# Axioms, Categories and Domains

Klaus Drescher

18. October 1999

In order to make the construction of new algebraic structures easier the concept of *domain constructors* has been made available since *MuPAD* Version 1.2. With the aid of this concept parameterized domains can be constructed. Knowledge about common properties of certain classes of domains may be utilized in form of categories and generic algorithms. This version of the paper reflects the state of the *domain constructors* in *MuPAD* version 2.0.

# Contents

# 1. Introduction

In Version 1.2 the new basic type *domain* has been introduced in MuPAD. A domain can represent an algebraic structure like the rational functions $\mathbb{Q}(x)$ or an abstract data type like the data type `stack`.
What are such algebraic structures good for? In many computer algebra systems (CASs) elements of an algebraic structure are simply stored as expressions. Thus the expression

```
((x^2-1) / (x+1)) - x + 1
```

can be understood as a rational function in $\mathbb{Q}(x)$. The problem now is that this rational function has the value 0 but the internal simplifiers of most systems will not simplify this expression to 0 directly because this is too time consuming. The system does not "know" that this is an rational function over $\mathbb{Q}$ and therefore has to "try out" many possibilities ad hoc in order to find a normal form. In MuPAD, for instance, the command

```
normal(((x^2-1) / (x+1)) - x + 1);
```

has to be explicitly executed so that the system simplifies the expression.
How should a CAS treat such expressions if they are, for instance, entries in a matrix which has to be inverted? The entries have to be simplified, otherwise a division by 0 may occur during the Gaussian algorithm.

- One possibility is to call the `normal` function before each division. This can be very time-consuming in cases where simplification is not necessary. Furthermore this only works in those cases where `normal` produces a normal form. There are, however, many classes of expressions for which `normal` cannot produce a normal form (consider algebraic functions, the residue classes of polynomial rings, etc.).

- Another possibility is to determine the type of the matrix entries and for each type to implement a special inverting routine. These routines will all be based on the Gaussian algorithm, the difference being that they normalize differently. This, however, has the disadvantage that a lot of code is duplicated and once again only those types that have already been anticipated can be processed, matrices of other types cannot be inverted.

- The third possibility is to pass the simplification routine for the matrix entries as a parameter to the inverting routine. Then the existing inverting routine can also be used for new types by the implementation of suitable simplification routines. This is, however, not very user-friendly and is susceptible to errors (the user has to remember the correct simplification routine and constantly apply it).

It would be much easier if the datum `((x^2-1) / (x+1)) - x + 1` "knew" that it was a rational function and how it should react to operations such as `+` and `*`. From Version 1.2 on this was possible in MuPAD. The users can define their own types and operations on these types, create elements and combine them with system operators. If the operators always produce a normal form then the inverting routine also works with matrices that contain these elements without explicitly having to simplify them.
A new type is implemented as a domain. In section 2 of this paper domains and the implementation of types shall be briefly described.
A domain always represents a specific type, e.g., the algebraic extension $\mathbb{Q}(\sqrt{2})$. If other extensions, such as $\mathbb{Q}(\sqrt{7})$, are to be represented then a new domain will have to be created for each. However, it would be very time-consuming to implement each of these domains "by hand". It is desirable to be able to create domains for all possible extensions dependent on parameters.
Of course, such domains can be created using procedures that contain suitable parameters. However, this has certain disadvantages as shall become clear later. In MuPAD parameterized domains are created using *domain constructors*. These shall be described in section 3.
Domains often only differ slightly. Some operations are newly defined while other operations can be taken over from similar domains. This "inheriting" of domain operations can be explicitly done in the domain constructors, thus avoiding code duplication. With this, in MuPAD, the paradigms of *object oriented programming* are used.
Algebraic structures are divided into categories. The structures of a category have certain characteristics in common, statements may be made about all the structures in a category. Categories are a means of abstraction. Similarly, domains can also be divided into *categories*. Operations can often be formulated so that they are valid for all domains in a category (e.g., the calculation of the gcd in an Euclidean ring is always possible using the Euclidean algorithm). Thus, in MuPAD, the possibility has been created for dividing domains into categories and to "abstractly" formulate operations for these categories, so that they can be used for all domains of the category. Like domains, categories can also be dependent on parameters and are therefore created using *category constructors*.
The characteristics of a domain are postulated with the aid of *axioms*. Axioms are simply attributes of domains. They may for example be used to distinguish between different cases in algorithms. As with domains and categories, axioms can also be dependent on parameters.
The potential advantages of this concepts have contributed to the success of the computer algebra system AXIOM [7]. Maple is also moving towards a similar direction with the GAUSS [6] package (without, however, being able to offer a similar good integration into the rest of the system).
This paper is also aimed at those who "only" want to work with raw domains. In this case the user should concentrate on section 2 and on Appendix B.

## Changes since Version 1.4

Since Version 1.4 lexical scoping has been introduced in the MuPAD language, which caused lots of changes in procedures and also in domain constructors.

- The former function `domain` has been renamed to `newDomain`. `domain` is now a keyword starting the definition of a domain constructor.
- The grammar of the MuPAD language has been extended to include domain, category and axiom constructors, see section 3 and following.
- The special name `this`, representing the domain at hand, has been renamed into `dom`.
- The actual values for constructor arguments and local domain values are no longer substituted into the domain entries, but are rather bound lexically by the constructor.
- The actual domain a method belongs to (i.e., the value of `dom`) is stored in the methods procedure and is no longer substituted into the methods body.
- The function `domattr` has been removed in favor of a general concept for "slots". The function `slot` must now be used to access domain entries.
- The second operand of a "slot-expression " like `A::b` may no longer be a keyword. The entries `"name"` and `"not"` have been renamed to `"Name"` and `"_not"`.
- Some categories have been renamed, see `Cat`. (A trailing "Cat" has been removed, for example the former `Cat::SetCat` is now simply called `Cat::Set`.)

## Changes since Version 1.2.2

Most notably, all the predefined constructors have been inserted into three additional library domains, in order to avoid global names and naming conflicts:

- All domain constructors and domains have been inserted into the new library domain `Dom`.
- The category constructors and categories have been inserted into the library domain `Cat`.
- The axioms have been inserted into the library domain `Ax`.

Thus the domain constructor for matrices now is called `Dom::Matrix` instead of simply `Matrix`, and the category of rings is called `Cat::Ring` instead of `Ring`.

The former global names may be exported from these library domains, with `export(Dom)` one gets all the former domain constructor and domain names for example.
The library package `domains` is now predefined and need not be loaded explicitly with `loadlib`.
The method names of the category `Cat::FactorialDomain` (formerly `FactorialDomain`) have been changed slightly, which involves the sub-categories and domains of this category.
With Version 1.3 each domain must have a unique key, which is an arbitrary expression stored in the domain entry `"key"`. The key must be defined when creating a new domain with `domain`. Note that the domains created by domain constructors implicitly get a key which is also used to print them.

## Changes since Version 1.2.1

There have been some major changes in the implementation of the `domains` package between MuPAD versions 1.2.1 and 1.2.2. Most of them did not influence the usage of the package. The super-domains, categories and axioms of a domain were cached in the domain. (Formerly they were always created on the fly.) This speeded-up the queries with `hasProp` and the creation of entries. Changes that had to be obeyed when implementing constructors were:

- An entry definition which should not be inserted into the domain has to evaluate to `NIL` instead of `FAIL`, see 3.1 and 4.1.
- To refer directly to category entries one has to use the `::` operator instead of the index-operator `[ ]`, see section 3.6.
- The algorithm which is used for searching the categories for the definition of entries had been changed. The categories are now searched in a breadth-first manner and the first definition found is returned, see section 3.6.
- The methods `"getAxioms"` and `"allAxioms"` of the domain `Dom::BaseDomain` return sets instead of lists, see section 6.

# 2. Raw Domains

In this section the "direct" handling of "raw" domains will be shown without the aids of domain constructors or the new language extensions for domains which has been introduced since MuPAD Version 1.4. This section should only be glanced over, as the details can sometimes be somewhat discouraging. Many of this initial discouragement will be relieved later when the details are handled by the domain constructors; one may come back as necessary.
In MuPAD, domains represent algebraic structures and abstract data types. In the following, we shall use the residue class ring $\mathbb{Z}_7 = \mathbb{Z}/7\mathbb{Z}$ as an example. (In the `Dom` package there is the domain constructor `Dom::IntegerMod`, with which this domain can be created; more about this later.) With the built-in function `newDomain`, a new domain is created. The new domain has the type `DOM_DOMAIN`:

```
>> Z7 := newDomain("Zmod7")
```

```
                                Zmod7
```

```
>> domtype(Z7)
```

```
                             DOM_DOMAIN
```

The string `"Zmod7"` is a key, which must be given when a domain is created. Each domain must have a unique key, which may be an arbitrary expression. (The key is needed to decide if domains are equal.)
If `newDomain(k)` is called and a domain with key `k` already exists, the existing domain is returned. A new domain is created only if no domain with key `k` exists.
The key is used to print the domain, but this may be changed easily:

```
>> Z7::Name := hold(Z7):
   Z7
```

```
                                 Z7
```

There are some points that need explanation: With the slot operator `::` a new slot is created in the domain under the index `"Name"`. When the domain is to be printed, the system searches for this slot. If a slot exists under `"Name"` then the value of that slot is printed. If there is no slot `"Name"` then the key of the domain is printed.
The key of a domain is also stored in a slot in the domain. The name of the keys slot is `"key"`.
A domain may have an arbitrary number of slots, there is no predefined slot other than the `"key"` slot. Thus domains have "open-ended" slots. A slot of a domain is also called a *domain entry*.

The user should imagine a domain as a special kind of table. In principle, as in a table, the user can enter any value under any index in a domain. However, caution is advised, as the domain entries under certain indices (like, for instance, `"Name"`) have a special meaning and are interpreted by the system according to the application.
The first thing is to decide how the elements of $\mathbb{Z}_7$ are to be represented. For $\mathbb{Z}_7$ this is canonical: for a residue class the representatives are chosen from the interval [-3,...,3] (symmetrical representation).
New domain elements are created with the built-in function `new`, e.g., the element zero:

```
>> a := new(Z7, 0);
```

```
new(Z7, 0)
```

The identifier `a` now has an element of the domain `Z7` as its value. Here also the output shall be improved later.
With `new` a domain element, which can have any number of operands, is created. The domain of the elements is stored as the "zeroth operand":

```
>> op(a); op(a, 0)
```

```
0

Z7
```

Of course, the function `new` cannot know if the domain element created makes any sense. Therefore, for creating new domain elements, a special operation should be defined that ensures the creation of valid elements:

```
>> Z7::new := proc(x: DOM_INT) begin
       if args(0) <> 1 then error("wrong no of args") end_if;
       new(Z7, mods(x, 7))
   end_proc:
```

The operation `Z7::new` creates from an integer the residue class whose representative is this number. Once more the domain entry under the index `"new"` has a special meaning for the system:

```
>> a := Z7(0); b := Z7(1)
```

```
new(Z7, 0)

new(Z7, 1)
```

With the function call `Z7(n)`, the procedure stored in the slot `"new"` of the domain `Z7` is called. The procedure is passed the argument `n` as the actual parameter. The element zero is created with `Z7(0)`, the element one in $\mathbb{Z}_7$ with `Z7(1)`.
Domain entries that are procedures or functions are usually called *methods*. The output of the domain elements can be improved with the help of the method `"print"`:

```
>> Z7::print := proc(x) begin
      hold(Z7)(op(x, 1))
   end_proc:
```

This procedure returns an expression of the form `Z7(n)`. This expression is printed instead of the default one `new(Z7, n)`:

```
>> a, b
```

```
                              Z7(0), Z7(1)
```

However, this version of the `"print"` method contains a potential source of errors: the call of the `op` function. If the domain has a method `"op"` then this is called instead of the system function `op`:

```
>> Z7::op := proc(x) begin "nonsense" end_proc:
   op(a), a
```

```
                     "nonsense", Z7("nonsense")
```

One says that the function `op` is *overloaded* by the method `"op"`. By changing the method `"op"`, the call `op(x,...)` with a domain element `x` of `Z7` only returns `"nonsense"`, whereby also the `"print"` method is affected.
Therefore there is a special `op` function for domain elements that cannot be re-defined by overloading: the function `extop` (*ext*ension *op*). With this the `"print"` method looks like follows:

```
>> Z7::print := proc(x) begin
       hold(Z7)(extop(x, 1))
   end_proc:
```

