Applying AXIOM to Partial Differential Equations

W.M. Seiler
Institut für Algorithmen und Kognitive Systeme
Universität Karlsruhe
76128 Karlsruhe, Germany
Email: seilerw@ira.uka.de

Abstract

We present an AXIOM environment called JET for geometric computations with partial differential equations within the framework of the jet bundle formalism. This comprises especially the completion of a given differential equation to an involutive one according to the Cartan-Kuranishi Theorem and the setting up of the determining system for the generators of classical and non-classical Lie symmetries. Details of the implementation are described and examples of applications are given. An appendix contains tables of all exported functions.

1 Computer Algebra and Differential Equations

Most casual users of computer algebra systems think that computer algebra and differential equations concerns basically the design of solution algorithms. But the real situation is fairly different. Although most general purpose systems provide a kind of `solve` command for differential equations, they actually employ mainly well-known techniques and some more or less clever heuristics to choose and apply them. Especially for partial differential equations no solution algorithm treating reasonably general and complicated systems is known, hence most computer algebra systems do not treat them at all.

Consequently, computer algebra applications to differential equations tend to work more indirectly. They help to study and understand properties of the solution space. Sometimes they may lead to special solutions. The reason for the use of computer algebra systems lies in the simple fact that most computations with differential equations tend to become soon very tedious. Hence such systems have proved to be an almost indispensable tool. We do not intend to give here a full survey but will briefly indicate the main directions emphasizing differential geometric methods. An excellent survey with a large bibliography oriented more towards algebraic approaches was given by Singer [53].

Almost any constructive method for differential equations has meanwhile been implemented in some computer algebra system. One can, however, distinguish certain approaches which have found most attention. One can classify them coarsely as follows:

- Symmetry analysis
- Singularity analysis
- Completion (construction of integrability conditions)
- Differential ideal theory
- Differential Galois Theory

Symmetry analysis. This approach has made the strongest impact on computer algebra applications to differential equations. There exist so many implementations that it is very difficult to keep an overview. We must refer here to a recent survey by Hereman [20] (an updated and extended version will appear in [21]). Most packages just set up the determining system for symmetry generating vector fields. A few of the packages also try some heuristics to solve it [19, 44]; usually they can cope, however, only with rather simple examples.

Singularity analysis. We must distinguish two approaches: In the Painléve analysis [22] one requires that every solution has at most poles as movable singularities. Movable means here depending on initial data. This method was originally developed for ordinary differential equations and later generalized to partial differential equations [61]. It represents one of the standard tests for (complete) integrability. Truncated Painléve expansions are also useful for the construction of Bäcklund transformations, Lax pairs and much more [60]. An implementation for ordinary differential equations was described by Rand and Winternitz [35]. Another direction is given by the construction of formal solutions in the neighborhood of singular points of the equations [14]. It is mainly applied to ordinary differential equations.

Completion. Any approach that claims to treat general systems of partial differential equations must deal with integrability conditions. The first systematic solution was probably given by the Janet-Riquier Theory[23] with the introduction of passive systems. Implementations of this ansatz have been undertaken by many authors, see e.g. [36, 45, 58]. The

so-called Differential Gröbner Bases [29] can also be seen as an extension of this approach. In geometric theories the notion of a passive system is replaced by involution. Hartley and Tucker [18] implemented the Cartan-Kähler approach [6] using exterior systems. An earlier version of our AXIOM implementation of the formal approach was published in Ref. [43],

Differential ideal theory. Here one tries to find a differential extension of algebraic ideal theory. Many of the ideas can already be found in the book of Ritt [39]. The main tools are either Gröbner bases or characteristic sets. Since the ring of differential polynomials is, however, no longer Noetherian, this generalization runs into problems, for algorithms along the lines of the Buchberger algorithm do not terminate in general [8]. As already mentioned above, Mansfield [29] introduced (and implemented) so-called Differential Gröbner Bases, but due to the use of pseudo-reduction they have weaker properties than their algebraic counterpart.

Differential Galois Theory. Already Lie was looking for a differential analog of the (algebraic) Galois Theory. What is nowadays called Differential Galois Theory resembles, however, only faintly his ideas. It is mainly based on the work of Kolchin [25] and culminates in the Singer algorithm for computing Liouvillian solutions of linear ordinary differential equations with rational coefficients [52]. An implementation of this algorithm has not been achieved so far. Only the simpler Kovačic algorithm for linear second order ordinary differential equations has been implemented [40]. Pommaret [32] has developed a Differential Galois Theory following Lie's ideas and using the formal theory.

A comparison of the impact made by symmetry analysis and by Differential Galois Theory, respectively, demonstrates clearly the importance of the availability of computer algebra tools. The latter one is a hardly known theory studied by a few experts only. The former one was in the same state for many decades following Lie's original work. One reason was definitely the extremely tedious determination of the symmetry algebra for any only slightly complicated equation. As soon as computer algebra systems emerged and were powerful enough, the first packages to set up at least the determining equations were developed. Since then Lie methods belong to the standard tools for treating partial differential equations.

We are in the sequel mainly concerned with the first and the third topic of above list. We will present an environment JET implemented in the computer algebra system AXIOM [24] to treat differential equations within the framework of the jet bundle formalism [41]. We want to stress here especially the fact that JET represents an *environment* and not a special purpose package.

Most computer algebra applications to differential equations consist of a package devoted to one special task. Since almost every author uses his own data structure etc. communication between different packages is very difficult. In contrast, in an environment like JET all packages employ the same structures and it is easily possible to exchange results or to process the output of one package with another one.

On the other hand, there might exist very good reasons to use different data structures in different applications, e.g. for reasons of efficiencies. But here we can take full advantage of the possibilities offered by AXIOM. By taking a categorical approach we basically fix only the signatures of many procedures. It is always possible to develop a domain based on a special data representation which is especially efficient for some specific task.

JET was developed as part of the Ph.D. thesis [47] of the author and this report is largely based on one chapter of it. It is organized as follows: The next three sections give a brief introduction into the main points of the underlying mathematical theory. Section 5 contains an overview over the used computer algebra system AXIOM highlighting the main

differences to more standard systems. The following four section describe in some detail the implementation of JET, whereas Sections 10 and 11 give examples of its application. Finally, some conclusions are given. An appendix contains tables of the exported functions for quick reference.

2 Involution

Formal theory uses a geometric approach to differential equations based on the jet bundle formalism. It is beyond the scope of this paper to give a detailed introduction into the underlying theory. The interested reader is referred to the literature [31, 47].

We will always work in a local coordinate system, although the whole theory can be expressed in a coordinate free way. Let X denote the space of the independent variables $x_1, \ldots, x_n$ and let the dependent variables $u^1, \ldots, u^m$ be fiber coordinates for the bundle $\mathcal{E}$ over the base space X. Derivatives are written in multi-index notation $p^\alpha_\mu = \partial^{|\mu|} u^\alpha / \partial x_1^{\mu_1} \cdots \partial x_n^{\mu_n}$ where $|\mu| = \mu_1 + \cdots + \mu_n$ is the length of the multi-index $\mu = [\mu_1, \ldots, \mu_n]$. Adding the derivatives p^α_μ up to order q defines a local coordinate system for the q-th order jet bundle $J_q\mathcal{E}$. A system of differential equation $\mathcal{R}_q$ of order q can be described locally by

$$\mathcal{R}_q : \left\{ \Phi^\tau \left(x_i, u^\alpha, p^\alpha_\mu \right) = 0 \,, \quad \tau = 1, \ldots, p \,;\ |\mu| \leq q \,. \right. \tag{1}$$

Geometrically, this represents a fibered submanifold of $J_q\mathcal{E}$.

At least some of the ideas behind the concept of involution can be understood best by considering the order by order construction of a formal power series solution. For this purpose, we introduce the *symbol* $\mathcal{M}_q$ of a differential equation $\mathcal{R}_q$. If $\mathcal{R}_q$ is locally described by the system (1), then its symbol is the solution space of the following linear system of (algebraic!) equations in the unknowns v^α_μ

$$\mathcal{M}_q : \left\{ \sum_{\alpha, |\mu|=q} \left(\frac{\partial \Phi^\tau}{\partial p^\alpha_\mu} \right) v^\alpha_\mu = 0 \,. \right. \tag{2}$$

(By abuse of language, we will refer to both the linear system and its solution space as the symbol).

The v^α_μ provide coordinates of the finite-dimensional vector space $S_q T^*X \otimes V\mathcal{E}$, i.e. we introduce one coordinate for each derivative of order q.[1] (2) is most easily understood by considering a quasi-linear system, i.e. a system linear in the derivatives p^α_μ with $|\mu| = q$. For such a system the symbol is simply obtained by taking only the linear highest order part and substituting v^α_μ for p^α_μ.

We make a power series ansatz for the general solution of the differential equation $\mathcal{R}_q$ by expanding around some point x^0

$$u^\alpha(x) = \sum_{|\mu|=0}^{\infty} \frac{a^\alpha_\mu}{\mu!} (x - x^0)^\mu \tag{3}$$

and substitute this ansatz into the equations (1) evaluating at x^0 . This yields a system of algebraic equations for the Taylor coefficients a^α_μ up to order q.

[1] S_q denotes the q-fold symmetric product, T^*X the cotangent bundle of X and V the verticle bundle. $J_q\mathcal{E}$ is an affine bundle over $J_{q-1}\mathcal{E}$ modeled on the vector bundle $S_q T^*X \otimes V\mathcal{E}$. One can thus consider $S_q T^*X \otimes V\mathcal{E}$ as the "vector space of all q-th order derivatives".

The remaining coefficients can be computed by linear algebra only. For the coefficients of order $q+r$ we use the *prolonged* systems $\mathcal{R}_{q+r}$ which are obtained by differentiating each equation in $\mathcal{R}_q$ r times formally with respect to all independent variables where the formal derivative is defined by

$$D_i\Phi^\tau = \frac{\partial\Phi^\tau}{\partial x^i} + \sum_\alpha \frac{\partial\Phi^\tau}{\partial u^\alpha}\, p_i^\alpha + \sum_{\alpha,\mu|} \frac{\partial\Phi^\tau}{\partial p_\mu^\alpha}\, p_{\mu+1_i}^\alpha \,. \tag{4}$$

Hence all prolonged equations are quasi-linear. If we substitute the power series ansatz into $\mathcal{R}_{q+r}$ and evaluate at x^0, we get an inhomogeneous linear system for the coefficients of order $q+r$. Its homogeneous part is determined by the prolonged symbol $\mathcal{M}_{q+r}$, i.e. the symbol of $\mathcal{R}_{q+r}$.

The Taylor coefficients a_μ^α of lower order appear in the matrix and in the right hand side of this linear system. Thus we are able to express the coefficients of order $q+r$ through the coefficients of lower order. This is the precise meaning of constructing a power series order by order.

This construction will fail, if non-trivial integrability conditions occur, i.e. equations of order $q+r$ which are functionally independent of the equations contained in the prolonged system $\mathcal{R}_{q+r}$ and which are satisfied by every solution of the system. Such equations arise usually by cross-differentiating and are detected only in some higher prolongation. They pose additional conditions on the coefficients of order $q+r$. Hence they must all be known to pursue the above described procedure. We call a system which contains all its integrability conditions a *formally integrable* system.

