -> a`
`\ mode x -> snd . divRem mode x`

`removeFromAssocList` 8 `:: Eq a => [(a,b)] -> a -> [(a,b)]`

`reordEPol` 39.2.1 `:: EPPOTerm -> EPol a -> EPol a`
bring e-pol to given epp ordering

`reordPol` 38.3 `:: PPOrdTerm -> Pol a -> Pol a`
bring polynomial to given pp ordering

`reordSymPol` 45.3.2 `:: PrttComp -> SymPol a -> SymPol a`
bring sym-pol to given partition ordering

`resGDom` 41 `:: r a -> (Subgroup a, Domains1 a)`
operation from constructor class `Residue`. For a residue group G/H,
it extracts the subgroup term and domain bundle for H.

`resIdeal` 41 `(fst . resIDom) :: Residue r => r a -> Ideal a`
extracts ideal from `ResidueI` data

`resIDom` 41 `:: r a -> (Ideal a, Domains1 a)`
Operation from constructor class `Residue`. For a residue ring `a/I`,
it extracts the ideal term and domain bundle for `I`.

in order not to convert it from string nor to search in `ranges`

`rHeadVarPol` 40.3 map r-polynomial r from R to $R[u]$.
r is not constant, u is its leading variable.
`:: Set a => Domains1 (RPol a) -> RPol a -> UPol (RPol a)`

`Ring` 15 a category
`class (AddGroup a, MulSemigroup a, Num a, Fractional a) => Ring a where`
`{fromi_m :: a -> Z -> Maybe a; baseRing :: a -> Domains1 a ...}`
Presumed: `add, mul` obey the ring laws on base set ...

`RingModule` 50 `DoCon` module exporting items for `Ring, GCDRing` ...

`root` 12.4 `:: Z -> a -> MMaybe a` root of n-th degree.
Operation from `MulSemigroup` category. It may yield
`Just (Just r) | Just Nothing | Nothing,` `r` a root in given domain

`rootOfNatural` 26.5

```
:: Natural -> Natural -> Natural -> (Natural, Natural)
   -- e       x          b           r        r^e
```

Integer part `r` of the `e`-th degree root of `x` for `e > 0`.
The root is searched in the segment `[0, b]`.

`rowMatrMul` 31.2 `:: CommutativeRing a => [a] -> [[a]] -> [a]`
product of a row by a matrix

`RPol` 40.3, 40 type “r-polynomial”
`data RPol a = RPol !(RPol' a) !a !RPolVarsTerm !(Domains1 a)`
In `RPol rpol' a vterm aDom,` `a` is sample coefficient, ...

`rpolHeadVar` 40.3 extract head variable
`(rp'HeadVar . rpolRepr) :: RPol a -> Maybe RPolVar`

`rpolRepr` 40.3 `(\ RPol f _ _ _) -> f) :: RPol a -> RPol' a`
extract ‘representation’ of r-pol

`RPolVar` 40.3 `type RPolVar = [Z]` r-pol variable

`RPolVarComp` 40.3 `type RPolVarComp = Comparison RPolVar`

`rpolVars` 40.3 `:: Char -> RPol a -> [RPolVar]`
variables `v(i,j)` on which r-polynomial `f` really depends.
The result is ordered by comparison from `f`.

`RPolVarsTerm` 40.3, 40.2 `type RPolVarsTerm = (RPolVarComp, String, [(Z,Z)])`
variable ordering and variable set description for `[pref[`$i_1, \ldots, i_n$`]  | ...]`

`rpolVComp` 40.3 `(rvarsTermCp . rpolVTerm) :: RPol a -> RPolVarComp`
extract variable comparison function

`rpolVPrefix` 40.3 `(rvarsTermPref . rpolVTerm) :: RPol a -> String`
extract prefix for variables

`rpolVRanges` 40.3 `(rvarsTermRanges . rpolVTerm) :: RPol a -> [(Z,Z)]`
extract variable ranges

`rpolVTerm` 40.3 `(\ RPol _ _ t _) -> t) :: RPol a -> RPolVarsTerm`
extract variable set description

`RPol'` 40.3, 40 inner representation of r-polynomial:

```
data RPol' a = CRP !a | NRP !RPolVar !Z (RPol' a) (RPol' a)
                                         deriving (Eq, Show, Read)
```

`rp'HeadVar` 40.3 `:: RPol' a -> Maybe RPolVar` head variable
`{NRP v _ _ _-> Just v,  _ -> Nothing}`

`rp'Vars` 40.3 `:: Char -> RPol' a -> [RPolVar]`
Lists variables, listed first — into depth, then to the right,
repetitions cancelled. `mode = 'l'` means f is linear ...

