Differential Equations in MuPAD I: An Object Oriented Environment

Marcus Hausdorf[1] and Werner M. Seiler[2]

[1] Institut für Algorithmen und Kognitive Systeme
Universität Karlsruhe, 76128 Karlsruhe, Germany
Email: hausdorf@ira.uka.de
[2] Lehrstuhl für Mathematik I
Universität Mannheim, 68131 Mannheim, Germany
Email: seiler@euler.math.uni-mannheim.de
WWW: http://iaks-www.ira.uka.de/iaks-calmet/werner/werner.html

Abstract. We describe an object oriented programming environment for differential equations realised in the computer algebra system *MuPAD*. It serves as a convenient and highly customisable basis for the implementation of sophisticated algorithms and facilitates the interaction between application packages developed by different authors.

1 Introduction

There exist a lot of computer algebra packages for differential equations implementing algorithms for many different tasks. As a simple example, we may take the symmetry analysis of some field theory in physics. Thus our starting point is a Lagrangian and the first step is the derivation of the corresponding Euler-Lagrange equations. The second step consists of setting up the determining system for the symmetry generators of these equations. As these systems tend to be rather large and overdetermined, one will probably need some simplification and completion. As fourth step one might try to solve the determining system. In the fifth step one would like to use the found symmetry generators for a symmetry reduction, preferably to some ordinary differential equations which should finally be solved, too.

For each of these six steps there exist computer algebra packages. However, they have usually been written by different authors. Furthermore, these authors may have had different tasks in their minds and not necessarily precisely the sequence of steps described above. In any case, chances are high that each of the authors implemented his/her own data structures for differential equations, as no computer algebra system provides standardised ones.[1]

Using the output of one package as input for another one requires usually to write some conversion routines between these different data structures. This is not only a tedious and unpleasant task (requiring some information about the interns of the used packages), the conversion is also a time consuming process at runtime. This problem can be largely avoided by using an object oriented approach.

In this report we will present an object oriented programming environment for differential equations realized within the `domains` package [3] of *MuPAD* [4,9]. Some years ago, we implemented a similar environment in the computer algebra system AXIOM [10,11]. In comparison, our *MuPAD* environment is much more user-friendly and many methods are more efficient. Especially, it is now possible to completely hide the usage of domains and categories, so that even users not familiar with object oriented programming can use applications packages based on the domains.

The report is organised as follows. The next section gives a brief introduction into object oriented programming in computer algebra, its realization in *MuPAD* and an overview over our environment. Sects. 3 and 4 describe the functionality of the already provided categories and domains, respectively. They also give some details on the implementation. Finally, we give some conclusions and an outlook in Sect. 5.

2 Overview

2.1 Object Oriented Programming in MuPAD

Object oriented programming in computer algebra has been pioneered by AXIOM [7] (formerly called SCRATCHPAD) and is closely modelled on an abstract algebraic approach to mathematics. Each *object* is element of a *domain (of computation)* which in turn belongs to a *category*. For example, in *MuPAD* the number 42 may be viewed as an element of the domain `Dom::Integer` which belongs (among others) to the category `Cat::Ring`. In addition, there exists the possibility to define *axioms*, but we will not need this here.

[1] In such a diverse field as differential equations this appears only reasonable, as for different tasks different data structures may be optimal.

Categories provide the possibility to define abstract data types. Their primary task is to specify the methods (or procedures) which each domain belonging to them must contain. For example, each domain in the category `Cat::Ring` must provide implementations for addition and multiplication of ring elements. A category may contain default implementations for some methods which are inherited by its domains (but the domains may overwrite the default and provide their own implementation).

In our context, categories allow us to abstract certain operations that are always needed in working with differential equations. Then we can provide different concrete implementations in form of domains. An algorithm may now be written in a *generic* form taking such a domain as a parameter. This provides a simple mechanism to exploit special features of certain classes of differential equations without implementing the same algorithm several times.

For example, we may apply an algorithm sometimes to linear and sometimes to nonlinear equations. But certain operations needed in the algorithm can be performed much more efficiently for linear equations. Classically, we could either implement the algorithm in its most general form and thus renounce exploiting the linearity or write two different versions of the algorithm. The first solution is inefficient for linear equations and the second one is tedious and difficult to maintain. In an object oriented environment, we simply develop two different domains for linear and nonlinear equations, respectively. Then we implement the algorithm only once in terms of domain methods and can still fully exploit the linearity when the algorithm is applied to linear equations.

To be fair one should mention that object oriented programming also introduces a certain amount of overhead, especially a higher number of procedure calls. The performance of the object oriented code will usually be lower than that of a specialised version for linear equations. The main advantages are thus for the programmer who obtains a code that is easier to maintain and to reuse and also more flexible to use.

MuPAD provides the necessary tools for object oriented programming within its `domains` package [3]. All categories are collected within the library `Cat`, all domains within `Dom`. In order to address a specific category or domain one uses the operator `::` (a notation familiar to C++ programmers). Thus the already mentioned category of all rings is obtained with the call `Cat::Ring`. The same operator `::` is also used for methods (or procedures) within a domain. If `DD` is some domain possessing a method `method`, it is called by `DD::method`.

A category may have super categories — e. g. `Cat::Field` is a specialisation of `Cat::Ring` — from which it inherits all methods. Each domain belongs to at least one category. In addition, it may inherit from one super domain. Any domain may overwrite the inherited implementation of a method.

All domains implemented within the `domains` package inherit from `Dom::BaseDomain` (and `Cat::BaseCategory`). This yields default implementations of a number of basic methods for the generation and the output of domain elements. If `DD` is a domain, one way to generate an element of it is to call `DD::new(...)` which can be abbreviated to `DD(...)`. What and how many arguments may be passed depends of course on the domain `DD`. If wrong arguments are given, `DD::new` will report an error. By default, `DD::new` calls the method `DD::convert`; each domain must provide an implementation of this method. It controls the various ways elements of the domain can be generated. Furthermore, it defines implicitly the internal representation of the domain elements.

For the output of a domain element the method `DD::print` is used (this is done automatically by *MuPAD*). By default, `DD::print` just calls `DD::expr` which is to some extent the opposite of `DD::convert`. `DD::convert` usually takes as arguments some basic *MuPAD* objects and converts them to an element of `DD`, whereas `DD::expr` takes an element of `DD` and converts it to a basic *MuPAD* object which can be printed on the screen.

2.2 Structure of our Environment

Our environment consists of two levels. The first one defines *differential variables*; the second one *differential functions*. Both levels are specified by a category and we provide a number of domains in both categories (see Fig. 1). Every user can add own domains and still make full use of the environment — as long as the "standards" set for the names and for the semantics of the methods declared by the categories are obeyed.