For formally integrable systems it is thus possible to construct order by order a formal power series solution. The arbitrariness of the general solution is reflected by the dimensions of the prolonged symbols, because at each order $\dim \mathcal{M}_{q+r}$ coefficients are not determined by the differential equations but can be chosen freely [48]. Formal integrability does, however, not suffice to determine these dimensions in advance without explicitly constructing the prolonged symbols. This leads to the concept of involution.

We introduce the *class* of a multi-index $\mu = [\mu_1, \ldots, \mu_n]$. It is the smallest k for which μ_k is different from zero. If we consider the symbol (2) as a matrix, then its columns are labeled by the coordinates v_μ^α. We order them by class, i.e. we always take a column with a multi-index of higher class left of one with lower class. Then we compute a row echelon form.

In this solved form the symbol is especially easy to analyze. Since we only need linear operations to obtain it, we can always perform the same operations with the full system $\mathcal{R}_q$ and thus assume that (2) yields the symbol directly in solved form. We denote the number of rows where the leading entry or pivot is of class k by $\beta_q^{(k)}$ and we associate with each such row its *multiplicative variables* $x_1, \ldots, x_k$.

It is important to note that if we prolong each equation only with respect to its multiplicative variables, we obtain independent equations, because each equation will have a different leading term. The question is, whether prolongation with respect to the non-multiplicative variables leads to additional independent equations. The symbol $\mathcal{M}_q$ is called *involutive*, if this does not happen, i.e. if

$$\operatorname{rank} \mathcal{M}_{q+1} = \sum_{k=1}^{n} k\beta_q^{(k)} \,. \tag{5}$$

The differential equation $\mathcal{R}_q$ is called *involutive*, if it is formally integrable and its symbol is involutive.

The above definition of the $\beta_q^{(k)}$ is obviously coordinate dependent. Thus it seems, as if the involution of a symbol depends on the chosen coordinate system, too. One can, however, show that almost every coordinate system leads to the same values for the $\beta_q^{(k)}$. These values are characterized by the property that all the sums $\sum_{i=k}^{n} \beta_q^{(i)}$, $k = 1, \ldots, n$, are maximal.[2] A coordinate system which leads to these values is called *δ-regular*. Definition 2 assumes that the $\beta_q^{(k)}$ are computed in such a coordinate system.

There exist ways to circumvent this problem. A straightforward method is to apply a generic linear change of coordinates, as this leads always to the correct values [31]. However, this approach is computationally very demanding. We developed recently an alternative method allowing to introduce the generic coordinates step by step as far as it is necessary. It is based on the so-called k-tableaux of a differential equation. We cannot explain it here but refer to Refs. [47, 50].

The prolongation of an involutive symbol is again involutive. Since prolonging an equation with respect to one of its multiplicative variables x_i yields an equation of class i, we get $\beta_{q+1}^{(i)} = \sum_{k=i}^{n} \beta_q^{(k)}$. Inductive use of this relation leads to

$$\beta_{q+r}^{(k)} = \sum_{i=k}^{n} \binom{r+i-k-1}{r-1} \beta_q^{(i)} \tag{6}$$

and together with Definition 2 to

$$\operatorname{rank} \mathcal{M}_{q+r} = \sum_{k=1}^{n} \binom{r+k-1}{r} \beta_q^{(k)} \,. \tag{7}$$

Besides the possibility to predict the number of arbitrary Taylor coefficients at any order, involutive systems have another advantage compared with formally integrable ones. There exists an easily applicable criterion to check whether or not a system is involutive. The problem of the definition of formal integrability is that one has to prove that a system does not generate non-trivial integrability conditions at any prolongation order, i.e. one must check an infinite number of conditions. This can, however, be done in a finite manner for systems with an involutive symbol.

Theorem 1 *Let $\mathcal{R}_q$ be a q-th order differential equation with an involutive symbol $\mathcal{M}_q$. If no integrability conditions arise during the prolongation of $\mathcal{R}_q$ to $\mathcal{R}_{q+1}$, then $\mathcal{R}_q$ is involutive.*

3 Completion to Involution and Arbitrariness

Since we have seen that involutive systems have many advantages, the question naturally arises whether they form only a very special class of systems and what to do with a non-involutive system. The answer is given by the Cartan-Kuranishi Theorem [27, 31, 47].

Theorem 2 *Any system $\mathcal{R}_q$ can be completed to an equivalent involutive one by a finite number of prolongations and projections (i.e. addition of integrability conditions).*

Since this theorem depends on some fairly deep results in the formal theory, we will not present a proof but only discuss an algorithm to perform this completion. It is based on Theorem 1 above and consists essentially of two nested loops. The inner loop prolongs the

[2] Note that this is different from requiring that the $\beta_q^{(k)}$ themselves take maximal values!

system until its symbol becomes involutive. The outer loop checks then for integrability conditions and adds them. The difficult part of the proof is to show the termination of the inner loop. The termination of the outer one follows from a simple Noetherian argument.

Involution of a symbol can be checked easily using Definition 2, if we assume that the coordinate system is δ-regular what we will do in the sequel. It requires only linear algebra. Whether or not integrability conditions arise during a prolongation, can be deduced from a dimensional argument.

Denote the projection of the system $\mathcal{R}_{q+1}$ into the q-th order jet bundle $J_q\mathcal{E}$ by $\mathcal{R}_q^{(1)}$. Its dimension can be computed indirectly from the identity

$$\dim \mathcal{R}_q^{(1)} = \dim \mathcal{R}_{q+1} - \dim \mathcal{M}_{q+1} \tag{8}$$

which reflects the fact that integrability conditions are connected with rank defects in the symbol. None has occurred during the prolongation from $\mathcal{R}_q$ to $\mathcal{R}_{q+1}$, if and only if this dimension is equal to $\dim \mathcal{R}_q$.

There are essentially two possible reasons for integrability conditions. The classical one is that it is possible by some linear combination of equations of order $q+1$ in $\mathcal{R}_{q+1}$ to eliminate all derivatives of that order. This is a generalization of the usual cross-derivative. The other one is that $\mathcal{R}_q$ contains some equations of lower order. In order to construct $\mathcal{R}_{q+1}$ all equations in $\mathcal{R}_q$ must be prolonged. If now some equations are of lower order, it might happen that their prolongation leads to new independent equations of order less than or equal to q. They must be taken into account in the projection to $\mathcal{R}_q^{(1)}$.

Fig. 1 shows this algorithm in a more formal language. $\mathcal{R}_{q+r}^{(s)}$ denotes here the system obtained after $r+s$ prolongations and s projections. $\mathcal{M}_{q+r}^{(s)}$ is the corresponding symbol. In this form it is comparatively straightforward to implement it in a computer program. The determination of the dimensions of the various submanifolds $\mathcal{R}_{q+r}^{(s)}$ poses the main remaining problem, especially for non-linear systems.

```
[1]     r ← 0; s← 0
[2]     compute R_{q+1}                                   {prolong}
[3]     compute M_q, M_{q+1}                              {extract symbols}
[4]     until R^{(s)}_{q+r} involutive repeat
[4.1]       while #multVar(M^{(s)}_{q+r}) ≠ rank M^{(s)}_{q+r+1} repeat
[4.1.1]         r ← r+1                                   {counter for prolongations}
[4.1.2]         compute R^{(s)}_{q+r+1}                   {prolong}
[4.1.3]         compute M^{(s)}_{q+r+1}                   {extract symbol}
[4.2]       if dim R^{(s)}_{q+r+1} − dim M^{(s)}_{q+r+1} < dim R^{(s)}_{q+r} then
[4.2.1]         s ← s+1                                   {counter for projections}
[4.2.2]         compute R^{(s)}_{q+r}                     {add integrability conditions}
[4.2.3]         compute R^{(s)}_{q+r+1}                   {prolong}
[4.2.4]         compute M^{(s)}_{q+r}, M^{(s)}_{q+r+1}    {extract symbols}
[5]     return R^{(s)}_{q+r}
```

Figure 1: Algorithm for the Cartan-Kuranishi Theorem

For ordinary differential equations this algorithm becomes very simple. Since there is only one independent variable, we find always an involutive symbol and cross-derivatives

are of course not possible. The only possibility for integrability conditions is the prolongation of lower order equations. For partial differential equations we recall that the other integrability conditions can always be found by considering the prolongations with respect to non-multiplicative variables.

To conclude this section we briefly recall some results of Ref. [48] concerning the arbitrariness of the general solution. (7) yields only the rank of the prolonged symbols, but their dimensions are more interesting. They can be expressed in a similar way, if we introduce the *Cartan characters* $\alpha_q^{(k)}$ of a differential equation

$$\alpha_q^{(k)} = m\binom{q+n-k-1}{q-1} - \beta_q^{(k)}, \qquad k = 1, \ldots, n\,. \tag{9}$$

They form a descending sequence

$$\alpha_q^{(1)} \geq \alpha_q^{(2)} \geq \cdots \geq \alpha_q^{(n)} \geq 0\,. \tag{10}$$

Now we can write

$$\dim \mathcal{M}_{q+r} = \sum_{k=1}^{n} \alpha_{q+r}^{(k)} = \sum_{i=0}^{n-1} \left(\sum_{k=i}^{n-1} \frac{\alpha_q^{(k+1)}}{k!} s_{k-i}^{(k)}(0) \right) r^i\,. \tag{11}$$

This is the *Hilbert polynomial* of the differential equation $\mathcal{R}_q$. $s_k^{(n)}(q)$ denotes here the modified Stirling numbers, combinatorial factors introduced in Refs. [47, 48]. They can be defined by the recursion formula

$$s_k^{(n)}(q) = s_k^{(n-1)}(q) + (q+n)\, s_{k-1}^{(n-1)}(q) \tag{12}$$

($s_0^{(n)}(q) = 0$ and $s_n^{(n)}(q) = (q+n)!/q!$). It is easy to see that there is a one-to-one correspondence between the coefficients of the Hilbert polynomial and the Cartan characters.

Analyzing the number of arbitrary Taylor coefficients in the power series expansion of the general solution and comparing with these dimensions yields the following result.

Theorem 3 *If there exists an algebraic representation of the general solution of the involutive differential equation $\mathcal{R}_q$ with Cartan characters $\alpha_q^{(k)}$ whose power series expansion can be constructed order by order, then it contains f_k free functions of k variables, where the f_k are determined by the recursion relation:*

$$\begin{aligned} f_n &= \alpha_q^{(n)}\,, \\ f_k &= \alpha_q^{(k)} + \sum_{i=k+1}^{n} \frac{(k-1)!}{(i-1)!} \left(\alpha_q^{(i)} s_{i-k}^{(i-1)}(0) - f_i s_{i-k}^{(i-1)}(q) \right)\,, \quad 0 < k < n\,. \end{aligned} \tag{13}$$

Such a representation can only exist, if the solution of (13) contains only non-negative integers.

