Domain Constructor: an experience with categorical and functional programming in algebra.

A talk in RICAM, Linz, April 2006.

S.D.Mechveliani
Program Systems Institute, Pereslavl-Zalessky, Russia,
e-mail: mechvel@botik.ru

1 Introduction

Acknowledgements

This talk has been prepared in March – April 2006, during the Special Semester on Gröbner Bases, organized by RICAM, Austrian Academy of Sciences, Johannes Kepler University.

The author is grateful to the supporters of his travel to Linz and of his work in Linz:
Russian Foundation for Basic Research (grant 06-01-10519),
organizers and sponsors of the Special Gröbner Semester.

Our general goal and problem is

to create an adequate language and a tool (library) for programming and reasoning in mathematics (and in programming too).

When in 1996 – 2001 we were developing the Domain Constructor library, we persuaded mainly the local goal of providing ourself with some reasonable generic, elegant, and *open-source* tool for programming computer algebra.

Also our intention was (and is) to enjoy the very process of reading/understanding theory with further implementing of the methods.

Concerning the general goal, our idea is that an adequate tool should meet the following requirements.

1. A language should support categorical programming, domain construction,
   in particular, the ability to construct dynamically domains depending on parameters.
2. The core language for programming should be elegant, *functional*, have high-order functions.

   (2.1) It is good for this language to have the "lazy" computation, or/and to provide the means to reset a strategy of computation.
3. A *logical language* extension and a tool should be provided for processing knowledge about algorithms, about mathematical data.
4. The language extension and tool should support *reflection*.
5. A rich library of direct computational methods must be provided in the tool — computer algebra. This library should be written in the core language.
6. A library for automatic reasoning is needed,
   expressed in the logical language: completion, induction, strategies of combining proof methods, and such.
7. Desirably, it should be taken for the start some existing reliable implementation for a subset of the features **(1)** — **(6)**, the wider subset the better.

   And many people would like this tool to be *open-source*.

And there should not be a gap between the tools for computing and tools for "reasoning". Because a human thinking has not such a gap.

What may be a simplest "knowledge" about data?

For example: *equations*. They are good to use both for computing and for reasoning.

Why functional language?

Because such one allows to consider programs as explicit symbolic expressions, as something like polynomials, and to operate in the algebra of programs. This approach treats adequately the notions of a value of a program and of program equivalence.

Why 'lazy' language?

The 'Lazy' evaluation model helps to write clear programs.

Concerning the choice of a basic programming language, our experiments were in the following succession:

`C, Refal, Lisp, ML, Axiom, Haskell, Cayenne, Maude, Aldor`.

The experience with `Haskell` [HFP, Ha] was, so far, the longest one, its main part lasted from 1996 to 2001.

Instead of `Haskell` (`Miranda` family), we can imagine using any elegant functional language, with pattern matching, rich polymorphic type system, with classes and instances, — and of the "lazy" computation model.

Domain Constructor (`DoCon`) is the result of this experiment. It is a computer algebra program written in `Haskell`.

This open-source program, and its manual book, can be found in Internet by its name.

In the `Haskell + DoCon` system, the points **(3), (4), (6)** are absent, and the point **(1)** is satisfied in half.

But for what it remains in **(1)**, **(2)**, **(7)**, the `Glasgow Haskell` implementation [GH] provides a strong support.

The mentioned deficiencies are the matter of our next study, started in about 2003.

We invite the listener to recall of various programming languages and tools and to find out in what degree each tool satisfies the requirements **(1)** — **(7)**.

In `DoCon`, the category instances (domains) are expressed via the `Haskell` type classes — but also through explicit processing of the domain description terms. This latter correction is due to the deficiency in the point **(1)**, and this correction complicates the tool architecture and concept.

And complication is our enemy!

2 The current tool abilities, the library of methods

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 staircase and diagonal form of matrix, solving a 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).
- 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:
  `UPol`, `Pol`, `RPol` models.
- `Vector` module over a ring P, in some cases, with the Gröbner bases structure.
- `Residue` ring by an ideal:
  `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.

2.1 Examples of possible contribution

- Setting interface to some freely available advanced computational number theory library (maybe, by Lenstra).
  This will join many efficiently implemented methods, such as for 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 TRW, AC-completion, and so on.

Remark on (tr)

It can be based on conditional TRW, various versions of unfailing completion procedure, equational prover.

Currently (2006) the author is developing such a library (`Dumatel`) aiming to link it to `DoCon`. We have implemented the many-sorted (unconditional) TRW, completion, with extension to Boolean terms, proof search in the predicate logic via this completion, inductive prover, with a certain search strategy, and with heuristics for the lemma invention.