The first level serves mainly as a user interface; the primary task of its domains is to define the format of the used differential variables, i. e. what are their names, how can they be entered, how does the output look like. No real computations happen in these domains besides trivial

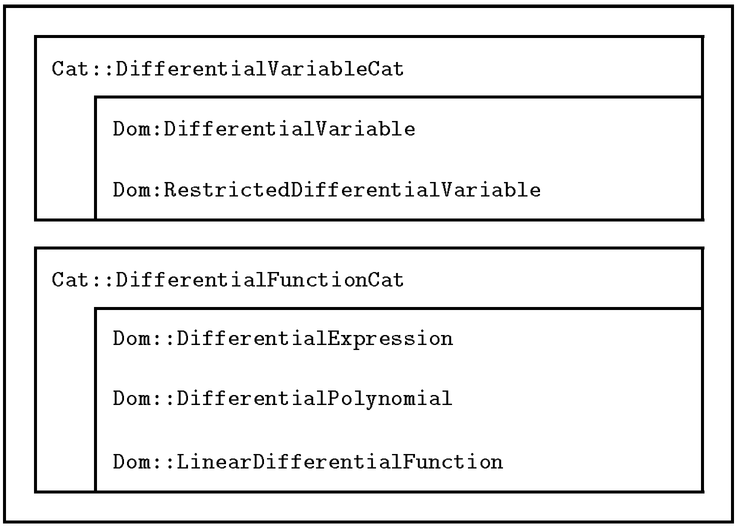

Fig. 1. Implemented domains and categories

ones like determining the order of a derivative. On the other hand, these methods are constantly used and should therefore be rather fast.

The domains in the second level correspond to specific types of differential functions like linear or polynomial ones. Each domain in this category takes as first argument a domain from `Cat::DifferentialVariable` specifying the differential variables on which the functions may depend. Besides standard arithmetical operations, the domains contain a number of differential methods like total differentiations and some simplification routines. Details will be given in the next two sections.

In a typical application of our environment, one first chooses a domain in `Cat::DifferentialVariable`. This tells *MuPAD* what are the independent and dependent variables. Then one chooses a domain in `Cat::DifferentialFunction`. Usually, this will declare some properties of the differential equations one is working with. With the following input we set the arena for working with arbitrary differential functions in the independent variables x, t and the dependent variable u.

MuPAD

```
>> DV := Dom::DifferentialVariable([x,t],u);
>> DE := Dom::DifferentialExpression(DV);
```

At first sight it might appear rather cumbersome to have to make such explicit declarations, but in extensive computations this pays off in form of much simpler in- and output. In addition, any procedure for differential equations will need the information what are the independent and dependent variables. In the traditional approach, these are passed directly as arguments. In our environment it suffices to define once these domains and any object of them always carries all the information in its datatype.

3 The Categories

The two basic categories underlying our environment are `Cat::DifferentialVariable` and `Cat::DifferentialFunction`. Both contain default implementations for many methods. Typically, only a few low level methods need to be written from scratch for implementing a new domain in one of the categories. However, for efficiency reasons it might often be worthwhile to overwrite some default implementations with specialised versions.

3.1 Differential Variables

The main task of `Cat::DifferentialVariable` is to provide a standardised interface to access the data of differential variables. This includes especially (multi) indices, the type or the order of a differential variable. The category distinguishes three different types of variables (encoded in strings):

`"Indep"`: This is the type of the *independent variables*. They play a special role in total differentiations and similar operations and carry only one index.
`"Dep"`: This is the type of the *dependent variables* (or unknown functions). They also carry only one index.
`"Deriv"`: This is the type of the *derivatives* (of the dependent variables with respect to the independent ones). They carry two indices, one of them is a multi or repeated index.

In the literature, one can find two different forms of multi indices. The classical *multi index* notation is mainly used for theoretical works.

If there are n independent variables, a multi index μ is an (ordered) n-tuple $\mu = [\mu_1, \ldots, \mu_n]$ where the entry μ_i denotes the number of differentiations with respect to the i^{th} independent variable. The order q of the corresponding differential variable is given by the length of the multi index $q = |\mu| = \mu_1 + \cdots + \mu_n$. In the second form which we will call, in lack of a better name, *repeated index* the index I of a q^{th} order derivative is a set of q integers between 1 and n: $I = (i_1, \ldots, i_q)$ where each integer i_k denotes a differentiation with respect to the ${i_k}^{\text{th}}$ independent variable. While in principle the order of the entries in I does not matter, we will usually assume that they are ordered: $i_1 \leq i_2 \cdots \leq i_q$.

Domains in `Cat::DifferentialVariable` automatically support both notations; the one internally used by a domain is given by the entry `notation` returning either the string `"multi"` or `"repeated"`. The auxiliary methods `m2r` and `r2m` transform one kind of index into the other one. We refer to the multi or repeated index as the "lower" index of a differential variable; the "upper" index is used to enumerate the independent and dependent variables. This reflects the usual mathematical notation for a derivative:

$$u^\alpha_\mu = \frac{\partial^{|\mu|} u^\alpha}{\partial(x^1)^{\mu_1} \cdots \partial(x^n)^{\mu_n}} = u^\alpha_{\underbrace{x^1 \cdots x^1}_{\mu_1 \text{times}} \cdots \underbrace{x^n \cdots x^n}_{\mu_n \text{times}}} \tag{1}$$

where we have assumed that there are n independent variables x^i, m dependent variables u^α and that $\mu = [\mu_1, \ldots, \mu_n]$ is a multi index.

The following alphabetic list contains all the methods and entries which must be implemented in any domain belonging to `Cat::DifferentialVariable`. For most of them, the category provides a default implementation; in fact, only six methods must be defined explicitly in the domain: `index`, either `multiIndex` (if `notation` returns the value `"multi"`) or `repeatedIndex` (if `notation` returns the value `"repeated"`), `notation`, `numIndVar`, `numDepVar` and `vartype`. They are marked by an asterisk in the list.

Methods in `Cat::DifferentialVariable`

`allRepeated(mi)`: Returns a list of all possible realizations of a given multi index `mi` as repeated index.

`class(dv)`: Class[2] of a differential variable `dv`.

[2] The class of a derivative with multi index $\mu = [\mu_1, \ldots, \mu_n]$ is defined as the smallest index i such that $\mu_i \neq 0$. For an *ordered* repeated index $I = (i_1, \ldots, i_q)$ the class

`depVariables`: Returns a list of all dependent variables.

`derivativeOf(dv1,dv2)`: Checks whether the variable `dv1` is a derivative of `dv2`. If yes, the corresponding multi or repeated index is returned (depending on the `notation` used by the domain), otherwise the result is an empty list.

`derivatives(q<,cl>)`: Lists all derivatives *up to* order `q`. If a second argument `cl` is given, all derivatives *of* order `q` and class greater than or equal to `cl` are returned.

`diff(dv,x)`: "Normal" differentiation where all differential variables are treated as independent; compare with `totalDiff` and see example in Sect. 4.1.

`dim(q)`: Counts the dependent variables and the derivatives up to order `q`.

`dimq(q)`: Counts the derivatives of order `q`.

`has(dv,x)`: Returns `TRUE`, if `dv=x`, and `FALSE` otherwise. Overloads the *MuPAD* function `has`.

`indVariables`: Returns a list of all independent variables.

*`index(dv)`: Yields "upper" index of differential variable `dv`.

`int(dv,i)`: Tries to integrates the differential variable `dv` with respect to the independent variable x^i. If this is not possible, `FAIL` is returned.

`length(dv)`: Returns 1. Overloads the *MuPAD* function `length`.

`m2r(mi)`: Computes repeated index for given multi index `mi`.

`makeIndexList(q)`: Generates all multi indices up to order `q`.

*`multiIndex(dv)`: Yields "lower" index of differential variable `dv` in multi index notation.

`multiRaise(mu,i)`: Raises the entry `i` of multi index `mu` by 1; `i` may also be a list of indices.

`newMulti(t,i,mi)`: Generates a new differential variable of the type `t` with index `i` and multi index `mi`.