For the precise meaning of the term "algebraic representation" we refer again to Refs. [47, 48]. Essentially it implies that no derivatives or integrals of the arbitrary functions occur. For first-order equations this recursion relation can be solved explicitly and it follows from (10) that the numbers f_k are always non-negative.

Theorem 4 *For a first-order differential equation $\mathcal{R}_1$ the numbers f_k are determined by*

$$\begin{aligned} f_n &= \alpha_1^{(n)} = m - \beta_1^{(n)}\,, \\ f_k &= \alpha_1^{(k)} - \alpha_1^{(k+1)} = \beta_1^{(k+1)} - \beta_1^{(k)}\,. \end{aligned} \tag{14}$$

4 Symmetry Theory

The most general definition of a symmetry simply states that it is a transformation that maps solutions into solutions. We will consider here diffeomorphisms $\phi : \mathcal{E} \mapsto \mathcal{E}$ and call them a *Lie point symmetry* of the differential equation $\mathcal{R}_q$, if $\phi^* \mathcal{R}_q = \mathcal{R}_q$. In practice it is usually impossible to find all such symmetries. But Lie has shown that it suffices to treat infinitesimal transformation, i.e. to consider vector fields on $\mathcal{E}$ of the form

$$\vec{v} = \sum_{i=1}^{n} \zeta^i(x,u) \frac{\partial}{\partial x^i} + \sum_{\alpha=1}^{m} \eta^\alpha(x,u) \frac{\partial}{\partial u^\alpha} . \tag{15}$$

Using the chain rule it is straightforward to define the prolongation $\mathrm{pr}^{(q)}\vec{v}$, a vector field acting on $J_q\mathcal{E}$, by

$$\mathrm{pr}^{(q)}\vec{v} = \vec{v} + \sum_{\alpha, |\mu| \le q} \eta^\alpha_\mu \frac{\partial}{\partial p^\alpha_\mu} \tag{16}$$

where the coefficients can be determined recursively

$$\eta^\alpha_{\mu+1_k} = D_k \eta^\alpha_\mu - \sum_{i=1}^{n} D_k \zeta^i p^\alpha_{\mu+1_i} . \tag{17}$$

If now a local representation of $\mathcal{R}_q$ is given by (1) and

$$\mathrm{pr}^{(q)}\vec{v}(\Phi^\tau)\Big|_{\mathcal{R}_q} = 0 \tag{18}$$

holds for all functions Φ^τ, then the flow generated by $\vec{v}$ is a symmetry in the sense of the above definition. One usually calls (18) the *determining system* of $\mathcal{R}_q$. It is an, in general over-determined, system of linear partial differential equations. This shows the decisive advantage of symmetry analysis: even if $\mathcal{R}_q$ is a non-linear equation, its determining system is always linear and thus usually much easier to tackle. This linearization corresponds to the fact that we use instead of the Lie group of finite transformations its Lie algebra.

From a computational point of view the biggest problem in evaluating (18) lies in the restriction to $\mathcal{R}_q$. Usually this is solved by assuming that one is given a local representation of $\mathcal{R}_q$ where each equation is solved for some derivative. Then one must simply perform the corresponding substitution. For more general equation one could use Gröbner basis techniques (cf. Section 9).

As a trivial application of the results of the previous section one can calculate the size of the symmetry algebra, if one is not able to solve explicitly the determining system (18). Reid [37, 38] has shown that it is even possible to obtain the structure constants of the algebra without solving (18), although there are some problems in the case of an infinite-dimensional algebra.

Reduction with respect to a symmetry group provides a standard technique for the construction of solutions. It is well-known that only for ordinary differential equations one can reconstruct the general solution of the original equation from the general solution of the reduced equation by quadratures. In the case of partial differential equations one obtains only special solutions.

The idea behind symmetry reduction is to look for solutions which are invariant under a symmetry group, i.e. the symmetry transformations map such a solution into itself. If the symmetry is generated by a vector field $\vec{v}$ of the form (15), then a group invariant

solution $u^\alpha(x^i)$ satisfies not only the considered differential equation $\mathcal{R}_q$ but in addition the *invariant surface condition*

$$\sum_{i=1}^{n} \zeta^i(x,u) p_i^\alpha = \eta^\alpha(x,u)\,, \qquad \alpha = 1,\ldots,m\,. \tag{19}$$

These are quasi-linear, first-order equations. In the case of a higher-dimensional symmetry algebra we must add one such set of conditions for each generator. If one can compute the general solution of (19), one substitutes it into the equations of $\mathcal{R}_q$ to obtain a reduction. Or, if we assume without loss of generality that $\zeta^n \neq 0$, we can solve (19) for the derivatives p_n^α and thus eliminate in $\mathcal{R}_q$ all derivatives with respect to x^n. This yields a differential equation in $n-1$ independent variables.

Bluman and Cole [4] were the first to observe that for this construction it is not necessary that $\vec{v}$ generates a symmetry of the differential equation. It suffices, if $\vec{v}$ yields a symmetry of $\mathcal{R}_q$ augmented by the invariant surface condition (19). Since the coefficients ζ^i, η^α of $\vec{v}$ are now also part of the differential equation, the resulting determining system is non-linear opposed to the situation in the classical Lie theory. This is a serious draw-back in the concrete application of this so-called *non-classical method*.

The transformations generated by such vector fields are sometimes called *conditional symmetries* [28], as in general they do not map all solutions of $\mathcal{R}_q$ into other solutions. They also do not form a Lie algebra, because every conditional symmetry depends on its own invariant surface condition (19).

Gauge symmetries form a special class in that they are fiber-preserving transformation of the bundle $\mathcal{E}$. Usually they depend on some arbitrary functions of all independent variables. (This implies that f_n cannot vanish for a system with such a symmetry.) In gauge theories one identifies solutions related by a symmetry transformation. In order to obtain information about the arbitrariness of the physically relevant part of the solution space we must adjust the Cartan characters.

Let us assume that the gauge transformation can be written in the following form

$$\begin{aligned} \bar{x}^i &= \Omega^i(x^j)\,, \\ \bar{u}^\alpha &= \Lambda^\alpha(x^i, u^\beta, \lambda_a^{(0)}(x), \partial\lambda_a^{(1)}(x), \ldots, \partial^p\lambda_a^{(p)}(x)) \end{aligned} \tag{20}$$

where γ_0 gauge functions $\lambda_a^{(0)}$ are entering algebraically, γ_1 gauge functions $\lambda_a^{(1)}$ are entering through their first derivatives etc. Refs. [47, 49] show how one can handle more general cases using a pseudogroup approach based on an implicit representation of the transformations by differential equations.

Under this assumption the gauge correction term $\Delta\alpha_q^{(k)}$ which must be subtracted from $\alpha_q^{(k)}$ to adjust for the symmetry can be computed recursively through [48, 49]

$$\Delta\alpha_q^{(k)} = \frac{(k-1)!}{(n-1)!} \sum_{l=0}^{p} \gamma_l s_{n-k-1}^{(n-1)}(q+l) - \sum_{i=k+1}^{n} \frac{(k-1)!}{(i-1)!} \Delta\alpha_q^{(i)} s_{i-k}^{(i-1)}(0)\,. \tag{21}$$

It is also possible to derive a similar formula for the gauge corrected Hilbert polynomial.

5 AXIOM

AXIOM [24] was mainly developed at the IBM research center in Yorktown Heights under the name of Scratchpad II. Since 1992 it has been marketed by NAG under the new trade

mark AXIOM. We are currently using Version 1.2. It is rather different compared with other general purpose computer algebra systems. Systems like REDUCE, Maple, Mathematica etc. differ not much in their principal structure: they have one basic data type for symbolic expressions and the core of the system consists of the simplifier. The real differences between the systems lay in the simplification strategy, in the power of the advanced packages (e.g. factorization, integration, equation solving) in the system library, in the user interface etc.

AXIOM has a completely different structure. It is inspired by category theory and the theory of abstract data types. Meanwhile, several attempts to incorporate this approach into other systems have been reported [16, 17]. We will describe it briefly in this section in order to give a reader unacquainted with AXIOM a chance to understand some of the following sections. More detailed descriptions of the AXIOM type system and the underlying philosophy can be found in Refs. [10, 11, 59].

There is a three level hierarchy: Any *object* is member of one (and only one) *domain* which itself is member of a *category*. The easiest way to understand this hierarchy is to consider a concrete example: The object 5 is a member of the domain `Integer` which is a member of the category `Ring`. In the language of computer science, domains represent *abstract data types*, whereas a category is considered as a *second order type*.

Besides categories and domains there are like in most computer algebra systems *packages*. They do not implement an abstract data type but consist simply of a collection of procedures. Their structure is very similar to a domain, i.e. they first specify the exported operations, then they provide an implementation for them. Unlike a domain, however, a package does not need a representation, as it has no members.

A category specifies which operations are possible in its domains, i.e. it declares their signatures. What it does not contain are any information on the representation of objects. The category `Ring`, for instance, requires that every domain belonging to it must have an operation `+` which takes two ring elements as arguments and returns a ring element. It does not, however, give a concrete implementation of this operation, because it will probably be different in the different domains belonging to `Ring`. The addition of two integers will be provided by the domain `Integer`.

As we will see later a category can also contain implementations of operations. The big difference between a domain and a category is that the former one specifies a *representation* for its objects in terms of other data types. Within the body of the domain constructor this representation type is considered as equivalent to the new data type generated by the domain. This means that if the representation consists e.g. of a record, it is possible to access the individual slots of this record in procedures of the domain. The representation is, however, invisible outside of the domain, i.e. operations from other domains or packages which use objects of this new type cannot see the slots of the record.

A category may be a sub-category of another one. For instance every field is a ring but with additional properties. Consequently, in AXIOM the category `Field` is a refinement of `Ring` by further operations like division, i.e. it *inherits* all specifications of `Ring` and adds new ones. This avoids the tedious duplication of signatures.

Although the system's name is now AXIOM, its weak point is actually the treatment of axioms. A field is not only distinguished from a ring by the fact that it allows division but also by the fact that its multiplication is always commutative. The only way AXIOM can handle such properties is through *attributes*. `Field` has e.g. the attribute `commutative("*")`. But this amounts in practice to not much more than a comment, because the system has no way to check whether or not the implementation of `*` in a domain is commutative. Thus it is more a remainder to the programmer. It is, however, possible

to ask whether a given domain has a certain attribute. This is useful, when domains or categories are passed as arguments (see below).

AXIOM's approach to symbolic computation has several advantages. It helps the programmer to ensure the correctness of his computations. Since it is a strongly typed system, every object must have a type, i.e. it belongs to a domain which determines what kind of operations can be performed with it. As long as the semantic of the used domains is sound, only well-defined operations can be called. For instance it is not possible to divide ring elements for the simple reason that no division is defined in the category `Ring`! Of course one can always circumvent this protection mechanism, but it still helps correct programming.

