# Introduction to Gauss

D. Gruntz and M. Monagan
Institute for Scientific Computing, ETH Zürich
gruntz@inf.ethz.ch and monagan@inf.ethz.ch

## 1 Motivation

The Gauss package offers Maple users a new approach to programming based on the idea of parameterized types (domains) which is central to the AXIOM system. This approach to programming is now regarded by many as the right way to go in computer algebra systems design. In this article[1], we describe how Gauss is designed and show examples of usage. We end with some comments about how Gauss is being used in Maple.

But first, what is wrong with the programming models in existing systems? Back in the early days of computer algebra systems, in the mid to late 1960's, when people were designing ALTRAN, Macsyma and REDUCE, the focus of systems research was on computing with multivariate polynomials with integer coefficients. Why? Because the most common problems solved by computer algebra systems entail integer arithmetic and polynomial arithmetic with integer coefficients. But what if instead of integer coefficients, the input has complex coefficients? Or coefficients containing $\sqrt{6}$, $\log 4$, $\sqrt{a+1}$, $e^{-2\alpha}$, $\cos(\omega)$ ? How did these systems handle these more complicated coefficients? Did they work correctly?

Many approaches to system design have been tried. We will group them under four main strategies from the point of view of writing programs, namely, the "automatic simplifier" approach, the "case-by-case" approach, the "parameterized simplifier" approach, and the "parameterized domain" approach. We will illustrate the different approaches by considering how one would code a routine to compute the determinant of an $n$ by $n$ matrix $A$ using Gaussian elimination. In this discussion we are assuming that the input matrix entries are expressions of type $F$. For example, they might be simply rational numbers, but they could be polynomials with integer coefficients.

### 1.1 The Automatic Simplifier Approach

Maple, and the other computer algebra systems like it, including Macsyma, Mathematica, and REDUCE, all have a general representation for mathematical formulae called expressions. All these systems provide a builtin simplifier which is automatically applied to every expression created. For example, if you input $x-x$ in any of these systems, the expression $x-x$ is created and then simplified to 0 automatically. But, because simplification is expensive in general, not all simplifications are done automatically. For example, $\cos(x^2)+\sin(x)^2-1$ is not simplified to 0 in any of the systems

[1] An earlier version of this article appeared in issue 9 of the Maple Technical Newsletter, 1993.

automatically. And therein lies a problem. If the automatic simplifier fails to simplify an expression to zero, a potential for error exists. Let us show where things go wrong. In this approach to coding, one writes code in the most obvious way and assumes that the builtin arithmetic operators and the builtin simplifier will work. In Maple, our determinant routine would look like

```
Det := proc(A,n) local d,i,j,k,m;
    d := 1;
    for k to n do
        for i from k to n while A[i,k] = 0 do od;
        if i>n then RETURN( 0 ) fi;
        if i<>k then
            ... # interchange row i with row k
            d := - d
        fi;
        d := d * A[k,k];
        for i from k+1 to n do
            m := A[i,k] / A[k,k];
            for j from k+1 to n do A[i,j] := A[i,j] - m * A[k,j] od;
    od; od;
    d
end;
```

The algorithm is correct. And the system accepts the program as a valid program. Yet it can return incorrect answers. Why? What do the arithmetic operations `+`, `-`, `*`, `/` that appear in the code actually mean? The result of adding two expressions is a new expression which is simplified by the builtin simplifier. Now, notice the comparison of `A[i,k]` with 0. Clearly, if the simplifier fails to determine that the expression `A[i,k]` $= 0$, then because the algorithm later divides by this value (in the statement `m := A[i,k]/A[k,k]`), we will get a wrong answer. Thus the correctness of this code depends vitally on how good the builtin simplifier is.

One approach to this problem is to make the builtin automatic simplifier very powerful. However, this solution poses problems. Firstly, it is inefficient. The simplifier must first analyze the input expression before the real work can begin. Secondly, it may be difficult or impossible to extend it correctly to handle new kinds of expressions. Not that computer algebra systems don't try. Some have all kinds of options, flags, and rules, that allow you to tell the simplifier how to do things differently. But it is an unsatisfactory solution in that there is no protection from getting wrong answers. And you do get wrong answers. In fact, all of the systems will get wrong answers in the presence of either trigonometric functions or nested radicals. Even though there may be routines in the system that can properly simplify such expressions.

If we actually look at the system's code for computing determinants, we would see that none of the systems are using the "obvious" code above. In Maple, we find a combination of two distinct approaches, namely

## 1.2 The Case-by-case Approach

In this approach, one must write a separate routine **Fdet** to compute $\det(A)$ for each $F$. The routine **Fdet** makes use of the constants **Fzero** and **Fone**, and subroutines **Fadd**, **Fsub**, **Fmul**, **Fdiv**, **Finv**, **Fequal**, which add, subtract, multiply, divide, invert, and compare elements in $F$ respectively, to do arithmetic in $F$. In Maple code, **Fdet** would look like

```
FDet := proc(A,n) local d,i,j,k,m;
    d := Fone;
    for k to n do
        for i from k to n while Fequal(A[i,k],Fzero) do od;
        if i>n then RETURN( Fzero ) fi;
        if i<>k then
            ... # interchange row i with row k
            d := Fsub(Fzero,d)
        fi;
        d := Fmul(d,A[k,k]);
        for i from k+1 to n do
            m := Fdiv(A[i,k],A[k,k]);
            for j from k+1 to n do A[i,j] := Fsub(A[i,j],Fmul(m,A[k,j])) od;
    od; od;
    d
end;
```

This solution will be efficient because the routines have been written for a specific $F$. The disadvantage is that if one wants to compute in a new coefficient domain $K$, not only must one implement the basic arithmetic operations for $K$, but one must also re-implement all the polynomial and linear algebra routines over $K$. Since there are many different kinds of domains, this rapidly becomes undoable.