And there always remains a huge space for the contributions:
the order-sorted TRW paradigm, AC-completion, and so on.

Also there are possible various optimizations in `DoCon`.

Performance comparison

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

2.2 Short comments about `Haskell`

A commentary starts with `'--'`. We often use the construct

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

familiar to the Lisp programmers. We also use intensively the pattern matching mechanism for computation. Also

- identifiers of type parameters, data (functions in particular) begin with small letters;
- data and the type constructor names, as `Integer, Z, Pol`,
  begin 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.

2.3 `Haskell` categories (classes) and their instances

We illustrate them by the folloing example. This is how *could* be defined the classic categories in `Haskell`.

```
-----------------------------------------------------------------------------
class  AdditiveSemigroup a  where  0   :: a
                                   (+) :: a -> a -> a
-- but the very Laws are implicit!

class  MultiplicativeSemigroup a  where  1   :: a
                                         (*) :: a -> a -> a

class AdditiveSemigroup a => AdditiveGroup a  where  neg :: a -> a

class (AdditiveGroup a, MultiplicativeSemigroup a) => Ring a
  where
  char :: a -> Integer

class Ring a => CommutativeRing a

instance AdditiveSemigroup Integer where 0   = 0 :: Integer
                                         (+) = Prelude.(+)
...

data UPol a = UPol [(a, Natural)]
                             -- UPol (list of monomials ordered by degree ...),
                             -- a monomial is a pair ...

instance CommutativeRing a => AdditiveSemigroup (UPol a)           -- a functor
  where
  0 = UPol []

  (UPol mons) + (UPol mons') = UPol (sumMonLists mons mons')
                               where
                               sumMonLists mons mons' =  implement it ...

Example 1:  (UPol [(1, 2), (3, 5)]) + (UPol [(-1, 2)])     -- :: UPol Integer
            =
            ...

Example 2:  ( UPol  [ (UPol [...], 2), (UPol [...], 5) ] )   +   (...)
                                                       -- :: UPol (UPol Integer)
            =  ...
-----------------------------------------------------------------------------
```

As a first approach:

(algebraic) domain is represented as a set of instances.

2.4 Dynamic parameter domains and sample argument approach

But the `Haskell` instances are not enough to represent a domain depending on an ordinary value (parameter), when this parameter is recomputed at the run-time.

Therefore, `DoCon` represents a domain by adding a sample element.
In particular, it declares

```
class AddSemigroup a where  zero :: a -> a
                            add  :: a -> a -> a,
```

where `zero` is a pseudo-constant.
For example,

`zero (Residue _ 3)  :: Residue Integer  =  Residue 0 3` and
`zero (Residue _ 4)  :: Residue Integer  =  Residue 0 4`

are zeroes of different domains.
`Haskell` treats them as the data of the same type.
And the user program `DoCon` makes difference.

For a parametric domain, the type is partitioned into domains depending on the parameter values contained in any sample element.

This partition is recognized by `DoCon` (mainly, at the run-time), or by a user program — when this application program cares of this recognition.

3 Example 1. Polynomial GCD over a residue ring

Denote

`a` <--> any Euclidean ring A, `AX = A[x]` `<--> UPol a`,

`R = A/(m)`, as depending on `m <- A`. `R` `<--> ResidueE a`.

Denote `P = R[x]`.

<u>Problem</u>

For the given polynomials `f, g` from `A[x]` find GCD for their projections to `(A/(m))[x]` for each `m <- ms`, `m <- A`

— but first, DoCon decides (automatically) which of these domains are correct for GCD.

```
module Main
where
import qualified Data.Map as Map (empty)
import DExport

projectPol :: (EuclideanRing a, FactorizationRing a)
              =>
              a -> UPol a -> UPol (ResidueE a)
projectPol    m    f      =                    -- A[x] -> (A/(m))[x]
  let
    unA = unity m                    -- unity of  A
    dA  = upEucRing unA Map.empty    -- the domain  A
    iI  = eucIdeal "bef" m [] [] []
    r1  = Rse unA iI dA              -- 1 of R
    dR  = upField r1 Map.empty       -- the domain  R
    p1  = cToUPol "x" dR r1          -- 1  of  P = R[x]
  in
  ctr p1 [(ctr r1 a, e) | (a, e) <- upolMons f]
```

`ctr r a` projects `a <- A` to the domain of `r` (generally, this is casting between domains).