While this point might appear a bit vague and fancy, a real advantage of the object-oriented approach lies in the possibility of generic programming. AXIOM offers two mechanisms for this purpose. The first one is to define procedures "categorically", i.e. one implements a procedure already at the level of a category. Then every domain in the category (and every sub-category) inherits this default implementation; it can, however, still provide its own implementation overriding the default one.

A prototypical example for this technique is the Euclidean algorithm to compute greatest common divisors. AXIOM provides a sub-category of `IntegralDomain` called `EuclideanDomain`. It contains a default implementation for the operation `gcd` using the Euclidean algorithm. Every domain which belongs to `EuclideanDomain` automatically inherits it. Of course this algorithm uses operations which are differently implemented in each of these domains like multiplication. This does not matter, as the category specifies the signature of multiplication and this information suffices for the compiler.

The possibility to pass domains as parameters to another domain (or package or category) yields another way of generic programming. Again AXIOM requires that every parameter is given a type. In the case of a domain this type is the category to which it belongs. This category also specifies all operations possible with objects of the domain. Thus it is only possible to use these operations. This mechanism is for instance used in our implementation of the completion algorithm for the Cartan-Kuranishi Theorem (cf. Section 8). The corresponding package takes as argument a domain which represents the class of differential equations considered (e.g. linear equations or polynomial equations). It only knows that this domain lies in the category `JetBundleFunctionCategory`.

Our completion algorithm requires several non-trivial operations like computing the dimension of a submanifold or simplification of a system. The implementation of these varies strongly depending on the kind of equations considered. But this does not matter for the completion procedure which can be written completely independent. Thus one can take full advantage of specific properties of special classes of differential equations and still use the same completion procedure.

A further advantage of AXIOM is the possibility to overload procedure names, i.e. several procedures can have the same name as long as they are distinguished by different signatures. Probably the heaviest overloaded name is `coerce`. This name is always used, if objects are transformed from one representation into another equivalent one. Every domain must contain a `coerce` procedure to the domain `OutputForm`, because this determines the way its objects are displayed in the output.

The implementation of categories, domains or packages requires the use of the compiler. Interactive work with AXIOM is performed using the interpreter. It has sometimes a slightly different syntax. The main difference is, however, that in the interpreter it is no longer necessary to enter an explicit type for every object.

Since this would be very tedious, the interpreter provides a type inference mechanism

which tries to derive the types of the objects in the input line. Only if there are ambiguities or the structure of the input is too complicated, the user must declare the types of at least some objects explicitly. The only information this mechanism needs are the signatures (here often called mode maps) of all used operations. (A description of it can be found in Ref. [56]).

Because of the huge size of the library it is not possible to load its complete content at the start of an AXIOM session. If the run time system needs an operation which is not "exposed", as this is called in AXIOM, it is automatically loaded. This autoload scheme does not work with user provided domains and categories. All those that will be used during the calculations must be loaded explicitly using the `)load` command of the interpreter. To facilitate the entering of the often very long type names the AXIOM compiler requires that every category, domain or package is assigned an abbreviation. We will also make use of this and denote everything by its abbreviation as soon as it is introduced.

The remainder of this section contains a short overview of a few domains etc. contained in the AXIOM library which have been used in our implementation. Due to the size of the library one of the biggest difficulties for an AXIOM programmer is to find the ones he needs... Basic data types comprise `Integer` with the subtype `NonNegativeInteger`, `Symbol` and `Boolean`. Rational numbers can be obtained as `Fraction Integer` (the domain constructor `Fraction` takes a ring as argument and returns its quotient field). More complicated data types can be constructed using e.g. `List`, `Vector`, `Matrix` or `Record`.

AXIOM provides a complicated hierarchy of categories for the basic structures used in algebra [12]. We will need only a few of these and most of them have already been mentioned. We will deal almost exclusively with rings, i.e. domains in `Ring`. Sometimes we will have additional structures or properties like a `GcdDomain` or a `PartialDifferentialRing`. The latter one adds partial differentiations with respect to elements of a given set. In our applications this set will always be the domain `Symbol`.

The data type for general expressions is `Expression Integer`. It corresponds roughly to the one basic data type used by most computer algebra systems. It can represent arbitrary expressions like $\sin(x+y)+f(x*y)$ with an unspecified function f. All the usual operations like plus, minus, differentiate, etc. can be performed within this domain. Like most other computer algebra systems AXIOM represents an element of it internally as a polynomial. The unknowns are so-called kernels. A kernel can be a simple variable like x but also a more complicated expression like $f(x*y)$. Basically everything that cannot be further simplified is a kernel (of course AXIOM disposes of a domain called `Kernel` for such objects). Kernels like $f(x*y)$ are generated using `BasicOperator`.

Polynomials are probably the most important objects in computer algebra. Hence AXIOM provides many different categories and domains for them. They vary basically in the chosen representation, as for different tasks different ones may be most efficient. `SparseMultivariatePolynomial` uses for instance a recursive representation and `DistributedMultivariatePolynomial` obviously a distributed one.

6 Implementation I — General Remarks

Most computer algebra applications to differential equations published so far consist of special purpose packages devoted to one specific algorithm. Communication between different packages, e.g. further processing of the output of one algorithm by another one, is then often rather awkward, as every package uses its own data structures and thus complicated conversion procedures must be implemented.

This problem could be avoided by implementing a general purpose environment for geometric computations. Such an environment should comprise basic data structures and procedures for jet bundles and differential equations as they are typically needed. Within this environment it would then be possible to implement packages for different purposes like completion to an involutive system, construction of the symmetry algebra etc.

We have started with the development of such a general environment for computations within the jet bundle formalism. We believe that AXIOM is especially well adapted for such a project because of its unique structure. A first version was implemented by J. Schü in his diploma thesis [42]. Together with some minor improvements it was presented in Ref. [43]. The current version is mainly distinguished from the earlier one by much more efficient data structures and the introduction of many simplification routines.

In its current state the environment comprises over 6000 lines of AXIOM code organized in 3 categories, 11 domains and 3 packages. There is a natural hierarchy consisting of four layers:

1. Jet bundles, i.e. data structures for local coordinate systems
2. Sections, i.e. functions defined on these charts
3. Higher structures like vector fields, differential forms or a special data structure for differential equations
4. Application packages

The first layer is basically for a better looking user interface. One could argue whether it is really necessary. The only "computations" that happen in it are the construction of the output form for jet variables.

Before we will give a detailed description of the implementation, we present an overview of all the categories and domains provided in the individual layers (see also Fig. 2). A list of their abbreviations can be found in Fig. 3.

Layer 1. The category `JetBundleCategory` specifies jet bundles with a fixed number of independent and dependent variables. The order of the jet bundle is not fixed but passed as argument to the procedures needing it. An instance of this category is given by the domain `JetBundle`. It implements jet bundles with arbitrary names for the independent and dependent variables. The domain `IndexedJetBundle` takes as parameters the numbers of these variables and symbols to construct their names. The domain `JetBundleSymAna` is only needed by the symmetry package (see below).

Layer 2. The basic category is `JetBundleFunctionCategory`. There is a subcategory `BaseFunctionCategory` for functions defined only on the base space X. Currently four instances are implemented. `JetBundleExpression` is basically an extension of `Expression Integer`. Thus it represents in some sense the most general domain in this category. A similar domain is given by `JetBundleXExpression` which contains only functions of the independent variables. Its main use is to serve as coefficient ring for linear functions realized in `JetBundleLinearFunction`. `JetLazyFunction` implements a very special instance. It takes another domain of this category as argument and sets on top of it a lazy evaluation scheme for partial differentiations. Its main purpose is to avoid unnecessary calculations when many prolongations are needed.

Layer 3. There are two domains `VectorField` and `Differential` for vector fields and one-forms, respectively, implementing the basic differential geometric operations with them. The domain `DifferentialEquation` provides basic operations

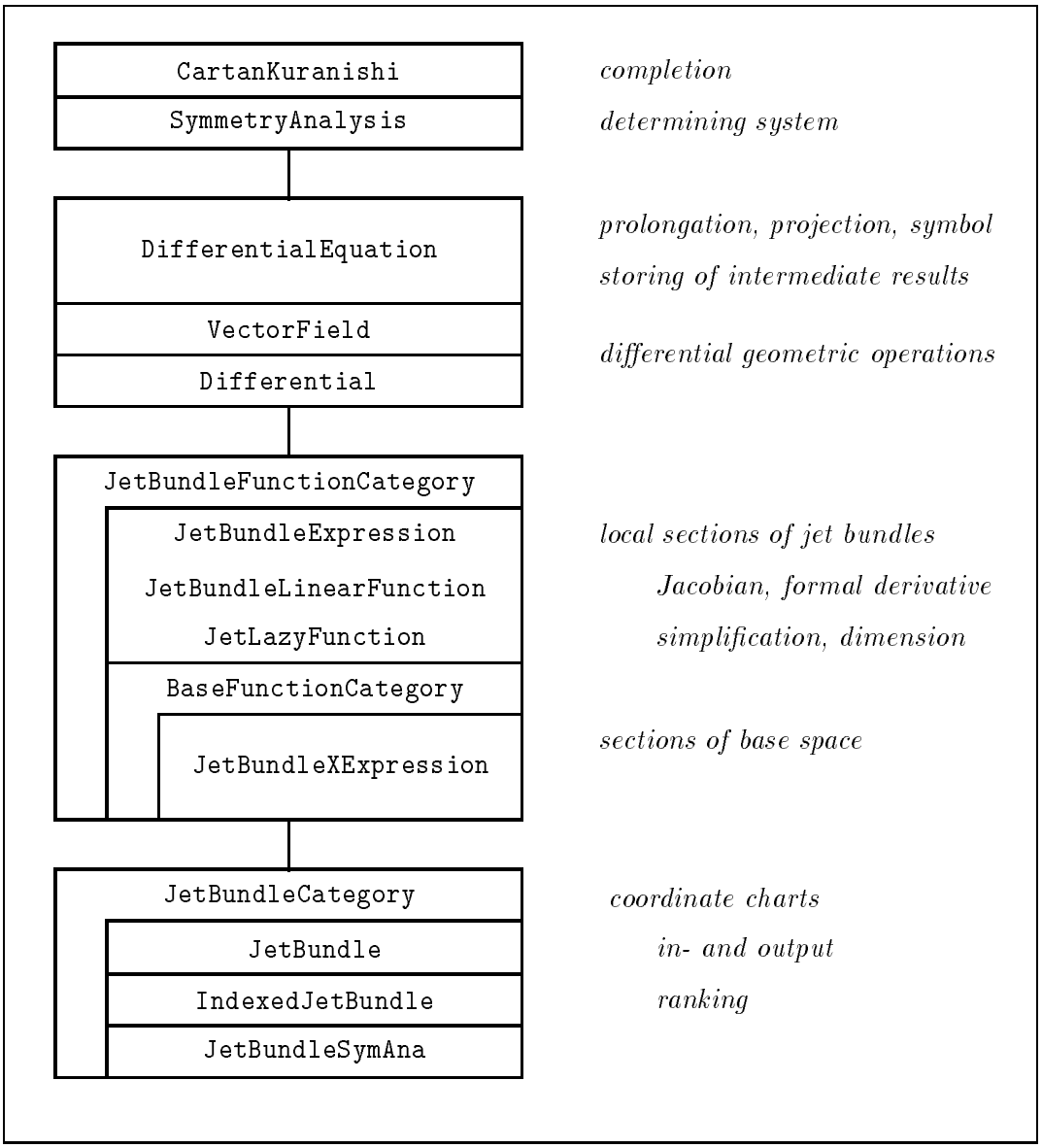

Figure 2: Overview of the implemented categories, domains, and packages.

like prolongation, projection or construction and analysis of symbols and tableaux. Its main purpose is to enhance the efficiency by storing many intermediate results like Jacobians or dimensions.