## 1.3 The Parameterized Simplifier Approach

As mentioned previously, all the systems have a general representation for formulae called expressions, and a builtin simplifier which performs certain simplifications automatically. But they also have other more powerful simplification routines. For example, in Maple there is **normal** which simplifies rational functions, **radsimp** for simplifying radicals, and **simplify**, a general purpose simplifier. There is also **testeq** which uses a probabilistic test for zero-equivalence based on the ideas of Schwartz [11]. See Monagan and Gonnet [10] for a description of the ideas behind this method.

The idea of the parameterized simplifier approach is to code the determinant routine to use a global SIMPLIFY function which can be assigned to a specific simplifier. For correctness, this function must recognize zero. The code in Maple would look like

```
Det := proc(A,n) local d,i,j,k,m;
    d := 1;
    for k to n do
        for i from k to n while SIMPLIFY(A[i,k]) = 0 do od;
        if i>n then RETURN(0) fi;
        if i<>k then
            ... # interchange row i with row k
            d := -d
        fi;
        d := d*A[k,k];
        for i from k+1 to n do
            m := SIMPLIFY(A[i,k]/A[k,k]);
            for j from k+1 to n do A[i,j] := SIMPLIFY(A[i,j]-m*A[k,j]) od;
    od; od;
    d
end:
```

This is an attractive solution because there only needs to be one determinant routine which can be relatively easily modified to work with different kinds of coefficients. Actually, some systems, Maple in particular, use a combination of strategies 2 and 3. For efficiency and algorithmic reasons, there are special implementations of the determinant algorithm for special fields. For example, in Maple there are special routines for the integers, the complex rationals (including the rationals), algebraic numbers, and polynomials over the integers.

This solution of providing a global SIMPLIFY routine addresses the issue of correctness. It also tries to address the issue of returning a "simplified" answer. In the latest version of Maple there are two global functions called `Normalizer` and `Testzero` which are both initially defined to use the Maple routine `normal`. The idea is that the `Testzero` routine provides a zero equivalence test and the `Normalizer` routine can be used to "simplify" expressions. This provides some more flexibility.

What is the problem with this approach? The first problem is that it is error prone. It still depends on the power of the simplifier function. If the simplifier is not able to recognize zero, wrong answers can be returned, with no indication that they may be wrong. The second problem is who writes the simplifier? Although it may be possible, by setting the various flags, and defining the right simplification rules, or simple coding, to create a correct simplifier, it is error prone, and does not inspire confidence in the user of the correctness of his programs. Another problem is the global nature of this approach. The user may be unaware of what a program needs to be able to do during an intermediate computation, and by changing the simplifier, break this intermediate calculation. So, we are led to

## 1.4 The Parameterized Domain Approach

The main idea used in the Gauss solution, pioneered by the AXIOM computer algebra system [7], (and other systems proposed in the literature [1, 3]), is that the second strategy is basically right. There needs to be separate functions for each arithmetic operation (also constants e.g. 0 and 1) for each type $F$ over which we want to compute. But instead of having a different determinant routine for each type, let's collect all the arithmetic operations for $F$ together into a unit, called a *domain*, and pass it as an argument to the determinant routine thus

```
Det := proc(A,n,F) local d,i,j,k,m;
    d := F[1];
    for k to n do
        for i from k to n while F[=]( A[i,k], F[0] ) do od;
        if i>n then RETURN(F[0]) fi;
        if i<>k then
            ... # interchange row i with row k
            d := F[-](d)
        fi;
        d := F[*](d,A[k,k]);
        for i from k+1 to n do
            m := F[/](A[i,k],A[k,k]);
            for j from k+1 to n do A[i,j] := F[-](A[i,j],F[*](m,A[k,j])) od;
    od; od;
    d
end;
```

The advantage of this approach is that this program will work for any $F$ and can be made as efficient as possible. The overhead is only the cost of looking up the domain $F$ for the routines used. Other linear algebra operations, and polynomial operations over $F$ are coded similarly. This is the first idea. Note, in practice, one may still want to include special implementations for special domains for efficiency reasons.

But who writes the code for the different $F$ domains? Given a domain $F$, it is possible to get the system to build new polynomial, and matrix domains over $F$ in the sense that the system itself will create the programs needed for polynomial and matrix arithmetic over $F$. This is the second idea. How are these domains implemented? How are these domains created? How does it all fit together? Does it all work?