`newRepeated(t,i,ri)`: Generates a new differential variable of the type `t` with index `i` and repeated index `ri`.

---

equals i_1. For an independent variable we define the class to be 0; for a dependent variable as the value of `numIndVar`.

*`notation`: Returns used notation as a string: either `"multi"` or `"repeated"`.

*`numDepVar`: Returns number of dependent variables.

*`numIndVar`: Returns number of independent variables.

`order(dv)`: Computes the order of the differential variable `dv`.

`orderly`: Returns `TRUE`, if the domain uses an orderly ranking.[3]

`orderLex(dv1,dv2)`: Graded reverse lexicographic ranking of the differential variables `dv1` and `dv2`.

`pureLex(dv1,dv2)`: Purely lexicographic ranking of the differential variables `dv1` and `dv2`.

`r2m(ri)`: Returns repeated index `ri` as multi index.

`random(<q>)`: Returns a random differential variable. The optional parameter `q` gives the maximal order; default is 6.

*`repeatedIndex(dv)`: Returns "lower" index of differential variable `dv` in repeated index notation.

`repeatedRaise(mu,i)`: Includes the entry `i` in the repeated index `mu`; `i` may also be a list of indices.

`totalDiff(dv,i)`: Total differentiation of `dv` with respect to the independent variable with index `i` (alternatively `i` may be an independent variable). Compare with `diff` and see the example in Sect. 4.1.

*`vartype(dv)`: Returns the type of the differential variable `dv` (as string).

——— *Methods in `Cat::DifferentialVariable`* ———

As one can see, rankings of differential variables should also be implemented in the domains in `Cat::DifferentialVariable`. Currently, only the purely lexicographic ranking and a graded reverse lexicographic ranking are implemented; the latter one is the default. Different rankings can simply be imposed by overwriting the method `_less`. Note, however, that the entry `orderly` should be adapted correspondingly.

Another caveat is that one must be careful with changing dynamically the ranking in a domain `DV` in `Cat::DifferentialVariable`. While of course any call to `DV::_less` will use the new ranking, this

[3] A ranking of differential variables is called *orderly*, if variables of higher order are always higher in the ranking than those of lower order.

change might not be noted by domains for differential functions in the variables of `DV`. Only newly constructed domains will definitely use the new ranking.

3.2 Differential Functions

The basic category for differential functions is called `Cat::DifferentialFunction(DV)`. It takes as argument a domain `DV` from `Cat::DifferentialVariable`. All domains belonging to this category represent functions of the variables in `DV`. The various domains correspond typically to different types of functions, e. g. linear or polynomial ones.

`Cat::DifferentialFunction` is only a specialisation of the category `Cat::AbelianSemiGroup` and not of `Cat::Rng` as one might expect. The reason is that we want to admit domains for linear functions, but these can only be added and not multiplied without loosing their linearity. This also makes difficulties with Jacobians. Normally, the Jacobian is a matrix over the same domain. But *MuPAD* matrices can be defined only over rings, so something special must be done for linear functions. This problem is discussed in more detail in Sect. 4.4.

Any domain in `Cat::DifferentialFunction(DV)` should provide (besides the arithmetical operations of a `Cat::AbelianSemiGroup`) the methods listed in the following table (sorted alphabetically). For most of them a default implementation exists. Only the methods `diff`, `has`, `indets`, `solve` and `subsOne` must in any case be newly implemented. They are marked by an asterisk in the list. Concrete examples for the use the methods can be found in the next section.

Methods in `Cat::DifferentialFunction`

`autoreduce(sys)`: Autoreduction of the list `sys` of differential functions. Note that reduction includes (total) differentiations in contrast to simplification.

`autosimplify(sys)`: Autosimplification of the list `sys` of differential functions. Note that simplification is a purely algebraic operation, i. e. no derivatives of the functions in `sys` are used.

`class(df)`: This is a simple lift of the corresponding method in `DV`; it returns the maximal class of the highest order differential variables contained in the differential function `df`.

*`diff(df,x)`: Partial differentiation of the differential function `df` with respect to the variable `x`. Compare with `totalDiff`!

`diffSubs(df,dv=e)`: In a differential substitution not only the differential variable `dv` but also all its derivatives are substituted by the expression `e` and its total derivatives.

`eval(df,sol)`: Evaluates the differential function `df` for some given functions. The list `sol` must have as many entries as there are dependent variables, and each entry must be a function of the independent variables only. These functions (and their total derivatives) are substituted in `df` for the differential variables.

*`has(df,dv)`: Returns `TRUE`, if the differential variable `dv` effectively occurs in the differential function `df`.

*`indets(df<,DiffVars>)`: Returns a set with all the indeterminates occurring in the differential function `df`. With the option `DiffVars` only those indeterminates are returned which are elements of `DV`, i.e. no parameters (see the example in Sect. 4.2).

`isConst(df)`: Returns `TRUE`, if the differential function `df` does not depend on any differential variable.

`jacobian(sys)`: Computes the Jacobian of the differential functions contained in the list `sys`.

`jacobianType`: This entry returns the data type of the Jacobian matrices computed by the method `jacobian` (some *MuPAD* matrix domain).

`leader(df)`: Returns the leading derivative of the differential function `df`. The ranking is implicitly defined by the method `_less` of the domain `DV`.

`linear`: This entry returns `TRUE` or `FALSE` depending on whether or not the functions represented by the domain are linear.

`multiDiff(df)`: Computes with as few differentiations as possible several derivatives of the same differential function `df`.

`multiSubs(df,dv1=e1<,dv2=e2,...>)`: Substitutes the differential variables `dv1,dv2,...` by the expressions `e1,e2,...` in the differential function `df` using the method `subsOne`.

`order(df)`: Returns the maximal order of the differential variables effectively appearing in `df`.

`polynomial`: This entry returns `TRUE` or `FALSE` depending on whether or not the functions represented by the domain are polynomial.

*`solve(df,dv)`: Tries to solve the differential function `df` for the differential variable `dv`.

*`subsOne(df,dv=e)`: Substitutes the expression `e` for the differential variable `dv` in the differential function `df`.

`totalDiff(df,x)`:[4] Total differentiation of the differential function `df` with respect to the independent variable `x`. This implies the use of the chain rule in contrast to partial differentiations with `diff`.

`Variables`: This entry returns the domain `DV`.

——— *Methods in `Cat::DifferentialFunction`* ———

Of importance are here the distinctions made between auto*reduction* and auto*simplification* and between *partial* and *total* differentiation. If we are given a differential function $\phi(x,u,p)$ where x represents the independent, u the dependent variables and p the derivatives, then the method `diff` computes the usual *partial derivatives* $\partial\phi/\partial x$, $\partial\phi/\partial u$, or $\partial\phi/\partial p$. If ϕ depends in addition on some parameters a, `diff` can also compute derivatives with respect to them. In contrast, the *total differentiation* is defined only for independent variables. Using multi index notation for the derivatives, it is given by

$$D_{x^i}\phi = \frac{\partial\phi}{\partial x^i} + \sum_{\alpha} \frac{\partial\phi}{\partial u^\alpha}\, p_i^\alpha + \sum_{\alpha,\mu} \frac{\partial\phi}{\partial p_\mu^\alpha}\, p_{\mu+1_i}^\alpha \tag{2}$$

where p_i^α is a short hand for the derivative with a multi index μ where all entries are zero except μ_i which is one (i.e. $\mu_j = \delta_{ji}$) and $p_{\mu+1_i}^\alpha$ represents the derivative with a multi index given by μ and μ_i raised by one. Thus if we take $\phi = u_x$, then $\partial\phi/\partial x = 0$ (and the result of a corresponding `diff` call will be zero), as ϕ does not depend explicitly on x. If we want to take into account the implicit dependency of u_x on x, we must use the total differentiation provided by `totalDiff` which uses the chain rule and will return $D_x\phi = u_{xx}$.