Layer 4. The two application packages implemented so far are `CartanKuranishi` and `SymmetryAnalysis`. The former one contains a completion procedure for the Cartan-Kuranishi Theorem. Additionally there are some procedures for calculations with Cartan characters, the Hilbert polynomial and for gauge corrections. The symmetry package is still in a very earlier stage. Currently it contains only a primitive procedure to produce determining systems.

Besides these there are three utility packages. `SparseEchelonMatrix` provides a special representation of sparse matrices designed for the fast and efficient computation of row

echelon forms, whereas LUDecomposition implements the LU decomposition method for the standard matrix type of AXIOM. JetCoordinateTransformation prolongs coordinate transformations in the base space $\mathcal{E}$ into the jet bundle.

JetBundleCategory	JBC
JetBundle	JB
IndexedJetBundle	IJB
JetBundleSymAna	JBSA
JetBundleFunctionCategory	JBFC
BaseFunctionCategory	BFC
JetBundleExpression	JBE
JetBundleXExpression	JBX
JetBundleLinearFunction	JBLF
JetLazyFunction	JLF
DifferentialEquation	DE
VectorField	VF
Differential	DIFF
CartanKuranishi	CK
SymmetryAnalysis	SYMANA
SparseEchelonMatrix	SEM
JetCoordinateTransformation	JCT
LUDecomposition	LUD

Figure 3: Abbreviations of the new categories, domains, and packages.

Examples of the application of this environment in concrete computations will be given in Sections 10 and 11. They contain complete examples with in- and output of AXIOM sessions. Tables with most of the exported procedures can be found in Appendix A. They also contain short descriptions of them and can thus be used as a kind of reference manual.

7 Implementation II — Jet Bundles

This section describes in some detail the implementation of the first two layers. JetBundleCategory was mainly introduced to facilitate the passing of jet bundles as arguments to domains and categories of higher layers. Otherwise it would be necessary to repeat each time all the parameters of a jet bundle. Furthermore, it is convenient for some special applications to use another representation than the one provided by the domain JetBundle. For instance, the notation with indexed variables is of advantage for many large systems; hence there is a domain IndexedJetBundle. Most operations are already implemented in JBC; JB and IJB provide essentially only the procedures for the generation and for the output of jet variables.

IJB uses as internal representation a record containing the upper and lower index of a jet variable and its type. It knows four different types: Besides the obvious ones Indep for the independent variables, Dep for the dependent ones and Deriv for the derivatives there is a special one called Const. It is assigned to the special "jet variable" 1 which is needed for differentiation and for the representation of linear functions (see below).

The upper index labels the different independent and dependent variables; the lower

index is needed for derivatives. IJB recognizes two different notations for the lower index. Internally always standard multi-index notation is used. For in- and output the user can choose between this and repeated index notation which is the default, as it is usually more convenient for derivatives of low order. The notation can be selected with the procedure setNotation.

Jet variables are generated with the procedures X, U and P. For the special cases of only one independent or only one dependent variable, respectively, it is possible to omit the label of the variables. P interprets its second argument as a lower index in the currently chosen notation. Pm and Pr always use multi-index and repeated index notation, respectively.

The implemented ranking is the inverse lexicographic ranking. In addition to < and > JBC provides a procedure weight which assigns each jet variable a unique integer such that a higher ranked variable always gets a higher number. Currently there are no provisions for other rankings, although this could be useful for computations as performed in Janet-Riquier Theory. Especially rankings of the type studied by Janet can easily be parameterized by matrices [23].

The coerce from JBC to Symbol consists of the straightforward construction of a symbol with the correct scripts. However, this operation "looses" all information about the jet variable. It is no longer possible to distinguish this symbol from others as a jet variable. Coercion to Expression Integer (EI) offers here much more possibilities. The corresponding procedure constructs a BasicOperator (BOP) which in turn can easily be coerced to EI.

An object of type BOP has associated with it a so-called property list which allows to store flags and indicators for properties of the object. We have introduced a new flag "%jet" to distinguish jet variables from other kernels. Furthermore we store the weight of each jet variable using the indicator "%weight". This indicator is used by the type Kernel Expression Integer to determine an ordering. Thus we can impose our ranking even after coercion to EI. This is crucial for the implementation of JBE (see below).

JB uses exactly the same representation as IJB. However, it employs always repeated index notation. JB takes two lists of symbols as arguments representing the names of the independent and dependent variables. This is in many applications more natural than indexed variables. For easier input JB provides two additional routines with symbols as arguments.

JBSA is more or less identical with JB but computes its parameters in a special way. It takes as argument another jet bundle. Its independent and dependent variables will be the independent variables of the new jet bundle. The other arguments represent the names used for the dependent variables. Such a domain is needed for some procedures in the symmetry package (cf. Section 8).

JetBundleFunctionCategory provides a default implementation for most operations defined by it. These can be coarsely divided into differential operations like partial or formal derivatives or computation of Jacobians, simplification routines, and some miscellaneous operations like determination of the leading derivative of a function or its effectively occurring jet variables. Furthermore there are the algebraic operations inherited from the category PartialDifferentialRing Symbol and some operations are lifted from JBC to allow for easier input.

Important among the miscellaneous operations are jetVariables, solveFor, subst, order, leadingDer, and dSubst, as these are much used by the default implementations of the other operations. For the first three of these no default implementation is provided. jetVariables returns a list of all effectively in a function occurring jet variables. solveFor tries to solve a function for a given jet variable. If that is not possible, it returns "failed".

`subst` substitutes a given expression for a jet variable in another given expression.

`order` yields the order as differential equation of a function; it is computed as the order of the leading derivative obtained from `leadingDer`. The default implementation of the latter one in turn calls `jetVariables` and determines the maximum of the resulting list. Whereas `subst` operates purely algebraic, `dSubst` also substitutes for derivatives of the given jet variable. This is done by repeated calls of `subst`.

For partial derivatives one must distinguish between differentiations with respect to objects of the type `Symbol` and differentiations with respect to jet variables, i.e. objects of a domain in `JBC`. The former one is inherited from `PartialDifferentialRing Symbol`. It is of no importance for the formal analysis, as such symbols are considered as parameters. Only differentiations with respect to jet variables are used in formal derivatives and thus in prolongations and similar operations.

Jacobians play an important role in the higher structures discussed in the next section. To compute the formal derivative of a function according to (4) requires the knowledge of all its partial derivatives. There are two operations `jacobiMatrix` implemented. In one a second argument containing a list of the effectively occurring jet variables is required. This avoids a call of the procedure `jetVariables`, which can be quite expensive in some instances of `JBFC`.

Similarly, there are several different operations called `formalDiff`. Some are for a single function, some for a system; some take the Jacobian as an additional argument, some compute it. One of them returns a list of the jet variables occurring in the prolonged function to facilitate the determination of its Jacobian. Such "tricks" are fairly important to obtain reasonable run times.

There are two kinds of simplification routines: `simplify`, `simpMod` and `simpOne` use only algebraic operations; `reduceMod` and `autoReduce` apply also differentiations. They serve different purposes. The former ones are used to simplify local representations of differential equations. Hence they can produce an equation of lower order only, if integrability conditions occur. Indeed this is the main task of `simplify`. The latter ones are used to obtain simple generating sets for differential equations. They eliminate for example equations which are just prolongations of others. `JBFC` contains default implementations for all four mentioned simplification routines.

`simplify` has two main tasks. Besides the already mentioned exhibition of integrability conditions it should eliminate functionally dependent equations. This is important for the computation of dimensions (see below). `simpMod` is similar to `reduceMod` (see below) and simplifies a system with respect to another one. `simpOne` tries to simplify individual equations. This entails removal of unnecessary coefficients and signs or reduction of exponents in the case of non-linear equations.

The default implementation of `simplify` uses basically the leading derivatives of the equations. Obviously two equations with different leading derivatives are functionally independent. If several equations with the same leading derivatives occur, `simplify` checks first whether one of them can be solved for it using `solveFor`. If yes it performs the corresponding substitutions in the other equations. If not it tries to analyze further jet variables occurring in the equations. But obviously this will not always suffice to decide functional independence. In this case an error message is issued.

`reduceMod` reduces a system with respect to another one, i.e. it uses the equations of the second system and their prolongations to simplify the first one. `autoReduce` reduces a system with respect to itself. The default implementation depends basically on the effectivity of `solveFor`, as it tries to solve each equation for its leading derivative and to substitute it in the other equations.

As we will see in the next section the power and the efficiency of the simplification routines and here especially of `simplify` are crucial for the performance of our environment in almost any calculation. Thus special care should be applied to them in the implementation of any domain belonging to `JBFC`. It is e.g. very useful, if `simplify` returns the system in such a form that its symbol is already as close as possible to its solved form. This does not only make the handling of the symbol easier, there is a high probability that the simplification of the prolonged system will also be much easier.

There are two operations for determining dimensions: `dimension` takes as arguments a system (i.e. a list of functions) and the order q of the jet bundle of which the differential equation is a submanifold. It computes the dimension of this submanifold. `orderDim` takes the same arguments, but it assumes that all equations are of the same order as the jet bundle and calculates the dimension of the submanifold in $J_q\mathcal{E}/J_{q-1}\mathcal{E}$. In other words, if an empty list is passed as "system", then `dimension` returns $\dim J_q\mathcal{E}$ whereas `orderDim` returns $\dim S_qT^*X \otimes V\mathcal{E}$ (cf. Page 3).

The default implementations of both procedures assume that the system is simplified and contains only functionally independent equations. It then subtracts the number of equations of the appropriate dimension. The reason for the introduction of `orderDim` will become clear in the next section, when we discuss the `dimension` procedure of `DifferentialEquation`.

The category `BaseFunctionCategory` specifies no additional operations. It contains only different default implementations for the procedures generating jet variables and for the coercion of jet variables to `BFC`. They result in an error message, if one attempts to generate a dependent variable or a derivative.

Simplification is especially easy in the case of linear systems. The domain `JetBundleLinearFunction` represents linear functions as a list of coefficients and a list of jet variables. This explains why `JBC` introduces the special "jet variable" `1`: it is here necessary to allow for a constant term. `simplify` calculates a row echelon form for the system. Since the lists in the chosen representation are always sorted, this puts the symbol at once in solved form. The coefficient domain must belong to `BFC`. Currently the only possibility is `JBX`. One could of course easily implement a special domain for functions with constant coefficients, but this is hardly worth while.