The answer to the last question is unfortunately no. The problem is that we don't know how to test arbitrary formulae containing radicals, elementary functions and special functions for zero. One might expect a system like AXIOM to be better than systems like Maple, Macsyma and REDUCE here. But, as we shall see, it is really not. That is because AXIOM will use an expression model for objects when it does not have a normal form. For example, consider the following Maple session.

```
> A := matrix([[1-sin(x),cos(x)],[cos(x),1+sin(x)]]);

                         [ 1 - sin(x)    cos(x)   ]
                    A := [                        ]
                         [   cos(x)    1 + sin(x) ]

> det(A);

                                     2          2
                         1 - sin(x)   - cos(x)

> rank(A);
Error, (in linalg[gausselim]) matrix entries must be rational polynomials
```

Maple does not know that the determinant is zero and this does lead to bugs. We see that the matrix rank routine in Maple refuses to operate. The designers restricted that routine to a class of expressions which they knew could be handled correctly. Does AXIOM do any better? Dissappointingly, no.

```
(3) ->A := matrix([[1-sin(x),cos(x)],[cos(x),1+sin(x)]])

         +- sin(x) + 1     cos(x)  +
   (3)   |                         |
         +    cos(x)     sin(x) + 1+
                                              Type: Matrix Expression Integer
(4) ->determinant(A)

                  2         2
   (4)   - sin(x)  - cos(x)  + 1
                                                     Type: Expression Integer
(5) ->rank(A)

   (5)  2
                                                        Type: PositiveInteger
```

## 2 Sample Session

Let us begin with a sample session. This sample session is intended to give an idea of how Gauss is used and to give an overview of some domains available in Gauss. The Gauss package is available after executing the command `with(Gauss)`.

```
    |\^/|     Maple V Release 3 (Ren. Stud. Sublic ETH)
._|\|   |/|_. Copyright (c) 1981-1994 by Waterloo Maple Software and the
 \  MAPLE  /  University of Waterloo. All rights reserved. Maple and Maple V
 <____ ____>  are registered trademarks of Waterloo Maple Software.
      |       Type ? for help.

> with(Gauss);
----------------------- Gauss version 1.0 ----------------------
Initially defined domains are Z and Q the integers and rationals
Abbreviations, e.g. DUP for DenseUnivariatePolynomial, also made
                                  [init]
```

In Gauss, a domain is represented by a Maple table, whose entries are procedures and constants, which are accessed by Maples subscripts. For example, given a domain $D$, `D[Input]` will access the input procedure (which converts a Maple expression into the data representation used by $D$. Note that all Gauss functions begin with an upper case letter. Initially, the two domains `Z` (the integers) and `Q` (the rationals) have been defined. Let us do some operations with them:

```
> Z[Gcd](8,12);
                                   4

> Q['+'](1/2,1/3,1/4);
                                  13
                                  ----
                                  12
```

Next we want to perform some operations in $\mathbb{Q}[x]$. First we have to create the domain of computation for objects in $\mathbb{Q}[x]$. This is done by calling a domain constructor, in this case the constructor `DenseUnivariatePolynomial` (or `DUP` for short). A domain constructor in Gauss is a Maple procedure (with zero or more parameters), which returns a Gauss domain. The domain constructor `DUP` takes two parameters, the coefficient ring and the name of the variable. The name *DenseUnivariatePolynomial* indicates that the data structure being used is a dense one. We now generate the domain $\mathbb{Q}[x]$ and call it `P`. Then we input a polynomial which is returned in the internal representation (in this case a Maple list of coefficients), compute the degree and square it.

```
> P := DUP(Q,x):
> m := P[Input](x^4-10*x^2+1);
                                  m := [1, 0, -10, 0, 1]

> P[Degree](m);
                                            4

> P['^'](m,2);
                          [1, 0, -20, 0, 102, 0, -20, 0, 1]

> P[Output](");
                           8      6       4      2
                          x  - 20 x  + 102 x  - 20 x  + 1
```

In Gauss, every domain has an Input and Output operation. These two operations provide conversions from the user representation of an object to the internal data structure used by Gauss, and vice versa. The user representation of a Maple object is the Maple sum of products representation, which is a sparse expression tree representation. For efficiency, however, the implementation of dense univariate polynomials in Gauss uses the "obvious" data structure, a vector of coefficients. The most efficient data strucțure available in Maple for this purpose is the Maple list (which is not a linked list, it is a read only vector).

The Input operation may fail to convert a Maple expression into the internal data structure representation, but `P[Input]` o `P[Output]` never fails. Gauss is assuming then that there are two representations of objects, which usually are not the same, a "user level representation" based on expressions, and an internal representation. This is very similar to Jenks and Trager's plans for a designing a new user level language called $B^\natural$ [8] which would interface an expression model (like Maple's) with the core AXIOM library.

