Graphics

General

Any property of a plot may be set interactively in ***VCam*** or defined in a plot-command.

VCam

Graphics front-end for 2- and 3-dimensional plotting, running under X-Windows and MacOS; thightly coupled with the ***MuPAD*** kernel. Defining functions may be entered and manipulated directly in ***VCam***, objects can be added or deleted by mouse-click.

2D-Plotting Facilities

Two-dimensional scenes, graphs, curves; multiple objects per scene; object style may be points, lines or impulses; options for titling, labeling, colour selection, smooth colouring by height, axes scaling and style, automatic viewing box, grid and smoothness.

3D-Plotting Facilities

Three-dimensional scenes, graphs, surfaces, space curves; multiple objects per scene; object style may be points, lines, impulses, wire frame, mesh, x-line, y-line, hidden line, colour patches, transparent; options for titling, labeling, colour selection, smooth colouring by height, axes scaling and style, automatic viewing box, camera point and focal point, grid and smoothness.

Programming Facilities

General

Functional programming language with procedural extensions and Pascal-like syntax. Procedures and statements are ordinary data objects and may be manipulated at run-time. Any language construct is available as system function. Any system function may be overwritten by user-defined procedures. Unnamed functions. Formatted output of procedures.

Language Constructs

Assignment, expression, statement sequence, **`if… then… elif… else… end_if, for… from… to… step… do… end_for, while… do… end_while, repeat… until… end_repeat, case… of… do… otherwise… end_case, proc()… begin… end_proc, break, next, error, return`**, parameter lists of varying length.

Evaluation

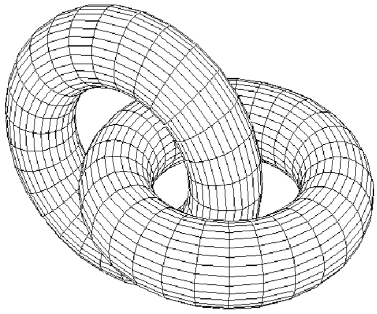
The programmer can control any aspect of expression evaluation and variable substitution: holding evaluation, evaluation in an outer context, defining substitution level, additional evaluation, retrieving values directly. Procedures may remember calculations to speed up execution. Results are stored in a history table which may be accessed by procedures.

Input and Output

Reading and writing of text and binary (M-Code) files, opening and closing of files, reading and writing of ***MuPAD*** data and raw text, printing output to file, writing a session protocol to file. Entering expressions or text interactively.

Debugging

Execution tracing, verbose procedure execution, different printing and error levels, error trapping. A source code debugger allows to step through procedures, print and observe values of variables, alter values of variables, execute instructions, display the procedure stack. The debugger is integrated into the kernel and may be used via command language or a comfortable window-based user interface (available under the X-Window-System).



Parallel Programming

General

There are two levels of parallelism in ***MuPAD*** : micro-parallelism on a processor cluster with high speed communication facilities (a multiprocessor machine for example) and macro-parallelism on a network of clusters with medium- to low-speed communication.

Micro-Parallelism

Automatic task creation and distribution by the system. Easy-to-use language constructs: parallel for-loop, parallel and global variables, task abortion.

Macro-Parallelism

Blocking and non-blocking message queues and pipes, global network-wide variables, topology informations.

Availability

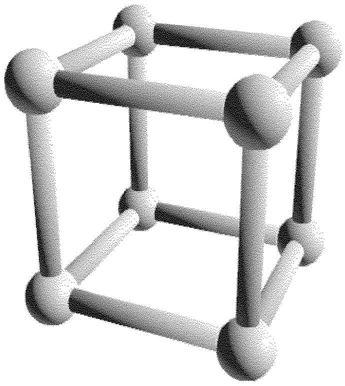
MuPAD Version 1.2 is now available for Sun SPARCstation (under SunOS and Solaris), IBM RS6000, Silicon Graphics, PC's (under LINUX) and Apple Macintosh. Prereleased versions exist for 80386-based MS-DOS systems and Sequent Symmetry. A version for SUN Multiprocessor systems is in preparation. Please call for others.

Documentation

MuPAD Benutzerhandbuch, (in german) by B. Fuchssteiner et. al., Birkhäuser, Basel, 1993.
MuPAD Tutorial, (in english) by B. Fuchssteiner et. al., Birkhäuser, Basel, 1994.
(English manual will soon be available.)

Further Information

MuPAD-Distribution, Fachbereich 17, University of Paderborn
D 33095 Paderborn, Germany.
E-mail: MuPAD-distribution@uni-paderborn.de



MuPAD

Multi Processing Algebra Data Tool

Fact Sheet

MuPAD – a system for symbolic and numeric computation, parallel mathematical programming and mathematical visualization.

Overview

The program consists of a small kernel written in 'C' and Libraries written in the ***MuPAD*** programming language. All features described may be used interactively or via the programming language. Control structures for programming are also available at the interactive level. Kernel and library are machine independant. In addition to the terminal-based user interface window-based interfaces exist for the X-Window-System and the Apple Macintosh.

Th***e MuPAD*** programming language has various built-in constructs for parallel programming, accessible at any level of programming experience. First parallel implementations exist.

MuPAD is devoted to the scientific community and freely distributed for non-commercial use.

Symbolic and Numeric Computation

Data Types

Integers and rational numbers of any length, floating point and complex numbers of arbitray precision, multivariate sparse polynomials, strings, Booleans, variable identifiers, dynamic lists, sets and tables, arrays of any dimension,
user defined types (domains) and objects, a rich set of expressions, procedures and statements, which are first class data objects.

Constants

E, PI, I, FALSE, TRUE, various system constants.

Data Manipulation

Operand extraction, counting and substitution, object type testing, indeterminate extraction, concatenation of strings and lists, set manipulation, extraction, sorting and substitution of list, set, table and array elements, mapping operands to functions, string searching, pattern matching, parsing, conversion of strings and polynomials to expressions and vice versa, run-time generation and manipulation of procedures.

Operators

Arithmetic (+, -, *, /, ^, mod, div), relational (<, >, <=, >=, =, <>), Boolean (and, or, not), set operations (union, intersect, minus), concatenation (., @), range (..), sequencing (, and $), domain attribute access (::), assignment (:=), last output (%), user defined operators. System functions and operators may be overloaded by user defined types.

Arithmetic

Absolute value, exponential, logarithm, square root, sign, trigonometric and hyperbolic functions and inverses, rounding, ceil and floor, truncation, modular computation, factorial, maximum and minimum, numeric evaluation, extraction of numerator and denumerator, fractional part, real and imaginary part, set uni-on, intersection and difference, set membership, differentiation and expansion of expressions, taylor series and order terms, differential operator.

Polynomials

Coefficient mapping and extraction, degree, leading coefficient, term and monomial (lexical and degree order), trailing coefficient, number of terms, nth coefficient, division and pseudo-division, evaluation, integer content, norm, factorisation.

Domains

User defined types and data structures, operator overloading, object oriented programming, generic algorithms.
As experimental library: parametric domains, domain constructors, categories, category constructors, axiomes.

Number Theory

Integer factorisation, integer GCD and LCM, extended Euclidean algorithm, prime number test, prime number generation, Eulers phi function, random number generation.

User Interfaces

General

Selection of printing and error level, line width, formatted or non-formatted output, verbose printing during procedure execution, recalling of earlier results. Line editor for raw terminal interface.

X-Windows Version

Look and feel of OpenLook, editing and recall of input and output, scroll back, two- and three-dimensional plotting with mouse-controlled interface, on-line manual and tutorial with hypertext functionality, pasting of examples from these documents into the input window, mouse-controlled interface for source code debugger.

Macintosh Version

Mac-like interface with notebook functionality, structured session documents, editing of text documents, graphics and hypertext system as in the X-Windows version.

```
local lp, lq;
begin
  if args(0) = 2 then
    o:= hold(DegreeOrder)
  end_if;
  if testargs() then
    if args(0) <> 3 and args(0) <> 2 then
      error("wrong no of args")
    end_if;
    if not isa::rational_poly(p) or not isa::rational_poly(q) then
      error("not a rational polynomial")
    end_if;
    if op(p, 2..3) <> op(q, 2..3) then
      error("polynomial types don't match")
    end_if;
    if o <> hold(DegreeOrder) and o <> hold(LexOrder) then
      error("unknown order")
    end_if
  end_if;

  # get LCM of leading terms #
  lp:= lterm(p, o);
  lq:= -lterm(q, o);
  groebner::term_lcm(lp, lq);

  # the S-polynomial #
  divide(%1, lp, Exact) * p + divide(%1, lq, Exact) * q
end_proc:

#--

groebner::term_lcm -- returns LCM of the terms p and q

groebner::term_lcm(p, q)

p,q - terms (polynomials mit one non-zero coefficient)

--#

groebner::term_lcm:=
proc(p,q) local n, i;
begin
  n:= nops(op(p, 2));
  degreevec(p);
  degreevec(q);
  max((%1)[i], (%2)[i]) $ i=1..n;
  poly(_mult(op(p, [2,i])^(op(%1,i)) $ i=1..n), op(p, 2..3))
end_proc:
```

Messages should only be returned in the case of an errors. If the results are valid then no error message should be returned with `error`. Such messages can be very disturbing when a function is called from another function. If a function in certain circumstances cannot return a solution it should return the message `FAIL`. If a linear system of equations is, for instance, not solvable the solution `FAIL` should be returned, but no error message should be given. All messages should be given in English.

File Names

When giving file names - perhaps for the functions `read` or `loadproc` - the path sould always be created with the function `pathname` so that it is independent of the operating system. In UNIX the expression

```
pathname("GROEBNER")."spoly.mu"
```

returns the string "`GROEBNER/spoly.mu"` and on a Macintosh it returns the string "`:GROEBNER:spoly.mu"`. By using `pathname` the library can be used with both operating systems. The individual componants of a pathname should not be longer than 8 characters (MS-DOS), and the use of capital letters in the name should be of no importance. Special characters such as /, \ and : must not be used. More details are to be found on the help pages of `pathname`.

Comments

Comments should be written in English so that our foreign collegues can also understand them. Each function should be given a description containing the following points: What does the function do? Which algorithms were used? Especially important: Where do the algorithms come from (the source should be given)? What significance do the parameters have? What is the return value of the function? Which side-effects does the function have? Which global variables have been used? In the function itself the individual steps and all local variables should be described. The importance of the global variables should also be explained. Furthermore, the conditions under which the variables can be loaded or even altered, and any possible side-effects, should be described. The use of "metacharacters" in comments has proved successful. These can for example be used to extract certain comments from the source. In the `groebner` library the description of the interface functions are in comments of the form `#++...++#`. In contrast the description of internal functions are in comments of the form `#--...--#`. Such comments can easily be extracted from the sources e.g.by using `awk`.

Appendix: The groebner Library

```
# groebner.mu -- the 'groebner'
library for Groebner-Bases #

#++ User Functions:

spoly - returns S-Polynomial of p
and q

greduce - reduces polynomial p
modulo polynomials in list S

gbasis - returns reduced Groebner-
Basis for polynomials in list S

The polynomials must have rational
coefficients. ++#

groebner:= domain():

groebner::name:= "groebner":
groebner::info:= "Library
'groebner': Calculation of
Groebner-Bases for polynomial
ideals":

groebner::interface:={hold(spoly),
hold(greduce),hold(gbasis)}:

loadlib("isa"):

loadproc(hold(groebner::spoly),
pathname("GROEBNER")."spoly.mu"):

loadproc(hold(groebner::greduce),
pathname("GROEBNER")."greduce.mu"):

loadproc(hold(groebner::gbasis),
pathname("GROEBNER")."gbasis.mu"):
```

The File spoly.mu

```
#++

spoly -- returns S-polynomial of p
and q

spoly(p, q [,o])

p,q - polynomials over the
rationals

o - term ordering (LexOrder or
DegreeOrder)

++#


groebner::spoly:= proc(p,q,o)
```

file contains MuPAD commands with which the files of the library can be loaded as required. For this the function `loadproc` is used:

```
dir:= pathname("GROEBNER"):
loadproc(hold(groebner::spoly),
dir."spoly.mu"):
```

Here `spoly` is a method of the library domain `groebner`. The method `spoly` is itself defined in the file `spoly.mu` in the directory `GROEBNER`. Only on calling `groebner::spoly` the file `GROEBNER/spoly.mu` is loaded and the function executed.

Orginization of the Files of a Library

Among the directories contained in `READ_PATH` or `LIB_PATH` there must be a directory called `LIBFILES`. This directory contains the library file for the library which is to be loaded with `loadlib`. A library file should have the same name as the relative library supplemented with the suffix `.mu`. As already mentioned it must be contained in a sub-directory with the name `LIBFILES`. Furthermore, each library should have its own directory containing the files of the library. This should be contained in the directory that contains the directory `LIBFILES`. The directory name should be the name of the library written in capital letters. The files of the library `groebner` are contained in the directory called `GROEBNER`. The directory contains the three files `spoly.mu`, `greduce.mu` and `gbasis.mu`. Altogether there are the files listed below:

Library-File	Implementation
LIBFILES	GROEBNER
groebner.mu	spoly.mu
	greduce.mu
	gbasis.mu

Tabelle 1: Library `groebner`

Directions for Programming

Testing Parameters

During the programming of procedures the user will often be confronted with the problem that the arguments must be exactly tested when a procedure is called. This is of course time-consuming. If however the procedure is called by other procedures the user can be fairly sure that the parameters are correct (except in the case of any programming errors). In this case an expensive "testing" of parameters is no longer necessary. One solution of this problem is to always write two procedures: one to test the parameters and an auxillary procedure to do the actual work (and which can be directly called from other procedures). This, of course, leads to the problem that the user has twice as many procedures as before and that the auxillary procedure must also be documented otherwise only the programmer himself can use it. Furthermore there are problems with the domains. Here the user will want to call the auxillary procedure directly, if possible. This will, however, not be possible because in general the user does not know the name of the auxillary procedure.
The function `testargs` offers a solution to this dilemma. With the aid of `testargs` the user can determine if a procedure has been called by the user directly or by another procedure and execute the relevant test. The expression `testargs()` returns `TRUE` if and only if it is executed in a procedure that was called from the interactive level - in other words by the user. If `testargs` is called in a procedure which was in its turn called by a procedure `FALSE` is returned. The user can control the testing of parameters as follows:

```
p := proc(x) local ...;
begin
if testargs() then
   Test parameter x ...
end_if;
   The actual algorithm follows here...
end_proc;
```

The condition "normally" was made above: testargs has two states, the *normal status* described above and a *test status*. In the test status testargs() always returns TRUE, i.e. the parameters are always tested. This is useful while searching for errors during programming. More details are to be found in the help pages of testargs.

Error Handling

Only those library functions listed by the function `info` can be exported: in this case only `gbasis`, `spoly` and `greduce`. To export all the functions of a library the user can simply enter the library in the `export` command:

```
>> export(groebner); "
Warning: 'spoly' already has a
value, not exported"
```

The warning is given because the identifier `spoly` already has a value (`spoly` has already been exported). In such cases the function is not exported. If a function whose identifier already has a value is to be exported then this value will have to be deleted:

```
>> spoly:= NIL:
>> export(groebner, spoly);
```

This is all that the inexpierienced user needs to know about libraries. However, for the implementation of libraries there are certain conditions which must be met to obtain the described functionality.

Conditions for Libraries

There are two aspects which must be heeded when implementing a library: The first aspect concerns the naming of the library functions. In order to avoid name clashes libraries are implemented as special domains, so-called *library domains*. The functions of the library interface are implemented as methods of the relative domain. If required the names of the interface functions can be made globally known by using the method `export`. The second aspect concerns the loading of a library. To make the functions of a library available they have be defined in a so-called *library file*. The MuPAD function `loadlib` makes use of this library file to load the library.

The Library Domain

The functions and variables of a library should be packed in a library domain so that name conflicts can be avoided. Each library domain should contain the entries "`interface`" and "`info`":

- `interface` is a set containing the names of the methods of the domain which can be directly called by the user as library functions. In practice, these methods form the interface of the library.
- `info` contains a string with information about either the library or a procedure.

The function `export` uses the entry "`interface`" to make the names of individual or all functions of the library interface globally known. The names of the interface functions and the entry "`info`" are used by the function `info` to return information about a library. We shall continue using the library `groebner` as an example. This library makes the three functions `spoly`, `greduce` and `gbasis` available to the user. The library domain and the methods of the domain are defined in the relative library file `groebner.mu` as follows:

```
groebner:= domain():
groebner::name:= "groebner":
groebner::info:= "Library
'groebner': Calculation of
Groebner-Bases for polynomial
ideals":
groebner::interface:={hold(spoly),
hold(greduce), hold(gbasis)}:
```

The entry "`name`" is used for the output of the library domain itself. The identifiers of the interface functions are bracketed with `hold`. This is a safety measure in case one of the identifiers should already have a value when the library is loaded. Otherwise not the identifier but its value is entered in the set.

The Loading of Libraries with loadlib

Normally a library is not implemented in one file but in many different files. One of the reasons for this is that the user will not want to load all the functions of a library but only those that are really necessary. This avoids the memory becoming blocked with unnecessary functions. With the function `loadlib` only the actual library file of the library is loaded. At the same time care is taken that the library is not loaded more than once. The library domain is defined in a library file, furthermore all functions related to the library with the relevant files are made known. These files are loaded on demand. The MuPAD function `loadproc` makes it possible to define functions so that their definitions are loaded from a file at the first function call. The mechanism of `loadlib` is really quite simple. In the example above the library file `groebner.mu` is simply loaded from the directory `LIBFILES`. In its turn this

The Structure of Libraries

Klaus Gottheil

In this document the conditions for the construction of libraries are proposed.
This concerns the conditions for the construction of library functions and libraries, the loading of routines and the file structure of libraries. A certain amount of uniformity in the construction of libraries leads to a great increase in their maintainability and makes the user's life easier by ensuring that the libraries can all be used in a uniform way.

The Libraries from the User's Point of View

Firstly we shall describe how the user can use a "typical" library. We shall use the library `groebner` as an example. This library is used to calculate Groebner Bases. Before the user can use the groebner library he must load it by using the `loadlib` command:

```
>> loadlib("groebner");
TRUE
```

The `TRUE` tells us that the library has really been loaded and that no errors have occured. If the library is loaded a second time with the command `loadlib` (perhaps indirectly through a read command), it is not re-loaded. A second call of `loadlib` results in:

```
>> loadlib("groebner");
FALSE
```

The result `FALSE` shows us that the library has not been re-loaded. If an error occurs during the reading of a library `loadlib` returns an error. After the library has been loaded the identifier `groebner` has a value:

```
>> groebner;
domain("groebner")
>> info(groebner);
"Library 'groebner':
Calculation of Groebner-Bases for
polynomial ideals"
"Interface:"
gbasis spoly greduce
```

The identifier groebner has a domain as its value. Because this domain is used especially to manage a library it is called a *library domain*. By using the function info the user can obtain information about the library domain. The library functions are entries in the Library-Domain. The library groebner makes only three functions available externally: gbasis, spoly and greduce. These are those *interface functions* shown by info. However, these functions are not stored under their own names. After loading, the function spoly, for instance, is not yet known:

```
>> spoly;
spoly
```

The function `spoly` is entered in the library domain under the index "`spoly`":

```
>> groebner::spoly;
proc(p, q, o)
local lp, lq;
begin
if args(0) = 2 then
  o:=hold(DegreeOrder)
end_if;
if testargs() then ...
```

This has the advantage of avoiding a collision between the names of different libraries. The user can use the functions as normal with `groebner::spoly`:

```
>> read("gbasis1"):
>> groebner::spoly(p1,p2,LexOrder);
poly(y^2*x*z*(-2)+y^2*5-y*x*z^3,
[y, x, z])
```

The user will often not want to use the function as the fairly clumsy `groebner::spoly` but will simply want to use the name `spoly`. To do this the name `spoly` must be exported. By using the function `export` the name of a library function becomes globally known:

```
>> export(groebner, spoly);
>> spoly(p1, p2, LexOrder);
poly(y^2*x*z*(-2)+y^2*5-y*x*z^3,
[y, x, z])
```

therefore, on using the second association method all entries are also found. A further example of an appropriate use of associative memory is the treatment of strings described in the next section.

The Interpreter

In order to work with the constructs of the MuPAD programming language the system contains, as already mentioned, an interpreter for executing the instructions and procedures. The main task of this interpreter is the construction of a stack for the different global and local variables and the formal parameters of the user defined procedures. Apart from this the retraction from procedures and the termination of loop constructs must be regulated. For the execution of both of these the same mechanism can be used because the program contructs do not differ in their internal presentation from the system functions. The organization of the stack plays a central role. Basically, this is set up by the relevant function calls of the evaluator which are fitted into one another like boxes. With this, for instance, the storage of the return address on leaving the procedure is unnecessary. A stack which stores all the values for assigned and local variables in each active procedure must be built up for the implementation of the dynamic scope rule. The stack of each variable is appendixed to the data structure of the appropriate identifier or the string under it. This is only possible because by the exact usage of the unique data representation for strings, two identical objects of the type `MMM_STRING` are never physically separately stored. To enable this, a further associative memory is formed at the system start and this contains all these objects. It is relatively simple to control the conformation to the unique data representation because new strings can only be generated by the concantenation of identifiers and strings or by the parser. This is carried out by each new string being controlled against the contents of the appropriate associative memory. The stack of an identifier (simplified) consists of a simple chained list that is used as a stack by objects being attached to the beginning of the list or dequeued. This takes place on entry into a procedure or when it is left. The individual entries contain the current valid value of the identifier. The reading of this value can be carried out by simply accessing the upper list element. This can always be quickly accessed because in the case of the value reading the relevant identifier is present under which the relative string and with this the stack are found. An additional numbering of the individual boxes allows the controlled dequeuing of parts of the stack, as is necessary, for instance, in the case of a run time error. In the parallel version of the kernel this concept must be extended, as different tasks have different values for one and the same variable. With this, different tasks can access common global variables. In order to realize this, each variable has many stacks, one for each cluster and processor. Due to skilful storage of its own stack depth and that of its predecessor, together with the number of the relevant processors, each task can efficiently access the contents of a variable.

Only in the case of identical signatures is an actual comparison of the data by recursively comparing the node's sons necessary. A good distribution of signatures makes certain that this case only seldom arises and that an actual comparison is rarely necessary when the expressions are not identical. The comparison of algebraic data is necessary in many modules of the kernel. The use of signatures in the kernel is of central importance in the simplifier during the sorting of operands, by the assimilation of identical sons and by the realization of the associative memory. The associative memory used here is realized with the aid of hash tables and, therefore, must always carry out a comparison to definitely identify an object.

Associative Memory

On the kernel level a separate module implements associative memory with various associative mechanisms available. Here any data can be joined with another in an associative memory. This offers a wide range of uses.

One area of usage for this memory is sets and tables. Sets are realized in the form of associative memory, as each element of the set is associated with itself. On entering a new element in a set it is first checked to see if a value is already associated with the relevant element. If this is not the case then the new datum can be entered into the set. Analogue is the implementation of tables. Here the index of an entry is associated with the relevant value. At the same time this memory has the possibility of giving itself a signature, dependent on its entries. This is necessary because a set or table is a valid algebraic datum and therefore must also be compared with other data.

Another wide field of use of this data structure is the evaluator. Here the many associative memories lead to a considerable acceleration. One phenomena in computer algebra is that identical subexpressions are repeatedly present due to substitution and evaluation and, therefore, must be repeatedly evaluated. To prevent the multiple execution of one and the same procedure associative memory is used on many levels in the evaluator to remember already evaluated algebraic data together with their solutions. When an algebraic subexpression is evaluated the first step is to test, with the aid of such memory, if the appropriate expression has already been evaluated. If this is the case, the solution can be directly taken over and the evaluation is finished. In the other case evaluation is carried out and finally, together with its solution, added to the associative memory. Unfortunately, the contents of such an associative memory are deleted after an assignment has been carried out because with this the values of variables change and the existing association can become invalid. Generally this affects only a small number of entries but a more exact analysis is hardly possible in a reasonable time due to the complex interdependencies. However, after the execution of an assignment, the variable to which a value is assigned is examined with the aid of a time stamp to see when it was last substituted by the evaluator. If this took place before the last deleting of the memory then a further deleting is not necessary.