Simplification becomes a much more difficult problem for general expressions as implemented in `JetBundleExpression`. The current implementation assumes a polynomial dependency of the jet variables. It was originally planned to provide a special domain for polynomial equations, but there have been very bizarre problems with AXIOM. For this reason `JBE` takes currently the place of such a domain.

Polynomial equations are probably the most general equations for which there exists a general algorithm to compute dimensions of submanifolds (or better varieties). In the jet bundle the differential equations become algebraic equations. They generate an ideal. Using a Gröbner basis [1] it is easily possible to determine the dimension of the variety defined by it. `dimension` applies the algorithm described by Kredel and Weispfenning [26].

This approach has, however, two serious disadvantages. Since we use the implementation of Gröbner bases provided in the AXIOM library, we cannot track the dependency of the resulting equations of the original equations. We will see in the next section that this is quite important for the domain `DifferentialEquation`. Furthermore the dimension of jet bundles and thus the number of unknowns occurring in the polynomials grows exponentially with the order. The same holds for the number of equations, if several prolongations are needed. But the complexity of Gröbner bases calculations explodes with the number of unknowns and the number of polynomials.

The implementation of `simplify` in `JBE` tries therefore to avoid the use of the AXIOM `groebner` procedure as much as possible. The main strategy rests on the observation that differential equations are usually sparse, i.e. not every jet variable occurs in every equation. A large part of the simplification can often be done by simply sorting the equations according to their leading derivatives. If there are several equations with the same leading derivative, `simplify` checks whether one of them can be solved for it. In this case this equation can be substituted in the other ones. Only if this is not possible, a Gröbner basis is computed — but only for the equations with the same leading derivative.

One could argue that this approach does not necessarily compute a Gröbner basis for the complete system and that thus the use of the algorithm of Kredel and Weispfenning is not correct. But this is not true. We obtain this way indeed a Gröbner basis but not a completely reduced one! Obviously we construct a basis with the same leading monomials as a reduced basis. By a well-known characterization theorem for Gröbner basis this implies that we get indeed such basis. In any case the algorithm of Kredel and Weispfenning needs only these leading monomials.

Solving an equation for a given jet variable provides of course also problems, if general expressions are allowed. Currently `solveFor` can handle only the case that the function depends linearly on this variable. But it may be non-linear in other variables. One could of course try to treat some more general cases, but we do not think that this is worth while the effort. For the purpose of simplification this strategy usually suffices, as prolongations lead to quasi-linear equations.

The domain `JetLazyFun` is rather special and still in an experimental state. But it shows the great possibilities offered by a system like AXIOM. The motivation for its development have been the following considerations. If we compare the calculations performed in our completion algorithm with other approaches to the construction of integrability conditions, it seems that we perform many unnecessary differentiations.

Most other approaches (take e.g. Reid's standard form [36]) prolong at first sight only those equations where one can suspect from the leading derivatives that cross-derivatives may generate integrability conditions. In our approach we must always prolong *all* equations in order to obtain all independent equations in the jet bundle of the next order. In principle, one knows that for the generation of integrability conditions only the prolongations with respect to non-multiplicative variables are of interest. The other prolongations yield automatically independent equations. Zharkov and Blinkov [62, 63] proposed a new approach to Gröbner bases (of algebraic equations!) based on the completion of an equivalent linear system of partial differential equations to an involutive one. Their algorithm tries to exploit this fact by considering only non-multiplicative prolongations.

A closer analysis reveals, however, that the difference between the two approaches is not as big as it seems. The generation of the potential integrability conditions alone does not suffice. One must also check whether they represent truly new independent equations. This requires simplification and hence reduction by other equations. But now one needs the prolongations with respect to the multiplicative variables, too. Algorithms like the one proposed by Zharkov and Blinkov or like the standard form algorithm of Reid compute these prolongations each time they are needed new. Thus there also occur redundancies. A priori it is not clear which approach will lead to less unnecessary differentiations.

One could argue that this is only a problem of implementation. One could store each computed prolongation for later reuse. But now the question arises how this can be organized efficiently. `JLF` could be viewed as a kind of synthesis of these two apparently so different approaches. It puts on top of another domain of `JBFC` a lazy evaluation scheme for partial differentiations. This means that the procedure `differentiate` does not really

compute a partial derivative but only stores a pointer to the function to be differentiated and the variable with respect to which it is differentiated. Only if later the full expression is needed, the differentiation is actually performed.

Such lazy evaluation schemes have been successfully applied to the implementation of infinite objects like series [7]. There the idea is to compute only as many terms as momentarily necessary, but to be able to calculate more terms if they are needed. In our case the idea is that the most expensive part of a prolongation, namely the many partial derivatives, is not really performed. As in the approach of Zharkov and Blinkov only the prolongations which are absolutely necessary to decide the existence of integrability conditions are computed.

Of course there is a price to be paid. The representation of `JLF` is a complicated three level hierarchy connected by pointers. To avoid as many evaluations as possible information about the leading derivatives and the occurring jet variables are stored. This introduces a considerable overhead. Thus the use of `JLF` is only effective, if one can really omit many differentiations. We will discuss this again in Section 10 based on examples.

The guiding principle in the implementation of `JLF` is to use the leading derivative of a function as much as possible. Testing for zero is for instance always a very important operation: `Ring` specifies a special operation `zero?` for this purpose. Many rings provide a special implementation of it which is more effective than the use of `=`. This is also the case in `JLF` where `zero?` uses the fact that the leading derivative of zero is the special jet variable `1`. Thus only if `leadingDer` yields this value an evaluation is necessary to see whether the function really vanishes.

This makes of course the effective implementation of `leadingDer` especially important. This is reflected in the choice of the representation. The lowest level represents a "lazy derivative". It consists of a record containing a function, a variable and a flag. If the value of the flag is `false`, then the differentiation has not yet been performed. The function is then the one to be differentiated. If the flag's value is `true`, the function represents already the result.

The next level represents a "lazy term", i.e. a coefficient and a pointer to a lazy derivative. In addition there are two flags and a jet variable. One flag indicates again whether or not the term has already been evaluated. In this case only the coefficient is relevant, as it contains the result. The jet variable provides an upper bound for the leading derivative of the term. The second flag indicates whether or not this bound is sharp, i.e. whether it is only a bound or really the leading derivative.

The top level record contains besides a list of pointers to lazy terms two flags, two lists of jet variables and a single jet variable. This single variable represents again an upper bound for the leading derivative of the total expression. One of the flags shows whether it is a sharp bound. The first of the two lists of jet variables represents all variables of which it is known that they occur effectively in the expression. The second one contains those which may occur. The list of lazy terms is considered as a sum. It is ordered by the leading derivatives of the terms. The second flag again indicates whether there are still unevaluated terms present. The representation of `JLF` is a pointer to such a record.

Pointers (in AXIOM realized with the domain `Reference`) are chosen in order to "communicate" the result of an evaluation. This is necessary, because arithmetic operations lead to a kind of proliferation of lazy derivatives or terms. If now such a lazy object is evaluated, the result should be available in all expressions containing it. This can be ensured through the use of pointers.

With this representation it is easy to implement `leadingDer`. It looks first whether the stored leading derivative represents only an upper bound. If not it is returned and no

evaluation was performed. Otherwise it starts using the procedure `eval1` to evaluate as many of the lazy terms as necessary to obtain a sharp bound.

We have already mentioned that `zero?` (and similarly `one?`) is based on `leadingDer`. But many procedures implemented categorically in `JBFC` use `zero?`, e.g. `jacobiMatrix` to avoid vanishing entries. This would lead to many unwanted evaluations. We have introduced the attribute `lazyRep` to distinguish domains like `JLF` with a lazy evaluation mechanism. In the case of such a domain `JBFC` tries to avoid procedures like `zero?` which might cause evaluation as much as possible. The same holds for the matrix domain `SEM` discussed in Section 9.

As explained above evaluation of a lazy derivative or a lazy term may effect several expressions. Thus an expression might have a more complicated representation than necessary, as some of its lazy terms are already evaluated. The procedure `collect` collects all such terms and leads hence to a simplification of the representation. It causes, however, no evaluation and changes of course not the value of the expression.

The implementation of `simplify` is very close to the default one in `JBFC`. As far as possible only the leading derivatives are used. But if several equations with the same leading derivatives occur, they must be evaluated either to try to solve them or to use the `simplify` procedure of the base domain. Many operations of `JBFC` cannot be implemented without evaluation. Hence they should be avoided as much as possible. They include `subst`, `solveFor` and all reduction procedure.

8 Implementation III — Higher Structures and Applications

This section describes some details of the implementation of the layers 3 and 4. The core of the third layer is of course the domain `DifferentialEquation`. As already explained in Section 6 its main purpose is to enhance the efficiency by storing some intermediate results. The two other domains `VectorField` and `Differential` could also be considered as utilities.

In principle we could use simply a list of functions of some domain in `JBFC` to represent a differential equation locally. The domain `DE` has, however, a more complicated representation. The most important point is that all equations are sorted by order. This allows to keep previous results, especially previous simplifications, almost unchanged by prolongations, as only integrability conditions can affect the lower orders. Such considerations play an important role for an efficient implementation of the completion algorithm outlined in Section 3.

The chosen data structure holds much more information than just the equations. The top level is a record of two lists: one contains the effectively occurring orders, the other one for each order a record. This record consists of a list of equations, namely all equations of the given order in the system, the Jacobian of these, and some flags indicating whether the equations of this order are already simplified (`Simp?`), whether they have already been prolonged (`Prolonged?`), or whether their contribution to the dimension of the differential equation (see below) has already been computed (`Dim?`).

Additionally the record contains an integer list (`Deriv`) to avoid the multiple generation of the same equation during prolongations. If an equation was obtained by prolonging another one, this list contains the number of the independent variable with respect to which that equation was prolonged. In the next prolongation the equation will be prolonged only with respect to independent variables with an index greater than or equal to this number.

Because of this list we try to keep track of the equations during simplification. If an equation is a combination of several other equations, then its value in `Deriv` is determined by the minimum of the values of the other equations. This strategy may not be optimal, but it is the best one which can be realized with reasonable effort.

The central operations in `DE` are `prolong`, `project`, `dimension` and `simplify` plus the ones for symbol and tableau. `simplify` is of course based on the corresponding procedure in `JBFC` and assumes that it finds all integrability conditions. It proceeds order by order starting with the equations of highest order. If at some order an integrability condition is detected, it is at once moved to the subsystem of the corresponding order. If algebraic equations occur, `simplify` checks whether they pose conditions on the independent equations. In that case an error message `"inconsistent system"` is issued. As soon as a subsystem has been simplified, the flag `Simp?` is set. If `simplify` is called again, it recognizes this flag and avoids a second, unnecessary simplification.

In the implementation of `prolong` one must take care of both mechanisms to generate integrability conditions: the classical one via elimination of the derivatives of highest order and the prolongation of equations of lower order. The first one is basically the task of `simplify` applied to the equations obtained by formally differentiating the highest order equations. To avoid unnecessary differentiations of lower order equations (and the subsequent simplification!) the list `Prolonged?` stores for each equation a flag whether or not its prolongation has already been computed.