Gauss can also compute with matrices and other objects. Let us compute the determinant and inverse of the 2 by 2 generalized Hilbert matrix. This is the matrix of rational functions

$$\begin{bmatrix} \frac{1}{2-x} & \frac{1}{3-x} \\ \frac{1}{3-x} & \frac{1}{4-x} \end{bmatrix}$$

We must first define the domain of rational functions using the constructor `RationalFunction` then the matrix domain using the constructor `SquareMatrix`. These domain constructors have abbreviations `RF` and `SM` respectively. This

```
> R := RF(Q,x):
> M := SM(2,R):
```

```
> A := M[Input]([[1/(2-x), 1/(3-x)], [1/(3-x), 1/(4-x)]]);

 A := [[[[-1], [-2, 1]], [[-1], [-3, 1]]], [[[-1], [-3, 1]], [[-1], [-4, 1]]]]

> M[Output](A);

                          1        1          1        1
                    [[- -----, - -----], [- -----, - -----]]
                        x - 2    x - 3      x - 3    x - 4

> R[Output](M[Det](A)); # the determinant

                                        1
                          -------------------------------
                           4       3        2
                          x  - 12 x  + 53 x  - 102 x + 72

> M[Output](M[Inv](A)); # the inverse

                  3     2                     3     2
             [[- x  + 8 x  - 21 x + 18, x  - 9 x  + 26 x - 24],

                  3     2                     3      2
               [x  - 9 x  + 26 x - 24, - x  + 10 x  - 33 x + 36]]
```
Next we show some calculations with univariate power series over $\mathbb{Q}$. This domain is defined with the constructor **LazyUnivariatePowerSeries** (LUPS). First we compute the series for $\cos(x)$:

```
> PS := LUPS(Q,x):
> a := PS[Input](x);
> c := PS[Cos](a);
                                      2          4      6
                       c := 1 - 1/2 x  + 1/24 x  + O(x )
```
Lazy univariate power series are *lazy* in the sense that coefficients are computed on demand, i.e. we can compute a series to higher order without having to recompute any previously computed coefficients. This also means that we will not loose any information e.g. by differentiating.

```
> PS[Diff](c);
                                    3           5      6
                          - x + 1/6 x  - 1/120 x  + O(x )

> PS[Output](", 10);
                     3           5            7              9       11
            - x + 1/6 x  - 1/120 x  + 1/5040 x  - 1/362880 x  + O(x  )
```
The actual data structure used is hidden in these examples. The Maple 'print/' mechanism is being used by the LUPS domain. On output, it automatically converts the data structure to a user readable format – similar to AXIOM's coerce to Expression.

As a last example we compute the Legendre polynomials from their generating function

$$\frac{1}{\sqrt{1-2\,x\,t+t^2}} = \sum_{k=0}^{\infty} L_n(k)\,t^k.$$

This example is taken from [2], where also the idea of lazy power series is explained. A description of the Maple implementation can be found in [5].

```
  > P := LUPS(LUPS(Q, x), t):
  > s := P[Input](1-2*x*t+t^2);
                                                         2
                                      p := 1 - 2 x t + t

  > P['^'](s, -1/2);
                               2   2                       3   3
   1 + x t + (- 1/2 + 3/2 x ) t  + (- 3/2 x + 5/2 x ) t

                             2         4   4                        3         5   5        6
        + (3/8 - 15/4 x  + 35/8 x ) t  + (15/8 x - 35/4 x  + 63/8 x ) t  + O(t )
```

Other domain constructors available in Gauss are Zmod($n$ : *posint*), GaloisField($p$ : *prime*, $k$ : *posint*), Gaussian($R$ : *Ring*), QuotientField($D$ : *IntegralDomain*), etc.

## 3 Coding a simple Domain

In this section, we define a simple domain which implements operations over the set $T_X$ of all monomials in $X = \{x_1, x_2, \ldots, x_n\}$,

$$T_X = \{x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n} \quad | \quad e_i \in \mathbb{N}_0\}.$$

This domain could be used to define multivariate polynomials. The operations we will define are the multiplication, the comparison and the Gcd of two terms, and of course the procedures **Input** and **Output** to use it with Gauss. For the representation of a term $x_1^{e_1} \cdots x_n^{e_n}$ we use the list $[e_1, \ldots, e_n]$ of exponents, and hence we call that domain **ExponentVectorDomain**. It is parameterized by the list of the variables $[x_1, \ldots, x_n]$.

```
      ExponentVectorDomain := proc(X:list(name)) local D, env;
```

First we generate a new domain by calling the constructor **newDomain** and define its name. In Gauss this is essentially an empty Maple table.

```
          D := newDomain();
          D[DomainName] := ExponentVectorDomain;
```

We then add the signatures for the procedures we want to offer. In the signatures, the type D stands for the domain which is constructed by this domain constructor and **Expression** stands for a Maple expression. Since multiplying two terms is the same as adding their exponent vectors, we call the multiplication operation '+' in our domain.

```
          defOperation( '+', [D,D] &-> D, D );
          defOperation( '<', [D,D] &-> Boolean, D );
          defOperation( Gcd, [D,D] &-> D, D );
          defOperation( Variables, List(Name), D );
          defOperation( Input,  Expression &-> D, D );
          defOperation( Output, D &-> Expression, D );
```