`rvarsOfRanges` 40.3 `:: [(Z,Z)] -> [RPolVar]`
all r-vars in the given ranges listed in lexicographic-increasing order

`rvarsVolum` 40.3 `:: [(Z,Z)] -> Z`
number of r-polynomial variables defined by ranges

`Rse` data constructor for `ResidueE`

`Rsi` constructor for `ResidueI`

`rvarsTermCp` 40.3 `= tuple31 :: RPolVarsTerm -> RPolVarComp`

`rvarsTermPref` 40.3 `= tuple32 :: RPolVarsTerm -> String`

`rvarsTermRanges` 40.3 `= tuple33 :: RPolVarsTerm -> [(Z,Z)]`

`sample` 9.2 `:: c a -> a` sample element for argument domain
Operation from constructor class `Dom`. Example:
for a polynomial `f`, `sample f` is sample for coefficient

`scalarMt` 31.2 `:: [a] -> b -> b -> [[b]]` scalar pre-matrix
$n \times n$ made from given elements `c, z`.
Example: `scalarMt "xxx" 1 0 ->` integer unity matrix of size 3

`scalProduct` 29.2 `:: CommutativeRing a => [a] -> [a] -> a`
`(\ u v -> sum1 (zipWith (*) u v))`

`semigroup` see `Subsemigroup, AddSemigroup, MulSemigroup`

`separate` 8 `:: Eq a => a -> [a] -> [[a]]`
break list to lists separated by given separator
Example: `';' "ab;cd;;e f " -> ["ab", "cd", "", "e f "]`

`sEPol` 39.2.1 `sEPol f g` differs from `sPol`
in that it is in `Maybe` format, returns `Nothing` when the
leading coordinates of `f, g` differ, otherwise, `Just (s-epol,m1,m2)`

`sEPVecP` 39.2.1 similar to `sPol, sPVecP`
`:: GCDRing a => EPVecP a -> EPVecP a -> Maybe (EPVecP a, Mon a, Mon a)`

`Set` 10.1 category Partially Ordered Set:
`class (Eq a, Show a, DShow a) => Set a where`
`{showsDomOf...fromExpr...compare_m...baseSet...}`

`SetGroup` 50
`DoCon` module exporting items for `Set, AddGroup,` …— before `Ring`

`SHook` 45.2.1 `type SHook = (Z, Z, Z, Z)`
A *skew hook* known from combinatorics (see also `Partition`):
weight of hook, No of starting block, No of row in block …

`sHookHeight` 45.2.1 `:: Partition -> SHook -> Z`
`numberOfRowsOccupied - 1`

`showsDomOf` 10.1, 10.5 `:: Verbosity -> a -> String -> String`
Operation of the `Set` category used in error messages.

`showsList` 3.7 an operation of the class `DShow`,
`:: ShowOptions -> [a] -> String -> String`

`shown` 3.7 `:: DShow a => Verbosity -> [a] -> String`
`shown v a = showsn v a ""`

`showsn` 3.7 `:: DShow a => Verbosity -> [a] -> String -> String`

`showsPrtt` 45.2.1 `:: Partition -> String -> String`
Example: `[(5,2),(4,1),(2,3)] "" -> "[5*2,4,2*3]"`

`showWithPreNLIf` 3.7

`showsWithDom` see `source/auxil/Set_.hs`
prints element and its domain, with their given string names.

`showsWithPreNLIf` 3.7

`smParse` 3.14, 47 `:: Set a => a -> String -> a`

Generic parsing from string by the sample element,
a composition of `lexLots, infixParse, fromExpr`

`solveLinearTriangular` 32.5

```
:: CommutativeRing a => [[a]] -> [a] -> Maybe [a]
                        -- mA       row
```

solves a system `xRow x mA' = row`
for the upper-triangular matrix `mA'` obtanained from `mA` ...