[4] A default implementation is provided only for the case that the domain is a ring.

Autosimplification is a purely algebraic operation. Its main goal is to eliminate all algebraic dependencies between the differential functions in a given list. Here u_{xx} and u_x are considered as algebraically independent, as they are distinct differential variables. *Autoreduction* eliminates in addition differential dependencies. A simple application of the chain rule yields $D_x(\sin u_x) = u_{xx} \cos u_x$, so the functions $u_{xx} \cos u_x$ and $\sin u_x$ are differentially dependent although they are algebraically independent. `autoreduce` therefore eliminates $u_{xx} \cos u_x$ from the list $[u_{xx} \cos u_x, \sin u_x]$, whereas `autosimplify` leaves it unchanged.

`autoreduce` and `autosimplify` will furthermore try to put the given list of functions in a kind of triangular form: each function in the output should have a different leading derivative. Ideally, these leading derivatives are completely eliminated from all other functions. Note that no general algorithm exists for the autoreduction or -simplification of arbitrary differential functions. Such algorithms can be given only for special classes of functions, e. g. for linear functions where autosimplification reduces to Gaussian elimination or for polynomials where Gröbner bases [2] can be used.

The default implementations of `autoreduce` and `autosimplify` in `Cat::DifferentialFunction` apply a simple heuristic relying essentially on the domain method `solve`. Each equation is solved for its leading derivative and then this derivative is eliminated in all other equations by substitution. This process continues, until no further eliminations can be performed. Obviously, for arbitrary differential functions it cannot be guaranteed that one can always solve for the leading derivatives and thus it is not sure that the remaining functions are indeed algebraically (or even differentially) independent.

Finally, some words are necessary to explain the approach taken for the implementation of the method `subs`. It is intended to free programmers from having to write code for argument checking and multiple substitutions each time they write a new domain belonging to `Cat::DifferentialFunction`. The categorical method `multiSubs` handles these tasks; however, it uses for a single replacement the method `subsOne` which has to be provided by each domain. Consequently, `subs` should be defined in the domain as `subs := dom::multiSubs`. As the default implementation in `Dom::BaseDomain` overwrites any categorical one, one cannot proceed this way directly in `Cat::DifferentialFunction`.

4 The Domains

With categories alone one cannot perform any computations. Therefore our environment provides already a number of instances, namely two domains for differential variables and three domains for differential functions. The latter ones cover the three most important types of functions: *linear functions*, *polynomials* and *general expressions*. The distinction between polynomials and general expressions is of interest for performance reasons. *MuPAD* possesses a special built-in data type `DOM_POLY` for polynomials and executes both arithmetical and differential operations much faster for polynomials than for general expressions.

In the sequel we will briefly describe these five provided domains. As their basic functionality is inherited from the categories, we will discuss only those methods which are specific for a given domain. In addition, we will give some indications on the underlying representation and the implementation of some important methods.

4.1 Differential Variables

The main domain for differential variables is `Dom::DifferentialVariable`. It may be considered as the simultaneous implementation of two different domains distinguished by the arguments they take. In the first usage, `Dom::DifferentialVariable` is given as arguments two lists (if a list has only one entry, one may omit the list brackets) with the names of the independent and the dependent variables, respectively. In the second usage, a domain with indexed variables is generated. Here one gives as arguments indexed identifiers like `x[3]` in order to obtain the variables x_1, x_2, x_3. Thus two different ways to generate a domain with three independent and two dependent variables are

MuPAD
```
>> DV := Dom::DifferentialVariable([x,y,z],[u,v]):
>> IDV := Dom::DifferentialVariable(x[3],u[2]):
```

The two so generated domains `DV` and `IDV` differ especially in their internally used representations: `DV` is based on *repeated index* notation and `IDV` on *multi index* notation, as one can see from the following two lines of output.

MuPAD

```
>> DV::notation;
>> IDV::notation;
```

Output

```
"repeated"

"multi"
```

`Dom::DifferentialVariable` supports several notations for the in- and output of differential variables. One is of course the standard *MuPAD* notation using `diff`. Alternatively one may use the `D` operator. However, for most purposes the most convenient notation is a rather condensed one mimicking the usual mathematical notation as good as possible using only ASCII characters. Here dependent variables always appear without arguments; for derivatives one uses the same symbols as for the dependent variables but uses them now as function names with a list of the independent variables with respect to which differentiations occur as argument. Finally, one may use the generic methods `newMulti` and `newRepeated` defined in `Cat::DifferentialVariable` not requiring any knowledge of variable names. The following five input lines all generate the same differential variable u_{xy} (we show the output only once, as it is of course always the same).

MuPAD

```
>> dv := DV(diff(u(x,y,z),x,y)):
>> dv := DV(D([1,2],u)):
>> dv := DV(u([x,y])):
>> dv := DV::newMulti("Deriv",1,[1,1,0]):
>> dv := DV::newRepeated("Deriv",1,[1,2]):
```

Output

```
u([x, y])
```

As one can see, for better readability the output of `Dom::DifferentialVariable` uses always the condensed notation. If one wants to convert back to one of the other input forms one can use the method `convert_to` with the options `"diff"` and `"D"`, respectively. The arguments of the remaining two calls could be retrieved with the methods `vartype`, `index`, `multiIndex` or `repeatedIndex`, resp.

MuPAD

```
>> DV::convert_to(dv,"diff");
>> DV::convert_to(dv,"D");
```

Output

```
diff(u(x, y, z), x, y)

D([1, 2], u)
```

Finally, it is possible to obtain the output as TEX; all domains in our environment are compatible with the *MuPAD*-TEX interface in the `generate` library. Again a condensed notation is used and not full differential quotients.

The following example shows the difference between the two methods `diff` and `totalDiff`: whereas the former one treats all differential variables as independent of each other and thus returns either 1 or 0, the latter one represents the total differentiation one needs for prolonging differential equations and similar operations.

MuPAD

```
>> DV::diff(dv,x);
>> dvx := DV::totalDiff(dv,1);
>> ri := DV::derivativeOf(dvx,dv);
>> DV::r2m(ri);
```

Output

```
0

u([x, x, y])

[1]

[1, 0, 0]
```

As $x \neq u_{xy}$, the first call returns 0. The second call adds a differentiation with respect to x. This is demonstrated in the third line where the "difference" between u_{xy} and u_{xxy} is computed as a repeated index. If the domain `IDV` had been used for this example, the result would have been the corresponding multi index, `[1,0,0]`, as determined by the auxiliary method `r2m` in the last line.

The internal representation of the domain `Dom::DifferentialVariable` is straightforward. It consists of five slots: (i) the type of the variable, (ii) the upper index, (iii) the lower index (or the empty list, if no lower index exists), (iv) the order and (v) the class of the variable. (iv) and (v) are not really necessary, however, it turned out that for many computations it is advantageous to store this information in the representation instead of computing it newly each time it is needed. Especially, comparisons with respect to the graded lexicographic order (our default ranking) are speeded up considerably.