Finally we must implement the defined operations. Since some operations must use local parameters of the domain constructor and since Maple does not support nested lexical scopes, we must simulate this behavior by substitution. For that we define the substitution pattern **env**. The implementation is straight forward. The order '<' is a pure lexicographical one.

```
          env := ['DD'=D];
          D['+'] := (x,y) -> zip((u,v) -> u+v, x, y);
          D[Gcd] := (x,y) -> zip(min, x, y);
          D['<'] := proc(x,y) local i;
             for i to nops(x)-1 while x[i]=y[i] do od;
             evalb(x[i]<y[i])
          end;
          D[Variables] := X:
          D[Input] := subs(env, proc(t) local x;
             [seq(degree(t,x), x=DD[Variables])]
          end);
          D[Output] := subs(env, proc(t)
             convert(zip((x,y) -> x^y, DD[Variables], t), '*')
          end);
          op(D)
       end:
```

This domain can now be used in Gauss as any other domain. For example, a domain $D$ of terms in $X = [u, v, w]$ is generated by the assignment

```
     V := ExponentVectorDomain([u,v,w]):
```

## 4 Categories and Domains

Let us go back to the determinant example in the first section. For which domains does the procedure **Det** work? What happens when we pass the integers as domain of computation? Of course, then the algorithm would fail, because the integers don't have a divide method. So, what we really need for computing the determinant is a field. To specify that a domain satisfies the axioms of an algebraic structure such as integral domain or field, every domain belongs to one or several *categories*.

A category defines an *abstract data type* $T$. It *specifies* a set of operations which must be provided by a particular implementation of $T$. This set includes operations to create and to manipulate objects and to obtain information from them. The category definition also specifies a set of properties which must be met by the defined operations, e.g. the commutativity of the addition in an Abelian group, etc. Furthermore, a category may supply default implementations of some operations, if they are expressible by other operations available in the same category. The only thing which is *not* specified by a category is the representation of the objects. In the terminology of object oriented programming a category is called an (abstract) class and the operations are called methods.

The instances (or implementations) of a category are the *domains*. A domain $D$ represents a (concrete) class of objects, i.e. it specifies the representation of the objects and implements the operations defined in the category. It may also supply further domain specific operations. Examples of domains we have already met are the integers and univariate polynomials over the rationals, which are both instances of the category *Ring*.

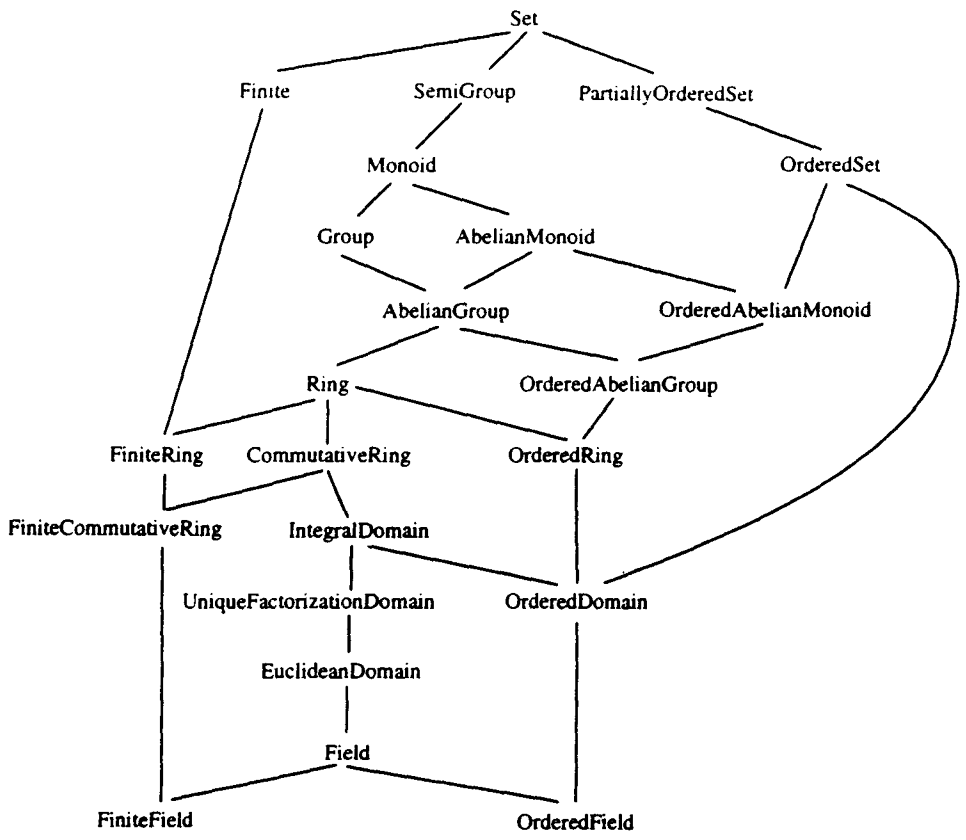

Figure 1: Categories in Gauss

Domains may be parameterized by other domains and objects. For example, a univariate polynomial domain is parameterized by its coefficient ring (a domain which must be a ring) and hence provides polynomial arithmetic and other polynomial operations for *all* possible coefficient rings. Every univariate polynomial domain is an instance of a category of univariate polynomials and offers the operations defined in that category.