Now, even if an `"op"` method is defined for `Z7`, the `"print"` method works as expected.
Apart from the function `op` the functions `nops` and `subsop` can also be overloaded for domain elements by defining appropriate methods. Thus, apart from `extop` there are also the functions `extnops` and `extsubsop`. These work similar to `nops` and `subsop`, except that they cannot be overloaded. However, the user should note that no ranges and operand paths can be given with the `ext...` functions, only single operand numbers.
So far, so good. But, what are domain elements used for? Until now they could only be created and printed. Well—exactly like the method `"op"` changed the system function `op` for domain elements, with appropriate methods many other functions can be overloaded for domain elements.
In MuPAD, addition is carried out by the built-in function `_plus`. By defining a method `"_plus"` for the domain `Z7` it gets its "own" addition that can be called with the "standard" `+` operator:

```
>> Z7::_plus := proc() begin
       Z7::new(_plus(map(args(), extop, 1)))
   end_proc:
```

Now elements of `Z7` can be added as usual:

```
>> Z7(1) + Z7(4)
```

```
                                Z7(-2)
```

Here, the operands of the domain elements (the representatives of the residue classes) are added together in `Z7::_plus`. With the result a new element of `Z7` is created.
With the implementation above care must be taken when adding different types:

```
>> Pref::typeCheck(Always):
   Z7(1) + Z7(4) + x

 Error: Wrong type of 1. argument (type 'DOM_INT' expected,
        got argument 'x - 2');
 during evaluation of 'Z7::new'
```

The `"_plus"` method is called with all operands of the sum, even if their types differ, and the methods of `Z7` doesn't know about identifiers like `x`. Types are never automatically converted because the system cannot know which type the user wants.
Note that in order to get the arguments of the method `"new"` checked one must first change the value of the preference `Pref::typeCheck` to `Always`. The default value `Interactive` of this preference causes type-checking only if the procedure is called "directly" (i.e., interactively) by the user.
Exactly like addition, multiplication can also be overloaded:

```
>> Z7::_mult := proc() begin
       Z7::new(_mult(map(args(), extop, 1)))
   end_proc:
```

Thus, the following expression is evaluated as expected:

```
>> Z7(2) * Z7(4) + Z7(1)

                                Z7(2)
```

The same holds for negation, subtraction, inversion and division, which may be defined by domain entries `"_negate"`, `"_subtract"`, `"_divide"` and `"_invert"`, respectively. Only the method `"_divide"` is shown here:

```
>> Z7::_divide := proc(x, y) begin
       Z7::new(mods(extop(x, 1) / extop(y, 1), 7))
   end_proc:
```

Here, the `mods` function calculates the modular inverse. The user can calculate with this in `Z7` as usual:

```
>> (Z7(2) * Z7(4) - Z7(3)) / Z7(5)

                                Z7(1)
```

Of course, other system functions, such as the function `_power`, can also be overloaded.

## 2.1. The Slot Operator

Given a domain and an index, the slot operator `::` returns the value of the slot stored under the index (similar to the index operator `[ ]` for tables). If there is no value stored under the index, then the operator raises an error:

```
>> Z7::foo

                                FAIL
```

Syntactically, the first operand of the slot operator has to be either an identifier, a variable or an arbitrary expression in *brackets*. Thus, `f::y` and `(f(x))::y` are valid expressions, whereas `f(x)::y` is invalid, because the first operand `f(x)` is not placed in brackets. The second operand of the `::` operator must have the same syntax as an identifier or variable, other expressions are not allowed. Furthermore another point should be noted:

> **Note:** The second operand of the `::` operator is converted into a string which is used as the index.

With the expression `D::_negate`, the name `_negate` is converted into the string `"_negate"` and used as index. (This can be seen when the entries of the domain `D` are inspected by using the function `op`.)
The reason for this seemingly crude rule is as follows: If the strings had to be written as usual with `"`, then the input would be very clumsy (apart from the fact that `D::"_negate"` would be not very nice to read).
As usual there is a functional equivalent to the slot operator. An expression of the form `dom::index` is equivalent to `slot(dom, "index")`. (In fact an expression of the form `dom::index` is internally converted immediately into a `slot` call by the parser.) When calling `slot` directly, arbitrary expressions may be used as indices.
The slot operator or a `slot` call may also be used on the left hand side of an assignment:

```
>> Z7::x := "foo":
   slot(Z7, "y") := "FOO":
   Z7::x, Z7::y

                              "foo", "FOO"
```

Currently only the basic domains `DOM_DOMAIN` and `DOM_FUNC_ENV` implement slots, but in principle the first argument of the slot operator can be of any type. One may define slots for new domains by overloading the `slot` function.
There is one special slot which is defined for any datum, this is the slot `"dom"`. The slot `"dom"` holds the domain of the datum:

```
>> XX::dom, (13)::dom, Z7::dom, (Z7(1))::dom

                 DOM_IDENT, DOM_INT, DOM_DOMAIN, Z7
```

The result of `D::dom` (or `slot(D, "dom")`) is the same as that of `domtype(D)`, but `D::dom` is in some occasions easier to read than `domtype(D)`. (Compare for example `D::dom::entry` with `(domtype(D))::entry`.)
In connection with the slot operator there is a stumbling block that the user may fall over while programming. Suppose you want to write a program which uses a slot name as a parameter, like this:

```
gotcha := proc(DOM, ind) begin DOM::ind end_proc:
```

This procedure always returns the domain entry with the index `"ind"`! The reason for this can be easily seen: the expression `D::ind` is equivalent to the expression `slot(D, "ind")`, and the string `"ind"` is not evaluated to the value of the parameter `ind`. This trap can be avoided by calling `slot` directly. The following procedure returns the desired result:

```
gotcha := proc(DOM, ind) begin slot(DOM, ind) end_proc:
```

A tip for domain experts: Slots for domains can be created on demand by defining a `"make_slot"` method. The significance is as follows: If `slot`, given an index, finds a value in a domain, then this is always returned. If no value exists `slot` searches for a method with index `"make_slot"` in the domain. If no such method exists then `slot` returns the value `FAIL`. If a `"make_slot"` method exists then this is called with the domain and the index as arguments. The value returned by the method is entered in the domain as the value of the original index and returned by `slot`.
By using this method entries can be created in a domain when necessary—when they are accessed for the first time. This fact is used during the creation of domains by domain constructors. Here only the necessary entries are created for a domain on the fly. (Often one creates many domains but uses only a few of their entries, so no memory space is wasted.)
Note that even if the `"make_slot"` method returns `FAIL`, this value is explicitly entered into the domain. This has the advantage that the time consuming creation need only be carried out once even when a value is missing. Thus a `"make_slot"` method will *not* add a new value under an index once `FAIL` has been inserted.

## 2.2. Overloading Built-In Functions

As a convention a built-in function (sometimes also called kernel function) is overloaded by a method which has the string with the name of the function as index. If a built-in function is evaluated, and one of the arguments is a domain element, then instead of the built-in function the corresponding method of the domain is evaluated and the result returned. In general the method is called with the actual parameters of the overloaded function. Examples for the overloading of the built-in functions `op`, `_plus` and `_mult` have already been shown above.
However this statement is so greatly simplified that it could nearly be called naive. On the one hand, basic types are also domain elements. An integer is, for instance, an element of the domain `DOM_INT`, as a call of the function

`domtype` shows. The processing of this "basic domains" by the built-in functions can usually not be overloaded. E.g., the addition of integers cannot be changed by defining a method `"_plus"` for the domain `DOM_INT`. This is inconsistent, unfortunately, however the overloading of these types would simply be too time consuming. (Exceptions to this rule are some methods that shall be described later.)
On the other hand, not all built-in functions can be overloaded and in general for those that can be overloaded not all arguments can be used for overloading. (I.e., only certain arguments result in the method of the corresponding domain being called.) It has already been mentioned that `extop` cannot be overloaded. Also `new` cannot be overloaded. In the function `map` only the first argument can be overloaded. Informations about the overloadable arguments of the different functions can be found on the help pages of the MuPAD manual.
One important function that is *not* overloadable is `_equal`. With `_equal` only the equality of expressions as data structures can be tested for, not the *logical* equality of expressions.
If a method is called via overloading then it is normally given the same arguments as the overloaded function. The arguments are evaluated before the method is called. The option `hold` has no effect in a method! Exceptions are certain methods that do not "canonically" overload functions, see the appendix B for examples.
What happens in a built-in function when a domain element is given as the argument, but the domain does not contain an appropriate method? This depends on the circumstances. Sometimes an error message is given, but if possible the function is executed further "as if nothing had happened". If no method exists but further arguments exist that could effect an overloading then the appropriate method is searched for in the domains of these arguments.
Usually, when overloading, the arguments of the built-in functions are not tested before the call of the method. The word "usually" indicates that there are exceptions to this. Some built-in functions do test their arguments before the method is called.
For the user, a method is easier to understand if it expects the same arguments as the corresponding built-in function. Finally a method should have a meaning analogous to the built-in function and not have a totally different meaning.

## 2.3. Overloading Procedures

There is no automatism that makes procedures overloadable. Each procedure—as reasonably as possible—should guarantee by itself that it may be overloaded. For instance, in the library procedure `normal` the first argument can be overloaded.
An overloading of the first argument of the procedure `gotcha` can, for example, be allowed as follows:

```
gotcha := proc(x, n)
```

```
begin
    // allow for overloading
    if x::dom::gotcha <> FAIL then
        return(x::dom::gotcha(args()))
    end_if;

    // carry on as 'usual'
    ...
end_proc:
```

It is tested if `x` is an element of a domain which contains the method `"gotcha"`. If a method exists then it is called with the original arguments and the result is returned.

## 2.4. Special Domain Entries

There are some domain entries and methods that do not canonically overload system functions. The entry `"Name"` and the method `"print"` have already been mentioned above. (The `"print"` method is called without the system function `print` being explicitly called.) These special entries and methods are briefly described in appendix B.

## 2.5. The Reference Effect

Domains are the only data structures in MuPAD that show the so-called *reference effect*:

> **Note:** When a domain is assigned to an identifier or a variable (either by the assignment operator `:=` or to the formal parameter of a procedure) then the contents of the domain is *not* copied. Only a reference to the contents is created. Changes to the assigned domain also change the original one.

The reference effect is demonstrated in the following example:

```
>> A := newDomain("A"):
   A::x := 1

                                  1

>> B := A:
   B::x := 13

                                 13

>> A::x

                                 13
```

The assignment `B::x := 13` has changed the original entry `A::x`!
The reference effect is necessary in connection with "generic algorithms". These are methods that may contain the domain they are entries of. If only a copy of

the contents of the domain could be inserted into such a "generic method", it could not call itself recursively for example, because it would not yet be contained in the domain it contains at the time it is created.
Take for example the following—not very meaningful—function `foo`:

```
foo := proc(x) begin
    if iszero(x) then 0 else foo(x-1)+x end_if
end_proc;
```

Now assume that a method similar to `"foo"` should be defined for a domain which is not known beforehand, but which is to be created at runtime. This could for example be done by:

```
make_foo := proc(DOM)
    option escape;
begin
    proc(x) begin
        if iszero(x) then 0 else DOM::foo(x-1)+x end_if
    end_proc
end_proc;
d := newDomain("spoof");
d::foo := make_foo(d);
```

The domain is defined via the lexical enclosing procedure `make_foo`, because later on the domain will no longer be stored as value of `d` and no other name is known now. If the actual contents of `d` would now be inserted into the procedure returned by `make_foo`, then this domain would not yet contain `d::foo` itself. Later on, when `d::foo` would be called, a runtime error would occur.
If a domain is to be copied then the function `newDomain` can be used:

```
>> A := newDomain("A"):
   A::x := 1

                                  1

>> B := newDomain("B", A):
   B::x := 13

                                 13

>> A::x

                                  1
```

By using `newDomain("B", A)` a copy of the domain `A` is created. Thus a change in `B` no longer affects `A`. The copy has the key `"B"`, which must be a new key (no domain with this key may exist beforehand).