A further special feature are the different association methods made available by the associative memory. In the first case, on reading from the associative memory, an equivalent entry is looked for in order to obtain its bound value. For this it is only necessary that the given datum and the relevant value in the memory both represent the same datum. In contrast to this in the second association method the data must be also physically identical, i.e. only a comparison of the physical addresses is carried out during the search for an entry. Naturally, this means a remarkable acceleration, but it also means that not all associations are recognized. This means that this method cannot be used in every area. An implementation of sets with this method leads inevitably to errors. Conversely this method is appropriate in the evaluator because if no association is found here the normal evaluation process is begun and the solution is not influenced. In this case only the execution time is lengthened. These two methods should be carefully considered in evaluations. On the one hand, in the second method more data must be evaluated, but on the other hand, the amount of searching in the associative memory is less. Whichever method is used is also influenced by the extent to which the unique data representation is observed. An exact observance leads to two identical objects always having the same physical address and,

expression operands is carried out on the basis of this order.
However, an evaluation can also be the analysis of an indexed identifier, where a value must be read from a table, array, or list. Tables, arrays, lists or sets also need to be evaluated before they can be worked with further. In the case of lists and sets this takes place by the evaluation of their elements. If at least one operand has been changed then the solution is stored in a new object. In all other cases only a logical copy is necessary.

Evaluation of the System Functions and Operators

The evaluation of the system functions and operators is carried out in MuPAD by a special mechanism. The basic idea is that the functionality of a system function or an operator stored object can be reached by a simple substitution of the relevant identifier. For this purpose this functionality is stored with the help of a so-called *function environment*. This is a separate category type that contains three sons. The first is responsible for evaluation, and the second controls the output of the relevant system function or algebraic expression. The third entry may consist of a table in which further characteristics can be stored. This is information about the type of expression build with this function environment or a procedure to differentiate or expand the expressions of this type. The first two sons consist in general of objects of the type `CAT_EXEC`, whose operands can consist of normal algebraic expressions or C functions. For this purpose the memory management offers the possibility of directly integrating C functions in special objects so that these can be used as Mammut objects. These C functions are used for the actual execution of evaluation or output. To support these tasks the other operands of a `CAT_EXEC` node contain information, like the operator priority, the system function name or the appearance of the operator symbol for the output. The name of the system function is, for instance, necessary to be able to return function calls which cannot be evaluated further. Alternatively the first or second son of a function environment can consist of a user defined procedure which then takes over the evaluation or output.
The function environment is stored in the stack under the relevant function name. At the start of the evaluation of a system function the left hand operand is evaluated to such a function environment. For this only a normal evaluation of the relevant operand is necessary. From the resulting structure the C function to be called or the procedure definition is calculated and carried out with the original function call as a parameter. If this is a C function then the operands which are attached to the relevant son of the function environment are passed on as further parameters. The result of this call is also the solution of the evaluation.
With this mechanism it is also possible to globally change an operator or a system function by assigning the function name to another or a changed function environment. The function environment can also be changed in order to alter, for instance, only the output of an operator.

Signatures

A basic concept of the kernel consists of the signatures. These are, as already described, a sort of hash value which is assigned to each datum subexpression in the kernel. The signatures make sure that two identical data structures are given the same signature value. Inversely identical signatures must not mean identical expressions. Apart from this signatures define a simple (and artificial) order on the set of all algebraic objects. This order is not a generalization of the lexical order of identifiers. The calculation of a signature value for any algebraic datum is carried out during the evaluation or simplification due to MuPAD only having the characteristics described above for data in normal form. Here, starting in the leaves of the relevant tree, the signatures are written in the nodes. The signatures of the leaves (identifier, strings and numerical constants) are calculated from the sequence of ASCII characters or from the signatures of the PARI data structures. The expression signatures that consist of many subexpressions are calculated with the help of the signatures of the individual subexpressions.
The signatures are an important aid in the comparison of expressions. In this case the signature values can be compared. If these are not identical then the expressions are also not identical.

for supporting the evaluator are stored. All the fundamental functions of the evaluator work on the level of objects belonging to the type `MMM_CATEGORY`, i.e. both the input and output of the functions consist of objects of this type. In order not to break this rule there is a separate category `CAT_NULL`, which corresponds to the `NULL` pointer.

The category `CAT_DEBUG` plays a special role. It is used to store information for the debugger. It enables the debugger to assign instructions in a program to a given text passage in the source text file. For this purpose data such as the line number and the file name of the respective file are stored. When the evaluator is in normal operation, i.e. the debugger is turned off, these categories are ignored or simply not produced. Only in the debug mode does the evaluation of such a category lead to a reaction by the integrated debugger or to communication with the mdx-tool, if debugging is being carried out with its help.

Due to the excellent construction of the data structures the realization of the manipulation functions such as `op` and `subsop` can be greatly simplified. As a rule the i-th operand can be obtained by accessing the i-th son of a node. In this case it is generally not necessary to examine the given datum in any more detail. The expression type is also determined by the first son. At this point it becomes especially clear how a carefully chosen data structure can simplify the necessary algorithms. Each special case in the data structure, which deviates from the rule above, leads to additional work in the realization of the manipulation functions. This is similarly true for many algorithms in the kernel.

The Evaluator

The evaluator tasks are the substitution of identifiers, the execution of system functions and the simplifying of expressions. The execution of user defined functions is also induced by the evaluator. The actual execution of the program is taken over by the interpreter, that in its turn needs the evaluator. Therefore both these modules cannot be clearly separated. Also reading from and writing in, for instance, tables and arrays is carried out by the evaluator. A further important evaluator task is the transformation of algebraic data into normal form, because only this enables the efficient comparison of algebraic objects. For this purpose the internal simplifier is used.

To assist the evaluator several associative memories are used. These prevent the multiple evaluation of the same datum and in this manner greatly increase the system speed.

Generally the evaluated solution of an algebraic expression is an entirely new expression. However if parts of the old datum can be used again because they have already been evaluated, i.e. not changed by evaluation, then the relative subtrees are logically copied and inserted in the tree depicting the solution. This ensures that the same datum is as often as possible not multiply stored. The logical copying is carried out by the memory management. Generally the system does not strictly adhere to the unique data representation. Should this happen then on each change in a datum and each generation it must be examined to see if a corresponding datum is already present as a data structure or if a tree exists which contains the relative subtree. The effort required for this is certain to exceed any use that can be obtained from it. Therefore the strategy in the kernel is to use the equality of a datum only when it can be recognised without further effort. One of these cases is the procedure for evaluating operands as described above. In the same way during a substitution the value of an identifier is read from the stack and the identifier is replaced by a logical copy of this datum. Naturally it may be necessary here to further evaluate the new value dependent on the substitution depth.

The process of expression evaluation is carried out recursively. To evaluate a sum the evaluator is firstly applied to each term. If they are operands of a commutative operator the evaluated sons are then sorted to make simplification in the next step easier. In this case such simplification consists of, for instance, the contraction of terms to a product. Equal terms can be easily recognized after sorting because they must be neighbouring operands. Numerical constants are added in exactly the same way and the solution is written as the last term of the sum. The signatures already described above, define an order on the set of algebraic expressions in a natural manner. The sorting of the separate

Here the nodes of the tree are formed by the Mammut objects and the references which originate in these objects (see above) are used as pointers between the individual nodes. This structure comes very near to that of the mathematical interpretation of an algebraic expression as an operator with any number of operands. Here the operator and operands are depicted as sons of a node. In this case the first son is interpreted as the operator. In order to ensure the already described uniformity the attempt is made to store nearly all algebraic data in such a functional form. Therefore, all operators of the system are functionally stored. The expression

```
a+b+c
```

is represented by the function call

```
_plus(a, b, c)
```

and consequently by a node with four sons `_plus, a`,`b` and `c`.

The work of the kernel and especially the evaluator consists on the one hand of evaluating expressions of any form and on the other hand of carrying out instructions such as assignments, loops and conditional branches. The interpreter, which is integrated in the evaluator, carries out procedures in exactly the same manner. This occurs automatically during the interactive input of instructions or on calling programs.

As, from a logical point of view, there is no appreciable difference between algebraic data and program constructs, such as procedure definitions and instructions, there is no difference on the data structure level between these objects. In this manner instructions and procedures are also functionally stored, so that an assignment of the form

```
a := b
```

is depicted as

```
_assign(a, b)
```

and therefore as a node with the sons `_assign, a` and `b.`

With this method it is possible to manipulate instructions and programs exactly like expressions. As a consequence these can be passed to functions, decomposed and reconstructed in any required order. The internal representation is directly accessible by the system user. For this purpose the so-called underline functions, such as `_plus` or `_assign`, can be used in the input. Lists, sets, tables and arrays deviate from the functional storage. Sets are realized with the help of associative memory, which is, of course, built of Mammut objects. Lists consist of one object whose references can all be interpreted as list elements. Arrays of various dimensions also consist of such list-like structures which may consist of several levels.

However, a node in the data structure of an algebraic expression does not only contain references to other nodes. It also contains information which takes up the same amount of memory space in each node. In this information is, amongst other things, the so-called *signature*, which gives a hash value for the subtree formed by these nodes and their sons. The usefulness of these signatures will be described in more detail in connection with the evaluator. As a data area of constant size which is present in every data object is being dealt with here, it is useful to include it in the static part of a node, in the header. As the header, as already mentioned, cannot be moved and, unlike the other parts of an object, can be reached by a simple reference, the signature can be very efficiently accessed.

The objects of the system management are subject to strict typification. This internal type must be given at the generation of each new object. There exists, for example, the type `MMM_PARI` which is used to store a numerical object of the PARI[2] packet, which realizes the arithmetic in the system. Another important type, `MMM_STRING`, is used for depicting any character string. Dependent on the node above, this can be interpreted by the system as an identifier or a string. Most of the objects have the Mammut type `MMM_CATEGORY`. From the point of view of the kernel these objects represent complete algebraic data and are once again typified. All objects of the type `MMM_CATEGORY` also have a uniform construction in their data part and differ only in the number of references which have been included in them. Apart from these references and one entry, which is used for typification, category objects contain a bit-array in which the datum sign and various flags

2. PARI is an arithmetic packet from C. Batut, D. Bernardi, H.Cohen and M. Olivier

prevent memory fragmentation, a typical problem of computer algebra systems. With this repeated demanding and releasing of different sized memory objects, a mixture of large and small, used and unused areas arises, so that memory demands for large objects cannot be satisfactorily met, although enough memory is available.
This can be avoided by moving the individual objects. In this way the many small free areas can be combined to a few larger areas for future use. Unfortunately this mobility of data partitions has the disadvantage that a double reference, a so-called *Handle*, is necessary in order to access its contents, due to the memory location of the relevant data being changed at will. By using the handle this problem is avoided. Another possibility of accessing the correct memory location is to mark certain references in this dynamic area so that their value is automatically updated when movement takes place. Both these possibilities have certain advantages when compared with the usual access with a simple reference. Which method is used in the kernel is dependent on the concrete situation. It is, for instance, certainly more advantageous during the step by step execution of a moveable area to mark the used pointers as references in the dynamic part and then to proceed as usual by counting the pointers forward and backward. The repeated access with a handle is certain to be more time-consuming.
The static part (header) of a Mammut object need not be moved because of its uniform size. This is due to the fact that each area of this form that is vacated can immediately be filled by a new header. Access to a header can be carried out with a simple reference. The header contains the reference to the dynamic part of the object, so this part contains the already mentioned handle. If the dynamic part is moved then only this reference in the relative header need be updated. In this way the user can access the moved object at all times. Among other things, the dynamic part of a Mammut object contains an area used for storing any number of references to other Manmmut objects. The size of this area, and therefore the number of references contained in it, can be determined when a new object is requested. The references are placed in sequential order in this area, so that certain entries can be accessed in a constant time. Due to this it is possible to efficiently realize the n-ary trees which are used by the kernel as data structures.
A further important aspect is the possibility of multiple logical representation of physically existing objects. With this the *Unique Data Representation* which tries to store identical algebraic expressions or parts of these only once, is supported. In the following, I would like to describe this process in more detail.
The parallel version of the system realizes the memory management of a *Shared Memory Machine*, so that the algorithms of the sequential kernel can be used unchanged in many areas. Large changes are only necessary in the realization of the stack and the implementation of the parallel elements of the programming language.

Data Structures

The choice of the appropriate data structures is of fundamental importance in the implementation of a computer algebra system. Nearly all kernel algorithms are employed in producing these objects, analysing them or altering them in one form or another. How elegantly, and therefore, simply and quickly, this can take place is to a great degree dependent on the construction of the data structures. As a consequence, the developer must clarify which sort of algorithms will work on these objects and how the working of these algorithms can be optimally supported by the data structures, before these structures are designed.
In this context an important aspect is a very near uniform construction of all data types. With this many procedures can be simplified and greater legibility and maintenance of the program can be attained. This is of great significance in a project which is undergoing continual development. A first step in this direction is the constant use of the Mammut module. Due to this, all dynamic data objects have a uniform structure that in many cases makes a more exact differentiation between data objects unnecessary. This simplifies and accelerates many algorithms
Another important point is that the data structures duplicate the inner logic of the depicted algebraic objects. From this point of view n-ary trees have been chosen as the basic structures.

Design and Implementation of the Kernel

Andreas Kemper

In the following contribution I would like to present a survey of the design and implementation of the computer algebra system MuPADs kernel. For this purpose the individual components of the system, their function and technical realization, together with the interaction of the components will be described. I will deal with the data structure for the presentation of algebraic objects and the mechanism for evaluating these data in detail. A further focal point is the presentation of the associative memory which is employed at various locations in the kernel, in order to increase the speed of the system and to bound the size of the data produced.

General Construction

Under UNIX the MuPAD package comprises of many, mostly independent processes, which, when running simultaneously, take over different tasks. The central element of these processes is the so-called *kernel*, in which the actual algebraic operations take place. The other programs are the tools, which make the use of the system easier and support the presentation of the algebraic data. All these programs communicate with the kernel and in some cases with each other.

The user can use the graphical user interface XMuPAD for support in the use of the system. To simplify the operation of the kernel's line oriented source text debugger the specially developed tool mdx can be used. This tool enables a comfortable graphical operation of the debugger. The documentation of the system is available on-line, in the form of the hypertext system HyTeX[1]. In addition there is a graphics tool for the graphical depiction of the objects handled by the system.

In the following I would like to deal only with the MuPAD kernel. This is also available in a terminal version without the surface and extra tools described above. Even this minimal version enables interactive use of the debugger realized in the kernel. Its algebraic capabilities are not in any way reduced by the other components being missing. From the users viewpoint the system consists of a cycle of user input, evaluation of the produced datum and output of the solutions. This reflects the internal realization of the kernel. In addition to the initialisation routine it carries out a loop that firstly calls the parser to read the user's input. Here the input is examined syntactically and then the appropriate data structure is built up. If this should be the reading of a program in the debug mode then information is provided for the debugger. The next, and far more complicated step, includes the evaluation of the produced data structure. The techniques used here will be described in detail later. The solution of the evaluation is returned in the form of a logically new data structure by this module. In the last step this data structure is displayed on the monitor by the output module.

Memory Management

The basis of the kernel implementation is *Mammut*, the *M*emory *A*llocation *M*anage*m*ent *U*ni*t*. This is a universally applicable memory management that optimally supports the demands of a computer algebra system. In the kernel, Mammut takes over the allocation, manipulation, and release of each dynamic object. With this Mammut provides the lowest level of the system.

Each Mammut object consists of a static part with a constant size and a dynamic part which can be independently moved by the memory management. The aim of this movement is to

[1] A development of the work group of N. Köckler at the University of Paderborn

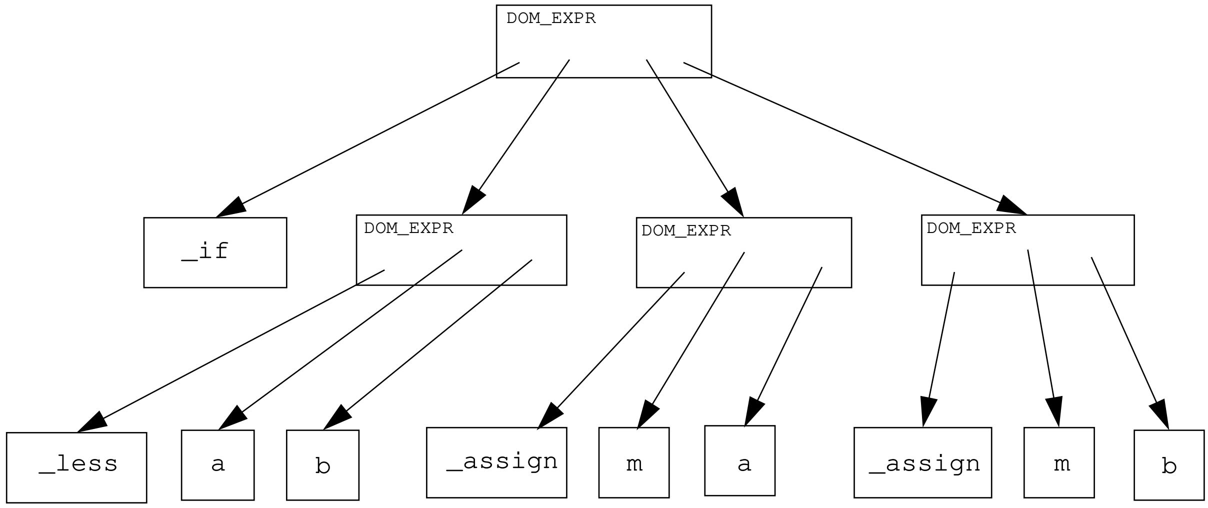


Fig. 11: The statement `if a<b then m := a else m := b end_if`

arithmetic due to the point-part of each datum consisting of a compounded memory block. Due to this a very laborious climb down to the operands, which would have been necessary with a binary oriented data structure, can be avoided. Apart from the high execution speed, a uniform data structure has a further advantage; many algorithms are simplified in the kernel. Typical examples of this are the functions used for data manipulation. Generally, the i-th operand is obtained by access to the i-th reference in the point-part of a datum. Usually it is not necessary to examine the datum more closely. By using a uniform data structure for algebraic expressions and instructions a further very important characteristic of MuPAD is revealed: The complete fusion of these objects. Apart from the natural differences in functionality in principal there is no difference between instructions and algebraic expressions. Instructions are algebraic data in MuPAD and can, therefore, be generated and manipulated during the system runtime exactly like data of any other type. For illustration we have included some examples of DOM_EXPR objects (see Figure 10 and Figure 11). The individual nodes are only represented schematically. An exact differentiation between the mem- and point-parts is not included to avoid confusion. For the same reason we have not taken the single data representation into account. Identical objects are represented multiply although internally they are only stored once.

Bibliography

[BBC*91] C. Batut, D. Bernardi, H. Cohen, M. Oliver and N.-P. Skoruppa. *The PARI-GP package.* mathPAD, 1(3):30-32, September 1991. Sonderausgabe DMV-Tagung.

[FGK+93] B. Fuchsteiner, K. Gottheil, A. Kemper, O. Kluge, K. Morisse, H. Naundorf, G. Oevel, T. Schulze and W. Wiwianka. *MuPAD Benutzerhandbuch.* Birkhäuser Verlag Basel Boston Berlin, 1993.

[Nau92] H. Naundorf. Mammut: *Memory allocation management unit.* Diplomarbeit, Universität-GH Paderborn, Juni 1992.

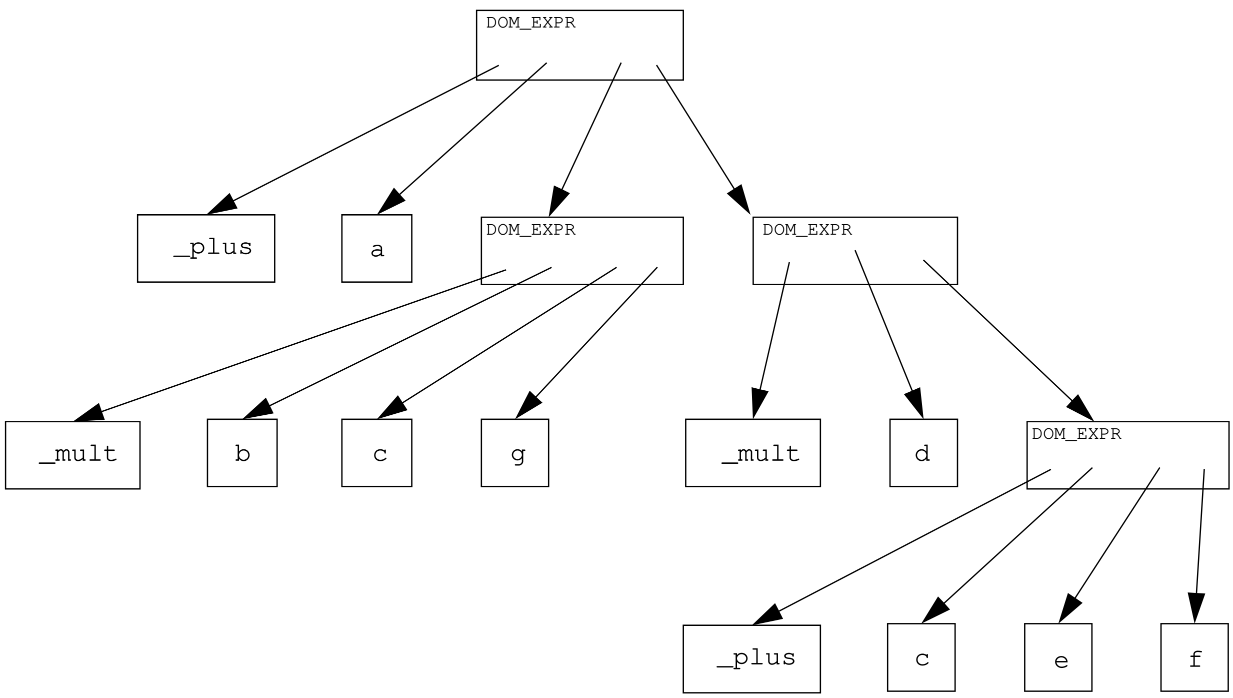


Fig. 10: The algebraic expression a+b*c*g + d*(c+e+f)

which the methods and other characteristics of the domain are stored. The other entries are also used to store methods and characteristics of the domain. In some cases access to the methods through the tables is too slow, therefore, this direct access mechanism has been made available. However, the user has no influence on this technique because it is only available inside the kernel.

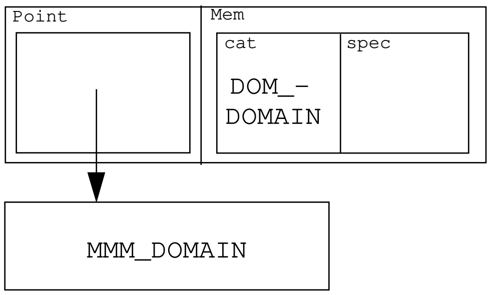


Fig. 8: Domains

The elements of a domain are stored in the type `DOM_EXT`. The `point`-part of such a datum may contain any number of entries. The first entry is a reference to the domain to which the element belongs. The other entries are references to objects which can be of any type. The user has free choice of representing a domain element.

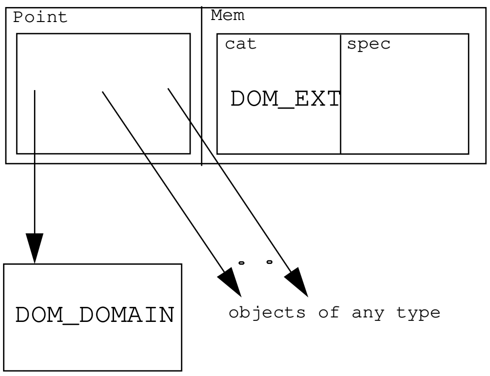


Fig. 9: Domain Element

The most general domain type in MuPAD is the expression. All algebraic expressions constructed with operators and their operands and all instructions of the MuPAD programming language are stored in objects of the type `DOM_EXPR`. The storage is done through references in the `point`-part of the `DOM_EXPR` object. For each operand of an algebraic expression there is a reference to this datum. The parts of an instruction that are stored are those components not consisting of key words. So that the evaluator can determine how it is to behave with a `DOM_EXPR` object it must be informed of the exact type of the object. For this purpose each object contains a further reference in the `point`-part. This entry, called the *zero operand*, is the first entry in the `point`-part and can refer to any MuPAD datum. This `DOM_EXPR` object is interpreted in functional notation. During the evaluation of this datum the following take place: Firstly, the type of the zero operand is checked. If this is a functional environment or a procedure the relevant routine is executed with the other sons as arguments. If there is an identifier without a value a function call which can be evaluated no further is returned. If the identifier can be evaluated, e.g. to a functional environment or a procedure it proceeds as above. With other domain types a function call is interpreted as the call of a constant function and the zero son is returned as the solution. With this a global concept for a functional programming language is realized. The zero son describes the function; the other sons are the arguments of the function. This representation, friends of LISP will be pleased to hear, is also outwardly available to the MuPAD user. For each operator and each instruction there is a functional equivalent in the form of an underline function. For instance, the addition operator has the function `_plus`; the instruction `:=` has the function `_assign`.