There exists also a second version of `prolong` requiring two arguments, the differential equation and an order. Only for the equations up to this order it is checked whether their prolongations have already been computed. This version is useful in the completion procedure described in the next section. It allows for the fast prolongation of a system obtained by a projection. In this case only the formal derivatives of lower order equations, namely of the integrability conditions determined during the projection, are needed.

`project` can be implemented very easily, if one assumes that it is applied only to prolonged equations which have already been simplified. This assumption is reasonable within the current context of completion to involution. For other tasks it might be useful to have a more general procedure available. It is, however, not obvious, how it could be realized for arbitrary classes of equations, as it requires then highly non-trivial, non-linear computations. With this assumption `project` amounts to nothing more than eliminating all records belonging to equations of higher order than the target order.

The determination of the dimension of a differential equation is a crucial operation in the completion algorithm. `DE` takes the following approach: Using the procedure `orderDim` of `JBFC` the contribution of each order is computed separately. It is then stored in the record containing the equations of this order and the flag `Dim?` is set. As long as no integrability conditions of this specific order are appearing this result can be reused, if e.g. the dimension of the prolonged equation must be computed. In the optimal case this requires only the determination of the contribution of the new equations of highest order.

`DE` specifies four operations for symbols: `extractSymbol` extracts it from the Jacobian of the highest order part of the system. `analyseSymbol` returns a record with the values of the $\beta_q^{(k)}$, the total number of multiplicative variables and the rank of the symbol. `prolongSymbol` computes directly the prolonged symbol instead of extracting it from the prolonged differential equation. `prolongMV` takes as argument a record as computed by `analyseSymbol` and returns the corresponding record for the prolonged symbol. The values are, however, only correct, if the symbol is involutive, as this procedure is based on (6).

Additionally there are two procedures `tableau` to compute the matrices of tableaux. One takes only one one-form as argument. The other one expects a list of one-forms and

can thus be used to compute k-tableaux. To enter one-forms the domain `Differential` must be used. It represents together with the domain `VectorField` the remainder of the third layer. The implementation of both is somewhat rudimentary, as we hope that some day AXIOM will contain a reasonable environment for differential geometric calculations and then it should be used instead of some special domains like the two mentioned.

Both use an identical representation consisting of two lists. One list contains the coefficients, the other one the basis elements. The domains for both are passed as arguments. The one for the basis elements must be a jet bundle from `JBC`, the other one a domain belonging to `JBFC`. The lists are always sorted using the lesser function of `JBC`.

Besides some elementary procedures to enter one-forms and vector fields, respectively, or to access the coefficients and basis elements we have implemented the usual arithmetic operations and a few geometric operations like commutators or the contraction of a one-form with a vector field. For vector fields we have additionally implemented a procedure `prolong` which prolongs a field given on the base bundle $\mathcal{E}$ to one acting on a jet bundle of given order using (17). Such a procedure is necessary for symmetry analysis.

The fourth layer contains currently the packages `CartanKuranishi` and `Symmetry-Analysis`. `CK`'s central procedure is called `complete` and completes a given differential equation to an involutive one. Whereas `complete` only displays some results (depending on the previous setting of the output flags using `setOutMode`) and returns nothing (or `void` in the AXIOM language), its second version `complete2` returns the involutive equation, the Cartan characters and a few other information. The user can set two further flags controlling reduction and simplification with `setRedMode` and `setSimpMode`. They are discussed in more detail in Section 10.

`complete` implements basically the detailed algorithm depicted in Fig. 1 on page 6. To achieve a reasonable performance it takes full advantage of the possibilities offered by the implementation of `DE`. Basically all the computations happen in two local variables: `PrevDe` contains the current differential equation which is checked for involution; `CurDe` represents its prolongation which is of course needed for the check.

If the inner loop of our algorithm requires a prolongation, `PrevDe` gets the previous value of `CurDe` and `CurDe` is newly calculated using `prolong`. Then the necessary information about the symbols are extracted. If the symbol of `PrevDe` is involutive, involution of the differential equation is decided by comparing dimensions using (8).

If integrability conditions occur, `PrevDe` is newly computed from `CurDe` using `project`. As explained above `prolong` puts such conditions directly in the correct place in the representation of `DE`. Hence the simple implementation of `project` suffices here. The new value of `CurDe` could now be computed from `PrevDe`, but this would be inefficient, as `CurDe` contains already all needed equations with the exception of the prolongations of the integrability conditions. Here `complete` makes use of the second form of `prolong` which computes only these equations.

Besides `complete` `CK` contains several procedures for calculations with the Cartan characters and the Hilbert polynomial. Especially it allows the user to compute the $\alpha_q^{(k)}$ from the $\beta_q^{(k)}$ and to calculate $H_q(r)$ from the $\alpha_q^{(k)}$ and vice versa. There is a procedure to evaluate the recursion relation (13) for a given differentiation order. Furthermore one can compute gauge corrections.

The implementation of these operations is straightforward. It only demands a procedure `stirling` to compute the modified Stirling numbers. They are determined using the recursion relation (12). To avoid repeated calculations of the same values, a hash table is set up storing all computed numbers.

`SYMANA` is still fairly primitive and in an experimental stage. It provides essentially

one procedure `detSys` to set up the determining system for symmetry generators. There exist different mode maps for this procedure. One can e.g. provide a special ansatz for the symmetry generator or a list of derivatives for which the equations of the system can be solved. The default is the most general ansatz and each equation will be solved for its leading derivative.

If the general ansatz is chosen, `detSys` returns the determining system in another data type, namely as functions over a new jet bundle with the coefficients as dependent variables. In this form the output can directly be converted into a `DifferentialEquation` and further analyzed. An example for this is given in Section 11. If one is searching for classical symmetries, it is of advantage to retract the output to a linear system using `linearize`.

In its current form `SYMANA` will have problems to compute the determining equations for a large system. Like all currently available symmetry packages it substitutes the equations of the original system solved for given derivatives in the conditions for the symmetry generators. Then it extracts in one loop the coefficients of all monomials. This approach is very inefficient and yields often many redundant equations. A later version will use a more sophisticated approach setting up the equations step by step and simplifying them at once.

Another project for the next version is to get rid of the restriction to equations which can be solved for some derivatives. If a Gröbner basis of the original system is given, then one can compute a normal form of all conditions and obtain this way independent of any solving the determining system. Note, that we use here standard algebraic Gröbner bases and not differential ones. This suffices, as we work in a jet bundle of fixed order.

`SYMANA` contains also a procedure `ncDetSys` to obtain the determining system for conditional symmetries as they are used in the non-classical method. It simply adds the invariant surface condition corresponding to the given ansatz for the symmetry generator to the original system and then calls `detSys`.

9 Implementation IV — Utilities

Since it is one of the basic ideas underlying the formal approach to differential equations to use linear algebra instead of differential operations, it is obvious that an efficient treatment of linear systems of equations is important for the performance especially of the `complete` procedure in `CK`. The determination of the $\beta_q^{(k)}$ requires to transform the symbol in row echelon form; some domains in `JBFC` may use the Jacobian in `simplify` or `dimension`, too.

Earlier versions of our environment used the standard `Matrix` domain of AXIOM. However, the matrices typically occurring in our applications have some special properties which can be better exploited by implementing a new domain. The most important one is that they are sparse, often even very sparse. This reflects simply the already mentioned fact that e.g. in a second-order equation usually not all jet variables up to order two are effectively occurring. Often even of the second order derivatives just one or two are present. Furthermore, all prolonged symbols have exactly the same percentage of non-vanishing entries, i.e. only integrability conditions can change the sparsity.

In a recent diploma thesis Berchtold [2] compared different algorithms and data structures to compute row echelon forms and determinants of sparse matrices. He showed that for integer matrices it is of advantage to use a special "sparse" data structure like lists or polynomials for the rows of the matrix. His conclusion that this would not be the case

for matrices with polynomial entries is, however, not correct and essentially due to his incorrect implementation.

Most of the matrices studied by Berchtold were still fairly dense compared with the matrices typically appearing as symbols. For such matrices Gaussian elimination amounts essentially to sorting the rows according to the position of the pivot, as the probability that two rows have their pivots in the same column is rather small.

Another result of Berchtold's thesis was that factor-free elimination methods like e.g. the Bareiss algorithm [15] are fairly inefficient. They tend to enlarge the entries considerably. In the case of a square matrix the last entry on the diagonal will contain in the end the complete determinant of the matrix! In our applications this can thoroughly effect the overall performance. Assume the `simplify` procedure of a domain in `JBFC` is based on such an elimination method. If later the resulting equations must be prolonged, these complicated expressions must be partially differentiated several times.

On the other hand divisions like they are used in the standard Gauss algorithm lead to a similar effect, as differentiation of fractions is quite expensive. It is therefore better to stay somewhat in the middle between these two extrema. The so-called primitive factor-free elimination divides each row by its greatest common divisor. This approach avoids collecting factors as in the Bareiss method but still does not generate fractions.

The greatest common divisor of a row can usually be computed fairly efficient, as experience shows that very often already the first two non-vanishing entries determine the final result. Actually there are two possible strategies: The classical one is compute the greatest common divisor iteratively stopping as soon as 1 is reached. Zippel [64] recommends a probabilistic strategy building random linear combinations of the elements and computing their greatest common divisor. With probability 1 this yields the correct result. But of course one must nevertheless check that one has not obtain a too large result. Which strategy is ultimatively the best is difficult to judge.

These considerations lead to the implementation of the domain `SparseEchelonMatrix`. Its internal data structure is especially designed for the fast computation of row echelon forms. On the other hand hardly any arithmetic operations like addition or multiplication are implemented, as these tend to destroy the sparsity and are furthermore rather inefficient in this special representation.

`SEM` takes two arguments: a domain belonging to `Ring` for the entries and a domain belonging to `OrderedSet` to label the columns. A row is now represented by a record containing two lists: one for the non-vanishing entries, one for their labels. The latter one is always sorted. A matrix consists essentially of a vector of such records.

The central operations of `SEM` are `rowEchelon` and `primitiveRowEchelon`. Both return not only the matrix in row echelon form but also the corresponding transformation matrix, the used pivots and the rank of the matrix. `primitiveRowEchelon` is only available, if the domain of the entries belongs to the category `GcdDomain`. Furthermore, the transformation matrix contains then fractions from the divisions by the greatest common divisors. `SEM` offers both above mentioned strategies for the determination of the greatest common divisor of a row. The user can choose it with the function `setGcdMode`.

The implementation of both operations is fairly different from the usual elimination methods. Instead of searching for a pivot in each column, we start by sorting all rows according to their pivots. After each elimination step changed rows are at once moved to their new places, so that the rows remain sorted all the time. The sorting is done using bubble sort. Since we can expect that our matrices are often already close to a row echelon form, this is probably the best choice. Furthermore it allows to keep easily track of the rows which is necessary for the construction of the transformation matrix.