# 3. Domain Constructors

Of course, it is not very efficient to construct a single algebraic structure, such as $\mathbb{Z}_7$ as a domain "by hand". It makes more sense to define a structure such as $\mathbb{Z}_n$, in order to obtain—dependent on the "parameter" $n$—all possible residue class rings over $\mathbb{Z}$. In MuPAD language: parameterized domains are to be created, so-called *domain constructors* are used for creating parameterized domains.
In the `Dom` package there is, for instance, a domain constructor `Dom::IntegerMod`[1] that creates the residue class ring $\mathbb{Z}_n$ for a non-negative integer $n$. Our favorite domain—the residue class ring $\mathbb{Z}_n$—is created by evaluating the expression `Dom::IntegerMod(7)`:

```
>> Z7 := Dom::IntegerMod(7)

                        Dom::IntegerMod(7)

>> Z7(2) - 1/Z7(3)

                               4 mod 7
```

Domain constructors are elements of the "meta"-domain `DomainConstructor`. New domains like `Z7` are created by a function call using the constructor as "function". (Internally this is realized with a `"func_call"` method in the domain `DomainConstructor`, but these details are hidden to the user and also to the domain programmer.)
The user will often want to implement a domain by taking over ("inheriting") an existing implementation and altering only some of its entries. Thus, a domain of square matrices can take over many of the basic operations from a domain of general (non-square) matrices. The new domain is derived from an existing domain. In such cases the original domain is called a *direct super-domain* of the new domain, the derived domain is called a *direct sub-domain* of the original domain.
If a direct super-domain also has a direct super-domain then this "super-super-domain" is simply called a *super-domain*. A super-domain is a direct super-domain or a super-domain of a direct super-domain. This is analogue for a *sub-domain*, whereby a domain is either a direct sub-domain or a sub-domain of a direct sub-domain. The domains created with the constructors of the `Dom` package, for example, form a hierarchy. The domain `Dom::BaseDomain` is the root of this domain hierarchy.
Note that multiple inheritance by more than one super-domain—like for classes in C++—is not possible with the domain constructors. One reason is the simple

[1] Please note that most the constructors of the MuPAD library are entries of one of the library package `Dom`, `Cat` or `Ax` in order to avoid naming conflicts.

structure of the domain elements created by `new`, it does not allow to inherit the operands of several elements without changing their indices.
A domain constructor must define the direct super-domain of the domains it creates. The constructor only has to add those entries whose implementation differs from that of the super-domain. Of course, the super-domain can also be dependent on parameters.

## 3.1. Defining Domain Constructors

A special syntax exists in the MuPAD language to define a domain constructor. The relevant rules of the grammar are shown below:

*domain-constructor:*
`domain` *factor domain-definition end-domain*
`domain` *factor* `(` *argument-seq*$_{opt}$ `)` *domain-definition end-domain*

*domain-definition:*
*local-declaration-seq*$_{opt}$ *domain-declaration-seq*$_{opt}$ *domain-entry-seq*$_{opt}$ *initializer*$_{opt}$

*local-declaration-seq:*
`local` *local-var-seq* `;`
`local` *local-var-seq* `;` *local-declaration-seq*

*domain-declaration-seq:*
*domain-declaration* `;`
*domain-declaration* `;` *domain-declaration-seq*

*domain-declaration:*
`inherits` *expression*
`category` *expression-seq*
`axiom` *expression-seq*

*domain-entry-seq:*
*name* `:=` *statement* `;`
*name* `:=` *statement* `;` *entry-seq*

*initializer:*
`begin` *statement-seq*

*end-domain:*
`end`
`end_domain`

Thus a domain constructor has the following general "pattern":

`domain` *name* `(` *parameters* `)`
`local` *variables* `;`

```
    inherits super-domain ;
    category categories ;
    axiom axioms ;
    entries
begin
    initialization
end_domain
```

The individual "pattern sections" have the following meaning:

*name* gives the name of the domain constructor as an expression. The constructor is assigned to the expression given by *name*. The expression is further used for output purposes and to define the keys of the domains to be created.

*parameters* is a sequence of the constructors parameters. This must be a sequence of formal arguments like in procedure definitions, which may have types and default values.

A constructor may also have no parameters, in this case the whole grammar part (*parameters*) is to be omitted. The singleton domain defined by such a constructor is created directly and assigned to the expression given by *name*.

*variables* defines the names of the local variables of the domains created by the constructor. This must be a comma-separated sequence of identifiers.

*super-domain* gives the direct super-domain from which the domains created by the constructor inherit their implementation. This must be an expression that evaluates to a domain. At least the domain `Dom::BaseDomain` should be given. (With this domain important basic operations are inherited. The domain `Dom::BaseDomain` is described in section 6.)

*categories* gives the categories in which the domains created by the constructor are contained. A comma-separated sequence whose elements evaluate to categories must be given. An element of the sequence may also evaluate to `NIL`; in this case no category is entered.

Only the "most special" categories need be given. If a domain belongs to the category of fields then it automatically also belongs to the category of rings. Categories of the super-domains are *not* inherited, because only implementation is inherited via super-domains, not mathematical meaning.

If several categories are given in a constructor, then the more specific should be put first into the list, because the categories are searched for entries in the order they are put into the list.

*axioms* gives the axioms for the domains. This must also be a sequence whose elements evaluate to axioms. Elements can also evaluate to `NIL`, in this case no axiom is entered.

Domains inherit the axioms of their categories. Only those axioms need to be given that are not already axioms of the categories involved. Axioms of the super-domains are *not* inherited, because only implementation is inherited via super-domains, not mathematical meaning.

*entries* is a sequence of any number of assignments of the form *name*`:=`*value*`;`. The entries must be separated by `;`, colons `:` are not allowed. With this the entries are defined in the domains to be created: The expression *value* is evaluated and the result inserted into the slot "*name*" in the new domain.

The expression *value* is only evaluated when a domain is created by the constructor. If the expression evaluates to `NIL` then this entry is not inserted into the domain.

The slot names are converted to strings, they are *not* bound to variables or identifiers. The syntax of the slot names must be the same as for identifiers.

*initialization* gives a statement sequence that is used to test the actual parameters and to initialize the local variables of a new domain.

One may think of the initialization section as the body of a procedure which is executed when a new domain is created by the constructor.

The following scoping rules hold for a constructor:

- A constructor introduces a new lexical context. The domain parameters, local variables and the special variable `dom` introduce new names in this context and hide variable and identifier names from a lexically outer context.
- The expressions for defining the super-domain, categories, axioms, entries and the initialization section are parsed in the context defined by the constructor. Names are bound lexically.
- Default parameter values and parameter types of the constructor are bound in the lexical context, in which the constructor is contained and *not* in the context of the constructor.

Thus the scoping rules for constructors are very similar to the scoping rules for procedures.
The following evaluation rules hold for the constructor:

- The contents of a constructor is not evaluated when the constructor is defined. It is only evaluated when a new domain is to be created by the constructor.
- The expressions defining the default domain parameter values and types are evaluated once when the constructor is defined, similar as with procedures.

- When a domain is created first a new procedure environment (also called *closure*) is created for it which contains the actual values of the parameters and variables. The parameters and variables are bound to the values in this environment.

  Then the initialization section is executed using this procedure environment. Here the actual parameters can be tested and the local variables can be initialized.

- The expressions defining the super-domain, categories, axioms and entries are evaluated on demand only, but always in the procedure environment of the domain.

The initialization section is evaluated like a procedure body during the creation of a new domain. Thus, the function `args`, for instance, can be used as in a procedure to access the actual parameters of the constructor. This will be explained in more detail in section 3.2.
A domain constructor for $\mathbb{Z}_n$ may appear as follows:

```
domain Dom::IntegerMod(Mod: Type::PosInt)

    // no local variables

    inherits Dom::BaseDomain;

    category
        if isprime(Mod) then Cat::Field
        else Cat::CommutativeRing
        end_if;

    axiom
        Ax::canonicalRep, Ax::normalRep;

/*--- entries ---*/

    new := proc(x) begin new(dom, x mod Mod) end_proc;

    zero := dom::new(0);

    one := dom::new(1);

    ...

begin
    if args(0) <> 1 then
       error("wrong no of args")
    end_if;
    if Mod < 2 then
        error("modulus must be > 1")
    end_if
end_domain;
```

The constructor is called `Dom::IntegerMod` and has the formal parameter `Mod`, the modulus. It has no local variables. The initialization tests for the existence of the actual parameter and if it is a valid number.
If the modulus $n$ is a prime number then the domain $\mathbb{Z}_n$ is a field, otherwise it is a commutative ring only. `Dom::IntegerMod` has `Dom::BaseDomain` as its direct super-domain and `Ax::canonicalRep` and `Ax::normalRep` as its only axioms. (The axiom `Ax::canonicalRep` states that mathematically identical domain elements also have identical representation as MuPAD expressions, `Ax::normalRep` states that zero has a unique representation.)
Both the identifiers `dom` and `Mod` are parsed in the lexical context of the constructor and bound to the procedure environment of the domain during the construction of a new domain. Thus, in `Dom::IntegerMod(7)` the parameter `Mod` has the actual value `7` and the variable `dom` has this domain as value.

## 3.2. Initialization and Local Variables

Each domain constructor may be given a list of formal parameters, a list with the names of local variables and an initialization expression sequence. During initialization the actual parameters are tested and then they are usually used to initialize the local variables. This may appear as follows with a constructor for polynomials:

```
domain Dom::Polynomial(R, Indets)
    local NumIndets;

    ...

begin
    if args(0) <> 2 then
        error("wrong no of args")
    end_if;
    if not R::hasProp(Cat::Ring) then
        error("illegal coefficient ring")
    end_if;

    // further tests...

    NumIndets := nops(Indets);
end_domain;
```

Here, `R` is the coefficient ring and `Indets` should be a list of polynomial variables. The local variable `NumIndets` should hold the number of polynomial variables. It is calculated during initialization. Thus, this number needs only be calculated once, it needs not be re-calculated in each method at run-time.
Because `args` may be used as in procedures for initialization, one can easily implement domain constructors with a variable number of arguments. With the constructor for polynomials above, the user can, for instance, define that the rational numbers are to be used as the coefficient ring if only a list of variables is given. The initialization code may look as follows:

```
begin
```

```
    if args(0) = 1 then
        R:= Dom::Rational; Indets:= args(1);
    elif args(0) <> 2 then
        error("wrong no of args")
    end_if;

    // further tests...

    NumIndets:= nops(Indets);
end_domain;
```

The initialization can be regarded as a procedure of the form

```
proc( parameters )
   name name;
   local variables;
begin
   initialization
end_proc
```

which is generated from the name, formal parameters, the local variables and the initialization section. This procedure is then called with the actual domain parameters. When the procedure finishes, its environment is used as the procedure environment containing the actual parameters and variables of the domain.

## 3.3. Constructors without Parameters

If a domain constructor has no formal parameters then it can only create one singleton domain. This domain is created immediately by the constructor and assigned to the name given in the constructors definition. The domain `Dom::Rational` for example is the domain of rational numbers. The domain is created by a constructor without parameters:

```
domain Dom::Rational
    inherits Dom::Numerical;
    ...
end_domain;
```

Note that this is not the case for a constructor with an empty parameter list, as in:

```
domain A()
    ...
end_domain;
```

Here `A` is a constructor (an element of the domain `DomainConstructor`) and the result of the call `A()` returns the singleton domain defined by the constructor.

## 3.4. Representation of Domain Elements

If a domain is no base domain, an element of the domain usually consist of a "container" holding a reference to the domain and the operands of the domain element. If `x` is an element of such an "explicitly represented" domain, then the expressions `domtype(x)`, `x::dom` and `extop(x, 0)` all return the domain of `x`, whereas `extop(x,n)` returns the `n`th operand of `x`.
The representation of domain elements is only implicitly given by the implementation of the operations of the domain, it can not be stated explicitly in the definition of the domains constructor. In the case of `Dom::IntegerMod` for example the domain elements all have the same simple form: they are created with the system function `new` and have exactly one entry, an integer representing the corresponding residue class. Note that the representation of the domain elements can also be made dependent on the domain parameters.
The use of `extop` to access the operands of a domain element is somewhat inconvenient and may lead to hard to maintain code, especially if a domain element has many operands. It may be difficult to remember the indices of the different operands or to re-arrange the operands. Sometimes alias definitions may be useful in order to access the operands, as in:

```
 alias(ResidueClassRep(x) = extop(x,1));

domain Dom::IntegerMod(Mod)
    ...
    _divide := proc(x, y) begin
        new(dom, ResidueClassRep(x) / ResidueClassRep(y) mod Mod)
    end_proc;
    ...
end_domain;
```

### 3.4.1. Representation of Sub-Domain Elements

A domain must take over the representation of the elements of its direct super-domain, in order that the methods of the super-domain can be used: The methods of the super-domain can only "know" the representation of "its" elements. Thus the "layout" and meaning of the operands may not be changed without needing to rewrite most of the super-domains methods.
Usually it is difficult even to append new entries in the sub-domain. The reason for this is that the methods inherited by the super-domain usually have no idea that elements of a sub-domain are to be created and *not* elements of the original domain. If the elements of the sub-domain carry inherently more information than the elements of super-domain generally there will be no way to compute this information from the operands of the elements of the super-domain. But this would be needed in order that the super-domains methods could create elements of the sub-domain.
Sometimes it may be the case that operands added by a sub-domain can be computed from the operands needed for the super-domain. In such a case one could define a method `"new"` for the sub-domain which would be able to create new elements given only the operands of the super-domain elements.