The chosen data structure for algebraic expressions is very near to the mathematical interpretation of such an expression; i.e. an associative operator connecting any number of operands. Storing is carried out in form of a n-ary tree. The root of the tree represents the entire expression. The following nodes are the operator and operands. The latter can be identifiers (leaves of the tree) or algebraic expressions formed by operators of a higher priority (sub-trees of the tree). The data structure of the n-ary tree, especially the identical construction of expressions and instructions, and their technical realization aid the basic routines such as the simplifier to attain a high execution speed. The individual references can be accessed by simple C address

ory is realized internally by the use of hash tables. The set elements to be stored are associated with an empty object. For tables, the table entries are associated with the relevant index. The `cat` component contains the concrete domain type.

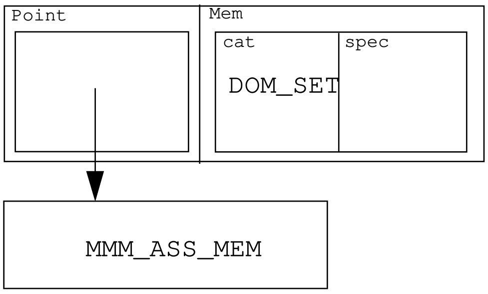


Fig. 5: Sets and Tables

The type `DOM_LIST` is used for grouping objects of any domain type to a datum while still maintaining their order. The `point`-part contains the references to the list entries which may have any domain type.

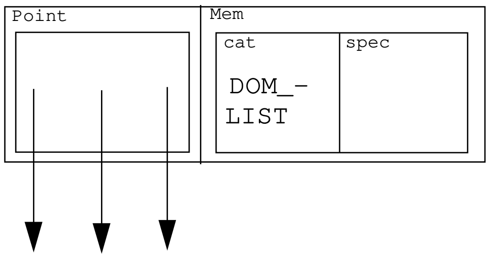


Fig. 6: Lists

The type `DOM_PROC` is used for the storage of procedures. The `point`-part of such a datum contains 6 references. The first is an `MMM_CODE` object. This is a reference to the internal routine `MEV_eval_interpreter`. This generic function is called for the evaluation of a procedure call. For this the standard arguments inside the evaluator, the procedure call in the form of an expression and the procedure definition are transferred. It updates the stack (the formal parameters are written in the stack with the values of the corresponding actual parameters, local variables are written in the stack without a value) and then evaluates the procedure body. The second entry in the `point`-part refers to an expression sequence of the formal procedure parameters. The third refers to the expression sequence of the local variables. An empty sequence is indicated by a `DOM_NIL`. The fourth entry refers to the procedure body. Generally this is an statement sequence. The fifth entry refers to the remember table of a procedure. The last entry refers to the name which in its turn can be any MuPAD datum. If these are missing `DOM_NIL` is to be found instead. The options of a procedure are stored in the `spec` component of the `mem`-part with the aid of three bits. The `OPTIONS_CHECKED_BIT` tells us that the options of the procedure have already been evaluated. The `HOLD_BIT` tells us that with a procedure call the transferred parameters will not be evaluated. Finally the `REMEMBER_BIT` makes sure that a remember table is set up and maintained.

Polynomials are stored internally in a sparse, distributive representation. This is carried out by using the memory management type `MMM_-DIST_POLY`. The exponents and the coefficients of the terms are stored in this. However, the user has no direct access to this structure. Further entries in the `point`-part are a list of variables and the coefficient ring of the polynomial. The sequence in the list shows the order of the variables. The first element of the list is the primary variable; the other entries are the secondary variables. The coefficient ring is given by the identifier `Expr`, an expression `IntMod(n)` or a domain.

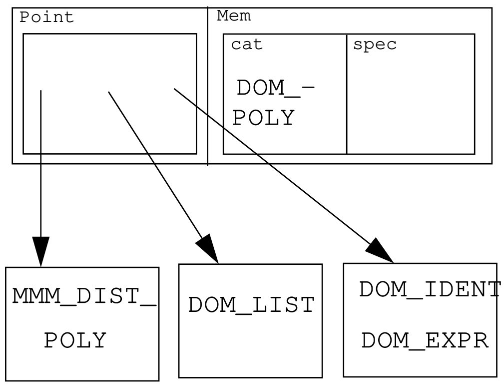


Fig. 7: Polynomials

The type `DOM_DOMAIN` is used to store domains. The `point`-part consists of a reference to the memory management type `MMM_DOMAIN` which, in its turn, contains several entries in the `point`-part. The first entry is a `DOM_TABLE` in

Elementary Domain Types

From the user's viewpoint the elementary domain types consist of one object and are generally not decomposable. All numerical types are processed with the aid of the PARI-GP-package (see [BBC+91]). This concerns both the internal representation and the arithmetic. The memory management type `MMM_PARI` is used for storage. For this purpose the PARI data structure is directly embedded in the `mem`-part of an S-pointer. By using a detour through two extra references the PARI long number arithmetic can be used in the kernel. All numerical data types are constructed as depicted in Figure 2. The actual datum is stored in PARI format in the `MMM_PARI` object and the concrete type, e.g. `DOM_INT` for integers is contained in the `cat` component.

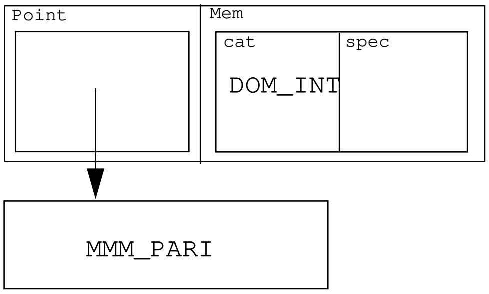


Fig. 2: Numerical Domain Types

For the storage of character strings (up to 2^32 - 1 in length) the type `DOM_STRING` is used. The string is stored as a normal C string in the `mem`-part of the `MMM_STRING` object. As further information the length of the character string is noted before the string.

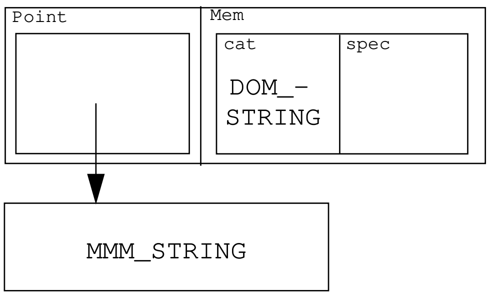


Fig. 3: Strings

Identifiers have the type `DOM_IDENT`. The sign of the identifier is stored in the `spec` component. Internally MuPAD has no subtraction operator. Instead, a substraction is represented as addition with the additive inverse. An identifier with a negative sign is logically treated as the expression `a*(-1)`. Therefore, the operands of `-a` are `a` and `-1`, and its type is multiplication ("`_mult`"). Division is handled analogue. Internally, a division operator does not exist. Division is represented as multiplication with the multiplicative inverse. Internally, this inverse is marked with the relevant sign in the `spec` component. An identifier with this sign is logically treated as the expression `a^(-1)`. The operands are `a` and `-1` and the type is exponentiation ("`_power`"). A further sign is the logical negation. Internally, this is also indicated by a bit in the `spec` component, but externally it is represented by a call of the function `_not`. The identifier name is stored in a `MMM_STRING` object. The `point`-part of the `MMM_STRING` contains some references that have different meanings according to the identifier type. For environment variables references to C routines are contained there. These are, for instance, called when assignment to the variable are made. For normal identifiers the run-time stack is located there.

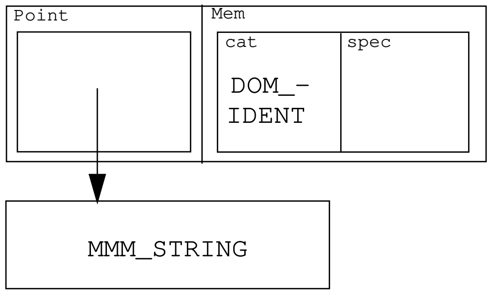


Fig. 4: Identifier

Compound Domain Types

By using the compound domain types many data (even with different domain types) can be combined to a single datum. With some types any number of data with any domain type can be combined, whereas with others a definite number of data with a certain domain type are necessary.

Sets and tables are stored in an associative memory (internal memory management type `MMM_ASS_MEM`). This content addressed mem-

- count - The number of S-pointers in the point-part.
- refs - The number of S-pointers that point to the header. By using this reference counter an expensive garbage collection can be omitted during run-time. An object is deallocated the moment the value 0 is reached in refs.
- status - Status information is stored in this structure. Internally, this is a bit-field with the size 16. The individual components are:
  - type - Specifies the memory management type of the datum.
  - system - A set bit shows that the data in the data area was not laid down by the memory management. In this manner libraries that manage their own memory can also be used.
  - unmovable - A set bit indicates that the memory cannot be moved.
  - memtype - The memory type of the memory.
  - lock - Is used to block the header in the parallel case.

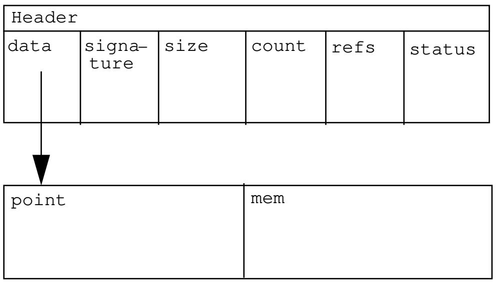


Fig. 1: S-Pointer

Figure 1 is a schematic diagram of an S-pointer. All headers generated by the memory management have a uniform size irrespective of the data contained. The data area of a header, i.e. the point- and mem-parts, can be of any size. In this way algebraic expressions consisting of associative operators and their operands can be represented internally by n-ary trees in a canonic manner. A node of the tree consists of references to the operator and its connected operands. Although outwardly the memory management only shows one data type, the S-pointer, this flexible structure is sufficient to realize many different data types. This takes place by different usages of the point- and mem-parts in the data area of the S-pointer. In this way the memory management makes many elementary types available. All these types are strictly typified, i.e. internally the user can recognize which type a datum has. This information can be gained from the type component of the status part of a header. This mechanism simplifies the internal realization of the MuPAD kernel. Principally it works with only one data type, the S-pointer. The user does not need to transfer extra information to functions that more closely specify the parameters. Functions work on S-pointers and the actual stored data and their types can be extracted from the datum without any problems. The elementary types of memory management are internally named MMM_<name>. Examples for this are: MMM_STRING, MMM_PARI or MMM_CATEGORY. The MuPAD user has no direct access to the memory management types. Only the data types on the domain level are available to him. These types all belong to the memory management type MMM_CATEGORY. These types have space for at least one structure consisting of two C short values in the mem-part. The first value cat specifies the domain type as a number; the second value spec has, according to the type, a differentiating meaning. In this part, for instance, the sign of a number or expression are situated. In the point-part the actual data are stored. In some domains references to memory management types can be found here, and other domains may contain references to further domains. In this manner all the types which are directly accessible by the MuPAD user are typified in two ways. Firstly, by the memory management type MMM_CATEGORY and secondly by the domain type. A complete description of Mammut can be found in [Nau92].

Domain Types

From the user's point of view all data types are of the Mammut type MMM_CATEGORY. I would now like to introduce you to some of these domains. In MuPAD we differentiate between elementary and compound domain types (a more complete overview of the available category types is given in [FGK+93]).

Data Structures and Memory Management

Karsten Morisse

Introduction

During the development of a computer algebra system data structures must be very carefully chosen. The kernel of such a system is nearly entirely occupied with generating, analysing or manipulating objects. To a large extent the efficiency of these operations is dependant on the construction of the data structures. The assignment of the data structures should take the following aspects into account:

- For the efficient storage of the objects that are found in many computer algebra systems dynamic changing of the data must be possible. Due to this the question of the data structure is closely connected with the question of memory management.
- Essentially, computer algebra consists of data manipulation, i.e. the execution of the basic operations: Extraction and substitution of data, generation and deletion of data. An efficient execution of these operations is a prerequisite for an efficiently working computer algebra system.
- A uniform construction of data structures simplifies many algorithms in the computer algebra kernel. This detail is also important for the servicing and maintenance of the algorithms.

In this contribution I would like to give an insight into the MuPAD kernel and introduce the data structures used. We must differentiate between the data structures from the user's point of view and those from the internal point of view. In reality all the data types available to the user, like sets, tables or expressions, are only modelled on one internal data type which can, in its turn, contain other internal types. The internal data types are provided by the memory management.

Memory Management

The memory management is a part of **Mammut** (**M**emory **A**llocation and **M**anage**m**ent **U**nit), which forms the interface between the kernel and the hardware (or operating system). Mammut is a universally applicable memory management that is nearly optimally suitable as the fundament for implementing a computer algebra system. Internally, for all data a uniform data structure, the *`S-pointer`*, is used. A S-pointer is a pointer to a header. Such a header is a (C-) structure which consists of the following components:

- `dat` - A pointer to a header. This is a reference to the data which are hidden behind the header. This data area is internally split into a so-called *mem*-part and a *point*-part. The `mem`-part can store data of any C-type, whereas the `point`-part consists of an arbitrary quantity of S-pointers. The `mem`-part and the `point`-part are stored in one memory block. With this the normal C-address arithmetic can be used to transit the data area.
- `signature` - Each datum is given a signature. It is internally guaranteed that identical objects are given the same signature. However, the reverse is not always guaranteed. It may happen that different objects have the same signature.
- `size` - Numbers the size of the data area that belongs to the header. This size is measured in terms of number of S-pointers. Therefore, this size consists of the sum of the `mem`-part and the `point`-part of a S-pointer.

dure in which the debugger stops, has already been reached, then the message `Bottom level reached` appears in the terminal window.

Clear all Deletes all breakpoints.

Clear The user can specify with a double click of the mouse in which line in the source text window the breakpoint is to be deleted. See *Stop at*.

Stop at The user can specify with a double click of the mouse in which line in the source text window a breakpoint is to be set. In order to give a specific procedure a breakpoint the user must load this into the source text window. See also *Goto proc*.

Print The user specifies the expression whose current value is to be shown by marking the expression with the mouse in the source text window. The value of the variable is displayed in the terminal window. See also *Execute*.

Display The user specifies the variable whose value is to be shown by marking the variable name with the mouse in the source text window. The value of the variable is displayed in the display window. Apart from this the current value of the variable is permanently shown from this point on. However, an update only takes place when the debugger is in input mode, i.e. when the program execution is halted.

Each of the previously described commands which can be carried out by clicking on the appropriate button with the mouse, can also be entered directly into the terminal window with the keyboard.

Internal Program Structure of the Debugger

Data produced during program execution never contain relevant debugger information. Only the data produced by the parser in a read-instruction contain the relevant information.
But, even here, only one piece of information per source text line is necessary, independent of the number of internal data produced by this line. Due to this the parser is the only module that generates debugger information and also makes sure that no unnecessary information is produced. In order to store the information the debugger needs an independent category type (`CAT_DEBUG`). This type stores debugger information. Information consists of a 32 bit integer that stores the line number and file name in code. Since a datum contains categories and the evaluator works on categories, from the internal viewpoint of the system, no difference exists between algebraic data, e.g. numbers, and debugger information. The debugger is neither an attachment nor an addition of an existing data structure, but is an extension of the system by a category type which neither changes the structure and characteristics of existing category types nor affects them. If a category of the type `CAT_DEBUG` is evaluated then the debugger kernel is activated and this then takes over the further processing. In the normal mode this category type is ignored by the evaluator. This results in the evaluator having no run-time loss in normal mode and the execution speed is only minimally affected even when debugger information is present. The parser is the only module whose execution speed is reduced when the debugger is turned on. But this does not affect the execution speed of a loaded program because only the evaluator is responsible for the program execution.

For this reason mdx is divided into five windows:

Status Contains general information about the current status. This includes, among other things, the line number in which the program was stopped and the line number of the file shown in the source window.

Source Shows the source text of the working file. After calling the debugger the window shows the first lines of the file if a file has been specified in the MuPAD call, otherwise the window remains empty. During the program execution, the part of the text which includes the line at which the program stops, is shown. The user can determine which part of the program is shown in the window by using the *Goto proc* button.

Buttons Often used debugger commands can be carried out by clicking them with the mouse. Commands that are not included here can be directly entered into the command window with the keyboard.

Command All output produced by the execution of the program, as well as some output evoked by the debugger commands, appears in this window. Input, to both MuPAD and the debugger, must be entered in this window.

Display The output of variable values, that are to be permanently shown, appears in this window.

The source, command and display windows have a scrollbar on the right hand edge, to enable the user to see the entire text in each window.

Command survey

Next Execution of all instructions in the current line. The debugger then stops at the next working line. If the executed instruction contains the call of a user defined function then this will be carried out without the debugger stopping in the function.

Step Analogue to *Next*, only now the debugger stops inside the user defined function.

Cont The program is carried out until an instruction is found that is in a line marked with a breakpoint, or until the program is finished. See also *Stop at*.

If the debugger stops then the status lines display in which line the debugger stops [Stopped at:], in which file this line is [Displayed file:], and in which procedure it stops [Procedure:]. The source text window contains the part of the program in which the line is at which the debugger has stopped. Which lines are included in this text section is given by [Lines: --]. The line in which the debugger stops is shown inversely.

Goto proc The specified procedure in the input field is shown in the source text window.

Quit Terminates the execution of the program if the debugger is in input mode. The debugger reports `Execution completed`, and the user is then in input mode. *Quit* in input mode terminates the MuPAD session. A running program can be stopped with Ctrl-C. The debugger is then in input mode.

Execute The command specified in the input field is executed. The command need not be only single instructions but can be put together from any series of instructions. One application is, for instance, the interactive influencing of a program execution by the user altering the variable values.

Where All procedures that have been called from the beginning of the program execution until this point in time are shown in the terminal window. Besides the procedure name the user is also informed of the call position in the form of a line number and a file name.

Up In the source text window the line is shown from which the shown procedure has been called. The status line of the debugger is also updated. If the procedure that the user has interactively entered, has been reached, then the message `Top level reached` appears in the terminal window.

Down This can only be used if at least one *Up* has been previously used. The procedure from which *Up* was called is shown in the source text window. If the current procedure, i.e. the proce-

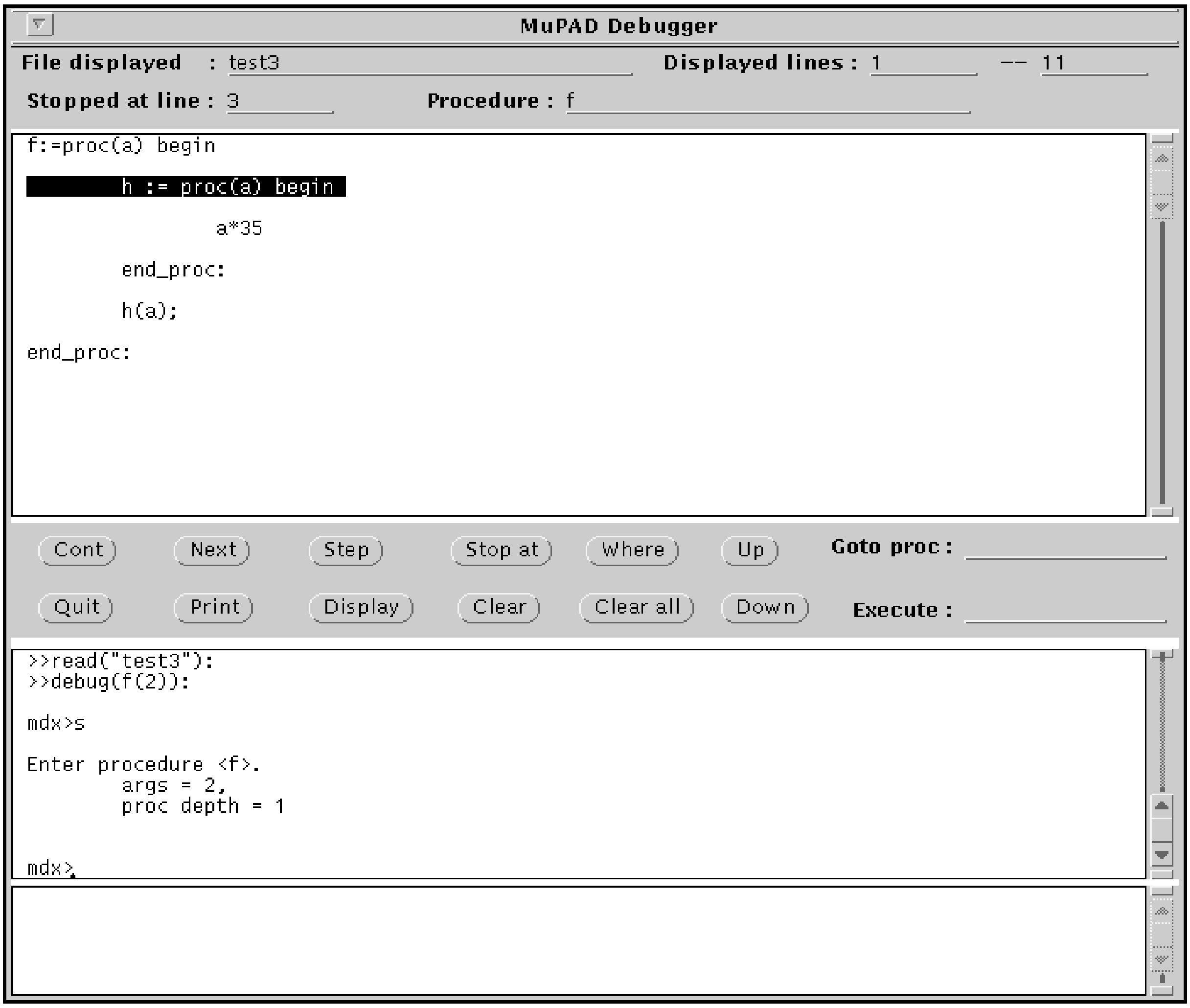
Interaction between the Debugger and the User

If MuPAD is started with the debugger turned on i.e. with `mupad -g`, then additional information for the debugger is produced internally. This information is transparent to the user. From the point of view of the user there is no difference to the standard mode of MuPAD. The interactive debug mode is activated with the system function `debug`. The argument of the function debug can be any instruction. If this instruction contains a procedure call the prompt `mdx>` appears and the user is now in the input mode of the debugger, in which commands for controlling the program execution or commands for information about the program status can be given (see page 48). During the execution of the instruction only commands for the debugger can be given. After the instructions have been carried out the debugger reports `Execution completed`, and the user is once again in the input mode of MuPAD.

mdx: The X-Frontend of the Debugger

The program *mdx* is one of the graphic user interfaces in MuPAD. This program requires either the window system X or OpenWindows. It improves the user comfort by:

- Simplifying the input of debug commands.
- Visual processing of the information given by the debugger.
- Simultaneously displaying the source text and the program output in separate windows.



Debugging

Oliver Kluge

The MuPAD-Debugger

MuPAD contains two possibilities of following the program execution in detail. The *trace mode* makes a protocol of the program execution. In this mode the user has no control over the execution of the program. By comparison the *interactive, line oriented source code debugger* offers the following possibilities:

- User controllable program execution (step by step or by setting breakpoints).
- Identification of source text lines in which a run time error has occurred.
- Displaying and changing of variable values.

This debugger mechanism, which is integrated in the system kernel, has a character/line oriented user interface and offers, even on a text-only output device (ASCII terminal), operation convenience and gives very detailed information in text form. An interface based on OpenWindows, which offers even more convenience, is the program *mdx*. This is not integrated in the MuPAD system kernel (see page 47).

The debugger is activated by the options in the MuPAD call. In order to do this the MuPAD program can be called with three different options. All three options can be used simultaneously.

A program, that is either to be executed under control of the user (debug mode) or to be protocolled (trace mode), must be a text file containing a syntactically correct MuPAD program. This text file is either specified as usual in the MuPAD call or read with the function `read`. Calling procedures, which are defined in this file, now gives detailed information about its program execution.

Under UNIX the debugger can be activated with use of the following command.

```
mupad [-g] [-t] [-v] [options]
```

-t : Turn on trace mode. During a program execution the number of the line in the source text in which the executed instruction is to be found is shown. In addition to these the place in the source text in which a (user defined) function is called and the function arguments are also shown. The place where the (user defined) function is left and the return value of this function are given.

-v : Turn on verbose mode. During the execution of a read instruction, i.e. during the reading-in of a MuPAD text file, the line numbers of the procedure definitions are displayed. Information about possible positions in the source text for the setting of breakpoints are displayed in the form `#line#` during the output of user defined functions.

-g : Turn on debug mode. In this mode the user can interactively enter commands with the keyboard to control the debugger. A survey of the control commands is given in the following table.