`solveLinear_euc` 32.5 `:: EuclideanRing a => [[a]] -> [a] -> ([a], [[a]])`
solution of linear system `M` $\times$ `(transp [x]) = (transp [v])`

`sortE` 8 `:: Comparison a -> [a] -> ([a], Char)`
Extended `sort`: the sign `'+' | '-'` of permutation
is also accumulated. Method: 'merge' algorithm.

`sPol` 38.3 s-polynomial for non-zero f, g
`:: GCDRing a => Pol a -> Pol a -> (Pol a, Mon a, Mon a)`
(related to Gröbner basis). It is $m_1 \cdot f - m_2 \cdot g$ ...

`sPVecP` 38.3 s-polynomial for `PVecP a`
`:: GCDRing a => PVecP a -> PVecP a -> (PVecP a, Mon a, Mon a)`

`SquareMatrix` 31.2 `data SquareMatrix a = SqMt ![[a]] !(Domains1 a)`
The main difference to `Matrix` is that
the `MulSemigroup, Ring` instances are defined too.

`squareRootOfNatural` 26.5

```
:: Natural -> Natural -> Natural
   -- x       bound
```

Integer part of the *square* root of `x`. *Newton's method.*
It is faster than `rootOfNatural`. The root is searched in `[0, bound]`.

`sub` 12 `:: AddSemigroup a => a -> a -> a`
subtraction: composed `sub_m` and `error`

`subgrCanonic` 14.2 `:: !(Maybe (a -> a))`
data field in description of `Subgroup H in a`

`subgrConstrs` 14 `:: ![Construction_Subgroup a]`
data field in `Subgroup` description

`subgrGens` 14.2 `:: !(Maybe [a])`
data field in `Subgroup` description: generator list.
`Just gs` means `gs` is a finite list of subgroup generators ...

`subgrOpers` 14.2 `:: !(Operations_Subgroup a)`
data field in `Subgroup` description

`Subgroup` 14.2 `data Subgroup a =`
`Subgroup {subgrType...subgrGens...subgrCanonic ...}`
description of subgroup of base group on `a`

`subgrProps` 14.2 `:: Properties_Subgroup`
data field in `Subgroup` description

`Submodule` 22.2 `data Submodule r a =`
`Submodule {moduleRank ... moduleGens ... moduleProps ... }`
description of submodule in a module `a` over a ring `r`

`Subring` 15.2 `data Subring a =`
`Subring {subringChar ... subringGens ... subringProps ...}`
description of a subring of a base ring on `a`

`subringChar` 15.2 `:: !(Maybe Z)` characteristic of ring.
Data field in `Subring` description.
`Nothing` means it is unknown, `Just n:` `char = n` ≥ 0

`subringConstrs` 15.2 `:: ![Construction_Subring a]`
data field in `Subring` description

`subringGens` 15.2 `:: !(Maybe [a])` subring generator list.
Data field in `Subring` term: `Just gs` means `gs` is a finite generator list for subring

`subringOpers` 15.2 `:: !(Operations_Subring a)`
data field in `Subring` description

`subringProps` 15.3 `:: Properties_Subring`
data field in Subring description

`Subsemigroup` 12.1 `data Subsemigroup a =`
`Subsemigroup {subsmgType ... subsmgUnity ... subsmgGens ...}`
description of subsemigroup of the base semigroup on `a`

`subsmgConstrs` 12.1 `:: ![Construction_Subsemigroup a]`
data field in `Subsemigroup` description

`subsmgGens` 12.1 `:: !(Maybe [a])` list of generators.
Data field in `Subsemigroup` description. `Just gs` means
`gs` is a finite list of subsemigroup generators ...

`subsmgOpers` 12.1 `:: !(Operations_Subsemigroup a)`
data field in `Subsemigroup` description

`symmSumPol` 45.3.4 $symF = n! \cdot (symmetrize\ f)$,
`:: CommutativeRing a => PrttComp -> Pol a -> SymPol a`
$n =$ `length vars`, $symF$ under given partition ordering

`SymPol` 45.3.2 Symmetric polynomial:
`data SymPol a = SymPol ![SymMon a] !a !PrttComp !(Domains1 a)`
Mainly, it is a list of sym-monomials. It is similar to `Pol`, only
the variables skipped (presumed to be an infinite set $x_1, x_2, \dots$)

`symPolHomogForms` 45.3.2 `:: AddGroup a => SymPol a -> [SymPol a]`
homogeneous parts `hs` of `f`, empty for zero `f`,
otherwise, `h(1)` is the homogeneous form of `lm f`, ...