Additionally care must be taken in order that the super-domains methods create the correct elements.
Have a look at the method `"_divide"` of the domain `Dom::IntegerMod`. The implementation shown above was:

```
_divide := proc(x, y) begin
    new(dom, extop(x,1) / extop(y,1) mod Mod)
end_proc;
```

If this implementation is inherited by a sub-domain, then a new element of the sub-domain is created via `new(dom, i)`, which is of course invalid if the elements of the sub-domain need additional operands.
If the elements of the sub-domain could be created by giving the integer representing the residue class ring only, the following implementation of `"_divide"` in `Dom::IntegerMod` would do:

```
_divide := proc(x, y) begin
    dom(extop(x,1) / extop(y,1) mod Mod)
end_proc;
```

In the sub-class one would now have to implement a new method `"new"` which would create elements given the integer representing the residue class ring only. Eventually the inherited method `"_divide"` would call this `"new"` method and return a correct element of the sub-domain.
There is certainly a trade-off in the design of a domain whether it should allow sub-domains to add new operands or not: The strategy shown above (using the method `"new"`) has the disadvantage that it may be much more efficient to call the built-in function `new` directly instead of causing a call to the method `"new"`.

> **Note:** Therefore most domains of the `Dom` package do currently not support sub-domains which add additional operands to their elements.

There is one important exception form this rule: The domain `Dom::BaseDomain` creates no elements and doesn't make any assumptions about the representation of elements, it only makes methods available. Thus it can be used, without limitations, as a super-domain for any domain.
Nevertheless even under this restrictions it is often possible to employ inheritance using a super-domains implementation, as many examples in the library package `Dom` show.

### 3.4.2. Façade Domains

While the domain elements of the domain `Dom::IntegerMod(7)` are explicitly created with the function `new`, other domains exist that do not create any elements of their own, but use the basic types directly to represent "their" elements. These domains are called *façade domains*.
An example for this is the domain `Dom::Rational`, the domain of the rational numbers. Elements of this domain are represented by integers or rational

(MuPAD) numbers, i.e., by elements of the domains DOM_INT or DOM_RAT. The basic domain DOM_RAT alone cannot be used as an algebraic structure because the integers (DOM_INT)—and thus 0 and 1—are *not* elements of DOM_RAT. The basic domains of MuPAD have almost no algebraic structure.
The representation of the rational numbers by the basic types has the advantage that the speed of the basic types, which are implemented in the MuPAD kernel, is fully available. Furthermore the kernel functions can be used as methods, these are much faster than procedures.
However the use of the basic types for representation has the disadvantage that the elements no longer "know" that they are elements of a "more involved" algebraic structure. `type(2/3)` always returns DOM_RAT, even when `2/3` is to be considered as an element of `Dom::Rational` in some context.
Furthermore, when defining methods for façade domains, it should observed that the system functions for basic domains may not be overloaded:

> **Note:** The method `"_plus"`, for example, of a façade domain `D` can only be the system function `_plus`.

The reason is that the elements of `D` are represented by basic types, thus `a + b` would return a different result as `D::_plus(a,b)` if the method `"_plus"` of `D` would not be the function `_plus`.
On the other hand, this also means that no basic type can be used for representation in a façade domain `D` which contains the method `"_plus"`, but for which the system function `_plus` is either not or "falsely" defined for that basic type.
When using basic types there is another limitation: sub-domains of domains that use basic types to represent their elements must take over the representation of their super-domains. They cannot add further information to "their" elements because the function `new` cannot be used for creating the elements.
In the MuPAD library there are the domains `Dom::Expression` for arbitrary MuPAD expressions and `Dom::ArithmeticalExpression` for arithmetical expressions. Facade domains should have one of these domains as a super-domain. Façade domains should further use the axiom `Ax::systemRep` to state that they represent their elements by basic types.

## 3.5. Parameterization of Domain Entries

The structure of a domain entry can depend on the actual parameters of the corresponding domain. With `Dom::IntegerMod` this concerns the method `"_invert"` that calculates the multiplicative inverse of a residue class. If the modulus is a prime number, i.e., the domain is a field, then the modular inverse can always be calculated. However, if the modulus is not a prime then the existence of an inverse depends on the corresponding residue class. This must be regarded in the method `"_invert"`:

```
_invert := if isprime(Mod) then
    proc(x) begin
        new(dom, 1/extop(x,1) mod Mod)
    end_proc
```

```
else
    proc(x) begin
        if igcd(extop(x,1), Mod) = 1 then
            new(dom, 1/extop(x,1) mod Mod)
        else
            FAIL
        end_if
    end_proc
end_if;
```

Here, the decision about which implementation is necessary takes already place during the creation of the method. When creating the method the "`if isprime(Mod) ...`" statement is evaluated after the variables have been bound to their actual values. The result of the evaluation is the appropriate procedure, which is then used as the `"_invert"` method.
The user can also enquire during runtime if a domain element belongs to a field and react accordingly. How the construction of a method exactly occurs is a development decision made during the implementation of the domain constructor. The first alternative has the advantage that during runtime no enquiries have to be made and therefore it is more efficient.
For enquiring about properties each domain defined by a constructor of the `Dom` package has the method `"hasProp"`. This method is inherited from the domain `Dom::BaseDomain`. With `"hasProp"`, the user can enquire if a domain belongs to a certain category for example:

```
>> Z7 := Dom::IntegerMod(7):
   Z7::hasProp(Cat::Field)

                                TRUE

>> Z6 := Dom::IntegerMod(6):
   Z6::hasProp(Cat::Field), Z6::hasProp(Cat::Ring)

                            FALSE, TRUE
```

Here, the user has enquired if `Z7` or respectively `Z6` belong to the category of fields and if `Z6` is a ring. (More about categories later.)

## 3.6. Searching for Domain Entries

The entries defined by a constructor are not directly created during the creation of a new domain. An entry is only created when it is accessed for the first time by `slot` (by defining a `"make_slot"` method). Thus only those entries are created that are in fact needed.
What entries does a domain created by a constructor "know"? In the first place, those directly defined in the constructor and then in addition the entries of the super-domains and the categories. It may be that an entry is defined by different constructors. An entry is searched for as follows:

- If the entry is defined directly in the constructor of the domain then this definition is used.

- If it is not defined in the constructor then the definition is searched for in the super-domains. The first definition is used that is found. (The categories of the super-domains are not searched; only the super-domains themselves.)
- If the definition is not found in the super-domains then it is searched for in the categories in breadth-first manner:

  Firstly, the definition is searched for in the categories given in the constructor. Then it is recursively searched for in the direct super-categories of the categories searched before. The first definition found is returned.
- If the definition is not found in the categories then `FAIL` is returned.

The algorithm used for searching the definition in the categories returns the definition which is "nearest" to the categories defined in the domain constructor. This will be the most specific valid definition among the categories.
With the method `"whichEntry"`, the user can enquire in which domain or category an entry is defined:

```
>> Dom::Rational::whichEntry("idealGenerator");

                        Cat::EuclideanDomain
```

Here, the method `"idealGenerator"` of the domain `Dom::Rational` is defined by the category `Cat::EuclideanDomain`. The method `"whichEntry"` is defined by the domain `Dom::BaseDomain`. The domain `Dom::BaseDomain` also defines some other useful methods for the "exploration" of domains; see the section 6.
The definition of the entries—as already indicated in the example of the method `Dom::IntegerMod::_invert` above—may be dependent on the domain parameters and variables. Sometimes a definition is only to be made for certain parameter values. For the domain `Dom::IntegerMod` a method could be imagined that should only be present when the domain is a field. If the domain is not a field then the definition of the entry should evaluate to `NIL` to achieve this behavior:

```
method_for_fields := if isprime(Mod) then
    proc(x: dom) begin ... end_proc
end_if;
```

If `Mod` is not a prime number then the if-statement returns `NIL` due to the non-existing else-part. The value `NIL` signals to the constructor that no definition exists for the parameters given and that it should search further.
The user can also refer to another method definitions directly, e.g., when the method normally would be "skipped" due to the search strategy for domain entries:

```
method_ab := (Dom::XYZ(Mod))::method_ab;
```

Here, the definition of the method `"method_ab"` is directly taken over from the domain `Dom::XYZ(Mod)`.
Furthermore, methods defined by categories can be used directly:

```
method_ab := CategoryXY::method_xy;
```

This possibility—though quite suggestive—may be somewhat surprising: After all `"method_xy"` is not a method of the domain `Category`, but a definition stored in the constructor of `CategoryXY`. To allow the access with the slot operator a `"slot"`-method for the domain `Category` is defined, see B.3.
The user can refer to category entries only inside of domain or category constructors. In any other context this "call" is incorrect and leads to runtime errors. Even the following call inside a method is incorrect:

```
method_ab := proc(x) begin
    CategoryXY::method_xy(x)
end_proc;
```

Here, during the execution of the method, the domain at hand is not mentioned and thus the entry `CategoryXY::method_xy` cannot be created.

## 3.7. Special Domain Entries

Domains that are created by a domain constructor always have some special entries. This entries must not be changed:

**`constructor`** The constructor of the domain; an element of the domain `DomainConstructor`.

**`closure`** The procedure environment of the domain, holding the actual values of its parameters and variables.

**`make_slot`** A method that creates the slots of the domain on demand.

**`super_domains`** A list containing all super-domains in the order they are searched for entries.

**`categories`** A list of all categories of the domain created so far in the order they are searched for entries.

**`categories_idx`** The index of the next category for which to create the direct super-categories.

**`axioms`** A set of all axioms of the domain created so far.

The other entries are created when needed by the method `"make_slot"`.
All domains that are created by a constructor should have `Dom::BaseDomain` as a super-domain and thus inherit its methods. In this manner, for instance, the method `"hasProp"` is inherited.

# 4. Categories

When new domains are defined it often becomes obvious that some domains have common features because they belong to a common class of algebraic structures. Thus, $\mathbb{Z}_6$ and $\mathbb{Z}_7$ are rings and have, amongst other things, the operations + and * and corresponding neutral elements. In each ring, the operation "exponentiation with a non-negative integer" can be easily implemented by repeated squaring. It would be quite boring to implement such an operation for each ring again. It is, of course, desirable to implement such an operation only once and use it for all rings.
A class of algebraic structures, like "the rings" is called a *category* in the MuPAD library. A category "postulates" certain basic operations and features of the domains that belong to it. The concrete implementation of these basic operations is the task of the domain. With these basic operations further operations for the domains of the category can be defined. These are realized as *generic algorithms*—like the repeated squaring mentioned above.
If a domain is newly created then the corresponding domain constructor determines to which categories it belongs according to the parameters. The constructor adds to the domain the operations defined in these categories. If the newly created domain belongs to the category of rings then the operation "exponentiation with a non-negative integer" is defined for the domain. (The operation is created only when needed—as already described in section 3.6—and not when the domain is created.) Of course one has the possibility, when defining a domain constructor, to implement such an operation directly in the domain constructor, if one wants to utilize special features of the domains.
The categories of a given domain may be "specializations" of more general categories. Thus, a domain that belongs to the category of fields also belongs automatically to the category of rings. In other words: every field is a ring. Categories only need to define those features and operations that have not already been made available by the more general categories.
In this context the category of rings is called *super-category* of the category of fields and the category of fields is called *sub-category* of the category of rings. Similar to the domains, a super-category that is directly given in a category constructor is also called a *direct super-category*. A category can have more than one direct super-category. The rings have, among others, the category `Cat::Rng` (ring without unit) as well as the category `Cat::Monoid` (non-commutative monoids) as direct super-categories. The categories do not only form a tree as the domains do, but a directed acyclic graph.
Note that the category graph depends on the domain at hand, *not* on the domains constructor: The rational numbers are a ring and therefore also a left

module over itself, i.e., they are a left module over the rationals; a fact which is stated in the category of rings. But of course not *any* ring is also a left module over the rationals. The graph of the categories of the integers is depicted in figure 4.