Functions	Commands
listing active procedures	(d)own, (u)p, (w)here
displaying/changing variables	(D)isplay, (p)rint, (e)xecute
breakpoint setting/ deleting	(g)oto proc, (C)lear, (S)top at, clear (a)ll
controlling program execution	(c)ont, (n)ext, (q)uit, (s)tep, CTRL-C

In contrast to the graphics user's interface *mdx* MuPAD offers the user no windows. Here the source text window, that shows the position in the source text in which the program is to be found, is missing. Information about this position in the source text is therefore not displayed visually but is described in text form.

clicking into the paint area and dragging the viewing box to the desired position.

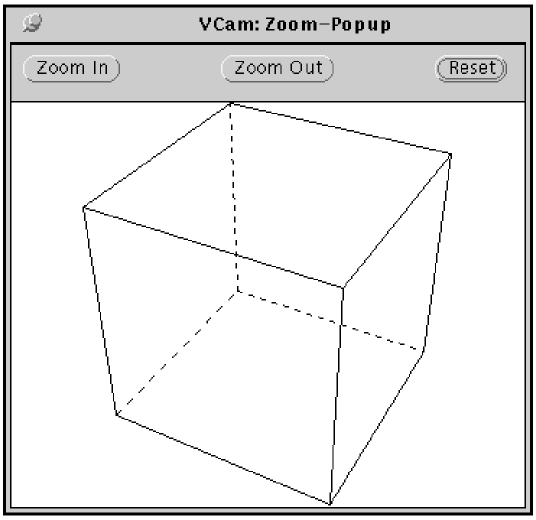


Figure 11: Zoom Window

Finally we should mention that the graphics tool is still in an early development stage, which can lead to some of the described options not being available. Therefore we would like to call your attention to the enhancements for the graphics tool which are being planned:

- Expansion of the three dimensional graphics by contour models and point lists.
- Implementation of the lighting model.
- Inclusion of animation routines.

Finally we would like to show you some examples which demonstrate the various graphics capabilities. The first picture shows a so called minimal surface, i.e. Enneper's surface. This surface is parametrized as follows:

```
x := u - 1/3*u^3 +u*v^2;
y := -v - u^2*v + 1/3*v^3;
z := u^2 - v^2;
```

The ranges for `u` and `v` are both `[-1.2, 1.2]`.

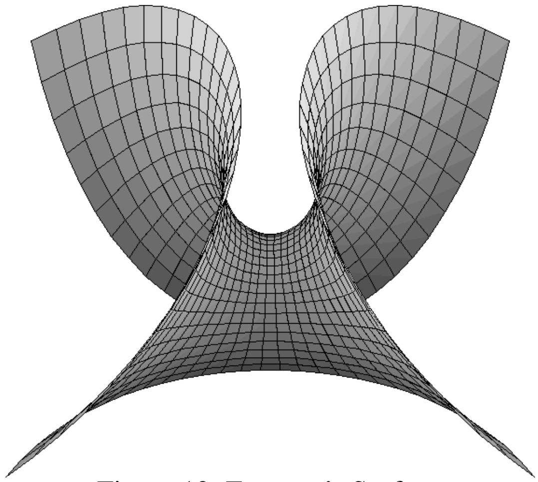


Figure 12: Enneper's Surface

The second graphics shows the corresponding Gaussian map of the above surface.

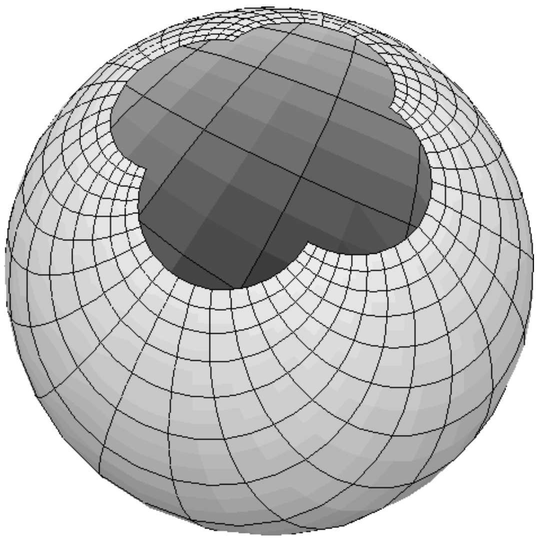


Figure 13: Gaussian Map for Enneper's Surface

Choice, to choose the categories of the shown values, Reset and Apply to revoke or apply the changes.

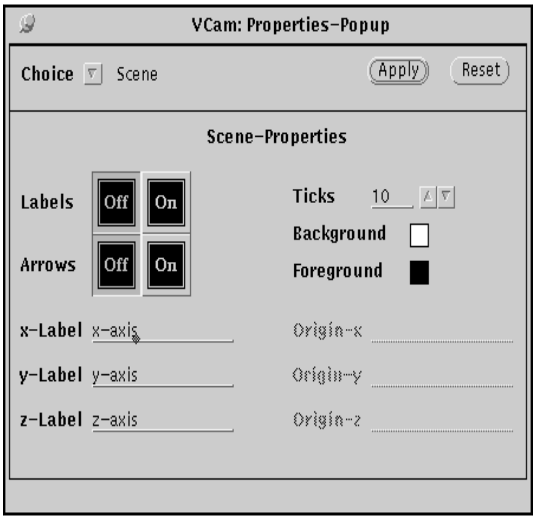


Figure 9: Properties Window

The user can activate the *perspective window* by switching the automatic perspective to manual perspective in the graphics manipulation window. This window is used to change the perspective in which the graphics data are drawn.
In MuPAD the perspective is determined by use of two three-dimensional points, the *camera point*, which defines the position of the camera, and the *focal point*, which specifies the direction in which the camera is held. These parameters can be changed interactively with help of the perspective window. Again this window contains the usual Reset button. In addition it contains:

- an area for entering the explicit perspective parameters in cartesian coordinates,
- an area for entering the respective perspective parameters in polar coordinates and
- a drawing area, in which the *viewing box*, i.e. the minimal box containing all graphics data, in the current perspective is shown.

The user has two possibilities of altering the perspective parameters: firstly he can determine the explicit values for the perspective parameters i.e. the camera point and the focal point. After this entry the user must press the <Return> button so that the viewing box is drawn in the new perspective. In the left hand corner of the canvas the user can see the viewing box with the old perspective. This enables him to see the effect the changes have had. Naturally these parameters can also be changed with the use of the polar coordinates. For this purpose the three sliders are used. The sliders on the right side of the window are serving for horizontal and vertical rotation, whereas the slider on the bottom of this window is used to change the distance between the camera point and the viewing box. Again the viewing box is drawn in both the old and the new perspective.

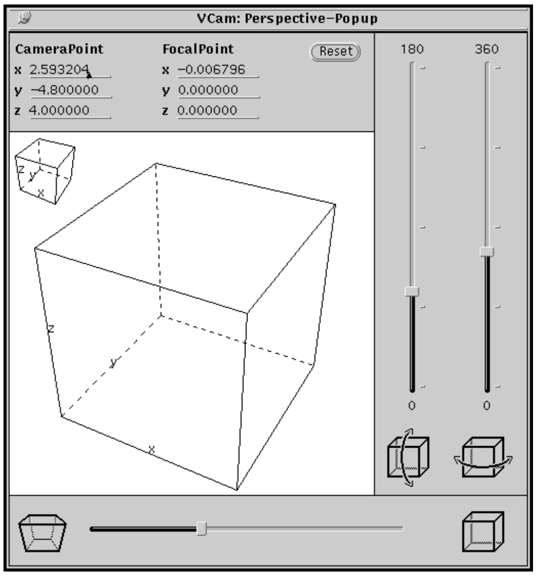


Figure 10: Perspective Window

The *zoom window* is activated by use of the Zooming menu of the graphics manipulation window. With the help of this window the user can alter the visible portion of the graphics.
The zoom window contains an area of control functions and a drawing area, in which the current viewing box is shown. The control area contains the following buttons:

- Zoom In:
  By activating this button the visible portion of the graphics will be enlarged by a predefined value.
- Zoom Out:
  This button is used to diminish the visible portion.
- Reset:
  This button is used to revoke the changes made in the zoom window.

Additionally the user is able to change the visible portion by use of the mouse. This is done by

- Directory:
  This text field is used to determine the path of the file in which the graphic is saved.
- File:
  Here the name of the file can be chosen.
- Save Scene:
  By activating this button the graphic drawn in the base window is saved in the specified format.

The *print window* is activated by use of the Print... button in the base window. The print window serves for printing the current scene, i.e. the graphics drawn in the drawing area of the base window. Therefore the Destination and the name of the printer are required. The scene is printed by activating the Print Scene button.

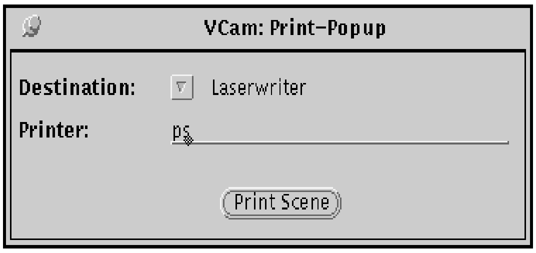


Figure 7: Print Window

In the *defaults window* the user can alter some of the preset default values. These values are stored in a file named `vcam_defaults`, which is read after starting the graphics tool. The default values are used to preset the options of the scene and the objects, so that the user does not have to specify all options of the scene and the objects. Furthermore the user is able to store his own default values with the help of the defaults window in his home directory, as will be explained later on. The defaults window is activated by clicking the Defaults... button in the base window. This window is divided into two parts. These are:

- an area with a series of control functions, with which the user can decide which default values he wants to change,
- an area in which the current default values are shown and changes can be made.

The control function area contains the following buttons and menus:

- Choice:
  This list contains the categories in which the default values are grouped. The current value is shown.
- Reset:
  This button is used for revoking changes.
- Apply:
  With this button the changes are verified and are made available to the user.
- Store:
  By use of this button the current default values are stored in a file named `.vcam_defaults` in the home directory of the user. This file will be read every time the graphics tool is started. So the user can store his personal default values in his own default file.

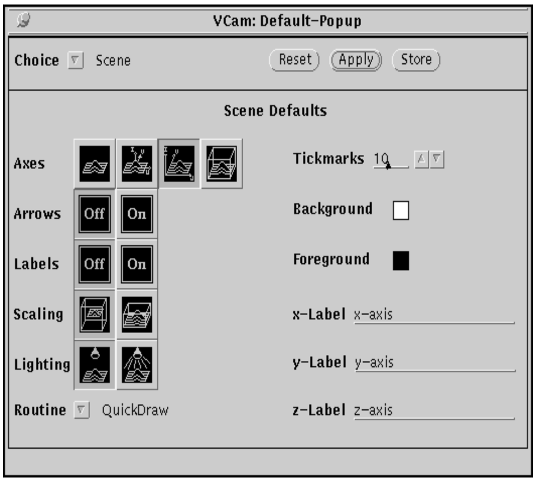


Figure 8: Defaults Window

The area in which explicit changes can be made is naturally dependent on the category chosen under Choice. For instance, in the figure above the user can see the current default values for a scene.These values are easily changed with the mouse. Similar subwindows are available for all other values in Choice.

The *properties window* is activated by use of the Properties... button in the graphics manipulation window. It is used to change further properties of the current scene, like the background and foreground colours, the number of tickmarks and so on.

The design of the properties window is similar to that of the defaults window. It is divided into two areas: an area with control functions and another area in which the explicit values can be changed. The control area contains the buttons

```
plot3d(CameraPoint = [0, 1, 30],
       Axes = NoAxes,
         [Mode = Surface,
            [sin(u)*cos(v)+1,
             sin(u)*sin(v), 0],
             u = [0, PI/2],
             v = [-PI, PI],
             Grid = [10, 28],
             Smoothness = [0, 0] ]);
```

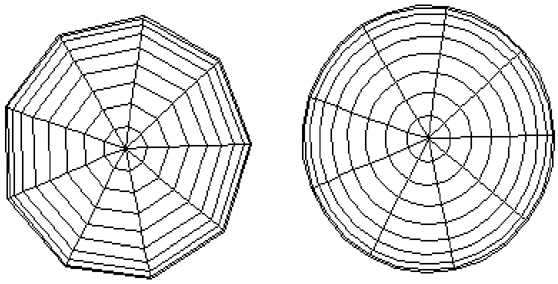


Figure 4: Demonstration of Smoothness

As already mentioned above, now we would like to describe the other pop-up windows which are activated by the buttons.
The *read window* is activated by choosing the item read scene... of the Scene menu in the base window.

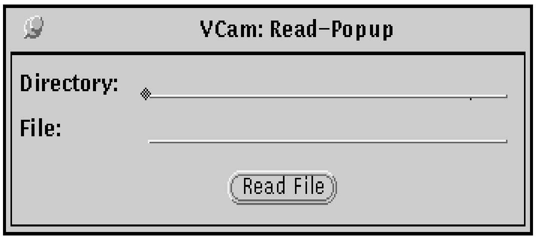


Figure 5: Read Window

This window is used to specify a file containing graphics data. This file is read by activating the Read File button and the corresponding graphics is drawn in the drawing area of the base window. The files, which can be read by the graphics tool, are either files in the ASCII format or files containing graphics data saved in binary format.
The *save window* is activated by use of the Save... button in the base window. It is used to save the graphics in various formats. The buttons in this window are used to choose the desired format and to specify the file in which the graphics data should be saved. Specifically, there are:

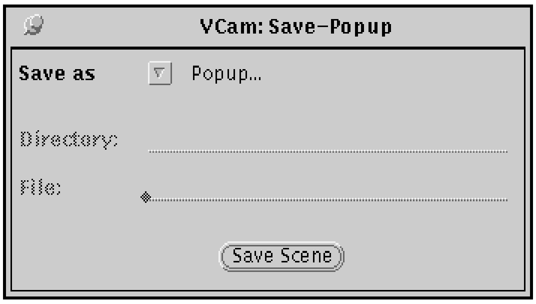


Figure 6: Save Window

- Save as:
  This button is used to choose the save format. At the moment seven options are available:
  - Popup...:
    The current graphic in the base window is saved in a further pop-up window. The user then has the possibility of comparing it with other graphics. By using the Replot button the saved graphic can be recalculated.
  - Raster-File:
    The shown graphic in the base window is saved as raster file. In this case both the text fields Directory and Name are activated. Here the user must give the path- and file names.
  - Gif-File:
    The graphic in the base window is saved in GIF format. Again the path- and file names must be given.
  - ASCII-File:
    The graphic is saved in an ASCII file and can be read again by the graphics tool.
  - MCODE-File:
    The shown graphic is saved in a file containing binary data. Like above binary files can be read again and the corresponding scene will be drawn in the base window.
  - Plot-Command (File):
    By use of this format, the corresponding `plot3d()` or `plot2d()` command of the shown graphic is saved in a file.
  - Plot-Command (XMuPAD):
    Like ahead, but the command is sent to XMuPAD and can be changed there.

- Plot:
  As already mentioned this button is used to enter the current graphic options in a new plot command, which is then evaluated. The graphic is then drawn in the already existing base window.

Of the functions in the lower part of the window only the most important shall be described here. All other functions are described in the MuPAD reference manual or in the help system in XMuPAD. In the lower part of the graphics manipulation window the specifications of the current object are shown. With the Add or Delete buttons the user can add or delete objects from an already present scene. Other important options are:

- Mode:
  the object mode. Four possibilities are included here:
  - Curve:
    This object is a space curve.
  - Contour:
    This is a three dimensional object, whose contour lines are to be drawn. This option is not yet available in the current version.
  - Surface:
    This object is a surface.
  - List:
    This is a list of points, which are to be given by the user. This option is also not available in the current version.

  The following options (and some others) are dependent on the object mode.
- Parametrization:
  The parametrization of the objects is given in the text field. Hereby, we differentiate between the following cases:
  - Mode = Curve:
    Because this is a space curve three parametrizations, dependent on one variable, have to be given. To make this clear, these text fields are called x(u), y(u) and z(u). These three parametrizations then define the x-, y- and z-coordinates of the object.
  - Mode = Surface:
    This object is a surface. In this case three parametrizations dependent on two variables are to be given. This is made clear by the names x(u,v), y(u,v) and z(u,v).
- Range:
  Here the parameter ranges have to be given. In Mode = Curve only one range needs to be given, whereas in Mode = Surface two parameter ranges are necessary.

All other options, like, for instance, the choice of plot style or colour functions, are optional. Here we would like to draw your attention to a few details in which the graphics tool differs from other computer algebra systems. In the MuPAD graphics tool, extra plot styles are at the users disposal. These are in Mode = Curve the plot style Impulses, with which the abscissae of the function values are drawn and in Mode = Surface the style Transparent, with which the user can simulate transparent objects. Furthermore the user may decide which parameter lines are to be drawn for the surfaces. By using the Smoothness value additional plot points can be included without the corresponding lines being drawn. This is shown in the following example:

```
plot3d(CameraPoint = [0, 1, 30],
       Axes = None,
         [Mode = Surface,
             [sin(u)*cos(v)+1.2,
             sin(u)*sin(v), 0],
             u = [0, PI/2],
             v = [-PI, PI],
             Grid = [10, 10],
             Smoothness = [0, 0]],
         [Mode = Surface,
            [sin(u)*cos(v)-1.2,
             sin(u)*sin(v), 0],
             u = [0, PI/2],
             v = [-PI ,PI],
             Grid = [10, 10],
             Smoothness = [0,2]]);
```

On entering this command the graphics shown in figure 4 appears. Two circular discs have been constructed, both with the same amount of sample points but with different Smoothness values. If the user wants a similar graphic to that on the right, but without using the smoothness function, then the amount of sample points will have to be raised. This leads to extra parameter lines being drawn. The command for constructing such an object is:

tion of the window, the graphic is drawn. The activation of the Plot button generates a new `plot3d()` command that contains the values shown in the graphics manipulation window as arguments. This `plot3d()` command is then sent to the MuPAD kernel. The evaluated result is then sent back to VCam and processed. The graphic appears VCam's base window.

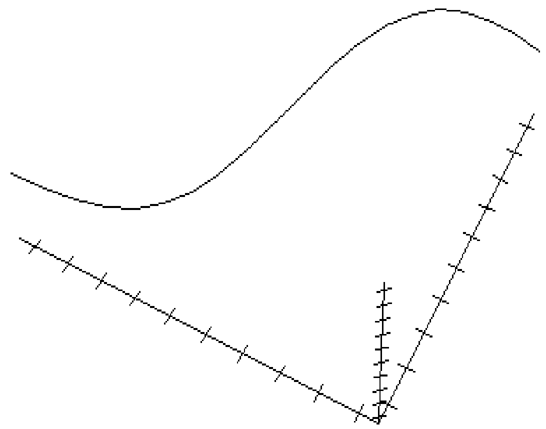


Figure 3: Example of a graphic

Here, once again, it must be expressly stated, that – unlike other computer algebra systems – MuPAD permits graphics commands to be entered directly in the graphics tool.

Now we come to the changes the user can make. In the area for the scene characteristics the following options can be changed or entered again:

- Title:
  Each scene can be given a title. The title will appear at a predefined position in the drawing area of the base window. This position can be changed interactively by clicking on the title and dragging it to the desired position.
- Routine:
  By use of this menu, the routine used to calculate the graphics data can be chosen. The option QuickDraw means that the graphics data are calculated depending on the numbers of Grid and Smoothness. The option QualityDraw means that the so called *fine structure algorithm* is used for calculating the graphics data. This means that depending on the smoothness of the object further plot points are evaluated automatically. However, this routine is not yet implemented in the current version of VCam.
- Axes:
  We offer a choice of four types of axis style. These can be chosen by means of the icons, i.e. None, Origin, Corner and Box.
- Scaling:
  With the help of the icons the user can choose between two scaling routines. In the first the graphical data are scaled so that distortion is only produced by perspective transformations, so that spheres appear as spheres and not as ellipsoids. In the second the data are scaled so that the canvas, that is the actual drawing area, is optimally filled.
- Zooming:
  With the help of this menu we offer the user some possibilities of altering the viewing volume. If the second icon is chosen, the so-called zoom window appears. This will be explained later.
- Perspective:
  This menu is used to switch from automatic perspective to manual perspective. In case the item manual is chosen the so-called perspective window is activated, but more about that later.
- Lighting:
  With the help of this menu the input of a lighting model is possible. This, however is not yet implemented at the current development stage.

Further functions can be called up by means of the three buttons in the upper part of the window:

- Properties...:
  This button is used to activate the so-called properties window, which serves for changing further properties of the current scene, like the labels of the axis and the number of tickmarks drawn at each axis. The properties window will be explained later.
- Reset:
  With the help of this button the user can restore the entries of the last plot command, or, if this does not exist, the default values are entered.

The Graphics

Thorsten Schulze

In this article we want to describe the MuPAD-graphics tool *VCam* (*V*irtual *Cam*era) by showing a sample session. The graphics tool is activated from the program XMuPAD by giving the MuPAD commands `plot3d()` or `plot2d()`. These commands can be given either with a series of arguments or without, whereby the second possibility was conceived for the inexperienced user. When the command `plot3d();` is entered, followed by <Return>, the so-called *base window* of the graphics tools appears.

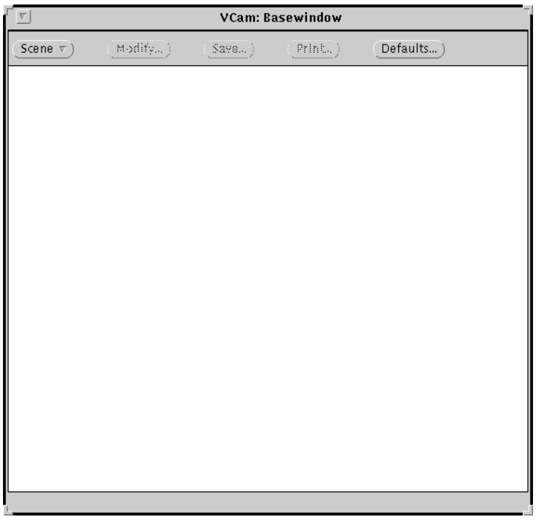


Figure 1: Base Window

This contains the actual plotting area for presentation of the graphics and a menu bar with control functions for the graphics tools:

- Scene:
  menu for the creation of new graphics
- Modify...:
  for modifying existing graphics
- Save...:
  for saving existing graphics
- Print...:
  for printing graphics
- Defaults...:
  for changing the default values

At first we want to produce a new graphic. This takes place, as already mentioned above, with the help of the Scene menu. If the item create 3D-scene is chosen, a further window appears, the so called *graphics manipulation window.*

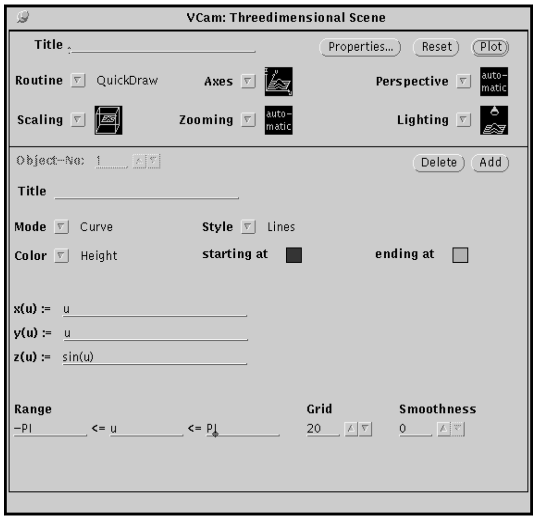
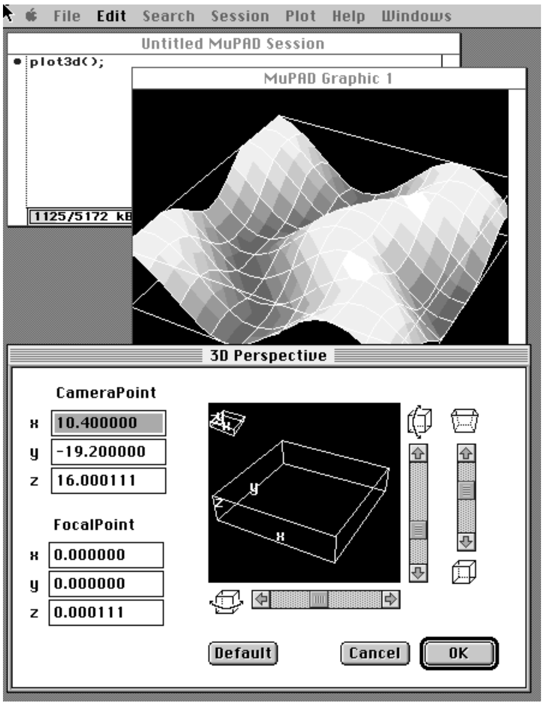


Figure 2: Graphics Manipulation Window

The partitioning of this window directly follows that of the basic user model which is based on photography. To understand this better, the user should imagine that he wants to take a photo containing certain information. For this purpose *objects* with individual parameters are grouped into a *scene* with further characteristics that are valid for all objects. Exactly this user's model reflects the graphics manipulation window. In the upper part of this window the scene characteristics are shown and in the lower part the object specifications are given. To produce a graphic, the user must first define the object mode. This is carried out by choosing the desired option in the Mode menu. For example, to depict a space curve the user should choose the option Curve. Now the user only needs to fix the parameters and the parameter range. For this the user enters the appropriate values into the text fields x(u), y(u), z(u) and <= u <=. When the user presses the Plot button in the upper sec-

PICT format. This allows export MuPAD graphics to any other Macintosh application.