In our case, `ctr r1 a` casts to `A/(m)`,
and `ctr p1 monomialList` casts a monomial list to `P(m)` in a natural way.

```
gcdMods :: (EuclideanRing a, FactorizationRing a)
           =>
           UPol a -> UPol a -> [a] -> [UPol (ResidueE a)]
gcdMods    f         g         ms  =
                               [ gcD [projectPol m f, projectPol m g]  |  m <- ms ]



-- Example of using  gcdMods,  with  A = Integer  (<--> dZ) :

main =
 let
   unX = cToUPol "x" dZ (1 :: Integer)   :: UPol Integer
                              --
                              -- unity of AX, serves also as a sample

   f = smParse unX " (x^2 + 2*x + 3)*(3*x + 1) "
   g = smParse unX " (x^2 + 2*x + 3)*(x + 2)^3 "
                             --
                             -- parse value from string using unX as a
                             -- sample defining the destination domain
   ms = [2, 3, 5]  -- [6]

   gcs     = gcdMods f g ms
   lastRes = last gcs            -- look into the domain description
                                 -- of the last result, for curiosity
   (_, setP ) = baseSet  lastRes Map.empty     --
   (_, ringP) = baseRing lastRes Map.empty
 in
 putStr $ concat
 ["\n",
  show gcs,                  "\n\n",
  show (osetCard setP),      "\n\n",
  show (subringProps ringP), "\n\n"
 ]
```

For example, for `ms = [2, 3, 5]`, it prints out

```
[x^2 + 1,  x^2 + 2*x,  x^3 + 4*x^2 + 2*x + 1]
Infinity
[ (Field, No), (Factorial, Yes) ... ]
```

And for `ms = [6]`, it prints

```
baseGCDRing r dom',    r = 1  <- D/I =  (Z/<6>)

Prime iI needed
```

4 Example 2. Constructing a cubic Cardano extension.

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`, $\mathtt{a} \neq 0$, $\mathtt{char(k)} \neq 2, 3$.

`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, perform various computations in terms of `x, y, z` in `E`, such as, for example, testing the Viete relations. A more advanced experiment here might be expressing of `y` as a polynomial of `x` over `k(d)`.

```
import qualified Data.Map as Map (empty, lookup)
import DExport

type Q    = Fraction Z       -- rational numbers - for coefficients
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 a b dK =
  let
    [unK, n4, n27] = map (fromi a) [1, 4, 27] -- integer images in k

    unA   = cToUPol "d" dK unK             -- 1 of A = k[d]
    dA    = upEucRing unA Map.empty        -- domain A
    dv    = varP unK unA                   -- "d" as element of A
    discr = - n4*a^3 - n27*b^2

    dd = case  root 2 discr  of            -- dd  is the minimal equation
                                           -- for the root of discriminant

      Just (Just rad) -> dv - (ctr unA rad)          -- linear equation in "d"
      Just Nothing    -> dv^2 - (ctr unA discr)      -- regular case
      _               ->
                      error ("cubicExtension a b _ _:  could not find whether"
                             ++" the discriminant is a square in  k \n"
                            )
    ddI     = eucIdeal "be" dd [] [] [(dd, 1)]
    unK1    = Rse unA ddI dA
    dK1     = upField unK1 Map.empty   -- domain of  K1 = k(d)
    d       = ctr unK1 dv              -- d <- K1 satisfies the equation dd
    varsUVR = ["u", "v", "r"]          -- variables for the cubic radicals over K1

        -- To obtain more clear formulas for the roots x_i of f in E,
        -- we define certain special ordering on the power products
        -- of  u, v, r:  compare according to the weight list
        -- [[1,1,0], [1,0,0], [0,0,1]]:
        --
    uvrComp (Vec [u, v, r]) (Vec [u', v', r']) =
                              case  compare (u+v) (u'+v')
                              of
                              EQ -> lexListComp compare [u, r] [u', r']
                              v  -> v
```

```
    o   = (("deg2lex", 3), uvrComp, [])
    unB = cToPol o varsUVR dK1 unK1
    dB  = upGCDLinSolvRing unB Map.empty         -- domain B
    d'  = ct unB d                               -- d image in B
    [u', v', r']  = varPs unK1 unB
    [m2, m3, m27] = map (times unB) [2, 3, 27]   -- integer images in B
    (m3_2, m27_2) = (m3/m2, m27/m2)
    kToB          = ctr unB . ctr unK1 . ctr unA
    [a', b']      = map kToB [a, b]              -- cast a,b to B

    radicals = [r'^2 + m3,                       -- equations for  u, v, r
                u'^3 - m3_2*d'*r' + m27_2*b',
                v'^3 + m3_2*d'*r' + m27_2*b',
                u'*v'+ m3*a'
               ]                                 -- :: [B k]
    (gs, _)    = gxBasis radicals
    (dI, iRad) = gensToIdeal gs [] [(IsGxBasis, Yes)]
                                   [(IsMaxIdeal, Yes)] dB Map.empty
    unE  = Rsi unB (iRad, dI) dB
    kToE = ct unE . kToB
    dE   = upField unE Map.empty                 -- E = B/iRad = K1(u,v,r)
    [u, v, r] = map (ctr unE) [u', v', r']       -- as elements of E

             -- Finally, the roots  x1, x2, x3  of  f  are represented
             -- by Cardano formulas in the radicals  u, v, uv, r :

    [e3, e6] = map (times unE) [3, 6]
    roots    = [(u + v)/e3,
                (r*v - r*u - u - v)/e6,
                (r*u - r*v - u - v)/e6
               ]
  in
  (dE, roots, 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`.

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 supplied with all necessary description 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 structures 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`.

Example of usage

Construct the extension of the field `Rational = Q` with the roots of

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

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