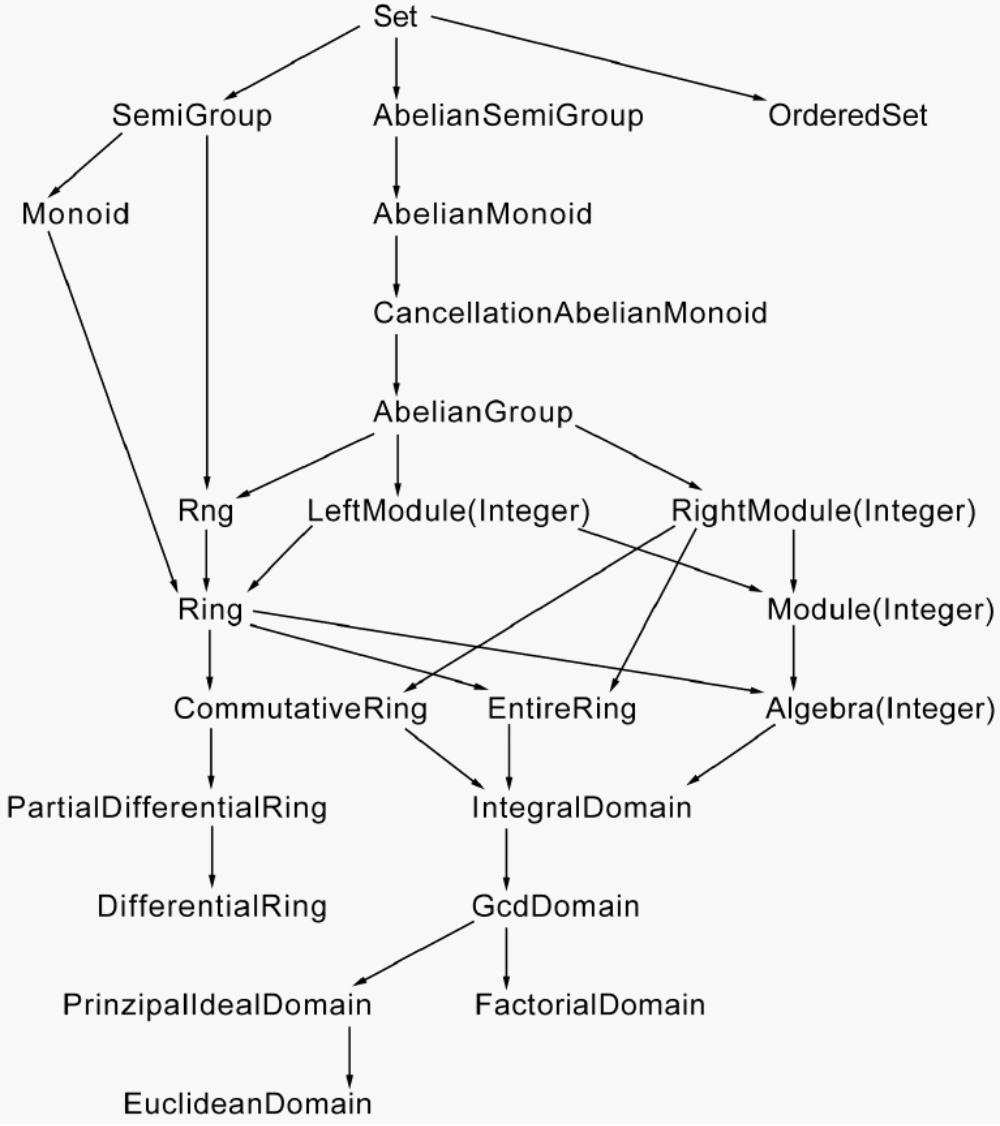


Figure 4: Graph of the Categories of the Integers

Categories may also depend on parameters. Thus, there are various different categories of polynomial rings, which are dependent on the coefficient ring. In order not to have to newly implement the corresponding category of polynomial rings for each possible coefficient ring so-called *category constructors* may be defined. These create categories dependent on parameters.
Categories are not defined directly by the programmer, rather they are always created by a category constructor. Table 2 in appendix A shows the predefined categories and constructors of the library.

## 4.1. Defining Category Constructors

The following rules of the MuPAD language grammar describe the syntax of category constructors. It is quite similar to the syntax of domain constructors:

*category-constructor:*
`category` *factor category-definition end-category*
`category` *factor* `(` *argument-seq*$_{opt}$ `)` *category-definition end-category*

*category-definition:*
*local-declaration-seq*$_{opt}$ *category-declaration-seq*$_{opt}$ *category-entry-seq*$_{opt}$ *initializer*$_{opt}$

*category-declaration-seq:*
*category-declaration* `;`
*category-declaration* `;` *category-declaration-seq*

*category-declaration:*
`category` *expression-seq*
`axiom` *expression-seq*

*category-entry-seq:*
*name* `;`
*name* `:=` *statement* `;`
*name* `;` *entry-seq*
*name* `:=` *statement* `;` *entry-seq*

*end-category:*
`end`
`end_category`

Thus the following "pattern" informally describes the syntax of a category constructor:

```
category name ( parameters )
    local variables ;
    category categories ;
    axiom axioms ;
    entries
begin
    initialization
end_category
```

This is the same pattern as for a domain constructor, only the *inherits* section is missing. The other sections have a similar meaning as those of a domain constructor:

*name* gives the name of the category constructor. Additionally the constructor (or the singleton category, in case the constructor has no parameters) is assigned to this expression.

*parameters* gives the formal parameters of the constructor. This must be a comma-separated sequence of formal arguments, which may have default values and types.

*variables* gives the names of the local variables of the constructor, a comma-separated sequence of identifiers.

*categories* gives the direct super-categories which contain the categories created by this constructor.

*axioms* give the axioms for the created categories.

*entries* is a sequence of names or assignments of the form *name*`:=`*value*`;`. The entries must be separated by `;`. Through an assignment the expression *value* defines the value of the slot *name*.

When only a name is given this indicates that this is a basic operation that has to be present in the domains of this category but whose implementation must take place elsewhere.

The slot names are converted to strings, they are *not* bound to variables or identifiers.

*initialization* must be a statement sequence that is used to test the actual parameters and to initialize the local variables of a new category.

The same scoping rules as for domain constructors hold. Especially the implicitly defined variable `dom` may be used to refer to the domain which uses a category created by this constructor to define (some of) its entries.
The following evaluation rules hold for the constructor. They are similar as for domain constructors:

- The contents of a constructor is not evaluated when the constructor is defined. It is only evaluated when a new category is to be created by the constructor.
- The expressions defining the default category parameter values and types are evaluated once when the constructor is defined.
- When a category is created first a new procedure environment is created for it which contains the actual values of the parameters and variables. The parameters and variables are bound to the values in this environment.

  Then the initialization section is executed with this environment. Here the actual parameters can be tested and the local variables can be initialized.

- The expressions defining the categories, axioms and entries are evaluated on demand only, but always in the closure of the category. Additionally the domain for which the expression is to be created is assigned to the variable `dom` in the categories closure before the evaluation takes place.

As in a domain constructor, an entry can evaluate to `NIL`. In this case the entry is further searched for as if it were not present. Apart from this an entry can also evaluate to the reserved identifier `toBeDefined`. This identifier means that the entry is a basic operation. (This is useful when an operation can only be defined for certain parameter values.) Thus, the entry `xyz;` is equivalent to `xyz := toBeDefined;`.

> **Note:** When defining entries no assumptions about the implementation of the domain for which the entries are to be defined should be made. Categories represent "knowledge" about their domains that is independent of the representation of the domain elements.

If several super-categories are given in a constructor, then the more specific should be put first into the list, like in a domain constructor. The reason is that the categories are searched for entries in the order they are put into the list.
We shall use a constructor for the category of Euclidean domains[1] as our example:

```
category Cat::EuclideanDomain

    // no parameters, no local variables

    category
        Cat::PrincipalIdealDomain;

    // no axioms

/*--- entries ---*/

    rem;

    ...

// no initialization

end_category;
```

An Euclidean domains is a factorial domain which has the additional operations `"rem"` etc.
Note that this category constructor has no parameter, so the singleton category defined by the constructor is created immediately and assigned to `Cat::EuclideanDomain`.

[1] The name `Cat::EuclideanDomain` may be somewhat misleading: This constructor indeed creates a category, not a domain, but the name "Euclidean domain" is too much folklore as that one would dare to use another one.

The operation `"rem"` is a basic operation whose implementation can not be given here. The basic operations must be "filled out" by the corresponding domain constructor. If the definition of this operation is omitted in a domain of the category `Cat::EuclideanDomain` then this leads to a run-time error when the operation is used for the first time.
The calculation of the gcd can be defined by the Euclidean algorithm using the basic operations at hand:

```
gcd := proc(x, y)
    local tmp;
begin
    x:= dom::unitNormal(x);
    y:= dom::unitNormal(y);

    while not dom::iszero(y) do
        tmp:= dom::rem(x, y);
        x:= y;
        y:= tmp
    end_while;

    dom::unitNormal(x)
end_proc;
```

The entries `"iszero"` and `"unitNormal"` are defined elsewhere in the super-categories (`Cat::AbelianMonoid` and `Cat::IntegralDomain`).
When this procedure is inserted as a method into a domain the domain in turn is inserted into the procedure and may be accessed inside the procedure by referring to the implicitly defined variable `dom`.
Instead of the method `dom::iszero` the system function `iszero` could also be called in the procedure above. The overloading of the function would lead to the same method being called. The given direct call is, however, somewhat more efficient.

## 4.2. Initialization and Static Values

Exactly as with the domain constructors, testing of the actual parameters and calculation of the local variables is carried out using the initialization section. In the initialization section the function `args` can be used to access the actual parameters.

## 4.3. Constructors without Parameters

As with parameter-free domain constructors, with a parameter-free category constructor the singleton category defined by the constructor is created immediately and assigned to the name of the constructor.
As with domain constructors this is not the case for a constructor with an empty argument list, like:

```
category C()
```

```
    ...
end_category;
```

## 4.4. Searching for Domain Entries

The method for searching domain entries defined by categories has already been described in section 3.6.
As in domain constructors, definitions of entries can be referred to by directly using the slot operator. The same limitations as for the domain constructors are valid for the access to category entries: the referring expression may only be evaluated directly by the constructor.

# 5. Axioms

The properties of categories and domains are postulated with so-called *axioms*. These are simply attributes whose existence can be enquired. For instance, the domain `Dom::IntegerMod` has the axiom `Ax::canonicalRep`. This axiom states that the domain elements are canonically represented, i.e., that two domain elements are mathematically identical if and only if they are identical as MuPAD expressions. Axioms of domains can be enquired with the method `"hasProp"`, in order to define, for instance, an operation dependent on these.

> **Note:** In the current implementation of the domains and categories of the MuPAD library the "evident" axioms for the categories are not stated explicitly.

There is no explicit axiom which states that the addition of ring elements is commutative for example. It is supposed that the user knows such facts given the information that a domain is a ring. Currently only those axioms are stated which represent "computational" informations about domains, like the axiom `Ax::canonicalRep` mentioned above.
Axioms may also depend on parameters. For this purpose so-called *axiom constructors*, exist, that create axioms dependent on parameters.

## 5.1. Defining Axiom Constructors

Because axioms are only attributes and have no further functionality their constructors are far simpler than domain or category constructors. The following section of the MuPAD language grammar describes the syntax of axiom constructors:

*axiom-constructor:*
`axiom` *factor local-declaration-seq*$_{opt}$ *initializer*$_{opt}$ *end-axiom*
`axiom` *factor* `(` *argument-seq*$_{opt}$ `)` *local-declaration-seq*$_{opt}$ *initializer*$_{opt}$ *end-axiom*

*end-axiom:*
`end`
`end_axiom`

Thus the following "template" informally describes the syntax of an axiom constructor:

```
axiom name ( parameters )
    local variables ;
begin
    initialization
end_axiom
```

The various sections of this template have the same meaning as for the domain and category constructors.
For axiom constructors the same scoping and evaluation rules hold as for domain and category constructors.
The axiom `Ax::canonicalRep` for example is defined as follows:

```
axiom Ax::canonicalRep
    // no parameters, local variables or initialization
end_axiom;
```

Because this axiom constructor has no parameter, the singleton axiom it describes is immediately created and assigned to `Ax::canonicalRep`. This is the simplest axiom one can imagine.

# 6. The Domain `Dom::BaseDomain`

Every domain that is defined in the `Dom` package of the MuPAD library is a subdomain of `Dom::BaseDomain`. This domain defines certain methods that are useful for every domain, for example methods that allow the user to query informations about a domain.
The domain `Dom::BaseDomain` creates no elements of its own and doesn't make any assumptions about the representation of domain elements. Therefore it can be used as a super-domain for any domain regardless how the elements of the domain are represented.
The following methods of `Dom::BaseDomain` may be used to query informations about a domain:

**`hasProp`** The expression `D::hasProp(x)` tests if the domain `D` has a certain property `x`, which may be a domain, category, axiom or constructor. If a domain is given it is tested whether it is `D` or a super-domain of `D`. If a category or axiom is given it is tested whether it is a category or axiom of `D`. If a constructor is given then it is tested whether it is the constructor of `D` or of a super-domain of `D` or of a category or axiom of `D`. A Boolean value is returned.