The Help Tool

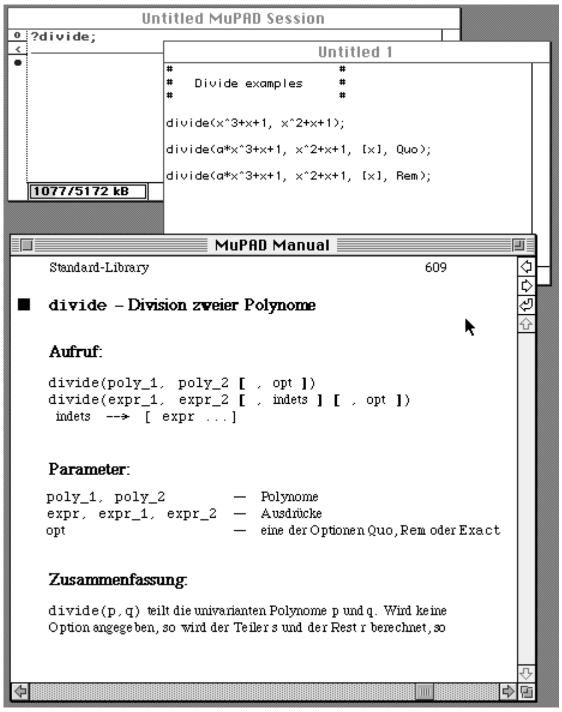
The built-in HyperText help-tool includes the full MuPAD reference-manual, the MuPAD tutorial and the help-pages. It serves as an on-line documentation for the complete MuPAD language.

In MacMuPAD, the help-tool is activated by selecting the appropiate menu item. After opening the help window the user may navigate through the document as in every hypertext system. The MacMuPAD HyperText system includes:

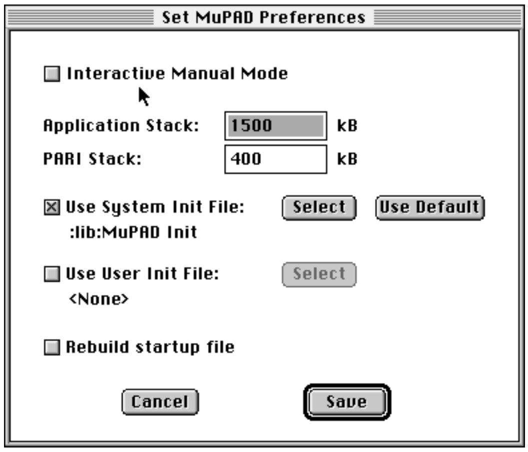
- Stepping through the document page by page
- Crossrefrences within the document text
- Jumping to important sections of the document with menu commands
- Stepping back to previously read pages
- Interactive examples within the document

In general, the MacMuPAD HyperText system offers the same functionality as the HyTeX system used in the UNIX version of MuPAD

Configuration



All important settings of MacMuPAD can be configured with menu commands or dialogs



which are stored in a Preferences file in the system folder of your Macintosh. This allows every user to costumize MuPAD for his/her personal needs even if MacMuPAD is loaded from a file server

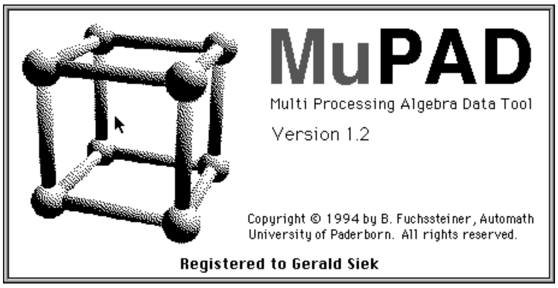
Requirements

MacMuPAD can be used on every Macintosh computer with at least 4 MB RAM and about 5 MB disk space. At present MacMuPAD needs system software version 7.0 or later

MuPAD for the Apple Macintosh

Gerald Siek

MacMuPAD is the MuPAD version for the Apple Macintosh. It includes the complete MuPAD language and functionality and offers the benefits of the user-friendly Macintosh interface. MacMuPAD was designed in accordance with the Apple User Interface Guidelines. Both experienced and novice Mac users will find MacMuPAD easy and straightforward to use.

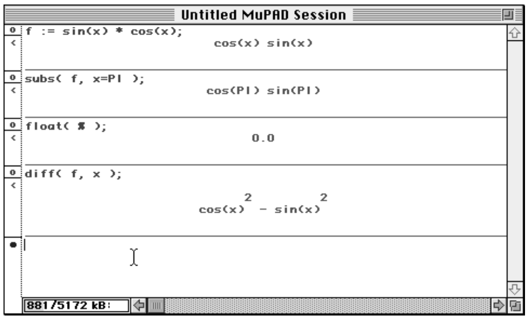


The basic components of MacMuPAD are:

- Session window
  All computations are displayed here
- Editor windows
  For addtional notes
- Graphics interface
  For creating 2D and 3D graphics
- HyperText help tool
  Including the complete MuPAD manual and tutorial

The Session Window

The session window is the main part of the MacMuPAD System. All input and output, all computations are displayed in this window.
The user can type the MuPAD commands as usual but may also use all well known editing features like cut&paste, cursor positioning by mouse, scrolling etc. MacMuPAD separates Input from Output and displays them in different colors. Furthermore, consecutive input and output lines are grouped in "blocks" which can be edited, deleted, copied or evaluated as a whole. The block types and their boundaries are displayed within the session window.



The user also may insert so-called “Text-Blocks”. These blocks may contain additional text which is not to be evaluated.
In addition to the session-window MacMuPAD offers a built-in editor for text documents. These editor windows can be used to enter additional text, programs or calculations. The contents of an editor window can be easily transferred to the session window to allow further evaluation. Both session and editor windows allow the user to perform editing commands like search & replace or choosing a different font and style. Of course, all documents created by MacMuPAD may be saved, re-opened or printed.
Important settings like the PrettyPrinter, fonts, or the textwidth can be configured with menu commands.

The Graphics Interface

The graphics interface is used to create and modify 2D and 3D graphics without typing cryptic commands. The user just opens the appropriate dialog and enters the object definition, it’s range, color and titles. New objects can be created with a single mouse-click, old objects can be deleted easily. Zooming, modifying the perspective or moving the titles - all this is done using the mouse. All pictures can be copied to the clipboard or saved in Macintosh

With this *Learning by doing* the user is quickly supplied with knowledge of the syntax to be used.

Implementation Details

XMuPAD is implemented with help of the XView-Library. This is a toolkit with various types of window, buttons, menus etc. as well as many functions for controlling and modifying these objects. XMuPAD is an interface purely for the user of the computer algebra system MuPAD. One of the first intructions from XMuPAD is the calling up of the function `fork()` and the starting of MuPAD as a new process. The input and output of this process are duplicated in a communication canal, through which XMuPAD and MuPAD then communicate. The on-line documentation HyTeX is also started by MuPAD as child process. The communication execution functions analogue to the connection between XMuPADand MuPAD: The input and output of HyTeX are duplicated in a communication chanel through which the two programs communicate. With it the internal marks, to which, on request of the user, the program jumps, are transfered to HyTeX.

the menu, the user can choose if capital and small letters are to be recognized.

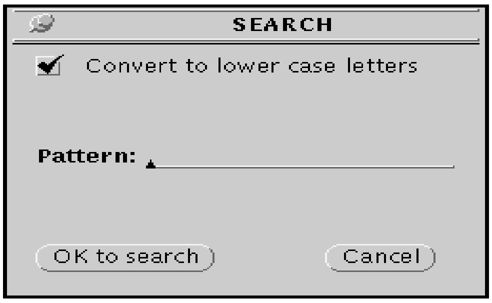


Fig. 9: Search Menu

Done Button

If the user clicks this button the HyTeX window disappears. However, it does not disappear completely and can be made to reappear with use of the help function or help buttons in XMuPAD.

Dynamic Buttons

The user has the possibility of marking his document with nodes and references to nodes. These nodes can be considered as internal text markers. They are not visible in the text. However, references to nodes are visible and can be seen in the HyTeX window as rectangles or boxes. If the user clicks on one of these boxes, the cursor jumps automatically to the corresponding node. In this way, the user has the possibility, with personally determined cross references, to skip through the document. In the MuPAD documentation there are two types of so called *dynamic buttons*. On the one hand there are those in the control panel,over the actual text window. With these buttons the user can jump to the previous or next chapter, section, etc. These buttons can change from page to page. On the other hand, there are the buttons in the text, where, for example,explanations of terms are realized as cross references.

If the user has made such a jump, naturally he will want to return to the starting page (especially with jumps of the second kind) as soon as possible. For this purpose there is the `Return` button. On jumping, with the dynamic button, to another page, the previous page is internally filed in a stack. A click on `Return` takes the uppermost element of this stack and jumps back to the corresponding page of the document.

Help Pages

As already mentioned above, beside the MuPAD reference manual, the on-line help system is also integrated in HyTeX, for example, in MuPAD or XMuPAD the user can request information about the system function `igcd` with the command

```
help("igcd");
```

or

```
?igcd
```

Then, the corresponding page is opened in HyTeX.

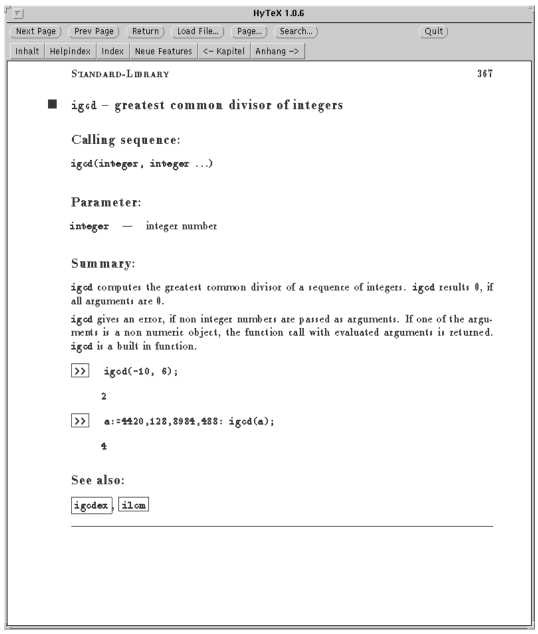


Fig. 10: Help page for `igcd`

The exact command syntax is described on such a help page. How the command works is described, some examples for use are given, and jumps to the help pages of related functions or commands are possible. Here, each example has also been given a box which is mouse sensitive, i.e. something happens when it is clicked. The difference to the dynamic buttons is that here no cross reference is carried out, but, the input of the example is transfered to XMuPAD and written in the XMuPAD text window. The user can now read the input again at leisure, if need be modify it and then execute it with `<Return>`.

offers the user a complete mouse orientated text editor, is opened. After the input of the desired text and clicking the `Done` button the text is passed to the MuPAD kernel.

HyTeX

HyTeX is a hypertext system developed at the University of Paderborn by the group of N. Köckler. In connection with XMuPAD, a specially modified version of this program is used. As can be seen from the name HyTeX, this system is based on the TEX- or LATEX-System from D. E. Knuth. Besides a special LATEX-Previewer it also contains some new commands. Therefore, text production is no problem for the TEXnician. HyTeX is used for the complete on-line documentation. This includes the MuPAD-reference manual and the help pages containing explanations of each MuPAD command. HyTeX is started as a child process from XMuPAD and is at first represented by an icon on the screen. The window is only opened after an explicit request from the user. This can be done in different ways. On the one hand he can open the contents list in the MuPAD reference manual by activating `Help` in the Tools Menu (see Tools Button on page 33). The second possibility is offered by the help function in MuPAD. With this the user can open the help page of a given command.

HyTeX-Window

The HyTeX window contains a LATEX previewer, which is provided with hypertext functions. The previewer gives various navigational possibilities through the documentation. These will be described below.

Static Buttons

This group of buttons contains those which, independent of which page can be seen, are always in the HyTeX window. They are all to be found in the upper control line.

Next- und Prev Button

With the help of these buttons, the document pages can be turned over one by one. `Next` turns the pages forwards, `Prev` turns the pages backwards.

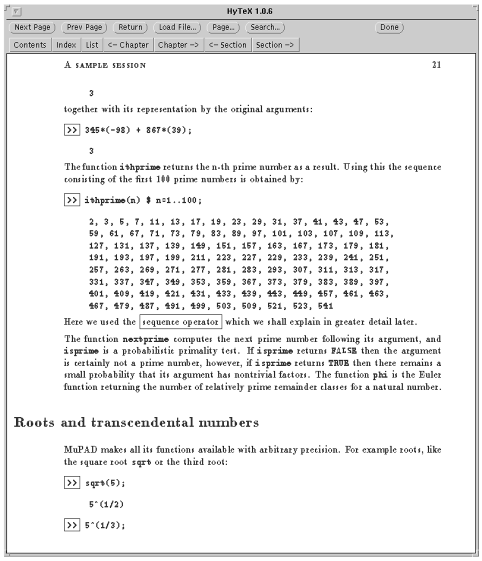


Fig. 7: HyTeX window

Return Button

This button is of importance in connection with the dynamic buttons and therefore, will be described later.

Load Button

The Load button allows the loading of other documents. Clicking on it opens another control panel, in which all the accessible documents are listed.

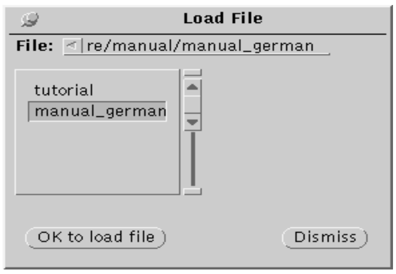


Fig. 8: Load window

Page-Panel und -Slider

The Page-Panel is used to directly jump to a chosen page. Instead of giving the page number directly with the keyboard, the chosen page can be set with use of the slider.

Search Button

This button is used for searching in the document. After activating it the search menu appears. Here the user can enter a text pattern, which is to be searched for in the document. In

pressed. If the `Recalculate Mode` is set to `ONE` only the selected input is recalculated. If the `Recalculate Mode` is set to `ALL` not only will this command be recalculated but all other following commands in the text window.

Replace Mode

With this menu item the `Replace Mode` is turned on and off. A description of how this mode works is to be found on page 31.

Seperate by lines

With this menu item the user has the possibility to seperate one input from the other with a line stretching over the whole window.

Tools Button

The `Tools` button opens the Tools Pulldown Menu. This menu comprises of entries to activate other programs. At the moment connections with the MuPAD-Debugger, the help system and the graphic tool VCam are implemented. By selecting `Help` the on-line documentation is activated. This contains both the complete MuPAD reference manual and help pages to all MuPAD commands (see also HyTeX on page 34). On selecting the `Debugger` the MuPAD-Debugger is implemented. So far it is not possible to refer back to previous MuPAD input because the debugger is started with a new MuPAD process. The `Tools` button is only active during the input phase.

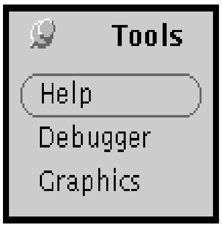


Fig. 6: Tools Pulldown-Menu

Special Cases - Deviations from MuPAD

The operation of XMuPAD differs in a few points from that of MuPAD.

UNIX Shell Command

With the function `system` (short form `!`) it is possible to give commands to the UNIX shell in MuPAD. The performance of the `system`-command demands special attention because it is not possible, for example, to start the Editor `vi` in a text window. In this case a new terminal window is opened in XMuPAD, within which the users default shell is started. The parameter of the `system`-command are executed. This window contains the `Done` button. On using this the window vanishes. However the contents of the terminal window are preserved. By means of a new `system` command the terminal window with the old contents reappears. Executing the `system` command in MuPAD is not without problems because the XView-Library offers no control mechanismus for the processes in the terminal window. Because of this in XMuPAD the `system` command provides no results in the terminal window. The user receives no information about if the displaced command has been executed, or, if the execution has already been ended. In an interactive session the user can see this in the terminal window. Furthermore the `system` command does not block XMuPAD, i.e. if `system` is called in a command sequence the following command is not delayed until the shell command has been completed, but both commands are executed simultaniously as two processes (one command is executed by the shell, the other by MuPAD).

User Interrupt

In MuPAD the user has the possibility with `<Ctrl-C>` to interrupt a running evaluation.There then appears a menu, in which the user can select between

- `abort` - Breaking off of the current calculation
- `continue` - Continuation of the current calculation
- `quit` - Ending the MuPAD session

In XMuPAD the user can generate an interrupt by clicking the `Interrupt` button. The user does not have the possibility to determine the further running of the calculation in a menu as the calculation is immediately stopped. However this is not a serious disadvantage as two of the three menu items are covered by the `Quit` and `Interrupt` buttons.

Execution of `textinput`

The user can enter interactive text during the running of a program and then assign it to a identifier with the function `textinput`. This command also demands special treatment in XMuPAD. In this case a further window which

commands `bytes()` and `time()`. The status button is only active during the input phase.

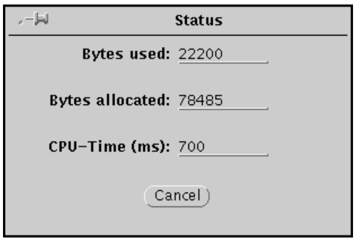


Fig. 2: Status Pop-up Window

Settings Button

On clicking this button a pulldown menu is opened with which the user has the possibility of carrying out different functions and can modify the input and output functionality. The pulldown menu has the OpenWindows typical pushpin with which the menu can be pinned to the screen. The `Settings` button is, as are all menu items, only active during the input phase.

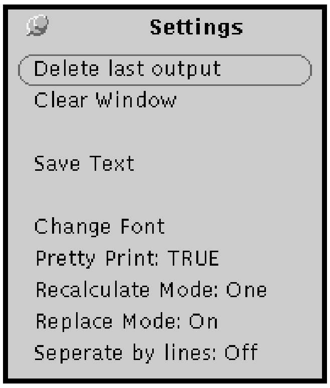


Fig. 3: Settings Pulldown-Menu

Delete last output

This item deletes the output of the last command. As the first item in the Pulldown-Menu this is also the default action. If the `Settings` button is pressed without the menu being open, this action is automatically executed.

Clear Window

This item deletes the contents of the text window. After this item has been activated the only text remaining in the text window is the MuPAD prompt >>.

Save Text

The menu item `Save Text` opens a further pop-up window, the Save pop-up window. The upper input field is used to enter the directory. The actual file name is entered in the lower field. The directory name need not be concluded with a `/`. It is inserted automatically between the two names. If `<Return>` is pressed whilst in the directory field the cursor goes to the file field. By pressing the `OK` button the contents of the text window are memorized in the specified file. If a file with this name already exists, a backup of the old file is made before it is overwritten. For that purpose a `%` is added to the name of the old file. The entered names are stored between two calls of the status pop-up window.

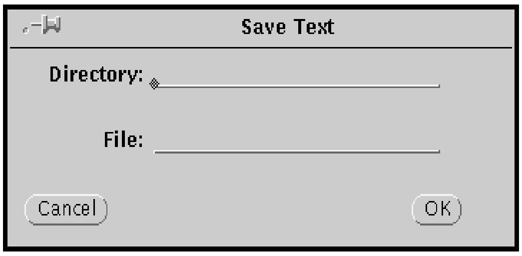


Fig. 4: Save Pop-up Window

Change Font

With the menu item `Change Font` a further pop-up window is opened. With this item the user can change the font of the window. The user can determine the font family, font style and font size.

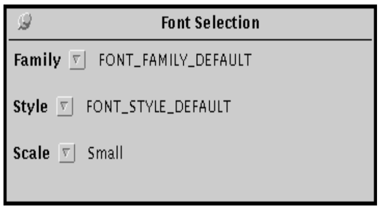


Fig. 5: Font Selection Window

With help of the following menu items different modes for the output in the text window can be set. The current status is shown in the menu.

Pretty Print

With this command the user can turn the formated output (Pretty Printer) on and off.

Recalculate Mode

It is possible in XMuPAD to execute a previous command again. To do this the old input is clikked with the mouse cursor and `<Return>` is

mouse (place the mouse cursor on the desired position and press the left mouse button) or the cursor keys. The command input is completed by `<Return>`. The current instruction is then read and sent to MuPAD to be executed. An instruction is normally only one screen line long, however; it is possible to enter commands spread over more than one line without the parser interpreting each line seperately. How this is done is described below. The result is printed as output directly under the input line. The result is followed by the MuPAD prompt character `>>`. The current insert position is now directly behind the prompt. After MuPAD has received the command, XMuPAD is in its evaluation phase. During this time the text window is inactive, and no input can be entered. This is shown by the mouse cursor having the form of a small clock.

Special Features

If there were no special features for the input of commands XMuPAD would be superfluous and a normal terminal window could be used.

<Shift Return> or <Enter>

In order to input commands structured over more than one line, the keys `<Shift-Return>` or `<Enter>` can be used to end a line. Through this the current line is concluded, a new line is inserted, and the current insert position moves to the beginning of this line. However, this newly entered command has not yet been transfered to MuPAD, i.e. it has not yet been executed. In this way a MuPAD command of many lines length can be entered. On entering `<Return>` the command is carried out. Hereby it is unimportant where the current insert position is. All lines above and below the insert position are considered part of the command. In other applications `<Return>` usually starts a new line at the current insert position. In XMuPAD a new line is started at the end of the input.

New execution of old input

By moving the current insert position into a previous command and pressing the `<Return>` key it will be executed again. The insert position may be anywhere within the command, even when this is more than one line long. If the `Replace Mode` is `OFF` in the Settings Pull-down Menu, the output of the old command is preserved and the new output is inserted before it. If the `Replace Mode` is `ON` the old output will be overwritten. In this case MuPAD gives no new input request, instead the current insert position moves to the end of the first line of the next command. If the `Recalculate Mode` is set to `ALL` then, not only will the current command be carried out, but also all other commands entered during the current MuPAD session. Here, the order in which the commands are carried out is laid down by the textual order and not the input order. In this way it is possible to carry out an interactive sequence of instructions if necessary with different starting values.

Insertion of new commands between old ones

As described above, the input of a command can take place at any position in the text window. It is also possible to enter a new command between the output of an old command and the input of the following command. Due to this the order of all entered commands is influenced, which is of importance for the realization of the `Recalculate Mode`.

Basic Window Buttons

The buttons in the basic window are ordered horizontally. At certain times some of the buttons are deactivated, i.e. clicking these buttons effects nothing. This can be seen as the black contour and lettering of the button are changed to light grey when deactivated. The individual buttons are described below.

Quit Button

The `Quit` button is self explanatory - it terminates XMuPAD. It closes all XMuPAD windows and also ends the MuPAD process. This button is always active.

Interrupt Button

This button sends a signal to the MuPAD process to terminate the current calculation. This button is only active during the evaluation phase. During the input phase it is deactivated.

Status Button

The `Status` button opens a pop-up window, which gives status information about MuPAD. The information gives the output of the MuPAD

The Graphical User Interfaces

Karsten Morisse

In order to work with the computer algebra system MuPAD no special input or output devices are necessary. A keyboard and ASCII terminal are all that are neded. To make the interactive work with MuPAD as convienient as possible, there is XMuPAD which is a graphical user interface for MuPAD under X-Windows and OpenWindows. Furthermore the Hypertext-System HyTeX, with the full MuPAD documentation, is at the users disposal.

In the development of the user interfaces much value has been laid on retaining the typical windows' characteristics, for example, the specification of options for defining the window size or window position, and the drag and drop mechanism. For descriptions of these characteristics the reader is refered to the OpenWindow and X-Window documentation. In this contribution the graphical interface is described. Apart from the environment and working methods of XMuPAD and HyTeX a short description of the technical implementation is also given. All specifications for keys, mouse buttons etc. refer to the Sun SPARCstation.

XMuPAD

XMuPAD is a fully independent program. It has no knowledge of computer algebra. It is implemented as an interface to the MuPAD kernel with the help of the XView-Library. XMuPAD is invoked with the command

`xmupad [`*options*`]`

The only important option for the user is the `-L` option. The language of the on-line documentation can be stipulated with this option. Possible parameters are `german` or `english`. After being called a *Basic Window* appears on the screen, which consists of two components

- a row of buttons,
- a text window for input and output.