A category may also *inherit* the definitions (and default implementations) of another category. For example, the category of univariate polynomials is also a ring and hence inherits all operation definitions and default implementations of the *Ring* category, as e.g. raising a polynomial to an integral power, which is per default implemented using binary powering. Gauss knows both multiple inheritance and conditional inheritance. For example, depending on the category type of the coefficient ring, a univariate polynomial domain may be an integral domain, a unique factorization domain or an Euclidean domain.

In Figure 1 you see the hierarchical world of the basic algebraic categories available in Gauss. The root of that hierarchy is the category **Set**, the class of algebraic sets. This is also the basic class to which every domain should belong and it supports the following basic operations:

|  |  |
|---|---|
| `=, <>` | boolean equality of domain elements |
| `Input` | for converting Maple expressions into the domain data representation |
| `Output` | for converting from the domain representation to an output form |
| `Random` | for generating a pseudo-random value from the domain |
| `Type` | for testing if a value is a valid domain element |

With this model it is possible to implement generic algorithms. That is, the algorithms may be written in their most general setting and they operate over all domains which belong to the category which is required. Another important point is that the algorithms work completely independent of the underlying data representation.

The elements of a domain are ordinary Maple objects and do not refer to their domain, in contrast to Axiom, where every object belongs to a domain. As a consequence, the domain must always be specified when calling generic functions. The **Type** method available in every domain allows for testing whether an object belongs to that domain. As a consequence, several domains may operate on the same objects. For example, the object 7 may belong both to the domain $\mathbb{Z}$ as well as to $\mathbb{Z}_{11}$, and no type coercions are necessary. The same model has been used in [6].

## 5 Coding in Gauss

Coding in gauss is quite straightforward, as indicated by the determinant example in the introduction, though a little cumbersome due to having to use a package call (in prefix notation) for every operation as you have seen in the examples. But since Maple hashing is very fast, this means that code access is fast too. The basic idea for writing code in Gauss for computing with elements of a domain is to pass the domain as an argument to the procedure, which is essentially passing a collection of routines for manipulating elements of the domain. E.g., let us write a routine to evaluate a univariate polynomial $a(x)$ at $x = b$. Our routine would look like this

```
Evaluate := proc(P,a,b) local R,k,d,r;
   if not hasCategory(P,UnivariatePolynomial) then ERROR(`...`) fi;
   R := P[CoefficientRing];
```

We pass the domain P as the first argument and check that it is a univariate polynomial domain. Since we need to do coefficient operations we assign the coefficient ring to the variable R. Next, we check the argument types of $a$ and $b$ as follows

```
   if not P[Type](a) then ERROR(`2nd argument must be of type P`) fi;
   if not R[Type](b) then ERROR(`3rd argument must be of type R`) fi;
```

Now we can do the polynomial evaluation using Horner's rule in the normal way. We need to use the **Degree** and **Coeff** functions from the univariate polynomial domain P, and the arithmetic operations `` `+` `` and `` `*` `` from the coefficient domain R.

```
   d := P[Degree](a);
   r := P[Coeff](a,d);
   for k from d-1 by -1 to 0 do r := R[`+`](R[`*`](r,b),P[Coeff](a,k)) od;
   r
end:
```

Let us test that procedure with a polynomial with matrix coefficients, which Maple cannot do directly:

```
> S := SM(2,Z):
> P := DUP(S,x):
> p := P[Input]([[1,0],[0,1]]*x^2 + [[-6,0],[0,-6]]*x + [[5,0],[0,5]]):
> q := S[Input]([[2,3],[1,4]]):
> Evaluate(P, p, q);
                              [[0, 0], [0, 0]]
```

which is what we expected, since $x^2 - 6\,x + 5$ is the characteristic polynomial of $\begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix}$.

## 6 Coding Categories and Domains

In this section we define a category (a category constructor is again a Maple procedure) which defines the operations on $T_X$, the set of exponent vectors $(e_1, e_2, \ldots, e_n)$ and hence we name the category **ExponentVector**. The main operations we have to specify are the addition of two exponent vectors, the degree function and the comparison of two terms. This category is parameterized by a list $X$ of names of the variables.

The category of the exponent vectors forms an ordered Abelian monoid. From that category the operations +, 0, <, <=, <>, >, >=, **Max**, **Min**, =, <> **Input**, **Output**, **Random** and **Type** are inherited. For the operations <=, <>, >, >=, **Max** and **Min** default implementations in terms of = and < are available. The category definition in Gauss has the following form. First an ordered abelian Monoid is generated. Then the new category name is inserted in the list of categories, and the signatures of additional operations (and constants) are defined.

```
ExponentVector := proc(X:list(name)) local D,n,r,env;
   D := OrderedAbelianMonoid();
   addCategory( D, ExponentVector );
   defOperation( Div, [D,D] &-> Union(D,FAIL), D );
   defOperation( '<>=', [D,D] &-> Union(-1,0,1), D );
   defOperation( TotalDegree, D &-> Integer, D );
   defOperation( Gcd, [D,D] &-> D, D );
   defOperation( Lcm, [D,D] &-> D, D );
   defOperation( Variables, List(Name), D );
   defOperation( Dim, Integer, D );
```