**`whichEntry`** The expression `D::whichEntry(e)` returns, given the name `e` of an entry of the domain `D`, the domain or category that defines the entry. If `D` has no entry `e` then the expression returns `FAIL`.

**`allEntries`** The expression `D::allEntries()` returns a set containing the names of all the entries known by the domain `D`.

**`undefinedEntries`** The expression `D::undefinedEntries()` returns a set containing the names of all the entries in the domain `D` that have not yet been defined. These are the entries that have to be defined for the members of the categories of `D`, but whose definition does not exist.

**`getSuperDomain`** The expression `D::getSuperDomain()` returns the direct super-domain of `D`.

**`allSuperDomains`** The expression `D::allSuperDomains()` returns a list containing all the super-domains of `D`. The sequence in the list corresponds to the hierarchy of the super-domains: first comes the direct super-domain and then its direct super-domain and so on.

**`getAxioms`** The expression `D::getAxioms()` returns a set containing the direct axioms of `D`.

**allAxioms** The expression `D::allAxioms()` returns a set containing all axioms of `D`.

**getCategories** The expression `D::getCategories()` returns a list containing the direct super-categories of `D`.

**allCategories** The expression `D::allCategories()` returns a list containing all the super-categories of `D`. The sequence of the categories in the list corresponds to the sequence in which the categories are searched for entries.

**info** The expression `D::info()` prints a short information about the domain `D`. This method is called by the function `info`.

The method `"info"` in turn uses the entry `"info_str"` if it exists. This string should contain a short descriptive hint what the domain may be used for, like for example the string `"domain of integer numbers"` in `Dom::Integer`.

The domain `Dom::BaseDomain` additionally defines some other methods which are not explained here, see the help page of the domain for documentation.

# 7. General Nonsense?

What advantages do the concepts described here have over the implementation of domains "by hand"? Well—the necessity of constructing domains that can be parameterized and inheritance of implementation is indisputable. The user can, of course, create such domains through normal procedures. However, such a procedure would be unstructured, every developer would create his own domains and define his own operations. The domain constructors standardize the implementation of domains and contribute to better maintenance. In addition the base domains takes over many standard tasks in the construction and thus reduces the developers work load, as well as avoiding unnecessary error sources and code duplication.
This is even more so when using categories. The advantages of generic algorithms, i.e., avoidance of the re-implementation for each possible domain, thus, reducing the implementation effort and the possibilities of errors, are quite obvious. The category constructors systematize the development of generic algorithms. And, last but not least, the user can expect uniformity in the use of domains.
These advantages are not fiction. M. Monagan has developed a library package for Maple V, GAUSS, which enables the definition of domains and categories, see [6]. In GAUSS domains are simply tables, domain constructors and categories are Maple functions that manipulate these tables. The domain operations are stored as table entries. Using the domains and their operators is a bit tedious because they are not integrated in Maple and cannot be addressed using the normal operators of the Maple language. In addition, the output of the domain elements cannot be altered to suit the user's own requirements.
Despite these deficiencies, GAUSS has been successfully employed to, for instance, calculate Gröbner bases in various polynomial rings, see [5]. Implementation details are hidden by the categories. Thus, different polynomial data structures and term orderings can be used with one another, but still only one generic algorithm needs to be implemented for calculating the Gröbner bases.
Most of the concepts in the constructors are based on ideas in AXIOM, see [7]. Many new algorithms have been implemented in AXIOM for the first time because there the user has the algebraic structures handy that he can use for his problem, independent of their representation. All objects in AXIOM are strictly typed. This is not possible in MuPAD because the MuPAD language is not strictly typed.
A debatable feature of AXIOM is the automatic type conversion: the system tries to determine the type of the user's input on entry, i.e., to convert untyped

expressions directly into domain elements. In MuPAD the user currently has to explicitly convert the type "by hand". It is, of course, in some cases easier for the user not to have to give the type. On the other hand, it is often difficult to determine the type of an expression that the user has in mind. For this AXIOM uses time-consuming and error-susceptible heuristics.

## 7.1. Where to go?

Open fields for improvements are:

- Design and implement a type coercion mechanism for interactive input. Such a method should be comfortable, fast and easy to understand. One would very carefully have to achieve a balance between comfort and soundness.

# A. The Constructors of the MuPAD Library

Here the currently implemented constructors of the MuPAD library are listed only. In [2], [3] and [4] they are documented in detail.

| | |
|---|---|
| `Dom::AlgebraicExtension` | simple algebraic extensions |
| `Dom::ArithmeticalExpression` | arithmetical expressions |
| `Dom::BaseDomain` | base of the domain hierarchy |
| `Dom::Complex` | complex numbers |
| `Dom::DihedralGroup` | dihedral groups |
| `Dom::DistributedPolynomial` | distributed polynomials |
| `Dom::Expression` | expressions |
| `Dom::ExpressionField` | expressions regarded as field |
| `Dom::Float` | floating point numbers |
| `Dom::Fraction` | fractions |
| `Dom::GaloisField` | finite fields |
| `Dom::Ideal` | ideals of a ring |
| `Dom::ImageSet` | infinite sets as images of sets |
| `Dom::Integer` | integer numbers |
| `Dom::IntegerMod` | residue class rings $\mathbb{Z}_n$ |
| `Dom::Interval` | interval arithmetic |
| `Dom::Matrix` | matrices |
| `Dom::MatrixGroup` | groups of matrices |
| `Dom::MonomOrdering` | monomial orderings |
| `Dom::Multiset` | multisets |
| `Dom::MultivariatePolynomial` | multivariate polynomials |
| `Dom::Numerical` | numbers |
| `Dom::PermutationGroup` | symmetric groups |
| `Dom::Polynomial` | polynomials |
| `Dom::Product` | homogeneous direct products |
| `Dom::Quaternion` | quaternions |
| `Dom::Rational` | rational numbers |
| `Dom::Real` | real numbers |
| `Dom::SparseMatrixF2` | sparse matrices over $\mathbb{Z}_2$ |
| `Dom::SquareMatrix` | square matrices |
| `Dom::UnivariatePolynomial` | univariate polynomials |

Table 1: The domain constructors of the library package `Dom`

| | |
|---|---|
| `Cat::AbelianGroup` | Abelian groups |
| `Cat::AbelianMonoid` | Abelian monoids |
| `Cat::AbelianSemiGroup` | Abelian semi-groups |
| `Cat::Algebra` | associative algebras |
| `Cat::CancellationAbelianMonoid` | Abelian monoids w. cancellation |
| `Cat::CommutativeRing` | commutative rings |
| `Cat::DifferentialRing` | ordinary differential rings |
| `Cat::EntireRing` | rings |
| `Cat::EuclideanDomain` | Euclidean domains |
| `Cat::FactorialDomain` | factorial domains |
| `Cat::Field` | fields |
| `Cat::FiniteCollection` | finite collections |
| `Cat::GcdDomain` | integral domains with a gcd |
| `Cat::Group` | groups |
| `Cat::HomogeneousFiniteCollection` | homogeneous finite collections |
| `Cat::HomogeneousFiniteProduct` | homogeneous finite products |
| `Cat::IntegralDomain` | integral domains |
| `Cat::LeftModule` | left-R-modules |
| `Cat::Matrix` | matrices |
| `Cat::Module` | R-modules |
| `Cat::Monoid` | monoids |
| `Cat::OrderedSet` | ordered sets |
| `Cat::PartialDifferentialRing` | partial differential rings |
| `Cat::Polynomial` | polynomials |
| `Cat::PrincipalIdealDomain` | principal ideal domains |
| `Cat::QuotientField` | quotient fields |
| `Cat::RightModule` | right-R-modules |
| `Cat::Ring` | rings |
| `Cat::Rng` | rings without unit |
| `Cat::SemiGroup` | semi-groups |
| `Cat::Set` | sets |
| `Cat::SkewField` | skew fields |
| `Cat::SquareMatrix` | square matrices |
| `Cat::UnivariatePolynomial` | univariate polynomials |
| `Cat::VectorSpace` | vector spaces |

Table 2: The category constructors of the library package `Cat`

| | |
|---|---|
| `Ax::canonicalOrder` | elements are ordered canonically |
| `Ax::canonicalRep` | elements are represented canonically |
| `Ax::canonicalUnitNormal` | unit normals are canonically |
| `Ax::efficientOperation` | the named operation is implemented efficiently |
| `Ax::closedUnitNormals` | unit normals are closed under multiplication |
| `Ax::normalRep` | zero is represented canonically |
| `Ax::noZeroDivisors` | no zero divisors exist |
| `Ax::systemRep` | facade domain |

Table 3: The axiom constructors of the library package `Ax`

# B. Special Domain Entries

There are some domain entries and methods that do not canonically overload system functions. These special entries and methods are briefly described here. The entries and methods are all indexed by strings. Thus, the slot operator `::` can always be used to access these entries.
Together with the methods, the arguments with which the methods are called are described. In the following, `D` is always a domain that contains the given entry. The expression `D::method(x, y)` means that the method `"method"` is called with arguments `x` and `y`. Dots like `...` indicates a—possibly empty—sequence of arguments.

**`key`** Each domain must have a unique key, which may be an arbitrary expression. Two domains with different keys are considered different. The key is defined when a domain is created via `newDomain`. The key of a domain may be changed with `slot` or the `::` operator, but it must not be deleted or changed into a key of another already existing domain.

## B.1. Creating Domain Entries

If `D` is a domain and the expression `D(a1,...,an)` is executed, a new element of `D` should be created using the arguments `a1` ... `an`.
This feature is implemented by calling the method `"new"` of the domain `D` with the given arguments, i.e., the call `D::new(a1,...,an)` is executed. Thus the method `"new"` does *not* overload the function `new`.
The method `"new"` usually should check the arguments and create a new domain element from them using the function `new`.

**`new`** If the expression `D(a1,...,an)` is evaluated for a domain `D`, this in turn executes the expression `D::new(a1,...,an)`. The method `"new"` should return a new element of `D` using the arguments `a1` ... `an`.

An error is raised if `D` has no method `"new"`.

Domain elements are finally created using the function `new`, as described above. If the domain is no base domain, an element of the domain usually must consist of a "container" holding a reference to the domain and the operands of the domain element. Usually just this container is created by the function `new`, which can not be overloaded.
This causes a problem for facade domains. If `D` is a facade domain then no container should be created by `new`, rather an element of another domain should

be returned. This may be implemented by defining a method `"new_extelement"` for the facade domain. (Note that a `"new_extelement"`-method is defined implicitly if one states the axiom `Ax::systemRep`.)
Similar reasoning holds for base domains: Elements of a base domain do not consist of a container holding the data, but rather of the "raw" data objects. To allow a call of the form `new(DOM_INT, 1)` to return `1`, a method `"new_extelement"` must exist for `DOM_INT`. Such a method in fact exists for all base domains.

**`new_extelement`** If for a domain `D` the call `new(D,a1,...,an)` is executed and `D` has a method `"new_extelement"`, then in turn the call `D::new_extelement(D,a1,...,an)` is executed and the result is returned as value.

If no such method exists a container for the new element of `D` is created by the function `new` which has the operands `a1` ... `an`.

## B.2. Creating Domain Entries

Domain entries may be created on demand:

**`make_slot`** If the function `slot`, given a domain `D` and an index `i`, finds an slot in the domain, then its value is always returned. If no slot exists, but the domain `D` has a method `"make_slot"`, then `D::make_slot(D, i)` is called. The value returned by this call is entered into the domain as value of a new slot with index `i` and returned by `slot`. If no `"make_slot"` method exists then `FAIL` returned by the function `slot`.

Please note that if `D::make_slot(D, i)` returns `FAIL`, then `FAIL` is explicitly inserted as the value of the slot `i` into the domain. Thus the method `"make_slot"` will *not* be called twice for the slot `i`.

## B.3. Accessing Slots

The function `slot` may be overloaded as usual:

**`slot`** Reading the value of a slot: If `slot(e, i)` is called for a domain element `e` and the domain `D` of `e` has a method `"slot"`, then `D::slot(e, i)` with the domain element `e` and the index `i` is called and the value returned. If no such method exists the error "unknown slot" is raised.