```
main =
  let
    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
    Just gs            = idealGens $ resIdeal x
    --
    -- gs  is the Groebner basis by which the projection to E is done.
    --     We shall print it, and see  E : k1  from this.

    -- Example of calculation in E.

    discr  = - (4:/1)*a^3 - (27:/1)*b^2
    n1     = fromi x 1                        -- 1 in E
    fRoots = [x^3 - x + n1 | x <- roots]   -- it should be [0, 0, 0],  0 of E

    vieteValues = [x + y + z,  x*y + x*z + y*z,  x*y*z]  -- testing Viete relations:

    vtCheck = map kToE [0:/1, a, -b]               -- this should give [0, a, -b]

    -- Try to express  y  as polynomial in  x  over  k1.
    --
    -- 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.

    [x', y'] = map resRepr [x, y]
    Just hs  = idealGens (resIdeal x)
    o        = lexPPO 2
    rels     = algRelsPols hs ["y", "x"] o [y', x']
        --
        -- the generators of algebraic relations for  y', x'  viewed
        -- modulo Ideal(hs) in  k1[u,v,r],  the relations displayed
        -- in variables "y", "x"

    relCol = transp $ Mt [rels] Map.empty   -- to print matrix in a nice way
  in
  putStr $ concat
  ["\n",
   "discriminant =         ",    show discr,
   "\n",
```

```
  "isField E =            ",    show $ lookup IsField propsE,
  "\n",
  "all isZero fRoots =    ",    show $ all isZero fRoots,
  "\n",
  "vieteValues =          ",    show [x+y+z, x*y +x*z +y*z, x*y*z],
  "\n\n",
  "Viete relation test =  ",    show (vieteValues == vtCheck),
  "\n\n",
 "Relations over d for  y, x =\n",                      show relCol,
  "\n\n",
  "See the Grobner basis for  E  and  (E : k1) :\n",  shows gsCol "\n"
 ]
```

By the output of this function, we see that `rels` consists of the source cubic equation on `x` and of a non-trivial *square* expression of `y` via `x`:

```
y + ((3/23)*d)*x^2 + ((9/46)*d + 1/2)*x - (2/23)*d
```

The existence of degree 2 expression for this could be predicted by considering dimensions.

Because the Groebner basis `gs` is shown as

```
[r^2 + 3,
 v^2 + (1/2)*d*u*r + (9/2)*u,
 u^2 - (1/2)*d*v*r + (9/2)*v,
 u*v - 3
],
```

and we see from this that `E : k1 = 3`.

`f` is the minimal polynomial (of degree 3) for `x` over `k1`.

Hence, it suffice the degree 2 polynomials to generate `E` over `k1`.

References

[Al] S. M. Watt et al. *Aldor Compiler User Guide.*
IBM Thomas J. Watson Research Center. <http://www.aldor.org>

[Bu] B. Buchberger.
Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.
CAMP. Publ. No.83–29.0 November 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.

[GH] *Glasgow Haskell Compiler.* `<http://www.haskell.org/ghc>`.

[Ha] *Haskell 98: A Non-strict, Purely Functional Language.*
Report of February 1999. `<http://www.haskell.org>`

[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).

[Ma] I. G. Macdonald. *Symmetric Functions and Hall Polynomials.*
Clarendon Press, Oxford, 1979.

[Me3] S. D. Mechveliani.
Cost bound for LLL-Grigoryev method for factoring in GF(q)[x,y].
The Russian version is in the Journal of Fundamental and Applied Mathematics ("Fundamentalnaya i prikladnaya matematika"), 2002, Vol. 8, release 1, Mathematical department of Moscow State University. `<http://www.math.msu.su/~fmp/>`.
The file with English version is available at
`<http://www.botik.ru/pub/local/Mechveliani/otherPapers/ovfEng*>`

[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/>`