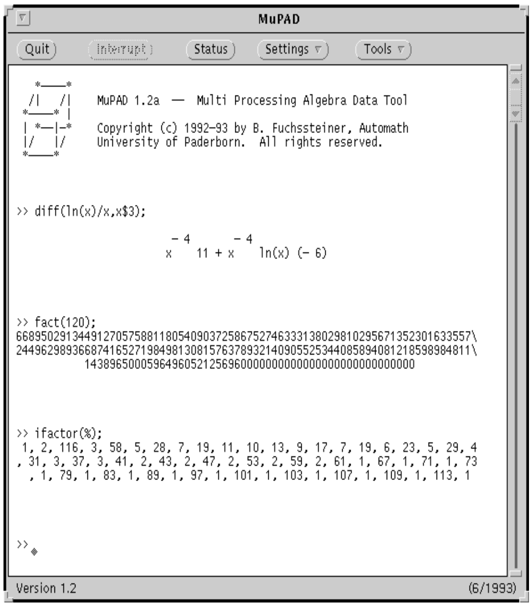


Fig. 1: XMuPAD basic window

Furthermore the MuPAD kernel is invoked as a child process on the UNIX level. XMuPAD's function can be divided into three phases, which are typical for interactive systems:

- During the input-phase the user enters commands into the text window.This phase is ended with the `<Return>` key.
- During the evaluation-phase the input of the first phase is interpreted.
- The third and final phase is the output of the solutions MuPAD has calculated.

The Text Window

The Text window is a usual window for the input of text, similar to the editor `textedit` in OpenWindows. On the right hand side is a scrollbar with which the visible text section can be changed.

Input

Commands are entered with the keyboard at the current insert position, which is made visible by the cursor. This insert position can be altered to anywhere inside the Text Window by using the

This is perhaps even more true in the use of categories. The advantages of generic algorithms —avoidance of re-implementation for all possible domains, and with this the lessening of implementation effort and a smaller chance of errors occurring — are quite obvious. The developer of generic algorithms is guided to a more standardized form by the category constructors. And, last but not least, the user can expect a standardization in the use of the domains.

These advantages are not fictional. M. Monagan has developed the library package GAUSS for Maple V which allows the definition of domains and categories, see [2]. In GAUSS domains are simple tables and domain constructors and categories are Maple functions that manipulate these tables. The domain operations are stored as table entries. The use of domains and their operators takes some getting used to because they are not integrated in Maple and cannot be operated with the normal operators in the Maple language. Apart from this the output of domain elements cannot be adapted to personal requirements.

Despite these deficiencies the package is being used successfully, for instance, for the implementation of a Gröbner bases package for different polynomial data structures and different term orderings, see [1]. The implementation details can be “concealed” due to them being made abstract by the categories. This means that different data structures and orderings can be used with one another and nevertheless only one “generic” algorithm for calculating the Gröbner bases need be implemented. Anyone who has seen the implementation of the “normal” Gröbner bases package `grobner` in Maple will be able to appreciate this abstraction.

Most of the concepts in the domains package can be traced back to ideas in AXIOM, see [3]. Axioms, domain and category constructors are language constructs in AXIOM. In contrast to the domains package these are anchored in the syntax. All objects in AXIOM are strictly typified which is, due to the lack of typification in the MuPAD language, not possible in the domains package. Many of the newer algorithms are firstly implemented in AXIOM because it contains rich algebraic structures that the user can utilize for his problems.

A conflicting feature of AXIOM is the automatic type conversion: The system tries to typify the user entries as soon as they are entered, i.e. to convert type-free expressions into domain elements. In the domains package the user must explicitly convert types “by hand”. Naturally, it is easier for the user not to have to give types. On the other hand it is extremely difficult to determine the type of an expression. For this AXIOM uses time consuming and error susceptible heuristics.

Normally, the user will not use the domains package directly at the “user’s interface” but will instead use it for the internal implementation of the library package. There will be “interface routines” that accept expressions as entries and then convert these internally to domain elements. This conversion is non-problematic as in this situation the implementor knows the problem that is to be solved and therefore also the type needed.

Until now the domains package is only available in an experimental form. The infrastructure exists, but the algebraic “meat” is missing. Some of the basic categories starting at the category “type” through the categories “sets”, “semi-groups”, “monoid” etc. to the category “field” already exist, but they do not have many of their more important operations. All in all much more experience will have to be gathered which will be beneficial to the infrastructure. However, this will be a part of a later MuPAD version.

Bibliography

[1] D. Gruntz. Groebner Bases in GAUSS. *MapleTech*, (9):36-46, 1993.

[2] D. Gruntz and M. Monagan. Introduction to GAUSS. *MapleTech*, (9):23-35, 1993.

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

As an example we shall look at a constructor for the category of Euclidean domains:

```
EuclideanDomain:=
CategoryConstructor::new(
# Name #
   EuclideanDomain,
# Parameters: none #
   [ ],
# Parameter-Test not necessary #
# Super-Categories #
   [ FactorialDomain ],
# Axioms: none #
   [ ],
# Entries #
   "rem",
   ...
):
```

In this simplified example the category constructor has no parameters. This means that the constructor can only generate a single category. An Euclidean domain is a factorial domain that contains the additional operations `"rem"` etc.. The operation `"rem"` is a basic operation whose implementation cannot be given here. If `"rem"` is carried out a run time error occurs. The basic operations must be "filled out" by the relevant domain constructors.
The calculation of the greatest common divisor can be defined by the Euclidean algorithm with the aid of the basic operations:

```
"gcd" = proc(x, y)
   local tmp;
begin
   x:= this::normalPart(x);
   y:= this::normalPart(y);

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

   this::normalPart(x)
end_proc
```

The entries `"iszero"` and `"normalPart"` are defined elsewhere in super-categories. Exactly as with the domain constructor operations the identifier `this` is replaced by the relevant domain during the construction of a domain.

Definig Axioms

Due to the fact that axioms are only attributes and have no further functionality their constructors are considerably more simple than the domain or category constructors. They are defined as follows:

AxiomConstructor::new(
   name,
   ⟨*parameters*,⟩
   ⟨*parameter-test*⟩)

name gives the name of the constructor. If the constructor has parameters they are given by *parameters*. Actual parameters are tested for their validity by *parameter-test*. To express that an operation can be inverted there is the axiom constructor `invertible`:

```
invertible:= AxiomConstructor::new(
   invertible,
   [ operator ],
   (if domtype(operator) <>
DOM_STRING
    then
      error("wrong operator name")
    end_if)
):
```

The expression `invertible("_mult")` generates an axiom that states that the operation `"_mult"` of a domain can be inverted.

General Nonsense?

What advantages do the concepts introduced here have over the implementation of domains "by hand"? The necessity of constructing parameterizable domains and the inheriting of implementations goes without saying. The user can, of course, also generate parameterizable domains with normal procedures. Such a method is doomed to be non-structured as each developer generates his own domains and defines his own operations. Domain constructors standardize the implementation of domains and allow for far better maintenance. Apart from this the domains package takes over many standard tasks during construction, reducing the developers work load and in addition unnecessary error sources and code duplications are avoided.

ulus) when a new domain is constructed. In `Zmod(7)` the operation `"new"` has the form:

```
proc(x) begin
  new(Zmod(7), x mod 7)
end_proc
```

The representation of the domain elements is implicitly set by the implementation of the operations. It can also be dependent on the domain parameters. In `Zmod` all domain elements have the same simple form: They consist of exactly one entry, an integer from the relevant residue class.

The algorithm structure may be dependent on the category of the relevant domain. In `Zmod` this concerns the operation `"invert"`, which calculates the multiplicative inverse of a residue class. If the modulus is a prime number and, therefore, the domain a field, the modular inverse can always be calculated. However, if the modulus is not a prime number the existence of the inverse is dependent on the relevant residue class. This must be reflected in the operation `"invert"` of the relevant domain:

```
"invert" = (if isprime(Mod) then
  proc(x) begin
    this::new(1/extop(x,1))
  end_proc
else
  proc(x) begin
    if igcd(extop(x,1), Mod) = 1
then
      this::new(1/extop(x,1))
    else
      FAIL
    end_if
  end_proc
end_if)
```

Naturally, the user can also inquire during the execution of the operation `"invert"` if the domain element belongs to a field and react accordingly. Exactly how this takes place is in end effect a design decision in the implementation of the domain constructor. The first alternative has the advantage that the user need not make any enquiries during the run time and is therefore more efficient.

The entries of a domain are generated as follows: Firstly the definition of the entry is searched for in the domain constructor. If it cannot be found there then it is looked for in the super-domain and then in its super-super-domains. If the entry is not defined in any of the super-domains the definition is searched for in the domain categories one after the other. In their own turn the categories carry out depth first searches in their super-categories for the definition.

The Definition of Categories

In the domains package each category must be generated by a category constructor. The definition of a category constructor is similar to that of a domain constructor:

`CategoryConstructor::new(`
  name`,`
  parameter`,`
  ⟨*parameter-test*`,`⟩
  super-categories`,`
  axioms`,`
  entries...`)`

The arguments have analogue meanings to those in the domain constructors:

name gives the name of the category constructor.

parameters gives the formal parameters of the constructor. This must be a list of identifiers.

parameter-test gives a statement that can be used to test the actual parameters. If no parameters exists then this statement is also not present.

super-categories gives the super-categories in which the categories generated by this constructor are contained.

axioms give the axioms for the generated category.

entries are a sequence of operator names or equations of the form *index*=*entry*. With an equation the *entry* is entered under the *index* in the domains of this category. If only an operator name is given this is an indication that this is a basic operation whose implementation must take place elsewhere.

The arguments are also evaluated during the generation of a domain of the so defined category. For `this` and the parameters the domain to be generated and the actual category parameters are inserted.

Constructing Domains

As already mentioned domains are generated with domain constructors in the domain package. A domain constructor is defined as follows:

DomainConstructor::new(
  name,
  parameters,
  ⟨*parameter-test*,⟩
  super-domain,
  categories,
  axioms,
  entries...)

The individual arguments have the following meanings:

name gives the name of the domain constructor. This is only used for the output.

parameters gives the formal parameters of the constructor. This must be a list of identifiers. The list can also be empty.

parameter-test gives a statement that is used to test the actual parameters. If there is no parameter this statement is missing.

super-domain gives the super-domain from which a new domain will inherit its implementation. Here a statement must be given that evaluates to a domain or to `NIL`. With `NIL` the domain has no super-domain.

categories gives the categories in which the domain generated by this constructor is contained. The statement given here must evaluate to a list of categories.
Only the "smallest" categories need be given. If a domain belongs to the category skew field then it automatically belongs to the category ring.

axioms gives the domain axioms. As with the categories this must be a statement that evaluates to a list of axioms. Only those axioms need be given that are not already entered by the categories.

entries is a sequence of equations with arbitrary length in the form *name*=*entry*. With this the entries in the new domains are defined. In the domains the value *entry* is stored under index *name*.

The arguments are not evaluated during the definition of a domain constructor. The evaluation of the arguments takes place when a domain is generated from a constructor: The statements for testing the parameter and for defining the categories and axioms can contain the identifiers from *parameters*. With the generation of a domain these "formal" parameters are replaced by the actual parameters of the domain. Only then are the so generated expressions evaluated. Apart from the formal parameters the entries can also contain the special identifier `this`. On generating a domain `this` is replaced by the new domain.
A domain constructor for $\mathbf{Z}_n$ may look like this:

```
Zmod:= DomainConstructor::new(
# Name #
  Zmod,
# Parameter: Modulus #
  [ Mod ],
# Parameter-Test #
  (if domtype(Mod) <> DOM_INT then
    error("no integer modulus")
  end_if;
  if Mod < 2 then
    error("modulus must be > 1")
  end_if),
# Super-Domain: not present #
  NIL,
# Categories #
  [ (if isprime(Mod) then
      Field
    else
      CommutativeRing
    end_if) ],
# Axioms: none #
  [ ],
# Entries #
  "new" = proc(x) begin
    new(this, x mod Mod)
  end_proc,
  "zero" = this::new(0),
  "one" = this::new(1),
  ...
):
```

The constructor is called `Zmod` and `Mod` is the modulus parameter. If the modulus n is a prime number then the domain $\mathbf{Z}_n$ is a field, if not it is only a commutative ring. `Zmod` has no super-domain and no axioms.
Both the identifiers `this` and `Mod` are replaced by the current values (the domain and the mod-

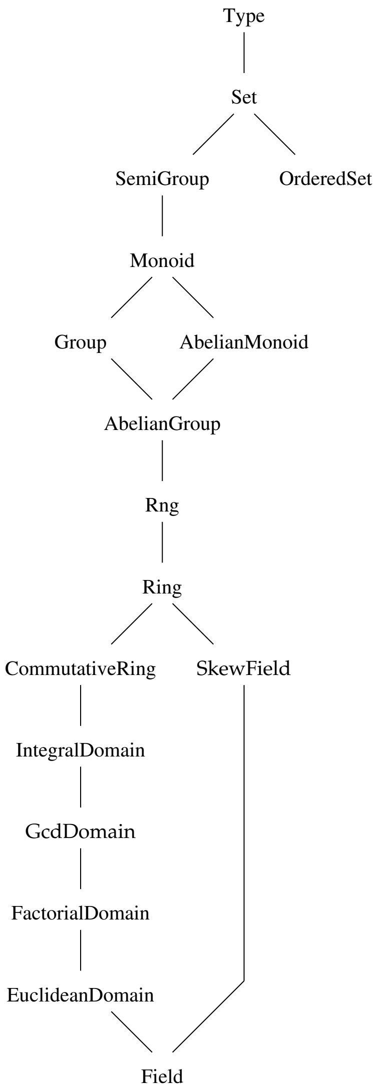


Figure 1: Hierarchy of the Categories in the Domains Package

Like the domains, categories also form hierarchies in the domains package. Thus every domain that belongs to the category skew field automatically also belongs to the category ring. In other words: Every skew field is a ring. Categories only need to define those characteristics and operations that are not made available by the more general categories.

In this context, the more general category of the rings is called the *super-category* of the skew fields and the more specialized category of the skew fields is called a *sub-category* of the rings. In contrast to the domains, a category can have more than one super-category. The commutative groups have, for instance, both the groups and the commutative monoids as super-categories. Categories not only form trees but also acyclic graphs. The — at the moment fairly simple — hierarchy of the categories defined in the domains package is depicted in figure 1.

Categories can, in their own turn, be themselves dependent on parameters. Thus, there are different categories of polynomial rings. These categories are dependent on the coefficients ring. In order not to have to newly implement the relevant category of polynomial rings for each possible category of coefficient ring the domains package contains so-called *category constructors*. These generate — as might be expected — categories dependent on parameters.

The characteristics of categories are postulated with the aid of so-called *axioms*. In the domains package these are mostly attributes whose existence the user can ascertain. The category ring has the axiom `unitElement("_mult")`. This axiom should show that a domain possesses a neutral element in connection with multiplication. The axioms can be requested by the category and domain constructors in order to define, for instance, an operation dependent on them. Axioms can also — as the example above shows — be dependent on parameters. Therefore, the domains package contains *axiom constructors* that generate parameter dependent axioms.

Categories in MuPAD

The objects described above — from categories to axiom constructors — are realized in the domains package as elements of special domains. Categories and axioms are elements of the domains `Category` and `Axiom`. The constructors for domains, categories and axioms are elements of the domains `DomainConstructor`, `CategoryConstructor` and `AxiomConstructor`.

If a new constructor is to be defined then the method `"new"` of the relevant "meta-domain" must be used. Entities of these "meta-domains", that is domains, categories and axioms, are generated in form of a function call with the method `"func_call"`. If, for example, `Zmod` is a domain constructor for $\mathbf{Z}_n$ then with the statement `Zmod(7)` the domain $\mathbf{Z}_7$ is generated.

Axioms, Categories and Domains

Klaus Gottheil

In order to simplify the construction of new algebraic structures, the experimental library package `domains` is available in MuPAD Version 1.2. With the aid of this package the user can construct parameterized domains. For this, knowledge of the common aspects of certain domain classes in the form of generic algorithms can be used.

Introduction

Starting with Version 1.2 the user is given a tool for the implementation of his own algebraic structures. This is realized in the MuPAD kernel by the data type "domain". These are nearly completely integrated into the MuPAD kernel and can use the already existing operations such as `+`, `*`, and `-`. The interactive use of the so defined structures is easy. An introduction to the domain concept is given in the article "Domains" contained in this issue of mathPAD.

Of course, it is not very effective to construct an algebraic structure like $\mathbf{Z}_7 := \mathbf{Z}/7\mathbf{Z}$ "by hand". Here, the user is more likely to want to construct a structure like $\mathbf{Z}_\mathrm{n}$ in order to obtain all possible remainder class rings of $\mathbf{Z}$ dependent on the "parameter" n. To use the MuPAD language: The user wants to generate a domain dependent on a parameter. To construct parameterized domains so-called *domain constructors* are used in the domains package.

The user will often want to implement a domain by taking over an already existing implementation ("inheriting" it) and changing some of its entries. The new domain is derived from an existing domain. In this case we can say that the original domain is a *super-domain* of the new domain and the derived domain is called a *sub-domain* of the original domain.

A domain constructor defined with the aid of the domains package permits the specification of such a super-domain. Naturally, the super-domain can in its own turn be dependent on the domain parameters of its own sub-domain. A domain constructor only needs to newly define those entries that differ in their implementation from the super-domain. The domains generated with the aid of the domains package form a hierarchy (a forest of trees).

The user will quickly notice that in the definition of new domains certain domains have certain things in common because they belong to the same class of algebraic structures. Thus $\mathbf{Z}_6$ and $\mathbf{Z}_7$ are both rings, they contain among other things the operators `+` and `*` and have neutral elements for these. In each ring the operation "raise to the power of a cardinal number" can be easily implemented by continued squaring. Of course, it would be extremely ineffective if such an operation had to be newly implemented for each ring. Naturally, the user will want to implement such an operation only once and then use it for each ring.

Classes of algebraic structures, such as "rings", are called *categories* in the domains package. A category "postulates" certain basic operations and characteristics of those domains belonging to it. The actual implementation of these basic operations is carried out by the domains. With the aid of these basic operations, further operations can be defined for the category. These are realized — as in the continuous squaring mentioned above — as *generic algorithms*.

If a new domain is now generated then the relevant domain constructor determines, by means of the parameters, to which categories the domain belongs. The constructor then enters the operations defined in these categories in the domain. If the newly generated domain belongs to the category ring then the operation "raise to the power of a cardinal number" is entered in the domain. Naturally, the user has the freedom to implement such operations himself in the definition of domain constructors if he wishes to use special characteristics of the domain.

Domains and abelian Groups

Holger Naundorf

A problem that many algebraists are confronted with is that of calculation in abelian groups. At the Paderborn university these were the Lp groups which occur in connection with the notation theory of algebraic curves.

On the one hand you may want to carry out addition of 2 elements of the group and multiplication of an integer and an element of the group and on the other hand you would like to be able to calculate using symbols. In the latter it is necessary to utilize associativity and distributivity - that is the normal calculation rules of `+` and `*`. Many computer algebra systems have problems dealing with this task. It is at the same time very difficult to solve. Here I would like to introduce the principle by which this can be solved in MuPAD.

Groups as Domains

As we have already indicated in this journal in the article "Domains" it is very easy in MuPAD to represent an algebraic structure by a domain. Normal addition can be used for the additive operation in connection with group elements - these are represented by domain elements. The normal simplification rules are still used for calculation with symbols. However one problem does occur and that is that the multiplication does not try to combine group elements with integers.

New Multiplication

At this stage, in many other systems the user would have reached the limit of his possibilities and will have to capitulate or look for other solution strategies. However, in MuPAD, the user has the possibility of overcoming the weakness of the system by newly defining how multiplication is to be carried out. In order to avoid the problem of having to implement the A- and D-law himself, the user will have to return to normal multiplication. To do this the user must temporarily store the contents of the variable `_mult` in another variable and then store the new algorithm for multiplication in `_mult`. This then carries out the following:

- Multiply the arguments of the function with the aid of normal multiplication. Store their solution in `x`. In this step the symbolic simplification is carried out and the various laws are applied.
- If `x` represents neither addition nor multiplication return `x` as the solution of the function.
- If `x` represents multiplication check if both a group element and an integer are contained in the product. If this is the case then multiply them and return the simplified product.
- `x` represents an addition and was obtained by applying the distributive law. Check each addend to see if a group element and an integer can be combined. Return the so-generated datum.

Conclusion

Although some of the weaknesses of MuPAD are exposed in this example it has shown the flexibility and power that this system offers. It is nearly always possible for the user himself to overcome the weaknesses that exist in MuPAD. This is extremely important when the user considers that no program can be perfect.

"negate": Determines how the additive inverse of a domain element is calculated.

"invert": Determines how the multiplicative inverse of a domain element is calculated.

"print": Determines how a domain element is read-out. If no entry exists a standard output is chosen.

```
e;
  new(Example,5,6,7):
d::print:=proc(x)
  begin
    [extop(x)]
  end_proc:
e;
  [5,6,7]
```

There are other internally used methods whose description would not be appropriate in this journal.

Usage

If a domain is to represent a ring or a field the user needs only to define the relevant necessary methods and then he can calculate in the normal way. At the moment there are still weaknesses in this area:

- With `+` and `*` only elements of the same domain can be connected. Multiplication with other types, such as integers, is not easily possible.
- Due to variables having no type it is not possible to recognize the elements of which type they represent, during calculation. The simplification laws used are those relevant to R. Here errors can be generated if, for instance multiplication in the user's own structures is not commutative.

Logical Interfaces

Due to the fact that at the time of implementation of a domain it is not yet clear if the chosen representation of a domain element is optimal a strict differentiation between a logical and a physical view of a domain element must be made. The logical view of a domain must be determined in the first implementation, the physical view can be changed later if necessary. Due to this the following basic rules should be followed:

- Each domain should contain a method `"new"` with which it can generate new domain elements. This function checks if an argument is correct and then generates the relevant domain element. The domain user should never generate new domain elements himself with `new`. In this way the user is prevented from generating non-standard or nonsense domain elements.
- The domain element user should not access the contents of a domain element with any of the `ext...` functions. For this purpose only the normal functions `op`, `subsop`, etc. should be used. These can be redefined with the relevant methods by the programmer if necessary.

Conclusion

To sum up it can be said that MuPAD enables object oriented and polymorphic programming by the integration of domains. The time needed for the development of programs can be drastically reduced. Apart from this the code size can also be reduced. A further advantage is that the user can configurate the system according to his own wishes and can intuitively calculate with non-predefined algebraic structures.

```
d1:=d:
d::self:=d:
bool(d=d::self and d=d1);
  TRUE
```

This is important, because methods - another name for a function stored in a domain - must often know which domain they belong to. As the reference effect is not always desirable the user can make a copy of the domain with the function `domain`.

```
d1:=domain(d):
```

Although `d1` has exactly the same entries as `d` the two domains represent different data and are, therefore, not recognized as the same:

```
bool(d=d1);
  FALSE
```

This is implemented in such a way because changes in one domain no longer affect the other domain:

```
d1::x:="y":
d::x;
  FAIL
```

Domain Elements

The concept of domains attains meaning with the possibility of generating objects - domain elements - that can have a domain as a type. Exactly as the number i has the type `DOM_INT` in MuPAD so the user can generate objects for any domain that have this domain as their type. A domain element can be generated with the aid of the function `new`:

```
e:=new(d,5,6,7):
```

generates an element of the domain d which contains the values 5, 6, and 7 as entries. The user can imagine a domain element as a sort of list. These entries can be accessed with the aid of the function `extop`:

```
extop(e,2);
  6
```

With the function `extsubsop` the entries can be changed (similar to the function `subsop`). The function `extnops` is used to determine the number of entries in a domain element.

Overloading Functions

The decisive point in connection with domain elements is that most system functions can be overloaded. This means that a system function which comes across a domain element searches for a method in the domain of the element with its name as an index, which can take over its execution. In a domain it can be easily laid down, for instance, how the sine of an element is to be calculated.

```
d::sin:=proc(x)
  begin
    sin(extop(x,1));
  end_proc:
sin(e);
  sin(5)
```

The function sin calls the found method with all its own arguments. Therefore it is also possible with the aid of domain elements to give the sine more than one argument. The functions `_plus` and `_mult` - which represent + and `*` - can also be overloaded. However, the usual simple overload mechanism does not come into play. Instead of this the method found in a domain is only called with all arguments that lie in this domain. From the individual solutions of the method an equivalent datum is constructed. Connections in an algebraic structure are therefore easily implemented.