The domain `Dom::RestrictedDifferentialVariable` is typically used for constructing coefficient domains of linear or polynomial functions (see Sects. 4.3 and 4.4) or by domains representing solutions of differential equations. It takes as first argument another domain in `Cat::DifferentialVariable` and imposes then some restrictions, specified by the second argument, on the variables. A typical example for such a restriction is that only independent variables are admitted.

Five different possibilities are provided for specifying the restriction. They always take the form of an equality where the left hand side denotes the type of restriction:

`Types`: Here one specifies the types of the differential variables which are admitted. If several types are allowed, a set with their names must be passed.

`Order`: If the order of the admitted variables is specified, then instead of an equality one might also use an inequality. Thus it is possible to generate domains containing either only derivatives up to a prescribed order or alternatively only derivatives of higher order.

`InSet`: In this case one specifies a set with all the differential variables admitted by the domain.

`OutSet`: This is just the converse of the last case; a set with all the variables *not* admitted is passed.

`Crit`: This represents the most general case where a criterion procedure is passed. This procedure should take a differential variable as argument and return `TRUE` or `FALSE` depending on whether or not the variable belongs to the domain.

It is not possible to combine these options. If, for instance, one wants to prescribe simultaneously the type and the order, one must write a criterion procedure and use the last possibility. The actually imposed restriction of a domain `RDV := Dom::RestrictedDifferentialVariable(DV,...)` can be retrieved with the method `mode`. Its possible results are:

`["Indep"]`: `RDV` contains all the independent variables of `DV`.

`["Less/Greater",q]`: `RDV` contains all differential variables of order less resp. greater than `q` (for example, in case of the restriction `Types={"Indep","Dep"}` `RDV::mode` returns `["Less",1]`).

`["General"]`: All other cases.

4.2 Differential Expressions

Our most general domain for differential functions is called `Dom::DifferentialExpression(DV)`. It is essentially a lift of the basic *MuPAD* type `DOM_EXPR` to the category `Cat::DifferentialFunction` and thus allows for computations with arbitrary differential expressions.

In order to avoid the need to always explicitly generate a domain `DV` belonging to `Cat::DifferentialVariable`, it is also possible to specify the domain via one or two optional arguments. These options contain essentially the arguments passed to the constructor `Dom::DifferentialVariable` or `Dom::RestrictedDifferentialVariable`, respectively. Thus in the following example the domain `DE` represents general differential expressions in the independent variables x, y, z, the dependent variables u, v and their derivatives. In contrast, the domain `RDE` represents only functions of the independent variables and consequently the attempt to generate a function of the dependent variable u yields an error. The automatically generated domain for the differential variables can be retrieved with the method `Variables`.

MuPAD

```
>> DE := Dom::DifferentialExpression(Vars=[[x,y,z],[u,v]]):
>> RDE := Dom::DifferentialExpression(Vars=[[x,y,z],[u,v]],
>>                                     Rest=[Types="Indep"]):
>> RDE::Variables;
>> de := DE(sin(a*u([y]))*u([x,y])^2-exp(v([z])+b));
>> rde := RDE(u^2);
```

Output

```
Dom::RestrictedDifferentialVariable(Dom::DifferentialVariable(
   [x, y, z], [u, v]), Types = Indep)

                                                2
               - exp(b + v([z])) + u([x, y])  sin(a u([y]))

Error: illegal arguments [(Dom::DifferentialExpression(
   Vars = [[x, y, z], [u, v]], Rest = [Types = "Indep"]))::new];
```

Besides the methods specified in the category `Cat::DifferentialFunction` some basic *MuPAD* functions have been lifted to `Dom::DifferentialExpression`. This includes `eval`,[5] `normal`, `simplify`, `radsimp`, `combine` and `rewrite`. For efficiency reasons, the default implementations of many methods in `Cat::DifferentialFunction` have been replaced by special versions.

The representation consists of two slots: the first one contains usually an element of the basic domain `DOM_EXPR`, the second one a set of differential variables, i. e. elements of `DV`. All differential variables appearing in the differential expression are contained in this set. However, in general the set will contain some further variables. While it is very useful for many operations to have such a set stored in the representation, it turned out that it is inefficient to always eliminate redundant variables (it is a rather expensive operation to check whether an element of `DOM_EXPR` effectively depends on a given variable).

Note that this second slot also affects tests for equality. It is a well-known problem in computer algebra that no normal representation exists for general expressions, i. e. it is not always possible to decide whether or not an expression is zero. In `Dom::DifferentialExpression` it may furthermore happen that two expressions look equal, as the have the same element of `DOM_EXPR` in their first slot (which determines the output), but still a check for equality returns `FALSE`, as one of them has redundant elements in the set in the second slot. In such cases the domain method `equal` will return `TRUE`.[6]

While the first slot usually contains an expression, there exist a few further possibilities. The simplest ones are that the differential expression is actually just a number (i. e. an element of one of the basic domain `DOM_INT`, `DOM_RAT` or `DOM_FLOAT`) or an identifier (which is considered by `Dom::DifferentialExpression` as a parameter). In all these cases, the second slot contains an empty set, thus we have a normal representation for them. A differential expression may also consist just of a differential variable; then the first slot contains this variable. The last possibility is an element of the domain `Dom::DifferentialFunction(DV)`.

[5] Note that called with one argument `eval` yields the lifted kernel function and with two arguments the method specified in `Cat::DifferentialFunction`.

[6] In general, it is a good strategy to use the domain method `equal` instead of the boolean operator `=` in order to compare elements of domains without a canonical representation.

This special domain is a (hopefully only temporary) hack around a problem with the differentiation of undetermined expressions. If (using the domain `DE` of our example above) an expression of the form `F(x,y,z,u,v)` is differentiated with respect to one of its arguments, i. e. with respect to a differential variable, *MuPAD* uses its differential operator `D` to represent the result. Unfortunately, it will try in subsequent computations repeatedly to simplify the result, although there is obviously nothing to simplify. In larger calculations (especially in Lie symmetry theory) this can waste surprisingly much computing time.

As an alternative, one may use the domain `Dom::DifferentialFunction(DV)` to represent such expressions. Then the differentiation is a completely formal operation (as it should be) and no subsequent evaluation attempts will happen. A user will almost never work directly with this domain, as `Dom::DifferentialExpression` provides an interface with the method `arbFunction`. The following example demonstrates the difference in the execution times of a simple differentiation: using internally `Dom::DifferentialFunction` is usually between two and three times faster.

MuPAD

```
>> de1 := DE(F(x,y,z,u,v)):
>> de2 := DE::arbFunction(F,[x,y,z,u,v]):
>> time(DE::diff(de1,x) $ i=1..100);
>> time(DE::diff(de2,x) $ i=1..100);
```

Output

```
                                 330

                                 150
```

We can now provide concrete examples for some of the procedures specified in the category `Cat::DifferentialFunction`. We start with the method `indets` and demonstrate the meaning of the option `DiffVar` for the expression `de` defined above. Without this option the parameters a, b are returned, too; otherwise only the differential variables are returned.