The operation <>= is used to define the ordering on terms which can either be lexicographical or total degree or any other ordering. The constant **Variables** keeps the names of the indeterminates and the constant **Dim** keeps the number of variables. The specification of the category is now complete. The only two operations which can be defined without knowing the representation are the two constants **Variables** and **Dim**.

```
   D[Variables] := X;
   D[Dim] := nops(X);
```

All other operations depend on the internal representation of the exponent vectors which is specified in the domains of that type and hence are not available at that point. But instead of delaying the implementation of the specified operations into the domains, we specify two additional operations **List2Vect** and **Vect2List**, which allow the transformations between the internal representation of a term $x_1^{e_1}\,x_2^{e_2}\cdots x_n^{e_n}$ (which is to be specified in the domain) and the list $[e_1, e_2, \ldots, e_n]$ of the $n$

integer exponents. With the help of this two operations, default methods for all the other operations (inherited from the *OrderedAbelianMonoid* or defined in this category) can be supplied.

```
        defOperation( List2Vect, List(Integer) &-> D, D );
        defOperation( Vect2List, D &-> List(Integer), D );
```

Below we show some default implementations based on these two new operations.

```
        env := ['DD' = D];
        D['+'] := subs(env, proc(x,y);
           DD[List2Vect](zip((u,v) -> u+v, DD[Vect2List](x), DD[Vect2List](y)))
        end);
        D['='] := subs(env, (x,y) -> evalb(DD[Vect2List](x)=DD[Vect2List](y)));
        D['<'] := subs(env, (x,y) -> evalb(DD['<>='](x,y)=-1));
        D[TotalDegree] := subs(env, proc(x) convert( DD[Vect2List](x), '+' ) end);
        D[Input] := subs(env, proc(t) local x;
           DD[List2Vect]([seq(degree(t,x), x=DD[Variables])])
        end);
        D[Output] := subs(env, proc(t);
           convert(zip((u,v) -> u^v, DD[Variables], DD[Vect2List](t)), '*')
        end);
        op(D)
     end:
```

An *ExponentVector* domain must only provide the methods List2Vect, Vect2List, <>=, the constant 0 (declared in the *Monoid* category) and the operation Type (declared in the *Set* category). For all other operations default methods are implemented in the category. These default methods may be overwritten if a more efficient implementation is available. Notice, that apart from the comparison method <>=, all other methods which must be implemented in a domain are related to the internal representation of the objects.

We show now the implementation of a minimal exponent vector domain which we call **MyExponentVector**. For the internal representation we use a Gödel coding, i.e. the term $x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n}$ is encoded by the integer $p_1^{e_1} p_2^{e_2} \cdots p_n^{e_n}$ (whereby the $p_i$'s are prime numbers). For the ordering of the terms we use the order induced by the natural order of the integers.

```
      MyExponentVector := proc(X) local D,P;
         D := ExponentVector(X);
         D[DomainName] := MyExponentVector;
```

With the call to the category constructor **ExponentVector**, this domain is asserted to be of type *ExponentVector* and the name of the domain is defined by an assignment. Then the representation specific implementations follow.

```
         P := [seq(ithprime(i), i=1..nops(X))]:
         env := ['DD'=D, 'PP'=P]:
         D[List2Vect] := subs(env, proc(l);
            convert(zip((x,y) -> x^y, PP, l), '*')
         end):
```

```
         D[Vect2List] := subs(env, proc(v) local n,l,p,q,i,j:
            n := v; l := NULL;
            for i to D[Dim] do p := PP[i];
               for j from 0 while irem(n,p,'q') = 0 do n := q od:
               l := l, j
            od: [l]
         end):
         D[0] := 1;
         D['<>='] := (x,y) -> if x=y then 0 else sign(x-y) fi;
         D[Type] := x -> type(x,posint);
```
Up to now, all defined operations are implemented, but for the chosen representation it is useful to overwrite some of the methods, because much faster implementations are available.

```
         D['+'] := (x,y) -> x*y: # integer multiplication
         D[Div] := proc(x,y) local q;
            if irem(x,y,q) <> 0 then FAIL else q fi
         end:
         D[Gcd] := igcd:
         D[Lcm] := ilcm:
         D['='] := (x,y) -> evalb(x=y):
         op(D)
      end:
```
In the following Gauss session we use our new exponent vector domain. We then generate two terms, compare them and compute their Gcd.

```
  > P := MyExponentVector([x,y,z]):
  > m1 := P[Input](x*y^3*z^3);
                                        m1 := 6750

  > m2 := P[Input](x^2*z);
                                         m2 := 20

  > P[Output](P[Gcd](m1,m2));
                                           x z
```
Further information about the design of Gauss can be found in [9]. Further examples of programing, including multivariate polynomials domains and routines for computing Grobner bases in Gauss, are given in [4].

## 7 Status Report

How efficient is Gauss?