`symPolMons` 45.3.2
`(\(SymPol ms _ _ _) -> ms) :: SymPol a -> [SymMon a]`

`symPolPrttComp` 45.3.2
`(\(SymPol _ _ cp _) -> cp) :: SymPol a -> [SymMon a]`

`syzygyGens` 18.3 `:: String -> [a] -> [[a]]`
operation of `LinSolvRing` category: linear relation module
generators for given list. `mode = "g"` is for polynomial ring ...

`syzygyGensM` 22.3, 22.4, 49
`:: r -> String -> [a] -> [[r]]`
Operation from `LinSolvLModule r a` category:

`takeAsMuch` 8 `:: [a] -> [b] -> [b]`
`takeAsMuch xs ys == take |xs| ys`, only it is better implemented.

`test` see 6, file `install.txt`
`T_.test "log"` runs automatic test for DoCon,
copying the result to the file `./log`

`testFactorUPol_finField` `:: Field k => UPol k -> [Bool]`
Tests partially factorization for polynomial from $k[x]$,
k a finite field ...

`times` 12.3 `:: AddSemigroup a => a -> Z -> a`
product by integer: composed `times_m` and `error`

`times_m` 12.2 `:: a -> Z -> Maybe a`
multiplication by integer operation from `AddSemigroup` category

`toDiagMatr_euc` 32.4 Diagonal form `d` of matrix `m`

`:: EuclideanRing a => [[a]] -> [[a]] -> [[a]] -> ([[a]], [[a]], [[a]])`

`d` is obtained by elementary transformations of rows and columns ...

`toEPermut` 28.2 `(\ Pm xs -> zip (sort xs) xs) :: Permutation -> EPermut`

`toEPrtt` 45.2.1 `:: Partition -> EPartition`

`concat . map (\ (j,m) -> genericReplicate m j)`

`toOverHeadVar` 38.3 Bring polynomial to tail variables:

`:: CommutativeRing a => Domains1 (UPol a) -> Pol a -> Pol (UPol a)`

$a[x_1, x2, \ldots, x_n] \ \rightarrow \ (a[x_1])[x_2, \ldots, x_n]$

`toPolOverPol` 38.3

from `a[xs  ys]` to `a[xs][ys]` or to `a[ys][xs]`

`toRPol` 40.3 converts polynomial to r-polynomial

`:: CommutativeRing a => Char-> RPolVarsTerm-> [RPolVar]-> Pol a-> RPol a`

r-variable set description and variable correspondence are given

`toRPol'` 40.3 converts polynomial to r-pol'

`:: CommutativeRing a => Char -> [RPolVar] -> Pol a -> RPol' a`

a bijective variable correspondence is given

`toSqMt` 31 `(\Mt xs d -> SqMt xs) :: Matrix a -> SquareMatrix a`

a square matrix must be given

`toSquareFree` 16 `:: a -> Factorization a`

operation from the category `GCDRing`. Returns $[(a_1, 1), \ldots, (a_m, m)]$:
`canAssoc a =` $a_1^1 \ldots a_m^m$, a_i square free, ...

`toStairMatr_euc` 32.2 staircase form of matrix:

`:: EuclideanRing a => String -> [[a]] -> [[a]] -> ([[a]], [[a]], Char)`

Gauss method modification with repeated remainder division.

`toSymPol` 45.3.4 `:: Eq a => PrttComp -> Pol a -> Maybe (SymPol a)`

given a symmetric polynomial, form sym-polynomial by gathering each monomial orbit into sym-monomial. Yields `Just sF` for symmetric f

`totient` 26.5 `:: Natural -> Natural`

the number of the *totitive* numbers `k` for `n > 1`
(that is `0 < k < n` and `gcd n k = 1`).