MuPAD

```
>> DE::indets(de);
>> DE::indets(de,DiffVars);
```

Output
```
{a, b, u([y]), v([z]), u([x, y])}

{u([y]), v([z]), u([x, y])}
```

Next we give an example for the computation of a Jacobian. We wrote a special domain `Dom::SparseMatrix` for representing Jacobians. It has a number of particularities. As the name already indicates, it is specialised to *sparse* matrices, as differential functions appearing in applications are typically sparse, i. e. they do depend only on a small subset of all possible differential variables up to a given order. Secondly, the columns of such matrices are labelled not by integers but by differential variables. In the following *MuPAD* commands we first define a system consisting of two differential functions, $u_{xx} - u_{yy}$ and $v_z + v_{xxx} + vv_x$. Then we determine its Jacobian `jac` and the domain of `jac`. In the last line we extract all column labels of `jac`.

MuPAD
```
>> sys:=[DE(u([x,x])-u([y,y])),DE(v([z])+v([x,x,x])+v*v([x]))]:
>> jac := DE::jacobian(sys);
>> SM := domtype(jac);
>> SM::allIndices(jac);
```
Output
```
+-                              -+
|  0, -1, 1, 0, 0,     0         |
|                                |
|  1,  0, 0, 1, v, v([x])        |
+-                              -+

Dom::SparseMatrix(Dom::DifferentialVariable([x, y, z], [u, v]),
   Dom::DifferentialExpression(Vars = [[x, y, z], [u, v]]),
   (dv1, dv2) -> DVless(dv2, dv1))

     [v([x, x, x]), u([y, y]), u([x, x]), v([z]), v([x]), v]
```

One can see here how the columns are indexed by the occurring differential variables sorted in descending order (the ranking determined by the method `_less` of the used domain for differential variables — compare with Sect. 4.1). `Dom::SparseMatrix` has three parameters: `DV` for labelling the columns, `DE` for the matrix entries and a boolean function for ordering the elements of `DV`. In the case of Jacobians the columns are always ordered decreasingly with respect to the order `_less` of the domain `DV`. Thus the first column of the matrix corresponds always to the leading derivative.

The following three methods have in common that they require several derivatives of the same differential function. In order to minimise the number of differentiations needed to compute these derivatives, the auxiliary procedure `derivativeTree` of the `DETools` library is used. It determines for a given set of multi indices a tree whose leaves represent the wanted derivatives. One can read off this tree in which order the differentiations should take place. A simple heuristic algorithm is used to determine the tree; it will be described in more detail in [1].

The method `multiDiff` takes a differential function and a list of multi indices and returns the total derivatives of the function with respect to the indices. `eval` substitutes functions (of the independent variables) for the dependent variables and their derivatives in a given differential expression `de`. A typical use is to check whether the functions form actually a solution of the differential equation `de=0`. `eval` uses internally `multiDiff` to determine the needed derivatives. The functions given as arguments should be elements of a domain of differential functions over the independent variables only (i. e. over `Dom::Restricted-DifferentialVariable(DV,Types="Indep")`. But in interactive calls automatic conversions are performed, so that one can simply enter:

MuPAD

```
>> DE::eval(de,[x^2*y^2*z^2,x*y*z]);
```

Output

```
                             2  2  4          2    2
      - exp(b + x y) + 16 x  y  z  sin(2 a x  y z )
```

`diffSubs` is similar to `eval` but more general: it substitutes an arbitrary differential expression for a single dependent variable or derivative and all of its derivatives.

MuPAD

```
>> DE::diffSubs(de,u([y])=u([z,z])+2*y^2);
```

Output

```
                                  2                          2
   - exp(b + v([z])) + u([x, z, z])  sin(a (u([z, z]) + 2 y ))
```

4.3 Differential Polynomials

`Dom::DifferentialPolynomial` can be used to generate domains representing differential functions which are polynomial in some or all differential variables. In the simplest call the constructor takes only one argument: a domain `DV` from `Cat::DifferentialVariable`. Then one obtains a domain which represents classical differential polynomials with constant coefficients, i. e. functions which are polynomial in the dependent variables and the derivatives and which do not depend at all on the independent variables. The domain `Dom::Rational` is used as coefficient ring.

One may also explicitly specify a coefficient ring `R` as second argument of the constructor. If the ring `R` belongs to the category `Cat::DifferentialFunction`, so that we have variable coefficients, its differential variable domain (obtainable with `R::Variables`) must be of the form `RDV:=Dom::RestrictedDifferentialVariable(DV,...)` where the dots stand for the chosen form of restriction (see Sect. 4.1). The functions contained in the so generated domain are polynomial in all differential variables of `DV` which are *not* admitted in `RDV`. The entry `coeffRing` returns the domain `R`, while the entry `constCoeff` signals whether constant coefficients are used.

For users unwilling to create domains for differential variables and the coefficient ring it is also possible to obtain differential polynomials in a way similar to that described in Sect. 4.2, by giving variables and restrictions directly as right hand sides of the options `Vars=[...]` and `Rest=[...]`. Here the restrictions apply to the *coefficient ring*; if they are missing, constant coefficients with dependent variables and derivatives in the terms are used.

The choice of the internal representation of `Dom::DifferentialPolynomial` requires some explanations of how polynomial arithmetics is performed in *MuPAD*. The basic polynomial type is `DOM_POLY`. An element of it consists of three operands: the expression representing the polynomial, the list of variables and the domain of the coefficients. Polynomials are created by a call of the kernel function `poly` with these three arguments (if the the coefficient ring is omitted, `DOM_EXPR` is the default). In order to utilise the fast polynomial addition and multiplication provided by the *MuPAD* kernel, the variable lists and coefficient rings of the operands must coincide. If the variable set is known *a priori* and if it is finite, this poses no problem.

In the case of differential polynomials, however, we have an infinite number of variables. Thus the variable lists of polynomials must be compared and adapted before adding or multiplying them and one loses some of the benefits of the faster arithmetics. In order to decrease the number of changes of the variable lists to a minimum, we chose the following approach: the internal representation of an element of `Dom::DifferentialPolynomial` consists of three slots.

<table>
<tr><td colspan="3">REP=poly</td><td rowspan="2">VARS</td><td rowspan="2">ORD</td></tr>
<tr><td>expr</td><td>VARSP</td><td>R</td></tr>
</table>

The first slot contains the polynomial as an element of `DOM_POLY` which in turn consists again of three slots as mentioned above. For the variable list `VARSP` we take *all* differential variables up to the order given by the third slot `ORD`. The slot `VARS` holds a superset of the set of all differential variables effectively occurring in `expr` (similar to the second slot in the representation of `Dom::DifferentialExpression`).

The variable lists are adapted only *before* two polynomials are added or multiplied. The values of `ORD` are compared. If the polynomials are of different order, the one of lower order is converted into list representation by the kernel function `poly2list`[7]. The auxiliary function `addZterms` augments the exponent vectors by the necessary zeros (the number of which is determined by the methods `dim` and `dimq` from `Dom::DifferentialVariable`); the result is changed back to a polynomial by another call of `poly`. For the same reasons as in Sect. 4.2 no adaptation of the variable list is performed on the result, i. e. if a variable is cancelled, it still appears in the list `VARSP` and the set `VARS`. This might also lead to a too high value of `ORD`.

All the points mentioned in the previous section on normalised expressions thus also apply for differential polynomials. The removal of superfluous variables from `VARS`, the determination of the correct value for `ORD` and the truncation of `VARSP` via a further auxiliary method `remZterms` can be enforced by the method `normalPoly`.