Gauss code is written in Maple, thus interpreted. The overhead of Gauss, in comparison with Maple code, consists of a Maple (hash) table subscript and procedure call for every operation. The subscript is cheap, the procedure calling mechanism in Maple is relatively expensive. But in many cases, we can directly make use of builtin Maple functions. We find that Gauss runs typically not much slower than the Maple interpreter. For polynomial arithmetic, however, Gauss is much slower than Maple because the functions **expand** and **divide** are builtin. What we have done to get back the efficiency, is to make Maple polynomial domains in Gauss which use the Maple representation for polynomials, and hence also Maple's builtin operations. In some cases, Gauss has turned out to be much faster than Maple because no time is wasted analyzing what kind of expression was input.

In an interpreted language like Maple, there is always room for improvement by adding builtin data structures and compiled functions which execute at machine speed to speed up critical parts of the system. The lack of a dense array data structure, and builtin functions for the arithmetic operations $+$, $-$, $\times$, and $/$ contribute to an overall loss in efficiency.

What is it like to program in Gauss?

Students and colleagues have had surprisingly little difficulty. The inconvenience of having to package call each operation has not been a problem. Note, one does not need to package call every operation since Maple operations, e.g. integer operations in subscripts and loops, are done by Maple. In many ways, always package calling operations is just simpler. One observation that we have made is that this way of coding is less prone to error than Maple coding. Also annoying is not having lexical scoping in Maple, and having to simulate it using substitutions. Lexical scoping is scheduled to be added to Maple.

What is it like to "use" Gauss?

It's not nice at all. But then we don't really intend that Gauss be used at the interactive level in Maple. Rather we see it as a tool for implementing library functions. The user will always work in the Maple representation of objects. To improve the interface we have developed a Maple style *evaluator*

**evalgauss**[ $D$ ]( $E$ )

which evaluates a Maple user level expression $E$ in the domain $D$. The data in the Maple expression $E$ are converted into the domain $D$, the operations are executed there, and the result is converted back into a Maple user expression. For example

```
> P := DUP( DUP(Q,t), x ): # bivariate polynomials Q[x][t]
> m := x^4-10*x^2+1;

                               4       2
                         m := x  - 10 x  + 1

> evalgauss[P]( Resultant(1-t*Diff(m),m) );

                                4        2
                        147456 t  - 3840 t  + 1
```

This facility is limited. All operations that appear in the expression are understood to be computed in $P$, i.e. in $\mathbb{Q}[x][t]$. One must be careful when using this facility. If the above bivariate polynomials were defined in the other order i.e. $\mathbb{Q}[t][x]$, then the resultant and derivative would be computed in $t$ instead of $x$.

Where do we see Gauss being used?

Not at the top level, but rather as a tool to implement algorithms in the Maple library. The principle reason for using Gauss there is to avoid code duplication when working over different rings and fields. For example, we are using Gauss to implement multivariate factorization over GF($q$). A goal that we have is to have all routines in Maple's mod directory compute over GF($q$) in Gauss where the finite field is defined by an arbitrary number of field extensions. Maple expressions are automatically converted into the appropriate domain in Gauss where the computation takes place, and converted back.

**Acknowledgement**

Throughout the article the reader familiar with AXIOM (see [7]) will have noticed that many of the ideas and much of the terminology has been taken from AXIOM. We acknowldege the use of many of the ideas from AXIOM [7] in Gauss. We also wish to thank Jürg Bolliger and Laurent Bernardin for helping develop the Gauss package.

## References

[1] Abdali S.K., Cherry G.W. and Soiffer N., An Object Oriented Approach to Algebra System Design, *Proc. of ISSAC '86*, pp. 24-30, ACM Press, 1986.

[2] Burge W.H. and Watt S.M., Infinite Structures in Scratchpad II, *Proc. 1987 European Conference on Computer Algebra*, Leipzig, GDR, Springer-Verlag LNCS **378**, pp. 138-148, 1987.

[3] Foderaro J., *Newspeak*, Ph.D. Thesis, University of California at Berkeley, 1983.

[4] Gruntz D., Gröbner Bases in Gauss, *Maple Technical Newsletter*, Issue 9, pp. 36-46, Birkhäuser, 1993.

[5] Gruntz D., Infinite Structures in Maple, To appear in *Maple Technical Newsletter*, (**1**) 2, Birkhäuser, 1994.

[6] Gruntz, D. and Weck, W. A Generic Computer Algebra Library in Oberon, *Proc. of AISMC'94*, Springer-Verlag LNCS, to appear.

[7] Jenks R. and Sutor R., *axiom - The Scientific Computation System*, Springer-Verlag, 1992.

[8] Jenks R. and Trager B., How to Make AXIOM Into a Scratchpad, *Proc. of ISSAC '94*, pp. 32-40, ACM Press, 1994.

[9] Monagan M., Gauss: a Parameterized Domain of Computation System with Support for Signature Functions. *Proc. of DISCO '93*, Springer-Verlag LNCS, **722**, 81-94, 1993.

[10] Monagan M.B. and Gonnet G.H., Signature Functions for Algebraic Numbers, *Proc. of ISSAC '94*, pp. 291-296, ACM Press, 1994.

[11] Schwartz J.T. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, **27**, 701-717, 1980.