`toUPol` 38.3 `:: Ring a => Pol a -> UPol a`

convert to univariate polynomial: remove pp-ordering term,
take the head variable only and head of each power product, ...

tuple<i><j>   8   selects the field No i
in an tuple of j elements, i, j = [3,4,5].
Example: tuple32 (_, x, _) = x

UMon   36.1   type UMon a = (a,Z)   univariate monomial

umonLcm   36.3   :: GCDRing a => UMon a -> UMon a -> UMon a
lcm of u-monomials over a gcd-ring.
\ (a,p) (b,q) -> case gcD[a,b] of g -> (a*(b/g), lcm p q)

underPPs   (38.5.3 {und})   :: [PowerProduct] -> (InfUnn Z, [PowerProduct])
A power product p is called 'under' power product with respect to
power product list pps if none of pps divides p ...

unfactor   17.1   :: MulSemigroup a => Factorization a -> a
example: $[(a,1),(b,2)]$ -> a b^2

unity   12.4   :: MulSemigroup a => a -> a
unity of semigroup given by sample:
composition of unity_m and error

unityFr   34.2   \x -> (unity x):/(unity x) :: Ring a => a -> Fraction a

unityIdeal   21.3.1   :: a -> Subring a -> Ideal a
unity ideal in a given base ring

unity_m   12.4   :: a -> Maybe a   operation from MulSemigroup category:
builds unity from the sample element. Relies on subsmgUnity

unparse   47   In unparsing, DoCon relies on the Show class

upAddGroup   23   :: AddGroup a => ADomDom a
\a -> fst . baseAddGroup a . upAddSemigroup a

upAddSemigroup   23   :: AddSemigroup a => ADomDom a
\a -> fst . baseAddSemigroup a . fst . baseSet a

updateProps   8   repeated propVOverList:
:: Eq a => [(a, PropValue)] -> [(a, PropValue)] -> [(a, PropValue)]

upEucFactrRing   23   :: (EuclideanRing a, FactorizationRing a) => ADomDom a

upEucRing   23   :: EuclideanRing a => ADomDom a

upFactorizationRing   23   :: FactorizationRing a => ADomDom a

upFactrLinSolvRing   23   :: (FactorizationRing a, LinSolvRing a) => ADomDom a

upField   23   :: Field a => ADomDom a

`upGCDLinSolvRing` 23 `:: (GCDRing a, LinSolvRing a) => ADomDom a`

`upGCDRing` 23 `:: GCDRing a => ADomDom a`

`upLinSolvRing` 23 `:: LinSolvRing a => ADomDom a`

`upMulGroup` 23 `:: MulGroup a => ADomDom a`

`upMulSemigroup` 23 `:: MulSemigroup a => ADomDom a`

`UPol` 36.1 Sparse univariate polynomial
  `data UPol a = UPol ![UMon a] !a !PolVar !(Domains1 a)`
  In `UPol mons c v aD`, `mons` are ordered decreasingly by `deg` ...

`upolInterpol` 36.3 Interpolates (rebuilds) polynomial
  $y = y(x)$ from table, $x, y \in$ `a`:
  `:: CommutativeRing a => UPol a -> [(a,a)] -> UPol a`

`upolMons` 36.3 `(\ UPol ms _ _ _ -> ms) :: UPol a -> [UMon a]`

`upolPseudoRem` 36.3 Pseudodivision in $R[x]$
  `:: CommutativeRing a => UPol a -> UPol a -> UPol a`
  The remainder is returned as described in ([Kn], Volum 2, section 4.6.1).

`upolSubst` 36.3 Substitutes g into f
  `:: CommutativeRing a => UPol a -> UPol a -> [UPol a] -> UPol a`
  Powers $[g^2, g^3, \ldots]$ either are given or computed by Horner scheme.

`upRing` 23 `:: Ring a => ADomDom a`

`up-function` (3.4, UP), 23
  function that builds a base bundle from initial bundle composing
  several base-operations. Example: `upRing (1,1) Map.empty -> (dD,rR)`

`vandermondeMt` 31.2 `:: MulMonoid a => [a] -> [[a]]`
  Vandermonde matrix: $[a_0, \ldots, a_n] \rightarrow [[a_i^j \ldots]| \ldots]]$

`varP` 35 (usable for univariate case)
  `:: (PolLike p, CommutativeRing a) => a -> p a -> p a`
  `\a f -> head . varPs a f`

`varPs` 35 `:: CommutativeRing a => a -> p a -> [p a]`
  operation from constructor class `PolLike`: convert variables from
  `f` multiplied by given non-zero coefficient to `p a`