Internal Methods

Not all operations on data are carried out by system functions in MuPAD. Domains also contain indices which do not represent system functions but are still important for the influencing of the behaviour of elements in a domain. Among these are:

"evaluate": Determines how a domain element is evaluated. If no entry is present the element evaluates to itself.

"func_call": Determines how a domain element behaves as the zero operand of a statement. With the aid of this method the programmer can generate his own types, which can represent functions or algorithms.

```
d::func_call:=proc(f,x)
  begin
    extop(f,1)*x^2 +
      extop(f,2)*x+extop(x,3)
  end_proc:
e(2);
  39
```

Domains

Holger Naundorf

Domains are the key to object oriented programming in MuPAD. Here we would like to briefly introduce the possibilities offered by these.

Introduction

Many computer algebra system users have the wish, sooner or later, to define their own types. For instance, MuPAD contains no type that represents the elements of the group Z_7 with addition. Even if this type should exist in a computer algebra system there is an infinite number of other types, all of which cannot be covered. If the user wants to write an algorithm that works on this group he must make sure that the combining of the elements is carried out correctly. Of course, the developed program is tailor-made for this group only. If the same algorithm is to be used for elements of other groups then it must be newly implemented. Normally only a few changes need be made - the combinations are replaced by the correct ones - but this is also error prone and time consuming. Theoretically, in MuPAD the user can write a function that automatically converts this program into one relevant to this function. However, this function is still to be written and the number of different algorithms generated take up a lot of memory space and all do nearly the same thing.

Domains

For this reason it is possible in MuPAD to define individual data types - the domains. A new domain can be generated with the aid of the function domain:

```
d:=domain():
```

A domain is a collection of data, that, with the aid of indices and the function `domattr`, can be read-out and altered. Both the indices and the data can be any MuPAD data.If no entry exists in the relevant index then `domattr` returns the value `FAIL`.

```
domattr(d,"name"):="example":
domattr(d,"name");
   "example"
domattr(d,"x");
   FAIL
```

In the description already given domains are very similar to the data type `DOM_TABLE`. This similarity is not accidental; for the implementation of domains the principle of tables has been used. However, the most important point in domains is that not only the user has the possibility of reading data from a domain, but that the MuPAD system does this, too. Now the entry `d;` returns `example` as its solution. For the domain output MuPAD returns the datum found in the index `"name"` - the object to be given out determines itself how it is to be given out. In general, this is referred to as object oriented. In many other situations MuPAD tries to read data from a domain. In MuPAD, strings are nearly always used as indices because most of the time these have an intuitative meaning and are not changed during evaluation. In this way the user does not have to protect the index with the function `hold` during access. In order to save the user paperwork MuPAD contains the operator `::`, a short form of `domattr`. The second argument must be a string which is allowed as the name of a variable. The second argument is written without quotes. With this the entry `d::name` is equivalent to the command `domattr(d,"name")`. In contrast to all other data types in MuPAD, domains are not copied before they are changed. Through this a change in a domain has an effect on all positions which refer to this domain. This is called the reference effect. In this way it is possible for a domain to contain a datum which has got this domain as a sub-datum:

polynomials can be divided if an exact division without remainder is possible.

`pdivide` carries out a pseudo-division of two polynomials.

`icontent` calculates the integer contents.

`norm` returns the norm of a polynomial.

`evalp` evaluates a polynomial at one ore more points.

`genpoly` generates a polynomial from a number by a radix expansion.

`iszero` tests if a zero polynomial is given.

The Polynomial Data Structure

In MuPAD, polynomials have a sparse distributed representation with lexicographic ordering of the monomials. This is true for both univariate and multivariate polynomials. The representation of a polynomial of the form

$$\sum_{i=0}^{n} c_i \cdot x_1^{e_{i,1}} \dots x_m^{e_{i,m}}$$

can be thought of as a list with the form

$$(c_n, (e_{n,1}, \dots e_{n,m}) \dots c_0, (e_{0,1}, \dots e_{0,m}))$$

Here $(c_i, (e_{i,1}, \dots e_{i,m})$ represents the ith monomial of the polynomial. Naturally, only those monomials are stored whose coefficients are not equal to 0. The sequence of the monomials in the list follows the lexicographic ordering of the exponents. The polynomial

$$3x^3y^2 - 7xy^5 + x - 5y^2 + 2$$

is equivalent to the list

(3, (3,2), -7, (1,5), 1, (1,0), -5, (0,2), 2, (0,0)).

The coefficients are stored as normal MuPAD expressions. The exponents, however, are stored as machine integers. The sum of the exponents of a term must not exceed 2^{31}-1, which should not cause practical problems. The exponents are densely stored, i.e. also exponents that have the value 0 are stored in the exponent list of a monomial.
Apart from the monomials a list of the variables $x_1,\dots,x_m$ and the coefficient ring are stored in the polynomial data structure. Furthermore the position of the leading monomial relevant to the degree ordering is also stored. The leading monomial can be quickly accessed relevant to both the lexicographic and the degree ordering.
“Outwardly”, for the user, the list of monomials described above is not visible. The function `op` always returns a polynomial expression as the first operand of a polynomial.

A Recursive Representation

Although polynomials are distributively represented, with the aid of domains a recursive representation is easily realized. For this purpose the MuPAD library contains the function `Poly`. With the aid of `Poly` domains can be generated whose elements are MuPAD polynomials.
With the aid of these domains polynomials can be defined whose coefficients are MuPAD polynomials, in other words they have a recursive representation. Here the advantages of domains becomes clearly visible.

must make certain basic operations available so that it can be used as a coefficient ring. (An introduction to the domain concept in MuPAD is given in the article "Domains" in this issue of mathPAD.)
The use of a domain as a coefficient ring has the advantage that the user receives a normal representation of the coefficients, provided that the domain can realize with such a representation. For expressions this is not always certain. (MuPAD has, for instance, no normal representation of rational functions. If the user uses MuPAD expressions to represent rational functions as coefficients then the zero function cannot always be recognized. With this the user may get polynomials whose coefficients are not equal to 0, but are a representation of the 0 in the ring of rational functions. Such polynomials are not in canonical form.)

Polynomial Arithmetic

Polynomials can be negated, added, multiplied, exponented and differentiated. The inverse of a polynomial is not defined. Polynomials can be used as operators. In this case they are considered polynomial functions.
In a sum or product all polynomials of one type are grouped together and then processed in these groups. A type conversion is not carried out. The expression

```
2 * poly(x,[x])
```

for instance stays like it is and is not returned as the polynomial `poly(2*x,[x])`. It would be very difficult to guess the type of the solution that the user wants. In the example above the desired solution could also be the expression `2*x`.
The user can work as usual with the functions `op`, `subsop`, `subs` and `subsex` on polynomials. However, one should note that these functions do not work very efficiently on the "visible" polynomial expression. In this case, one of the special polynomial functions should be used whenever possible.
A call of `op` with a polynomial as an argument always returns three operands: A polynomial expression, a list of variables and a representation of the coefficient ring.
Polynomials are never implicitly evaluated due to speed. Also functions that have a polynomial as an argument and return an expression as the solution do not evaluate the solution. If the solution of such a function is to be evaluated the call must be enclosed in `eval`.

Operations on Polynomials

The polynomial operations made available in MuPAD version 1.2 will only be listed in brief here. Further information is available in the user's manual. As already mentioned an expression and optionally a list of variables can be given instead of a polynomial in all functions. This can make interactive use easier.

`poly` converts an expression into a polynomial. Furthermore, with `poly` the user can change the type of a polynomial.

`expr` converts a polynomial into an expression.

`degree` returns the total degree of a polynomial or the degree relevant to a variable.

`degreevec` returns a list of the exponents of the leading term.

`lcoeff` returns the leading coefficient.

`lmonomial` returns the leading monomial.

`lterm` returns the leading term.

`nthcoeff` returns the nth coefficient.

`nthmonomial` returns the nth monomial.

`nthterm` returns the nth term.

`nterms` returns the number of terms in a polynomial.

`tcoeff` returns the coefficient of the last term.

`coeff` returns a sequence of all coefficients or the coefficients of a term with a certain factor.

`mapcoeffs` maps a function to the coefficients of a polynomial.

`multcoeffs` multiplies all coefficients of a polynomial with a factor.

`divide` carries out an Euclidean division of two polynomials. Furthermore, multivariate

Polynomials

Klaus Gottheil

From version 1.2 on special data structures and algorithms for polynomials will be made available in the MuPAD kernel. These enable quick basic operations to be carried out on polynomials.

Introduction

The computer algebra systems Maple and Mathematica do not know any special data structures for polynomials. They use "normal" expressions for the generation of polynomials. This has the advantage that no conversion of polynomials and expressions is necessary. Furthermore, the use of expressions as polynomial is intuitive because, from the users point of view, certain expressions and polynomials are "the same".

The MuPAD team decided in the planning of version 1.2 to introduce some special data types for polynomials (`DOM_POLY`). One of the reasons for this is that in MuPAD the terms of sums are sorted by the internal simplifier. This sorting does not conform with the canonic generation of polynomials (of any ordering). Another reason is the memory space occupied by expressions. The new data structure is far more compact.

Apart from polynomial expressions the variables and coefficient ring are also stored in a polynomial data structure. This type information is useful when programming using polynomials otherwise they must be transferred as extra parameters.

To make the life of an interactive user easier expressions can be given as arguments instead of polynomials in the relevant functions. These are then internally converted to polynomials.

The User's Point of View

As already mentioned polynomials have the data type `DOM_POLY`. A normal expression can be converted into a polynomial by the function `poly`. The call

```
p := poly(x*(x + a*y) + 12, [x,y]);
```

generates the polynomial `p` with the variables `x` and `y`.

Polynomials are given out in functional form, like tables, with the aid of the function `poly`. The polynomial `p` defined above is printed as

```
poly( x^2 + a*x*y + 12, [x,y] )
```

As a special feature the user can also define the coefficient ring of the polynomial. Apart from polynomials with expressions as coefficients the user can also define polynomials for the ring $\mathbf{Z}_n := \mathbf{Z}/n\mathbf{Z}$ with positive integers n and also with domain elements as coefficients.

The ring $\mathbf{Z}_7$ is written as `IntMod(7)` in MuPAD ("integer modulo 7"). The "ring of MuPAD expressions" is signified by `Expr` in the polynomials.

Particulars of the Coefficient Ring

From the particulars of the coefficient ring the type of polynomial can be deduced. The residue class ring $\mathbf{Z}_n$ is one of the most required rings. Polynomials in the form $\mathbf{Z}_n$ are used, for instance, for factorization. Operations on these polynomials must be as fast as possible.

One possibility of calculating in $\mathbf{Z}_n$ is to redefine the operations like `_mult` and `_plus` before calculation so that they calculate in $\mathbf{Z}_n$. When the polynomial operations evaluate the coefficients as normal, due to the changes in the basic operations, the calculation is carried out in $\mathbf{Z}_n$. However, this has two disadvantages: Firstly the continual redefining of the basic operations and the management of the "current operations" is tiresome, secondly the basic operations as MuPAD procedures are very slow when compared with the kernel routines. Due to this the basic operations for possible coefficient rings are implemented in the kernel.

Apart from $\mathbf{Z}_n$ and expressions, a domain can also be used as a coefficient ring. This domain

and not simplified any further.
The function `val` also has an expression as a parameter, which, exactly like `hold` is not evaluated. Furthermore all identifiers in the expression which are bound to a value, are substituted with this. This expression is then evaluated no further, but immediately returned as the solution of `val`. For instance after the assignment

```
a:=b; b:=2;
```

the expression

```
val(a+3+0)
```

is evaluated to `b+2+0` and the expression

```
val(b+4)
```

to `2+4`, but both are simplified no further. The most important use of `val` is the quick and direct access to the value of a variable, without a time consuming evaluation of the value taking place.
The function `level` is given an expression as its first parameter and, optionally, an integer as its second parameter. An expression with the form `level(`a`,`n`)` is evaluated, by the expression a being evaluated to the maximal substitution depth n instead of to the "current" depth given by `LEVEL`. The function `level`, therefore, temporarily overwrites the variable `LEVEL`. The variable `MAXLEVEL` remains unchanged, on reaching the substitution depth `MAXLEVEL` an error message will still be given. If the second parameter is missing then a nearly infinite maximal substitution depth is assumed (to be exact 2147483647).
On the interactive level, with the statement

```
a:=b; b:=c; c:=3;
```

the expression

```
level(a,1)
```

is evaluated to `b` and the expression

```
level(a,2)
```

to `c`. Also, the function `level` can be especially used to fully evaluate a variable in a function.
`context` can be used for evaluation of arbitrary expressions in the context of an calling procedure instead of the actual context. If `context` is called in a procedure, then the argument is firstly evaluated in the actual context. The result is then evaluated in the context, from where the procedure has been called. For instance:

```
i := 2;
f := proc(x)
   option hold;
begin
   x, context(x)
end_proc
```

By calling `f(i)` the sequence `i,2` is returned.

If matrixes or non-commutative operators are to be multiplied using symbols then the user will need to use his own functions. The calculated expression is then returned as the solution of the evaluation.

The Substitution Depth

In the evaluation of identifiers the *substitution depth* plays an important role. What is to be understood by this?
If the original expression is considered before evaluation then the substitution depth is 0. Each time an identifier is substituted by its value the substitution depth is increased by 1 and the value of the identifier is evaluated with this new substitution depth. After evaluating the value the old substitution depth is taken on again.
This means that the substitution depth gives the recursive depth of the evaluation of identifiers.
Evaluation can be controlled by declaring a maximal substitution depth. If the maximal substitution depth is reached then the identifier is not evaluated any further.
Interactively entered expressions are fully evaluated (fully means the user can still give a maximal substitution depth). However, normally the expression is evaluated to the point where no unbounded identifiers are present. After the assignment

```
a:=b; b:=c; c:=13;
```

the variable `a`, when interactively entered, is evaluated to `13`, as expected.
In procedures the maximal substitution depth is 1, i.e., identifiers are substituted by their values, these are, however, not evaluated further. In the following procedure `f` the local variable `z` is evaluated to `y` and not to the parameter `x`:

```
f:= proc(x)
  local y, z;
begin
  z:=y; y:=x; z
end_proc
```

The call `f(13)` returns the solution `y` and not `13`, as might have been expected.

Control of the Substitution Depth

The user can explicitly set the maximal substitution depth with the variables `LEVEL` and `MAXLEVEL`.
During evaluation substitution is carried out (when necessary) to the depth `LEVEL`. With the assignment

```
a:=b; b:=c; c:=13;
```

`a` is normally evaluated to `13` in interactive input. If however `LEVEL` is set to `2` then `a` is evaluated to `c` and `b` to `13`.
By the way, `LEVEL` can be set to values other than 1 during procedures. In the procedure

```
f:= proc(x)
local y, z, LEVEL;
begin
  LEVEL:=2; z:=y; y:=x; z
end_proc
```

`z` is evaluated to the parameter `x` and not to `y`. The call `f(13)` results in `13` and not `y`.
The variable `MAXLEVEL` is used to detect infinite recursions, as in the seemingly harmless statement sequence

```
a:= a+1; a;
```

If `a` has no value before the assignment, then `a` is evaluated to `a+1` in the second statement, this then to `a+2`, this to `a+3` and so on, ad infinity.
In order to detect such situations, a substitution depth of `MAXLEVEL`, during evaluation, is taken as an error and evaluation is terminated with the message “Recursive Definition”. Naturally this is only a heuristic, which may lead to false error messages in certain cases, but in most cases it is correct.

Fine Control of Evaluation

So far, for controlling evaluation, the procedure option `hold` and the variables `LEVEL` and `MAXLEVEL` have been described. The system functions `hold`, `val`, `level` and `context` allow even greater control of evaluation.
The function `hold` can have any expression as a parameter. This is not evaluated but returned as the result of the call. So, with `hold` the evaluation of an expression is prevented. For instance, the expression

```
hold(a+2+0)
```

is evaluated to

```
a+2+0
```

Here, the expression `g()` always returns the value `16`. In this context, the value of `z` outside of `g` makes no difference to the call of `g`. During the evaluation of `f(2)` in the procedure `g`, the local variable `z` from `g` is used (this was not "known" in the definition of `f`).

Execution of System Functions

In the system functions, parameter evaluation is dependent on the given function. Most functions evaluate all their parameters before the execution of the "actual" function. However, some functions do not, or only partially, evaluate their parameters.
The assignment `_assign()`, for instance, does not evaluate its first parameter (the right hand side of the assignment), if this is an identifier. Otherwise the variable `b` after the assignments

```
a:=b; a:=2;
```

would have the value `2` — an extremely confusing situation. However, the left hand side of the assignment is evaluated. Through the assignment

```
b:=2; a:=b;
```

the value `2`, and not the value `b`, is bound to `a` during interactive input.
Another example for incomplete evaluation of parameters is the procedure definition with `proc... end_proc`. In this case the procedure definition is not evaluated. It would be very confusing if, with the assignment

```
x:=13;
f:= proc(x) begin x*2 end_proc;
```

the (naturally incorrect) procedure definition

```
proc(13) begin 26 end_proc
```

was generated.
Of course, the execution of system functions is dependent on the given function. In the following, only a few directions for the execution of functions, like "+", "*" and "^", which represent the algebraic operations, are given:
With these functions the evaluated parameters are at first simplified and then transformed into standard form, in order that the expressions can be more easily compared with other expressions.
Simplification is carried out with simple rules. Firstly the associativity of the operation is used, in order to "flatten" the expression tree. Expressions in brackets of the form

```
_plus(a, (b,3))
```

can be simplified to

```
_plus(a,b,3)
```

due to the associativity of addition.
After this rules like $a + 0 = a$ or $a \times 0 = 0$ are used. As a consequence of this all 0's are removed from sums and multiplication with results in 0. The use of these rules can be — and this drawback cannot be denied — quite hazardous. One such hazardous rule is $a\ /\ a = 1$; because it is possible that later a can take the value 0, so the substitution with 1 is wrong in this case. If the variable `a` is defined by

```
a:= b*(c+1)/b;
```

with `b` not yet having a value, then `a` is given the value `c+1` through simplification. If afterwards `b:=0` is defined, then the value of `a` is still falsely defined as `c+1`, the variable `a` does not "know" the variable `b` from its definition any more.
Unfortunately, here, the user must live with this hazard, as otherwise expressions can very quickly "explode". (Exploding expressions, while being physically harmless to the user, can render the system useless and therefore cause psychic damage.) The user should be aware of the hazards of the simplification rules.
The calculation of constants also belongs to the simplification rules. This includes transformations like

```
_plus(3,a,4) → _plus(a,7)
```

After simplification the expression is normalized. Here, for instance, in sums and products constants are moved to the end of an expression and the remaining sub-expressions are sorted. The expression

```
z*3*a1*a
```

is transformed into

```
a*z*a1*3
```

by sorting. The sorting order is difficult for the user to predict, but luckily this is nearly always not relevant.
The expression `b*a` is always transformed to `a*b` through sorting. Therefore, sums and products must always be conceived as commutative.

If the evaluation of the first subtree of a function call results in an identifier, then the function is not bound to a definition. In this case, all the remaining subtrees are also evaluated and the new tree is returned in the form of a "formal function call". If `f` is, for instance, an identifier to which no value is bound, then `f(4+3)` will be evaluated to `f(7)` because `_plus(4,3)` will be evaluated to `7`.
If the first subtree of a node is evaluated neither to a function definition nor an identifier then the evaluation is terminated with an error. Therefore the expression `"_plus"(2,3)` is invalid, because the string cannot be evaluated to the function definition for the sum.

Execution of Procedures

As already mentioned, a function is executed with the subtrees of the call as parameters. If the function is a procedure which is defined in the language, then there are two forms of call: Normally, the parameter expressions are evaluated and the procedure is executed with these parameters. The statements

```
f:= proc(x)
begin
  x*2
end_proc;
a:=2;
f(a);
```

define the procedure `f` and then assign the value `2` to `a`. As expected the call `f(a)` returns the value `4`.
Conversely, if in the procedure definition the option "`hold`" is given, then the parameters are not evaluated before execution. The statements

```
g:= proc(x)
  option hold;
begin
  x*2
end_proc;
a:=2;
g(a);
```

define the function `g` and again assign the value `2` to `a`. The call `g(a)` then returns the expression `a*2` as the solution: The parameter `a` is not evaluated, the function `g` returns the identifier `a` multiplied by `2` as the result.
The execution of a procedure takes place in three steps: Firstly the actual parameters are bound to the identifiers of the formal parameters. Then the local variables of the procedure which have been declared "`local`" are established. In the last step the procedure is evaluated, as already described. The last evaluated value is then the return value of the procedure call.
The local variables are virtually "fresh" identifiers, to which, at first, no value is bound. If identifiers with the same name exist before the procedure call, then these values are not visible during the procedure. Instead, during the evaluation of the identifiers the local variables are used. All other identifiers are handled as global variables; in this context the current, valid values of the identifier for this run time are used.
In the following procedure `f`, for instance, `x` is the parameter and `y` is a local variable, while `z` is a global variable, because the identifier `z` is declared neither as a parameter nor a local variable:

```
f:= proc(x)
  local y;
begin
  y:=2; x*y*z
end_proc
```

After the assignments `y:=3;  z:=NIL;` the value `3` is bound to `y`, while `z` (due to the assignment `z:=NIL`) has no value. Because in the expression `x*y*z` the local variable `y` is evaluated to `2`, afterwards the expression `f(2)` gives the value `z*4`. However, after the assignment `z:=3; f(2)` returns `12`.
The described method has one consequence, which is called *dynamic scoping*: For global variables those identifiers are used which are visible at the execution time. In most "normal" procedural programming language, like Pascal, those variables are used which are statically declared through the program text (through the outer procedure). This is called *lexical scoping*.
In the following procedure `g`, the local variable `z`, which was global in connection with the function `f` above, is defined:

```
g:= proc()
  local z;
begin
  z:= 4; f(2)
end_proc
```

Evaluation

Klaus Gottheil

In MuPAD terminology, evaluation is the execution of an expression by the interpreter. It can be said, with good reason, that evaluation is the "actual purpose" of the interpreter. In this context, what does "execution" mean?

MuPAD expressions

Before evaluation is tackled, the term *expression* must be explained: A expression is — expressed very simply — a tree.
The leaves of an "expression-tree" are basically identifiers (i.e. variable identifiers), constants (numbers, Boolean constants, strings) and special constructs, that are needed to identify system functions and procedures.
Identifiers have, apart from the fact that they have names, another useful characteristic: Values can be bound to identifiers, i.e., an expression can be assigned to an identifier. (This is done with the assignment operator "`:=`".) Through evaluation, an identifier, which is bound to a value, is substituted by this value (i.e., by this tree). For instance, with the assignment `a:=b; b:=2;` the value `b` is bound to the variable `a` and the value `2` to `b`.
The inner nodes of a tree are function calls or complex data structures, like lists, sets or tables. In the following, complex data structures will not be dealt with, in order not to make the description too complicated.
With function calls the first subtree identifies the function and the other subtrees identify the current parameters of the call. Such a node can be written as

$$F (P_1,..., P_n)$$

in "functional notation", where the function F and the parameters P_i can stand for subtrees (not only for leaves). In Lisp these nodes would be written as (F P_1 ... P_n).
The statements of the MuPAD language are all realized as functions and are also accessible to the user as functions (e.g. as `_assign` for the assignment "`:=`" or as `_if` for the "`if then else`" statement). For each statement there is a so called *underline-function* (so-called, because by convention all the names of these functions begin with an underline "_"). The statement

```
if a < b then a+2 else b end_if
```

can be just as well written as

```
_if( _less(a,b), _plus(a,2), b)
```

Here the legibility obviously suffers (Lisp accustomed bracket-artists will, of course, have no problems with this).

Evaluation of expressions

Through evaluation an expression is transformed into another one, in other words one tree is replaced by another. Evaluation is a recursive process:
A leaf, which is not an identifier, is not altered through evaluation. Likewise, if an identifier has no value bound to it, then it is also not altered. When an identifier has a value bound to it, then this value (i.e. this expression tree) is evaluated and the solution returned. Following the statements `a:=b; b:=c; c:=2;` the identifier `a` is normally, during interactive input, evaluated to `2`. (More to the qualifications "interactive" and "normally" later.)
As already mentioned the inner node of an expression is a function call. Initially the first subtree of the node is evaluated. If the evaluation of the first subtree results in a function definition then the corresponding function is carried out with the remaining subtrees of the node as parameters. During the evaluation of the call `sin(a+b+2)` the identifier `sin` is evaluated to the function definition of the system function "sinus" and the function can then be called with the expression `a+b+2` (or the equivalent `_plus(a,b,2)`) as a parameter.

element in its queue named *name*. As special queues those with the name "`work`" are available. If a cluster is idle, it tries to read a datum from this queue and evaluate it.