[7] It consists of a list of monomials, each of them again being a list of two elements: the coefficient (from the domain `R`) and the *exponent vector*, which is a list of integers where each entry holds the degree of the corresponding element in the variable list.

`Dom::DifferentialPolynomial` is also a member of the category `Cat::Polynomial`. There are two reasons why we do not use `Dom::DistributedPolynomial` or one of its descendants for the representation. Adaptations of the variable lists are done there before and after each operation and we simply did not succeed in making this domain work with differential variables. So a great deal of methods had to be implemented in `Dom::DifferentialPolynomial` anew, making the code rather large. Currently, it works only with restrictions possessing `mode=["Less",q]` or `mode=["Indep"]` (which is internally treated as `["Less",0]`). The case `mode=["Greater",q]` requires a completely different implementation, as the variable lists are now finite, and for `mode=["General"]`, certain methods, e. g. `totalDiff`, have to be changed. Since these cases rarely turn up in applications, they have been deferred until needed. The same applies for `Dom::LinearDifferentialFunction` (see Sect. 4.4).

The `convert` routine of `Dom::DifferentialPolynomial` takes either one or two arguments; in the latter case, the first argument holds the polynomial in dense list representation and the second argument is the variable list. With the following commands, first a domain `DP` is constructed which represents differential functions which are polynomial in all derivatives and where the coefficients can be arbitrary expressions in the independent and dependent variables (the order of a dependent variable is 0). Then an element `p` of this domain is generated using a list notation. Of course, it would have been also possible to enter `p` directly in standard *MuPAD* notation.

MuPAD

```
>> DP := Dom::DifferentialPolynomial(Vars=[[x,y,z],[u,v]],
>>                                    Rest=[Order<1]):
>> p := DP::convert([[sin(u),[2,0,1]],[2*x+exp(y),[1,2,0]]],
>>                   [u([x]),u([x,y]),v([z,z])]);
```

Output

```
                                2                            2
   sin(u) (v([z, z]) u([x])  ) + (2 x + exp(y)) (u([x, y])  u([x]))
```

If `convert` is given merely one argument, any *MuPAD* expression that can be converted into a differential polynomial is allowed. Especially, `Dom::DifferentialPolynomial` provides methods to convert to and from all other currently existing domains in the category `Cat::DifferentialFunction`.

As arithmetical operations the methods of a (commutative) ring have been implemented: `_plus`, `_negate` and `_mult`. They use the respective fast polynomial kernel operations as described above. All methods required by the category `Cat::Polynomial` (and some more) have also been lifted from the corresponding polynomial functions. This includes `multcoeffs`, `mapcoeffs`, `coeff`, `lcoeff`, `tcoeff`, `nthcoeff`, `nterms`, `lterm`, `nthterm`, `lmonomial`, `nthmonomial`, `mainvar`, `degreevec`, `degree`, `evalp` as well as the methods for Gröbner bases, `gbasis` and `normalf`. If working with a monomial ordering other than the default `DegreeOrder` note that the variable list itself is ordered according to how `DV::derivatives` returns its result. Currently, this means a `DegreeOrder` ranking on the variables.

An important polynomial operation for many applications is *pseudodivision*. It is used, for example, in (differential) algebra in the computation of characteristic sets [8]. The procedure `pseudoRemainder` takes as argument two differential polynomials f and g and a differential variable y such that $deg(f, y) > deg(g, y) > 0$. It returns a list $[r, q, d, s]$ such that $d^s f = qg + r$ where d is the leading coefficient of g and $deg(g, y) > deg(r, y)$. The procedure `pseudoReduction(f,g,y)` computes a *polynomial remainder sequence* by repeatedly applying a pseudodivision with respect to `y`. It returns the last element of this sequence, which is then free of `y`, and a list of all the leading coefficients appearing in the divisions.

Finally, implementations for all the deferred methods of `Cat::DifferentialFunction` are provided. `diff` uses the operator for polynomial differentiation `Dpoly`. `solve` can only solve for differential variables appearing linearly (this can be checked by the method `isLinear`); in all other cases, it returns `FAIL`. A completely new implementation is given for `totalDiff`. It turned out that the approach via the Jacobian taken in `Cat::DifferentialFunction` is inferior to working directly on the list representation.

The following comparison of the execution times for the addition and the total differentiation of two random polynomials in `Dom::DifferentialPolynomial` and `Dom::DifferentialExpression`, respectively, demonstrates very clearly the usefulness of having a special domain for differential polynomials.

MuPAD
```
>> dp1:=DP::random(): dp2:=DP::random():
>> de1:=DE(dp1): de2:=DE(dp2):
>> time(DP::totalDiff(dp1,1));
>> time(DE::totalDiff(de1,1));
>> time(dp1+dp2 $ i=1..100);
>> time(de1+de2 $ i=1..100);
```
Output
```
1680

5630

90

1300
```

The differences in the execution times are so drastic, because we are dealing with rather large polynomials: `dp1` is here of total degree 97 in 58 differential variables up to order 5 and consists of 18 terms; `dp2` is of degree 60 in 51 differential variables also up to order 5 and consists of 13 terms. For smaller examples and if differential polynomials of differing orders are treated the speedup is usually smaller due to overheads, but nevertheless it is clearly worthwhile to use polynomial arithmetics whenever possible.

4.4 Linear Differential Functions

`Dom::LinearDifferentialFunction` is in many respects similar to `Dom::DifferentialPolynomial` but for linear instead of polynomial functions. For the arguments exactly the same rules apply. The linearity concerns only those differential variables which are not admitted in the coefficients. As it is not preserved under multiplication, this domain belongs only to the category `Cat::LeftModule(R)`, where `R` denotes again the coefficient ring, and not to `Cat::PartialDifferentialRing` as all the other domains in `Cat::DifferentialFunction`.

Another difference lies in the behaviour of the method `jacobian`. Usually, it returns a matrix whose entries are of the same type as the differential function. For linear functions this is not possible, as the matrix domains of *MuPAD* require a ring for the entries. In domains generated by `Dom::LinearDifferentialFunction` the method `jacobian` computes the Jacobian only with respect to those variables in which the functions are linear and returns a matrix over the coefficient ring `R`.

One must also be cautious with total differentiations. For constant coefficients or if the coefficient ring has the restriction `mode=["Indep"]` no problems arise. In all other cases, differentiation of the coefficients may yield nonlinear terms in variables in which the function should be linear. Hence for all other restrictions `totalDiff` returns an error.

Linear differential functions are finite sums with summands of the form coefficient times differential variable and possibly one single element of the coefficient ring to which we refer in the sequel as the *inhomogeneity* (or *"right hand side"*, if we think of differential equations). Consequently, we let `Dom::LinearDifferentialFunction(DV,R)` inherit its representation from the domain `Dom::FreeModule(R,DV)` supplying finite linear combinations of elements from `DV` over the ring `R` and add an additional slot for the inhomogeneity. The representation consists thus of one slot for storing a list containing all monomials in the form `[coefficient,variable]` and one slot for the inhomogeneity. The presence of the latter makes it necessary to reimplement many of the methods provided by `Dom::FreeModule`.