Changing the value of a slot: If `slot(e, i, v)` is called for an expression `e` which evaluates to an element of domain `D` and the domain `D` has a method `"slot"`, then `D::slot(e, i, v)` with the expression `e`, the index `i` and the new value `v` of the slot is called. The method should create a new domain element in this case which has a slot `i` with the given value `v`. This new element is returned as the value of the call of the function `slot` and assigned to the expression `e`. The expression `e`

may not only be an identifier or variable, it may also be a function call, an indexed expression or a slot expression.

Note that the function `slot` may also be overloaded for base domains other than `DOM_DOMAIN` and `DOM_FUNC_ENV`.

## B.4. Evaluation

In general domain elements are not further evaluated. This can be changed by using an appropriate method. Both these methods are optional; the second method `"posteval"` is very specific and not usually necessary.

**`evaluate`** The method is called when a domain element is evaluated; the call `D::evaluate(e)` has to evaluate the domain element `e`.

**`posteval`** When a substitution depth of 1 is reached, a datum is generally not evaluated further, only the value of the datum is returned. (This is normally the case, for example, inside procedures.) The method `"evaluate"` is not executed if the substitution depth is 1. Instead the `"posteval"` method is called when the substitution depth 1 is reached during the evaluation of a domain element. `D::posteval(e)` has to evaluate the domain element `e`. If no such method exists `e` is not evaluated further.

## B.5. Output

**`Name`** An expression with the name of the domain, which is printed instead of the domain. If the name is a string, it is printed without quotes (").

**`print`** Method for the output of a domain element. `D::print(e)` has to return an expression which is used to print the domain element `e`. The user should not use the `print` function in this method!

## B.6. MCode

MuPAD data may be written to and read from byte streams in the so-called MCode format. The input and output of domains can be controlled for MCode:

**`create_dom`** If this entry exists when the domain is written to a MCode stream, the domain entries are generally *not* written to the stream, only this entry is written. It is assumed that the reader of the stream can create the domain by evaluating the entry `"create_dom"`.

If the entry `"create_dom"` does not exist all the entries of the domain are written to and read from the stream.

If the stream is to be read from another kernel for example, the other kernel can easily create the domain `Dom::Integer` given the domains name, there is no need to transfer the domain entries. Domains inheriting from `Dom::BaseDomain` inherit a `"create_dom"` entry which simply contains the domains name.

## B.7. Arithmetic

Whereas the arithmetical functions `_plus`, `_mult` and `_power` existed since the very first versions of MuPAD, the functions `_negate`, `_subtract` and `_invert` are relatively new:
Given inputs of the form `-e`, `e-f` and `1/e`, the parser creates the expressions `_negate(e)`, `_subtract(e,f)` and `_invert(e)` which are then evaluated. With "normal" expressions `e` and `f`, expressions of the form `(-1)*e e+(-1)*f` and `e^(-1)` are returned. For domain elements the functions `_negate`, `_subtract` and `_invert` may be overloaded as usual:

**`_negate`** A method for negating a domain element. It is called for an input of the form `-e`. `D::_negate(e)` has to return the opposite of the domain element `e`.

**`_subtract`** A method for subtracting domain elements. It is called for an input of the form `e-f`. `D::_subtract(e,f)` has to return result of subtracting `f` from `e`.

**`_invert`** A method for inverting a domain element. It is called for an input of the form `1/e`. `D::_invert(e)` has to return the inverse of the domain element `e`.

## B.8. Accessing Operands

The functions `op` and `subsop` resolve operand paths automatically. The methods `"op"` and `"subsop"` only need to return, or respectively, change the direct operands of the domain elements. If, for example, in an expression `op(x,[2,1,3])`, the second operand of `x` is a domain element `y`, then the method `"op"` of the domain `D` of `y` is called as `D::op(y,1)`. The system function `op` then extracts the third operand of the value returned by the method.

**`op`** The function `op` already resolves operand paths. The method `"op"` only needs to return the "direct" operands of a domain element. A call of the method `"op"` therefore has one of the forms `D::op(e)`, `D::op(e,i)` or `D::op(e,i..j)` with integers `i` and `j`.

**`subsop`** The function `subsop` also resolves operand paths automatically.

## B.9. Type Testing and Conversion

The function `testtype` is used for testing types. The idea is as follows: The call `testtype(e, T)` should return `TRUE` if the datum `e` has the type `T`. Here the type `T` can be a domain or a special *type expression*. If `T` is a domain, `testtype` should also return `TRUE` if `e` may be converted into an element of `T` by one of the domains involved.

**Domains as Types** If the call `testtype(e, T)` returns the value `TRUE` with expression `e` and domain `T`, then `e` can be converted into an element of `T` either by the domain `D` of the datum `e` or by the "target domain" `T`.
Both domains are responsible for conversion. The domain `D` of `e` is "asked" first if it can convert `e` into type `T`. If this fails the domain `T` is asked if it can convert `e` into one of its elements. Thus, both domains should offer methods that can execute a conversion.
The function `testtype` first calls the method `"testtype"` of the domain `D` of `e` (in the form `D::testtype (e, T)`). If the method returns `TRUE` or `FALSE` then the function returns the same value.
If however, the method returns `FAIL` then this means that the method could not decide if a conversion is possible. In this case the method `"testtype"` of the domain `T` is called (in the form `T::testtype(e, T)`). If the method returns a Boolean value then this is returned. If the method returns `FAIL`, then the value `FALSE` is returned (none of the domains is capable of executing the conversion).
Note that the `"testtype"`-method is also called if `e` is an element of a base domain or if `T` is a base domain.

**`testtype`** The expression `D::testtype(e, T)` must return the value `TRUE` if the domain `D` can convert the expression `e` into an element of the domain `T`. The domain `D` can be either the domain of `e` or the domain `T`! The method should only return `FALSE` if it "knows" that such a conversion is not possible. The method has to return `FAIL` if it cannot execute the conversion itself, but if the conversion might possibly be done by another domain.

It is not necessary for the domain `T` to create its own elements. Abstract domains can also be defined. These may be used to test if a datum has a particular form.
The following example defines the domain `NUMERIC`, which may be used to determine if a datum is a number. The domain is defined as follows:

```
NUMERIC := newDomain("NUMERIC");
NUMERIC::testtype := proc(x, y)
begin
    if contains({ DOM_INT, DOM_RAT, DOM_FLOAT, DOM_COMPLEX },
                domtype(x)) then
        TRUE
    else
        FAIL
```

```
    end_if
end_proc;
```

In this case it is clear that the first argument `x` cannot be an element of the domain `NUMERIC` because the domain does not create any "own" elements. (There is no method `"new"`.) Thus, `x` must be an element of a different domain and `y` must be the domain `NUMERIC` itself. The method only has to test if `x` is a number.
If the method `"testtype"` returns the value `TRUE` then the domain—if it is not an abstract domain—should offer a corresponding conversion routine:

**`convert`** With `D::convert(e)` the datum `e` has to be converted into an element of the domain `D`. The method has to return `FAIL` if a conversion is not possible.

**`convert_to`** With `D::convert_to(e, T)` the domain element `e` (an element of `D`) has to be converted into an element of the domain `T`. The method has to return `FAIL` if conversion is not possible.

**`expr`** This is a special case of `"convert_to"`. With `D::expr(e)` the domain element `e` (of `D`) must be converted into an expression built by elements of the basic domains which may be used for printing for example. Once more `FAIL` has to be returned if conversion is not possible.

Note that the function `expr` is overloaded by a method `"expr"` as usual.

This routines should never return an error when conversion is not possible.

**Type Expressions** In the previous section a type was defined by a domain. However, domains are also possible whose *elements* represent types. Such elements can for example represent certain types of expressions (e.g., lists with integers as elements). Thus, the second argument of `testtype` needs not only be a domain but can also be a domain element which represents a type.
If the call `testtype(e, T)` returns `TRUE` for a type expression `T` then `e` is assumed to already have the properties asked for by `T`. Generally no conversion methods need to be defined in this case by the domains of `e` or `T`.
A type expression may be any datum which is not a domain. Thus the `"testtype"`-method is even called if `T` is an element of a base domain.

**`testtype`** The expression `DT::testtype(e, T)` has to return `TRUE` if the argument `T` is a type expression which is an element of the domain `DT` and if `e` has the appropriate type. The method can return `FALSE` if it "knows" that `e` does not have the appropriate type. It has to return `FAIL` if it cannot decide if `e` has the correct type.

Thus a `"testtype"`-method for a domain of type expressions (`DT` above) has not only to consider the cases where `T` is a domain (either the domain of `e` its own domain `DT`), but also the case that `T` is an element of `DT`—adding some more complexity.

A simple example of type expressions are strings. The user can ask if `x` is a sum with `testtype(x, "_plus")`. Here the string `"_plus"` represents the type "sum", the strings are type expressions. This is implemented with an appropriate `"testtype"` method of the domain `DOM_STRING`:

```
DOM_STRING::testtype := proc(x, T) begin
    if domtype(T) = DOM_STRING then
        bool(type(x) = T)
    elif domtype(x) = T then
        TRUE
    else
        FAIL
    end_if
end_proc;
```

The function `type` returns a string representing the type as the operator of an expression. This string is compared with the "searched for" type `T`.

## B.10. Function Calls

Sometimes it is desirable to consider a domain element `e` as a function, i.e., expressions of the form `e(a,...)` are to be evaluated. The user can consider the "function call brackets" `()` as an operator with `e`, `a`, `...` as the operands. This sort of operator does not exist in MuPAD but it can be "simulated" by using the method `"func_call"`.

**`func_call`** When an expression of the form `e(a1,..., an)` with a domain element `e` is evaluated then the result of the call `D::func_call(e, a1,..., an)` is returned, where `D` is the domain of `e`, if the original expression is not on the left hand side of an assignment.

With this method all the arguments `a1, ... an` are *not* evaluated before the method is called (it does not matter if the the option `hold` is set in the method `func_call` or not). The domain element `e` is evaluated.

**`set_func_call`** If an assignment of the form `e(a1,..., an) := v` is evaluated where the expression `e` evaluates to an element of domain `D`, then the result of the call `D::set_func_call(e, a1,..., an, v)` is assigned to the expression `e`.

This method is needed in order to "simulate" the assignments to remember tables of functions. A new domain element, which is a copy of the value of `e` and additionally returns the value `v` when called as function with the arguments `a1,..., an`, should be created by the method. This new domain element is returned and then is assigned to `e` automatically. Note that `e` is not evaluated before the method is called. The other arguments are evaluated as usual.

Note that the method `"set_func_call"` is also be called if the expression `e` is not only an identifier or variable, but a function call, indexed identifier or slot expression.

The method "func_call" can be overloaded for most of the basic types. It is, for instance, defined for the basic type DOM_POLY. If p is a polynomial with two variables then p(x, y) evaluates the polynomial at the points x and y:

```
DOM_POLY::func_call := proc(p: DOM_POLY)
    local indets;
begin
    indets:= op(p, 2);
    if args(0) <> nops(indets) + 1 then
        error("wrong no of args")
    end_if;
    evalp(p, op(zip(indets, context([args(2..args(0))]), _equal)))
end_proc;
```

The polynomial p is evaluated with evalp. The arguments are previously evaluated in the calling context using the function context.

## B.11. Indexed Access

In a similar way to how domain elements can be considered as functions, they can also be considered as data structures which store a value under an index. Examples for such data structures are lists, tables and arrays. Indexed access to these domain elements can be simulated using the method "_index". In principle it is the function _index that is overloaded, however the left hand side of an assignment needs special treatment:

**_index** When an expression of the form e[i1,...,in] with an element e of domain D is evaluated then the result of the call D::_index(e, i1,..., in) is returned if the original expression is not on the left hand side of an assignment.

**set_index** When an expression of the form e[i1,...,in] := v is evaluated (whereby the expression e evaluates to an element of domain D) then the result of the call D::set_index(e, i1,..., in, v) is assigned as a value to the expression e.

To implement the storage of a new value in the domain element given by e, the method should proceed as follows: Store the value v under the index i1,..., in in a copy of the domain element given by e and then return the changed domain element. This is then assigned to e automatically.

As with "set_func_call", the method "set_index" is also be called when the expression e is not only an identifier or variable, but a function call, indexed identifier or slot expression.