Pipes allow the establishment of a direct two-way communication channel between two clusters. (Because any cluster can write into a queue, it is possible that the receiver has problems identifying the sender of a message in a queue.) With `writepipe(`*name*`,`*i*`,`*value*`)`, the expression *value* is written in the pipe called *name* of the cluster *i*. Through `readpipe(`*name*`,`*i*`)` a cluster reads the next message in the pipe *name* connected to cluster *i*.

Net-variables constitute a distributed shared memory. An identifier `a`, which is assigned the value `b` by `global(a,b)`, has this value in all clusters.

```
pi:= proc()
  local DIGITS;
begin
  DIGITS:=200;
  float(PI)
end_proc
```

returns π to exactly 200 digits. After calling `pi()`, the system variable `DIGITS` still has the same value as before the call, i.e. it was only altered during the procedure `pi`. However, for this the environment variable must be declared as `local`!

Parallel statements are easy to handle in the domain of *micro-parallelism*, and offer an optimal adaptation to the hardware in this domain. Micro-parallelism can be employed efficiently when communication time between the processes involved is minimal. The system carries out load balancing and process communication automatically. The user only has to mark the parallel executable tasks. The constructs of *macro-parallelism* make a means available for the optimal use of the network topology. Load balancing and process communication are no longer carried out automatically but have to be done by the user.

Micro-parallelism is oriented on the model of the shared memory machine. With the available constructs processes are generated and deposited on a task stack. Idle processors examine this stack and remove tasks for processing. There are a parallel `for`-loop and a parallel block, both of which are also available in the sequential version of MuPAD

```
for identifier from beginning to end
step step width parallel
  private private variables;
  statement sequence
end_for
```

The range `from` *beginning* `to` *end* is run through with the step width *step width*. For each run the index variable *identifier* is set to the current value and a task is generated to execute the *statement sequence*, which is then deposited on the task stack. The *private variables* are a (possibly empty) series of identifiers which are separated by a `,`. The usage of these variables by the tasks is analogous to local variables in procedures.

A second form of the parallel `for`-loop runs through all operands of an expression:

```
for identifier in expression parallel
  private private variables;
  statement sequence
end_for
```

As in the sequential case, all expression operands are assigned, one after the other, to the identifier. For each value a task is deposited on the task stack.

A parallel loop is only then ended when all the tasks it has generated are finished. The return value is a expression sequence containing the return values of each task. The result of `for i in [1,2,3,4] parallel i^2 end_for` is the sequence `1,4,9,16`.

Parallel blocks execute statements in parallel:

```
parbegin
  private private variables;
  statement sequence
end_par
```

Each statement in the statement sequence is deposited as a task on the task stack. In order to execute statements sequentially inside a parallel block, a sequential block is available:

```
seqbegin
  statement sequence
end_seq
```

Macro-parallelism can be efficiently used in distributed systems. In this case, all communication has to be explicitly defined by the user. The system does not care for load balancing or scheduling. As a model a network of clusters, each running independently is used. Clusters can communicate with each other by global variables or by exchanging messages via queues or pipes. Clusters are addressed via positive integers.

Queues store incoming messages for a cluster. Clusters may send messages to the queues of other clusters. Queues have names which can be any expression. A queue is organized on the first in first out principle. With `writequeue(`*name*`,`*i*`,`*value*`)`, the expression *value* is written in the queue called *name* of the cluster *i*. With `readqueue(`*name*`)`, a cluster reads the first

are executed. The `otherwise`-part may be omitted.
With the `break` command the user can jump to the end of a `case`-statement, the rest of the statements in the `case`-statement are not executed. With the `next` command the comparison of the expression with the values of the `of`-parts is resumed: Like at the beginning of the `case`-statement, the comparison of the expression with the values of the `of`-parts is resumed starting with the `next`-statement and the statements are executed after a matching value is found.

Procedures have the form

```
proc (parameters)
   local local variables;
   option options;
   name name;
begin
   statement sequence
end_proc
```

The *formal parameters* are a (possibly empty) series of identifiers separated by a `,`. The *local variables* are also a sequence of identifiers separated by a `,`. *Options* are keywords separated by a `,`; `hold` and `remember` are accepted. The *name* can be any expression and is used only for output, a procedure can not be called by using its name. The declaration of the local variables, options and name may be omitted. If the specification of the name is missing, then by assigning a procedure to an identifier, the identifier is taken as the procedure's name. With

```
max:= proc(x,y)
begin
   if x > y then x else y end_if
end_proc
```

the procedure `max` is defined, it calculates the maxima of two numbers. With the assignment `f:=max`, `f` is assigned the same procedure as `max`. Procedures are data like expressions or lists.
On calling a procedure, the formal parameters are assigned copies of the actual parameters. (A "call by value" is executed, a "call by reference" does not exist.) Then the body of the procedure is executed. At the beginning local variables are unassigned.
A *free variable* in a procedure (a variable that is neither a parameter nor a local variable) is automatically a global variable. A procedure can be defined in the body of another procedure. In this case the local variables of the outer procedure can be global variables of the inner one, as in

```
g:= proc(x)
   local y, f;
begin
   y:= x;
   f:= proc(x) begin x*y end proc;
   f(x)
end_proc
```

Here the inner procedure `f` is a local variable of the outer procedure `g`, `y` is a local variable of `g` and a global variable of `f`. `g(x)` evaluates (very clumsily) to `x^2`.
Dynamic scoping is used for the free variables, i.e. global variables are bound during run-time. (In a procedure, those global variables which were last defined in one of the procedures called so far, are used.)
Normally, a procedure returns the value of the last executed statement. A procedure can also be left by using the function `return`. The parameters of `return` are the return values of the procedure. The above mentioned procedure `max` can also be defined as follows:

```
max:= proc(x,y)
begin
   if x > y then return(x) end_if;
   y
end_proc
```

The return value of a procedure can be an expression sequence, with this a procedure can return several values at the same time.
Options for procedures are, as already mentioned, `hold` and `remember`. Normally, the current procedure parameters are evaluated before the procedure is executed. This mechanism is prevented with the option `hold`. The option `remember` causes a procedure to "remember" the return values and if possible simply to return these values instead of executing the procedure again. By this means especially recursions can be greatly accelerated at the cost of using more memory.

Environment variables are system variables that are automatically reset to their former values after a procedure has been executed. The following procedure

If-statements have the form

```
if condition then
   statement sequence
elif condition then
   statement sequence
   ...
else
   statement sequence
end_if
```

The conditions must be Boolean expressions. If the first condition results in `TRUE`, then the statements in the `then`-part are executed, if not then the conditions in the `elif`-part are examined. If these result in `TRUE`, then the statements in the `elif`-part are carried out. Any number of `elif` parts, whose conditions are examined one after the other, can follow. If none of the conditions result in `TRUE`, then the statements in the `else`-part are executed. The `elif`- and `else`-parts may be omitted. Examples are `if x > y then x else y end_if` or `if a mod 3 = 0 then a:= a+1; return(a) end_if`.

For-loops have two possible forms. In the familiar form a range of numbers is run through:

```
for identifier from beginning to end
step step width do
   statement sequence
end_while
```

The range *beginning* to *end* is run through with the step width *step width*. In each run the index variable *identifier* is set to the current value and the *statement sequence* is executed. `for i from 2 to 8 step 2 do print(i) end_for` results in the output of `2`, `4`, `6` and `8`.
The step width may be omitted, in which case the step width 1 is used. In `for i from 1 to 3 do a.i:= i end_for` the assignments `a1:=1`, `a2:=2` and `a3:=3` are executed.
The alternate form *end* `downto` *beginning* can be used instead of *beginning* `to` *end*. In this case the loop runs backwards and the index variable is reduced by the step width each time. In `for j from 4 downto 2 do t[j]:= t[j-1] end_for` the assignments `t[4]:= t[3]`, `t[3]:=t[2]` and `t[2]:=t[1]` are executed in the given order.

In the second form of the `for`-loop all operands of an expression are run through:

```
for identifier in expression do
   statement sequence
end_for
```

The operands of the *expression* are assigned, one after another, to the *identifier* and then the loop body is executed each time. In `for t in a+b+c do print(t) end_for` the values for `a`, `b` and `c` are displayed one after the other.

While-Loops have the form

```
while condition do
   statement sequence
end_while
```

Here, the *statement sequence* is executed as long as the *condition* results in `TRUE`. The condition is evaluated before the statement sequence is executed.

Repeat-Loops have the form

```
repeat
   statement sequence
until condition end_repeat
```

Here, the *statement sequence* is executed as long as the *condition* results in `FALSE`. The condition is evaluated at the end of the statement sequence.
A loop can be terminated by `next` or `break`. With `next` the execution of a statement sequence is terminated and the next run of the loop is started, with `break` the loop is terminated.

Case-statements have the form

```
case expression
   of value 1 do statement sequence 1
   of value 2 do statement sequence 1
   ...
   otherwise statement sequence n
end_case
```

Here, the *expression* is compared with the expressions *value 1*, *value 2*, etc. one after the other. If the expressions match, then all the following statement sequences of the `case`-statement are executed. (Also the sequences of the remaining `of`- and `otherwise`-parts!) If the expression does not match any of the `of`-parts, then only the statements of the `otherwise`-part

rally, when defining new identifiers the result must be a syntactically valid identifier.
The `@` operator is used for composition of functions. For example, `(tan@exp)(x)` returns `tan(exp(x))`.
The `::` operator is the short form of `domattr`, `D::a` is equivalent to `domattr(D,"a")`.

The type of an expression is the domain to which the expression belongs. With simple expressions this is identical to the expression type. Complex expressions have the type `DOM_EXPR`. The type can be ascertained by the function `domtype`.

The expression type is the type of the corresponding object for simple expressions and for operators the operator type. Expression types are given by the function `type`. `type(`*expr*`)` returns the type of *expr*, `testtype(`*expr*`,`*type*`)` results in `TRUE` when *expr* has the type *type*. The expressions `type(a*b)`, `type(x)` or `type(a+b+c)` result in `"_mult"` (product), `DOM_IDENT` (identifier) or `"_plus"` (sum). The expression `testtype(x,DOM_IDENT)` results in `TRUE`.

The Operands of an expression can be determined by the function `op`. `op(`*expr*`)` results in a sequence containing all the operands of the expression `e`. `op(`*expr*`,`*n*`)` returns the nth operand of `e`. `op(`*expr*`,0)` gives the operator of `e`. If `e= a+b+sin(3*c)` then `op(e)` is the sequence `a,b,sin(c*3)`, `op(e,1)` is `a` and `op(e,0)` is `_plus`.
By using `op`, operands can also be ascertained recursively. If `e` is defined as above, then `op(e,[3,1])` results in the expression `c*3`, `op(e,[3,1,2])` is `3` and `op(e,[3,0])` is `sin`.
Furthermore, by using `op`, elements of complex data structures, such as sets, lists and tables, can be accessed. Thus, `op([a,b,c],2)` results in the second element, `b`, of the list.

The number of operands in an expression can be determined by the function `nops`. The expression `nops(a+17+3*c)` results in the value `3` (for the three operands `a`, `17` and `3*c`).

The substitution of partial expressions can be carried out with the functions `subs`, `subsex` and `subsop`. On calling `subs(`*expr*`,`*old*=*new*`)`, the partial expression *old* in expression *expr* is replaced by *new*. The partial expression *old* must be complete. It is not possible to replace some operands of an operator singly and retain the others. Therefore, in the expression `e` above `3*c` can be replaced by `d` by using `subs(e, 3*c=d)`, but the partial expression `a+b` cannot be replaced by the function `subs`.
With `subsex` incomplete partial expressions can be replaced, the call is the same as that for `subs`. `subsex(e,a+b=d)` results in the expression `d+sin(3*c)`.
With `subsop(`*expr*`,`*n*=*new*`)` the nth operand of *expr* is replaced by *new*. `subsop(e,3=d)` results in the expression `a+b+c`. Internal partial expressions can be specified by giving paths instead of the simple number n (see `op`).

Statements are expressions, assignments or conditional statements. During interactive input, a statement must always be terminated by `;` or `:`, in order to be carried out. When `:` is used the result of the statement is not shown on the screen.

Statement sequences are series of statements which are separated by `;` or `:`. When `:` is used the result of the statement is not displayed on the interactive level.

Assignments normally bind values to identifiers, as in `a:=12+x`. They have the form *identifier* `:=` *expression*. After assigning `NIL`, as in `a:=NIL`, an identifier has no value.
An indexed identifier, such as `l[2]` (a list, table or array element), an identifier generated by the concatenation operator, such as `x.2`, a domain attribute, such as `D::a`, and a call of a function, such as `f(2)`, or a combination of such constructs can be placed on the left hand side of an assignment, instead of a simple identifier.
The result of an assignment is normally the right hand side of an assignment. If the assignment is `NIL` then the result is the identifier itself. The result of `a[3]:=12+x*0+4;` is `16`, for `b:= NIL` the result is `b`.
An assignment of `NIL` to the system variables results in them being set back to their default values. Therefore, these identifiers always have a value.

modulo n) and Domains. More than 20 functions are available for working with polynomials.

Domains are used for the definition of new data types by the user. By the function domain, new domains are generated, by domattr or the :: operator the methods of a domain can be accessed. Many system functions can be overloaded by domain methods. For instance by defining a domain method "_plus", the usual operator + can be used for addition of domain elements. Domains enable to implementation of polymorphic algorithms.

Expressions are data structures in MuPAD and not calculation statements as in most other programming languages. Expressions are not always directly evaluated. Evaluation is dependent on the context. During evaluation the identifiers are substituted by values, function calls by their solutions and arithmetic expressions are simplified and sorted for easier comparison.
Identifiers are normally recursively substituted as far as possible during the evaluation of expressions (so that in the expression no more identifiers with values are present). However, this does not apply to procedures. In procedures the identifiers are replaced by their values only once. An expression consists of a simple expression or an operator and its operands. A simple expression is a constant or an identifier, a function call, an expression sequence, a list, a set, a polynomial or a domain. Operators are operators like +, * and $. The operands of an expression are themselves any expression. Examples of expressions are a+3*PI+sin(x) or [f(i) i=1..n].
Tables and arrays cannot be directly included in expressions. There are only identifiers that have entire tables or arrays as values, or indexed identifiers, like t[3], which have a single array or table element as value.
A function call, with its actual parameters, is internally understood as an operator with its operand. Conversely for every operator there is a system function which realizes the operator. An operator is executed by calling the relevant system function with the operands as the parameters.

Trees are defined by the recursive definition above. Every expression is a tree, operators and function calls are inner nodes, operands and parameters are children, simple expressions are leaves.

Arithmetical operators are +, -, *, /, div and mod. div and mod are integer division and integer remainder respectively. 1/a mod b results in the modular inverse of a modulo b.

Rational operators are =, <>, <, <=, >, and >=.

Logical operators are and, or and not.

Set operators are union, minus (difference) and intersect.

Special operators are the range operator .., the sequence operator $, the concatenation operators . and @, and the domain attribute operator ::.
An range-expression has the form *lower_limit* .. *upper_limit*. The lower and upper limits are expressions that result in integers. Such a range represents the integers between the lower and upper limit.
The simplest form of a sequence-expression is *expression* $ *quantity*. Here an expression sequence with a *quantity* of elements is defined, with each element equal to the *expression*. The expression a $ 3 results in the sequence a,a,a.
With *expression* $ *variable* = *range* a expression sequence with the element *expression* is defined. Here, however, the running variable in the expression is replaced with the respective current value from the *range*. The expression i*PI $ i=0..3 results in the sequence 0,PI, 2*PI,3*PI.
A sequence of integers is defined by $ *range*. This consists of all integers between the lower and upper limits of the range. The expression $2..6 results in the sequence 2, 3, 4, 5, 6 and $-2..-2 in the number -2. The expression $1..0 results in the empty series.
The concatenation operator is used to concatenate lists and character strings, and to define new identifiers. [a,b].[c,d] produces the list [a,b,c,d], "Can"."teen" produces the character string "Canteen". The identifier force_x is defined by force."_x". Natu-

The Language in Brief

Klaus Gottheil

In this passage the language will be described — in about 6 pages. Naturally this is very short, but the most important language concepts should become clear. The evaluation of the language shall be described in the article "Evaluation".

Comments are one of the most important language elements. In MuPAD they are enclosed in `#`, as in `# comment #`.

Constants are numerical constants, the Boolean constants `TRUE` and `FALSE`, and character strings. Numerical constants are integers, e.g. `175`, rational numbers, e.g. `-17/3`, floating-point numbers, e.g. `12.3e-4`, and complex numbers, e.g. `2/5+4*I`. (When giving exponents the user must use a small `e`.) Character strings are enclosed in ", e.g.: `"Otto"`.

Numerical values can have any number of digits. The precision of floating-point numbers can be predefined with the system variable `DIGITS`.

Special Constants are `I` for the imaginary unit, `E`, `PI` and `EULER` for e, π and C respectively. The constant `NIL` is used to remove values from identifiers.

Identifiers are formed of by an alphabetical character followed by any number of alphabetical characters or digits, e.g. `deriv` or `x_12`. (The underline _ is an alphabetical character.) An identifier can have a value assigned to it by using the assignment operator`:=`. At first an identifier has no value assigned to it. An identifier is of no particular type and can be assigned any value.

Function calls have the form *procedure(parameter)* or *procedure()*. Here the parameter list is a series of expressions separated by a `,`. Procedure names are frequently identifiers, e.g. `reset()`, `sin(1)` or `append(1,x)`. Procedure names can also be given in the form of expressions.

Expression sequences are ordered series of expressions separated by the comma operator `,`, e.g. `2, x, y, 3/4`.

Lists are expression sequences enclosed in `[ ]`. The expressions are separated by `,`, e.g. `[a,b,3/7]`, `[x]` or `[]` (the empty list). The list entries can be accessed by `[ ]`, for instance, `l[3]` is the third element of list `l`. By using `append(l,7)` the value `7` can be attached to `l` and the new list is returned.

Sets are, like lists, expression sequences, which are enclosed in `{ }`, e.g. `{a, 2.23, Lisbet}` or `{}`. A set contains no identical elements and is not ordered.

Tables are any number of equations with the form `index=value`, where `index` and `value` can be any expression. Tables are produced by the function `table`, e.g. `t:=table()`. The table entries can be accessed by `[ ]`, e.g. `t[x]`, `t[7]` or `t[z]`. (Here `x`, `7`, `z` are the index values of the respective entries.)

Arrays have any number of dimensions and are used for the storing of vectors, matrices, tensors etc.. Arrays are produced by the function `array`, e.g. `v:=array(1..4)` (vector `v[1]`...`v[4]`), `m:=array(2..6,3..4)` (matrix `m[2,3]`, `m[2,4]`,... `m[6,4]`) or `t:=array(1..3,1..3,1..3)` (a three dimensional array containing `t[1,1,1]`,... `t[3,3,3]`). Arrays cannot be appended. At first the array elements have no values.

Polynomials are represented by a data structure which is created by the function `poly`, for instance `poly(x^3-2*x^2+x-1,[x])` or `poly(x*y^3+2*x-2,[x,y],IntMod(7))`. Admissible coefficient rings are `Expr` (expressions), `IntMod(n)` (ring of residue classes

What's New in MuPAD 1.2

Karsten Morisse

Since the release of MuPAD 1.1 a lot has happened. Naturally all errors that have been passed on to us (and those we have found ourselves) have been eliminated. However, we have invested more time in new and further developments. In this short article I would like to introduce the more important innovations.

The user can now define new data types (so-called *domains*) at the user's level with the aid of a new type (`DOM_DOMAIN`). With this it is possible to implement polymorphic algorithms: Instead of multiply implementing an algorithm for the different instances of an algebraic structure, a single generic implementation coupled with the definition of special domain methods is sufficient. Examples for working with domains can be found in the article "Domains" in this issue of the mathPAD journal. The quick handling of domains is enabled by them being implemented in the kernel. The library contains an experimental version of a category concept. A category represents a class of algebraic structures. By using categories generic algorithms for these classes can be implemented.

Many of the functions in the MuPAD kernel can now be overloaded, so that, for example, by defining a domain method `_plus` the normal operator notation for addition can be used.

The kernel has an intrinsic data type (`DOM_POLY`) for the handling of polynomials. This is a sparsely distributed representation. The work with this data type is supported by more than 20 functions. An overview of these functions is contained in the article "Polynomials" in this mathPAD journal. On the library level routines for the recursive representation of polynomials are also available. The library also contains experimental versions of gcd and factorizing algorithms for polynomials.

For the performance analysis of programs written in the MuPAD language a profiler is included. This gives information about the frequency and the duration of the programs called.

The graphic tool VCam has been completely reworked. This concerns both the handling of the interfaces and the internally used structures. Objects can now be drawn in colour.

The use of the debugger has become more user friendly. Breakpoints can now be combined with conditions so that, for instance, errors can be searched for very quickly and specifically in complex loops.

To avoid naming conflicts the library packages are implemented with the aid of domains. They are only loaded on demand.

Many new functions have been implemented (a specific list is contained in the on-line manual). The functionality of some functions has been reworked. Whilst doing this we have made every effort to guarantee upward compatibility.

MuPAD is a joint effort of many people. In addition to the authors of this issue major contributors were Gudrun Oevel (Graphics), Andreas Sorgatz (Compiler and Module Management development), Ralf Hillebrand (Prettyprint and Linux port). Mrs. McIntosh-Schneider was of great help in preparing the English manuscript.

We are grateful to our friend Waldemar Wiwianka, MuPAD's project leader from the beginning, who was tragically killed in an accident in July 1993. MuPAD owes a great deal to his enthusiastic work and the perspectives he helped to develop.

In case you would like to obtain a copy of MuPAD please send an e-mail to

MuPAD-distribution@uni-paderborn.de

A Reference Manual (in German) and a Tutorial (in English), which allows a guided tour through the system, are available from Birkhäuser Publishing house in Basel.

Editorial

Benno Fuchssteiner

At the occasion of the CEBIT 1994 the present issue of the mathPAD journal is entirely devoted to the computer algebra system MuPAD, a development of the AUTOMATH Institute at the University of Paderborn. On the following pages some of MuPAD's developers report in a leisurely style about the result of their work and the aims they have followed in the development of this system.

MuPAD is a parallel-processing, object oriented computer algebra system, of which sequential versions are available as well. The sequential MuPAD version is the topic of this mathPAD issue. MuPAD is designed to solve simple as well as complex problems in Science, Engineering and Mathematics. Special care has been taken to provide state-of-the-art interfaces for utmost ease-of-use even for absolute beginners in computer algebra. At present MuPAD is available on Unix Workstations, Macintosh, PC (under LINUX 0.99 p/14) and Unix based shared memory machines; ports to other platforms are coming.

The design of MuPAD was originally driven by the desire to efficiently handle large data (up to the Gigabyte range) generated by our algorithms for investigating the group theoretical structure of nonlinear systems. However, MuPAD outgrew this goal and was developed as a general purpose system and should be used as such.

As other computer algebra systems, MuPAD consists of basic system functions which, for speed and efficiency, were implemented in the kernel of the system. Further components are a high level language, allowing the user comfortable programming (including parallel constructs on sequential machines) and the interface modules.

The system functions include the necessary arithmetic of arbitrary length. This arithmetic is realized by incorporating the PARI[1] system into MuPAD.

A comfortable graphics module which allows efficient computation and display of two-dimensional and three-dimensional plots is provided with the system.

At present the MuPAD interfaces include a debugger, running either interactively within the system or as a more comfortable separate tool under X. We also provide an on-line hypertext help system, which for Unix versions is based on the HyTeX[2] system allowing a hierarchical search in the manual and its helpfiles; for the Macintosh the hypertext system is our own product. A filmtool, a compiler and a module management unit are under development.

MuPAD was developed as a service to the scientific community. As a consequence MuPAD, although copyrighted material, has a special distribution policy. It will be distributed to scientific and educational non-profit organizations anywhere in the world free of charge, however a license has to be acquired. At present 300 site licenses have been distributed worldwide and MuPAD is approximately running on 1100 machines.

Many organizations have supported the development of MuPAD. We are indepted to the Deutsche Forschungsgemeinschaft, the Heinz-Nixdorf-Institut, the Mathematische Forschungsinstitut Oberwolfach, and the University of Paderborn.

---

1. PARI, Copyright (c) by C. Batut, D. Bernardi, H. Cohen, and M. Olivier
2. HyTeX, Copyright (c) by N. Köckler, University of Paderborn, Germany

mathPAD

Vol 4 / No 1 / March 1994

Special MuPAD issue

Editors:
Benno Fuchssteiner
Klaus Gottheil
University of Paderborn
33095 Paderborn, Germany
e-mail:
`benno@uni-paderborn.de`
`kg@uni-paderborn.de`

ISSN 0941 – 9187
V.i.S.d.P.: B. Fuchssteiner