`varToRPol` 40.3
  `:: AddSemigroup a => RPol a -> a -> RPolVar -> RPol a`
  makes r-pol $a \cdot v$ after given r-pol sample, coefficient, variable

`varToRPol'` 40.3 makes r-pol' $a \cdot v$
`:: AddSemigroup a => a -> RPolVar -> RPol' a`

`vecHead` 29.2 `:: Vector a -> a`
`\v -> case vecRepr v of {x:_ -> x;  _ -> error ...}`

`VecMatr` 50, 29, 31 `DoCon` module exporting items for vectors, maetrices

`vecMax` 37.2 `:: Ord a => Vector a -> Vector a -> Vector a`
`\v -> Vec . (zipWith max) (vecRepr v) . vecRepr`

`vecMutPrime` 37.2 `:: AddMonoid a => Vector a -> Vector a -> Bool`
`\v -> case zeroS $ vecHead v of`
`z -> and . (zipWith (\x y -> (x==z||y==z))) (vecRepr v) . vecRepr`

`VecPol` 39.1.1 `data VecPol a = VP ![Pol a] !EPPOTerm`
representation for `Vector (Pol a)`. It provides a field for
extended pp ordering. This makes a free module with grading.

`vecPolToEPol` 39.2.1 Inverse to `epolToVecPol`
`:: CommutativeRing a => EPPOTerm -> Vector (Pol a) -> EPol a`

`vecRepr` 29.2 `(\ Vec xs -> xs) :: Vector a -> [a]`

`vecSize` 29.2 `(genericLength . vecRepr) :: Vector a -> Z`

`vecTail` 29.2 `:: Vector a -> [a]`
`\v -> case vecRepr v of {_:xs -> xs;  _ -> error ...}`

`Vector` 29 Direct power of a domain
`newtype Vector a = Vec [a] deriving (Eq)` non-empty list required
`Vector a` denotes a parametric family of vector domains …

`WithCanAssoc` 16 `:: Property_GCDRing`
`(WithCanAssoc, Yes)` means `canAssoc, canInv`, are correct
algorithms for the canonical associated element, canonical invertible factor

`WithFactor` 17 `:: Property_FactrRing`
`(WithFactor, Yes)` means `factor` is correct factorization

`WithGCD` 16 `:: Property_GCDRing`
`(WithGCD, Yes)` means `(Factorial, Yes)` and
`gcD` a correct algorithm for greatest common divisor of a list

`WithIsPrime` 17 `:: Property_FactrRing`
`(WithIsPrime,Yes)` means `isPrime` is a correct primality test

`WithPrimeField` see 15.2, `Operation_Subring` name for one of items in Operations_Subring.

WithPrimeField' see 15.2, Operation_Subring
data constrtuctor for WithPrimeField

WithPrimeList 17 :: Property_FactrRing
(WithPrimeList, Yes) means (primes _) is correct method for list of primes

WithSyzygyGens 18.4 :: Property_LinSolvRing
(WithSyzygyGens, Yes) means (syzygyGens <anyMode>)
is a correct algorithm for the linear relation generators

Z 8 type Z = Integer ignore Int
DoCon treats this Haskell type as model of Ring of Integers,
supplies it with the instances EuclideanRing, OrderedRing ...

zeroEPol 39.2.1 (\t f -> EPol [] t f) :: EPPOTerm -> Pol a -> EPol a

zeroFr 34.2 (\x-> (zeroS x):/(unity x)) :: Ring a => a -> Fraction a

zeroIdeal 21.3.1 :: Properties_Ideal -> Subring a -> Ideal a
Zero ideal in non-zero base ring. Some ideal properties given in argument

zeroS 12.3 :: AddSemigroup a => a -> a
makes zero from sample: composed zero_m and error

zeroSubring 15.4 :: a -> Subring a -> Subring a
zero subring in a non-zero base ring

zero_m 12.2 :: a -> Maybe a makes zero from sample.
Operation from AddSemigroup category. Relies on subsmgUnity

zipRem 8 :: [a] -> [b] -> ([(a,b)], [a], [b])
version of zip that preserves remainders

Z module 50, 26
DoCon module exporting operations and instances for Z = Integer