## B.12. Coefficient Rings of Polynomials

A domain D can be used as the coefficient ring of a polynomial. For this the domain must contain certain methods. In the following e and f are always elements of D:

**_plus** The call `D::_plus(e,f)` must return the sum of the arguments.

**_negate** The call `D::negate(e)` must return the opposite of `e`.

**_mult** The call `D::_mult(e,f)` must return the product of the arguments.

**_power** The call `D::_power(e, i)` must return the power `e^i` for an integer `i` greater than 1.

**zero** A zero element of the domain has to be stored under this entry.

**one** An unit element of the domain has to be stored under this entry.

The following methods are not absolutely necessary, but can be useful for conversion and zero test:

**convert** The call `D::convert(x)` has to convert the expression `x` into an element of `D`.

**expr** The call `D::expr(e)` has to convert the domain element `e` into an expression.

**iszero** The call `D::iszero(e)` has to return `TRUE` if and only if the domain element `e` is zero.

If no method `"iszero"` exists, the elements of the domain are compared to the entry `"zero"` for zero-testing. Thus the method `"iszero"` defaults to

```
e -> bool(e = dom::zero)
```

The following methods are only necessary if the functions `divide`, `norm`, `diff` or `polylib::randpoly` are to be used:

**_divide** The call `D::_divide(e1, e2)` must return the quotient of the domain elements `e1` and `e2`. The method must return `FAIL` if division is not possible.

**norm** The call `D::norm(e)` must return the norm of the domain element `e`, an expression which may be converted to a number by using `float`.

**diff** The call `D::diff(e, x)` must return the derivative of the domain element `e` with respect to the variable `x`.

**intmult** When differentiating, the call `D::intmult(e, i)` must return the `i`-fold multiple of the domain element `e`; `i` is a non-negative integer. One may implement this method by repeated additions for example.

**random** When creating random polynomials and no other random generator is given then this method is called. `D::random()` should return a random element of `D`.

## B.13. Domains Created by Constructors

Domains created by constructors have certain entries that are needed by the constructor. These are `"constructor"`, `"closure"`, `"make_slot"`, `"super_domains"`, `"categories"`, `"categories_idx"` and `"axioms"`. They are described in section 3.7 and must not be altered.

## B.14. Domains as Library Packages

Domains may not only be used to represent algebraic structures or abstract data types. An example for a (perhaps at first glance, unusual) application of domains are the library packages in MuPAD. All procedures of a package are entered into a so-called *library domain*. This has the advantage that the procedure names can not conflict with procedure names from other packages. The procedure of the `groebner` package computing S-polynomials is, for instance, referred to by the expression `groebner::spoly`. The library domains are usually created as raw domains.
By using the function `export` the names of those procedures of the package, that may be "externally" used, can be made "globally" visible. These procedures are also called the *interface* of the package. (The other procedures are only intended for internal purposes.) After entering `export(groebner, spoly)` the interface procedure `groebner::spoly` can also be directly addressed with `spoly`.
The function `info` displays a short information about the package as well as a list of the functions that can be exported.
Two entries are necessary in a library domain so that `export` and `info` can work correctly:

**`info`** A string or a method. If the entry is a string then the string and the interface functions are printed. If the entry is a method then it is called as `D::info()` by `info`.

**`interface`** A set of identifiers. These are the names of the interface functions which can be exported by `export`. The identifiers should be enclosed by `hold` to prevent evaluation.

In the `groebner` package this looks like follows:

```
groebner := newDomain("groebner");
groebner::info := "Library 'groebner': ....";
groebner::interface := { hold(spoly), ... };
```

# C. An Example: Multi-Indices

As an example for the implementation of a simple domain constructor we will use the constructor `MultiIndex`. The constructor has one or two parameters which give the dimension of the indices and optionally an ordering for the indices. The default ordering is lexically.
`MultiIndex(3)` creates the domain of the 3-dimensional lexically ordered multi-indices:

```
>> M := MultiIndex(3)
```

```
MultiIndex(3, Lex(3))
```

A multi-index may be created by its components, it is printed as a list:

```
>> a := M(1,2,0)
```

```
[1, 2, 0]
```

Accessing the second component:

```
>> a[2]
```

```
2
```

The absolute value of an index is given by the sum of its components:

```
>> abs(a)
```

```
3
```

Creating a second index:

```
>> b := M(2,2,2)
```

```
[2, 2, 2]
```

Adding them adds the components pair by pair:

```
>> a + b
```

```
[3, 4, 2]
```

Comparing them with respect to the lexical ordering:

```
>> bool(a < b)
```

```
TRUE
```

Comparing with another index:

```
>> bool(M(2,2,3) < b)
```

```
FALSE
```

The ordering must be given as an element of the domain `Dom::MonomOrdering`. Elements of this domain may be used to compare lists of non-negative integers—just what we need.
The definition of the constructor is as follows:

```
domain MultiIndex(dimen: Type::PosInt, ordering)

// no local values

inherits
    Dom::BaseDomain;

category
    Cat::CancellationAbelianMonoid, Cat::OrderedSet;

axiom
    Ax::canonicalRep;

    ... // the methods follow here

// finally the init procedure of the domain
begin
    if testargs() then
        if args(0) = 2 and domtype(ordering) <> Dom::MonomOrdering then
            error("illegal ordering")
        end_if
    end_if;
    if args(0) = 1 then ordering:= Dom::MonomOrdering(Lex(dimen))
    elif args(0) <> 2 then error("wrong no of args") end_if
end_domain;
```

During initialization it is tested if the optional parameter `ordering` exists. If it does not exist then the lexical ordering `Dom::MonomOrdering(Lex(dimen))` is used as the default value.
The direct super-domain is `Dom::BaseDomain`, the categories defined directly are `Cat::OrderedSet` ("ordered set") and `Cat::CancellationAbelianMonoid` ("abelian monoid with cancellation"). The only axiom is `Ax::canonicalRep` ("canonical representation"). Note that we could also have used `Dom::Product` as the super-domain (the constructor of direct product domains), but then almost no work would have been left (too bad for an example).
Because of the super-domain `Dom::BaseDomain` and categories `Cat::OrderedSet` and `Cat::CancellationAbelianMonoid` a whole series of entries are defined for the constructor. These are:

```
"TeX", "_leequal", "_less", "_negate", "_plus", "_sub-
tract", "allAxioms", "allCategories", "allEntries",
"allSuperDomains", "coerce", "convert", "convert_to",
"create_dom", "equal", "equiv", "expr", "getAx-
ioms", "getCategories", "getSuperDomain", "hasProp",
```

```
"info","intmult","iszero","key","max","min","new",
"new_extelement", "print", "printMethods", "sort",
"subs", "subsex", "testtype", "undefinedEntries",
"whichEntry","zero"
```

For most of these entries there is already a default implementation inherited from the super-domain `Dom::BaseDomain` or from the categories. Only the following entries are basic operations and therefore have to be entered:

```
"_less", "_plus", "_subtract", "convert", "expr",
"print","zero"
```

One may get a list of this entries by using the methods `"allEntries"` and `"undefinedEntries"`. (Define the constructor first without any entries and then use the methods `"allEntries"` and `"undefinedEntries"` with an "example domain" to get the entry lists.)
In this example each multi-index is represented as follows: It has a single operand, which is a list of non-negative integers. The numbers in the list are the components of the index.
The method `"convert"` converts a datum into a multi-index or returns `FAIL` if this is not possible. In our example `"convert"` can be called with three different arguments:

`D::convert(x)` returns `x`, if `x` is a multi-index of the correct dimension.

`D::convert(L)` returns a multi-index if `L` is a list of non-negative integers of the correct length.

`D::convert(j1,...,jn)` returns a multi-index, if `j1` to `jn` are non-negative integers, whereby `n` is the dimension of the multi-index.

The method is implemented accordingly:

```
convert := proc(l) begin
    if args(0) <> 1 then
        if args(0) = 0 then return(FAIL) end_if;
        l:= [ args() ]
    end_if;

    if domtype(l) = dom then return(l) end_if;
    if not testtype(l, Type::ListOf(Type::NonNegInt, dimen, dimen))
    then
        return(FAIL)
    end_if;
    new(dom, l)
end_proc;
```

`new(dom, l)` creates a new multi-index, `l` is the list of components.
The zero element `"zero"` simply contains a list of zeros:

```
zero := new(dom, [0 $ dimen]);
```

Addition adds the list elements component by component. The method `"_plus"` is a bit intricate because it has to cope with an arbitrary number of arguments:

```
_plus := proc(x, y) begin
    case args(0)
    of 1 do
        return(x);
    of 2 do
        if domtype(x) <> dom or domtype(y) <> dom then
            return(FAIL)
        end_if;
        return(new(dom, zip(extop(x,1), extop(y,1), _plus)));
    otherwise
        y:= _plus(args(2..args(0)));
        return(x+y);
    end_case
end_proc;
```

Of course one could do better than returning `FAIL` if the arguments are no multi-indices.
The method `"_subtract"` has to carry out subtraction if this is possible. Because negative components are not allowed the method must return `FAIL` if one occurs:

```
_subtract := proc(x: dom, y: dom) begin
    x:= zip(extop(x,1), map(extop(y,1), -id), _plus);
    if nops(select(x, _less, 0)) <> 0 then
        return(FAIL)
    end_if;
    new(dom, x)
end_proc;
```

Note the somewhat tricky way to negate the list elements of `y` by using `-id` and the use of `select` to search for negative components.
The method `"_less"` compares two indices. For this the optional ordering `ordering` is used:

```
_less := proc(x: dom, y: dom) begin
    bool(ordering(extop(x,1), extop(y,1)) = -1)
end_proc;
```

If an element of the domain `Dom::MonomOrdering` is applied to two lists it returns one of `-1`, `0` or `1` if the first argument is less than, equal or greater than the second argument.
The method `"expr"` returns the list representing the index, the method print simply uses the method `"expr"` to output a multi-index:

```
expr := proc(x: dom) begin extop(x,1) end_proc;

print := expr;
```

In addition to the methods above, the constructor defines four other methods which are not strictly needed—according to the domains categories—but which are quite useful:

`"_index"`, `"abs"`, `"random"`, `"set_index"`

The method "_index" allows indexed access of components. "abs" returns the absolute value of a multi-index, i.e., the sum of its components. "random" creates a random multi-index. "set_index" allows components to be changed:

```
_index := proc(x: dom, i: Type::PosInt) begin
    extop(x,1)[i]
end_proc;

abs := proc(x: dom) begin
    _plus(op(extop(x,1)))
end_proc;

random := proc() local l, i; begin
    l:= [];
    for i from 1 to dimen do l:= append(l, random()) end_for;
    new(dom, l)
end_proc;

set_index := proc(x: dom, i: Type::PosInt, v: Type::NonNegInt) begin
    extsubsop(x, 1=subsop(extop(x,1), i=v))
end_proc;
```

Moreover other methods are defined for which a default implementation exists but that can be more efficiently or "better" implemented here. "convert_to" and "TeX" can be better implemented. The method "convert_to" can convert a multi-index into a list or sequence. "TeX" returns a string with the index components in brackets:

```
convert_to := proc(x: dom, T) begin
    case T
    of DOM_LIST do return(extop(x,1));
    of "_exprseq" do return(op(extop(x,1)));
    of dom do return(x);
    end_case;
    FAIL
end_proc;

TeX := proc(x: dom) begin
    "(".expr2text(dom::convert_to(x,"_exprseq")).")"
end_proc;
```

The methods "intmult" and "_negate" can be implemented more efficiently. "intmult" multiplies an index with a non-negative integer. The opposite of an index is calculated by "_negate". But an opposite exists only for the zero index:

```
intmult := proc(x: dom, i: Type::NonNegInt) begin
    if i < 0 then error("negative factor") end_if;
    new(dom, map(extop(x,1), _mult, i))
end_proc;

_negate := proc(x: dom) begin
    if dom::iszero(x) then dom::zero else FAIL end_if
end_proc;
```

These are all the methods defined by `MultiIndex`.

# Bibliography


[1] K. Drescher. "The Constructors of the domains Package". *Automath Technical Report* No. 2, Univ. GH Paderborn 1995.

[2] S. Wehmeier et. al. "The Domain Constructors of the MuPAD Library". *MuPAD Library Document* No. xx, SciFace Software 1999.

[3] K. Drescher et. al. "The Category Constructors of the MuPAD Library". *MuPAD Library Document* No. xx, SciFace Software 1999.

[4] K. Drescher et. al. "The Axiom Constructors of the MuPAD Library". *MuPAD Library Document* No. xx, SciFace Software 1999.

[5] D. Gruntz. "Groebner Bases in GAUSS". *MapleTech*, (9):36–46, 1993.

[6] D. Gruntz, M. Monagan. "Introduction to GAUSS". *MapleTech*, (9):23–35, 1993.

[7] R.D. Jenks, R.S. Sutor. *AXIOM, The Scientific Computation System*. Springer, 1992.