The method `convert` either accepts a list of pairs consisting of an element of the coefficient ring and a differential variable, optionally extended by a coefficient representing the inhomogeneity, or any expression that can be legally converted into a linear differential function. For example, we can generate linear differential functions with variable coefficients as follows:

MuPAD

```
>> LDF := Dom::LinearDifferentialFunction(Vars=[[x,y,z],[u,v]],
>>                                         Rest=[Types={"Indep"}]):
>> lf1 := LDF::convert([[2*x,u([x])],[y,u],3*z]);
>> lf2 := LDF::convert(2*x*(u([y])+v([z])));
```

Output

```
                          3 z + u y + 2 x u([x])

                          2 x u([y]) + 2 x v([z])
```

In order to check whether or not a linear differential function possesses an inhomogeneity, the method `isHomogeneous` can be used; the inhomogeneous term itself is extracted with `inhomogeneity`:

MuPAD

```
>> LDF::isHomogeneous(lf1);
>> LDF::inhomogeneity(lf1);
```

Output

```
                            FALSE

                             3 z
```

The implemented arithmetical methods are those specified in the category `Cat::LeftModule(R)`: `_plus`, `_negate` and `_mult`, where the latter means the scalar multiplication with elements of the ring `R`. Due to the similarity of the internal representation with the one used by polynomials, the polynomial methods mentioned in Sect. 4.3 are also provided by `Dom::LinearDifferentialFunction`. They differ slightly from the corresponding *MuPAD* functions:

- `lterm` and `nthterm` directly return the corresponding differential variable.
- `coeff` returns a list of all the coefficients of the linear differential function given as argument; if a differential variable is passed as second argument, the result is only its coefficient.
- In addition, the methods `tterm` and `tmonomial` for determining the lowest term and monomial as well as `terms` returning a list of all terms have been implemented.

Since no entry `one` exists in `Dom::DifferentialVariable`, it is not possible to represent a constant term. Hence any method which should return the constant term returns `FAIL` instead. Nevertheless, the inhomogeneity counts as a term when determining their number with `nterms`.

MuPAD

```
>> LDF::nterms(lf1);
>> LDF::nthterm(lf1,3);
```

Output

```
                             3

                            FAIL
```

Of the methods specified in `Cat::DifferentialFunction`, a new version of `totalDiff` is provided; the above made restrictions on when this procedure works apply. `solve` can solve only for those differential variables in which the function is linear and returns `FAIL` otherwise.

The domain contains some additional methods written in view of upcoming applications. These include the following ones:[8]

`changeIndVars(ldf,newVars,NewFromOld<,LDF>)`: This method performs a change of the independent variables in the linear differential function `ldf`. The names of the new variables are given in the list `newVars` and expressions of the new variables in terms of the old ones in the list `NewFromOld`. The optional argument `LDF` specifies the domain of the output, although there is a default one.

`changeDepVars(ldf,newVars,OldFromNew<,LDF>)`: The same for the dependent variables. Note however that for this method the old variables have to be given in terms of the new ones.

`makeMatrix`, `makeSystem`: These methods convert between a list of linear differential functions and its coefficient matrix. The matrix domain is identical to the one used for the Jacobian (i. e. the domain `Dom::SparseMatrix(DV,R)`). Inhomogeneities of the functions are ignored in the conversion.

As a simple example we consider a change of the independent variables in the linear function `lf2` introduce above. The second line shows the automatically generated output domain.

MuPAD
```
>> nlf2 := LDF::changeIndVars(lf2,[X,Y,Z],[X=3*x,Y=y,Z=z+y]);
>> domtype(nlf2);
```
Output
```
                2 X u([Y])   2 X u([Z])   2 X v([Z])
                ---------- + ---------- + ----------
                    3            3            3

Dom::LinearDifferentialFunction(
   Dom::DifferentialVariable([X, Y, Z], [u, v]),
   Dom::DifferentialExpression(Dom::RestrictedDifferentialVariable(
      Dom::DifferentialVariable([X, Y, Z], [u, v]), Types = Indep)))
```

[8] The first two methods have been supplied by Jay Belanger (Truman University).

5 Conclusions and Outlook

In this report we presented a programming environment for differential equations. We have not considered any serious application; but two packages for the completion of systems of differential equations and for Lie symmetry analysis built upon our environment will be described in two further reports of this series [5,6]. Their implementations will demonstrate some of the advantages and the flexibility of the object oriented approach to computer algebra.

Most general purpose computer algebra systems do not really use data types and even where they do, it is not very transparent to the user. The other extreme is a strongly typed system like AXIOM where the user must declare the type of each object (or must hope that the system can infer the type which is often very time consuming). This can be very inconvenient, especially if one just want to do some quick computations. *MuPAD* tries to combine these two approaches by distinguishing between basic types and the domains in the `domains` package. The user can enter some expressions without bothering about their types and perform all kinds of computations. But he can also work in a strongly typed environment for more sophisticated tasks. Bridges between the basic types and domains are provided by the methods `convert` and `expr` which each domain should possess.

The topic of a fourth report [1] in this series will be a new *MuPAD* library `DETools`. It allows users which are deterred by concepts like categories or domains to use sophisticated methods like those contained in the symmetry and the completion package; the library acts here as an interface that automatically chooses appropriate domains. In addition, it contains many others methods for solving, manipulating or visualising differential equations. A lot of them are again based on the environment described in this report.

All the categories and domains mentioned in this report will be contained in the forthcoming release 2.0 of *MuPAD*. Unfortunately, our environment cannot be used with the current version 1.4 of *MuPAD* due to some changes in the language.

Acknowledgements

Eduard Niess (Universität Karlsruhe) ported large parts of our earlier AXIOM version of an object oriented programming environment for differential equations to *MuPAD*. Jay Belanger (Truman University) contributed some methods to `Dom::LinearDifferentialFunction` and reported a number of bugs. This article was written with his EMuPAD package interfacing *MuPAD* and LaTeX within the Emacs editor. Finally, the authors would like to thank the *MuPAD* team at Universität Paderborn for their continuing support.

The second author is supported by Deutsche Forschungsgemeinschaft. He thanks J. Calmet for his hospitality at the Institut für Algorithmen und Kognitive Systeme at Universität Karlsruhe.

References

1. J. Belanger, M. Hausdorf, and W.M. Seiler. Differential equations in MuPAD IV: The DETools library. In preparation.
2. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1992.
3. K. Drescher. Axioms, categories and domains. Automath Technical Report No. 1, Universität Paderborn, 1996.
4. B. Fuchssteiner *et al.* *MuPAD — Multi Processing Algebra Data Tool.* Birkhäuser, Basel, 1993.
5. M. Hausdorf and W.M. Seiler. Differential equations in MuPAD II: A completion package. In preparation, 2000.
6. M. Hausdorf and W.M. Seiler. Differential equations in MuPAD III: A Lie symmetry package. In preparation, 2000.
7. D. Jenks and R.S. Sutor. Axiom – *The Scientific Computation System.* Springer-Verlag, New York, 1992.
8. B. Mishra. *Algorithmic Algebra.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1993.
9. W. Oevel, F. Postel, G. Rüscher, and S. Wehmeier. *Das MuPAD-Tutorium.* Springer-Verlag, Berlin, 2000.
10. J. Schü, W.M. Seiler, and J. Calmet. Algorithmic methods for Lie pseudogroups. In N. Ibragimov, M. Torrisi, and A. Valenti, editors, *Proc. Modern Group Analysis: Advanced Analytical and Computational Methods in Mathematical Physics*, pages 337–344. Kluwer, Dordrecht, 1993.
11. W.M. Seiler. Applying Axiom to partial differential equations. Internal Report 95–17, Universität Karlsruhe, Fakultät für Informatik, 1995.