The package `JetCoordinateTransformation` provides two procedures `transform` to prolong coordinate transformations of the base bundle $\mathcal{E}$ into higher order jet bundles. Its parameters are two jet bundles and two vectors (one for the independent and one for the dependent variables) containing expressions for the old coordinates in terms of the new ones. One procedure computes the transformation law for an old jet coordinate; the other one transforms an expression in the old coordinates using the first one and `subst`.

In the current implementation all expressions must be of the type `JBE`. The main reason for this restriction is that this represents by far the most flexible domain, as its representation is based on the domain `Expression Integer`. In it arbitrary transformation laws can be specified. Furthermore for most domains in `JBFC` it is possible to write a `retract` or `coerce` procedure to convert its objects into objects of `JBE` and vice versa.

The implementation of `transform` is straightforward. It makes use of the chain rule. Ref. [5] gives a simple recursive formula for the transformation law of a derivative of order q given the one for a derivative of one order less. It requires the inversion of the Jacobian of the transformation of the independent variables. This inverse matrix is computed using an LU decomposition. To enhance the efficiency transformed jet variables are stored in a hash table similar to `stirling` in `CK`.

We have implemented for this purpose a package `LUDecomposition` containing three procedures: `LUDecomp` computes a LU decomposition using the algorithm of Crout [34]; `LUSolve` uses such a decomposition to solve a linear system with given right hand side; `LUInverse` determines the inverse for a matrix with given LU decomposition.

10 Examples I — Completion

This and the next section demonstrate the application of the above described environment to concrete examples. The mentioned running times refer to the AXIOM implementation on an IBM RS 6000 Model 530 with 64MB memory.

To start we treat a classical example due to Janet [33]

$$\mathcal{R}_2 : \begin{cases} u_{zz} + y\, u_{xx} = 0\,, \\ u_{yy} = 0\,. \end{cases} \tag{22}$$

The upper part of Fig. 4 on page 32 shows the necessary input. As first step the different needed domains and packages are constructed and assigned to variables to facilitate the input. We use a jet bundle (`jb`) with 3 independent variables named `x` and one dependent variable named `u`. Derivatives are denoted by `p`. For the equations we use linear functions (`jbl`) whose coefficients can be arbitrary expressions in the independent variables (`jbx`). `de` is the data type used to represent the differential equation; `ck` denotes the package with the completion procedure.

The equation is set up in two steps: First the two equations (`eq1,eq2`) of (22) are defined. Jet variables are generated with the procedures `X, U, P`. Then a differential equation (`janet`) is generated from this system. The `setOutMode` command controls the amount of output generated and whether or not it should be in TEX. The `setRedMode` command tells `complete` to reduce the found integrability conditions as much as possible. Finally the procedure `complete` from `ck` is called. In the last two lines we have helped the interpreter to locate the used procedures by explicitly stating that they are from the package `ck`.

The output of `complete` is also contained in Fig. 4. With the chosen setting of the output flags we get a trace of the dimensions of all intermediate systems and symbols

and of all found integrability conditions. The output is directly in TEX. The computation follows exactly the steps of the treatment in Ref. [33]. It is, however, a non-trivial fact that the program produces the integrability conditions in the simple form shown in Fig. 4. Actually many more integrability conditions are produced, for there occur quite a lot of non-multiplicative variables in the higher prolongations. They are, however, not independent. `complete` must invoke several simplification procedures to avoid redundant equations and unnecessary coefficients. In this example it succeeds in reaching the simplest form, because we deal with a linear system. For non-linear equations this is not necessarily the case.

The running time for this example is about 18 sec. It rises to 28 sec, if one takes `JBE` as data type for the equations instead of `JBLF`. There are two main reasons for this striking difference: differentiations are much faster in `JBLF` and simplification of linear systems can be performed more efficient. Although in the end the `simplify` procedure of `JBE` produces the same result as the one in `JBLF`, the latter one finds the necessary steps easier and thus faster.

Due to the many prolongations necessary in this example it also demonstrates very nicely the advantages of our special domain for sparse matrices. Compared with an earlier version of our environment which used the standard `Matrix` domain of AXIOM the mentioned running time improved by more than a factor two. It is easy to see that system (22) leads indeed to very sparse matrices for the symbols. Furthermore the procedure `simplify` in `JBLF` computes already a row echelon form of the whole system and thus of course also of the symbol. Hence the call of `rowEchelon` in `analyseSymbol` is in principle redundant. But because of the special way the elimination algorithm is implemented this takes hardly any computing time.

One could think that this example should be an ideal test case for the lazy differentiation scheme implemented in `JLF`. However, the result is disappointing. Instead of accelerating the computation the running time rises to almost 1 min. A closer look solves the mystery. We encounter three times a non-involutive symbol. Each time two prolongations are necessary. Tracing the intermediate systems shows that after the second prolongation all lazy differentiations used in the previous one have been fully evaluated.

The reason is that there occur many non-multiplicative variables in the equations of higher order and deciding whether or not the corresponding prolongations generated integrability conditions requires the evaluation of most of the equations. Thus in this example it is indeed necessary to compute almost all prolongations. Another aspect of this at first sight perhaps surprising performance is that the basic assumption for the development of `JLF`, namely that partial differentiations are expensive, is not correct for linear equations. With the chosen representation of `JBLF` the differentiation with respect to a dependent variable or a derivative requires only the extraction of an element of a list. Since the lists are not too long, this is very fast.

As an example of a non-linear system we treat the Euler Equation of hydrodynamics in a slightly unusual way to obtain an over-determined system.

$$\mathcal{R}_1 : \begin{cases} \partial_t u^i + \sum_{j=1}^{D} u^j \partial_{x^j} u^i = 0 \,, \qquad i = 1, \ldots, D \,, \\ \sum_{j=1}^{D} \partial_{x^j} u^j = 0 \,, \end{cases} \tag{23}$$

where D denotes the space dimension. In our computer calculation we take the usual value of $D = 3$ and write x^4 instead of t.

In- and output are shown in Fig. 5 on page 33. The input is very similar to the previous example. The main difference is that we use another domain for the equations, namely `JBE`. They can now be arbitrary expressions in the independent and dependent variables. The integrability condition can be obtained by taking the divergence of the first D equations and subtracting the t-derivative of the last equation

$$\sum_{i,j=1}^{D} \partial_{x^i} u^j \partial_{x^j} u^i = 0 \,. \tag{24}$$

Due to the simplification procedures this equation and some of the original equations appear different in the output. As explained in Section 7 the `simplify` procedure in `JBE` tries to solve equations for their leading derivatives and to substitute them into the other ones. If this strategy is successful, it leads to kind of normal form of the system which is better adapted for the analysis of the symbol.

The time used for simplification also dominates the running time. Although this example is much smaller than the Janet example in terms of the number of prolongations needed, it takes much longer, namely about 8 min. This time is really surprising, as the simplification does not require the computation of Gröbner bases. Nevertheless, it is frequently necessary to solve equations for their leading derivatives and to substitute the result in other equations. This process is quite time-consuming in `JBE`.

We have used our basic domains for jet bundles in these two examples: in the Janet example `JB`, for the Euler Equation `IJB`. One could see that the input of the equation was slightly more complicated for `JB`, because the interpreter needs some help in order to identify the types correctly. One can of course also use the syntax applied for the Euler Equation in `JB`. But this requires that one knows the used ranking of the variables. The purpose of `JB` is, however, to stay as close as possible to the natural notation.

The two examples differ also in the setting of the reduction and simplification control flags with `setRedMode` and `setSimpMode`. The choice of a good setting requires in some sense to know already a priori what will happen during the completion. In principle one would prefer to avoid the setting of both flags, as they only need computation time.

`setRedMode(1)` affects basically only the output. Without it many spurious integrability conditions and/or equations in the generating system displayed at the end may appear. They are either algebraically dependent of each other or derivatives of some other equations. Reduction tries to detect this. In examples like the Janet example where many prolongations are needed and integrability conditions appear at different orders, this is usually useful. In the case of the Euler Equation one would, however, gain nothing.

Deciding whether `setSimpMode(1)` helps or not is more difficult. As explained in Section 8 the individual equations of a system are stored order by order. With the default setting simplification only occurs within the different orders. Equations of lower order are not used. This may lead to more complicated expressions, because some simplifications are overlooked. For instance it might happen in the Janet example that some equations of higher order contain u_{yy}, although it is zero. But it turns out that in this example it is more expensive to search for such simplifications than to continue with the unsimplified expressions.

The situation is often different for non-linear systems. Here it is much more important to avoid complicated expressions, as otherwise the standard simplification within one order needs much more time. In the case of the Euler Equation one gains almost a factor of two through simplifying modulo the equations of lower order.

To stress this point we will briefly discuss a non-linear modification of the Janet ex-

ample. Consider the system

$$\mathcal{R}_2 \;:\; \left\{ \begin{array}{lll} u_{zz} + v\, u_{xx} & = & 0\,, \\ u_{yy} & = & 0\,, \\ v_z & = & 0\,, \\ v_y - v_x & = & 0\,. \end{array} \right. \tag{25}$$

The completion runs completely analogously to the classical Janet example, i.e. the projections occur at the same places and $\mathcal{R}_5^{(2)}$ is involutive. The only difference is that in the second projection two further integrability conditions[3] occur, thus there are three:

$$\begin{gathered} 2v_x u_{yxx} + v_{xx} u_{xx} = 0\,, \\ 4v_x^4 u_{xxxx} + v_{xx}(4v_x^2 - v v_{xx})(2v_x u_{xxx} + v_{xx} u_{xx}) = 0\,, \\ 2v_x v_{xxx} - 3v_{xx}^2 = 0\,. \end{gathered} \tag{26}$$

It is obvious that here the simplification routines have hard work to do. Indeed the running time for this example is slightly more than one hour. It was not possible to run it without simplification modulo lower order equations. We stopped the computation after about 10 hours!

We showed in Ref. [51] that the Dirac algorithm in the theory of dynamicla systems with constraints [13, 55] does nothing else than to complete the equations of motion to involution. Thus we can also use our environment for constrained dynamics. Tombal and Moussiaux [57] presented a MACSYMA program for the Dirac algorithm. It can, however, handle only Lagrangians which are quadratic in the velocities. Furthermore they reported that their program was too slow to handle complicated systems like the Maxwell Equations. Our environment is not only capable of treating much more general systems, it proves the involution of the Maxwell Equation in a four-dimensional space-time in about 15sec.

As a simple example we consider the motion of a particle in three dimensions whose motion is confined to the 2-sphere of radius 1. Setting all masses to 1 we have to analyze the following equations of motion

$$\mathcal{R}_2 \;:\; \left\{ \begin{array}{l} \ddot{x} - 2\lambda x = 0\,, \\ \ddot{y} - 2\lambda y = 0\,, \\ \ddot{z} - 2\lambda z = 0\,, \\ x^2 + y^2 + z^2 - 1 = 0\,. \end{array} \right. \tag{27}$$

The output of a corresponding AXIOM session is shown in Fig. 6 on page 34. λ is denoted by L. Since we are dealing with ordinary differential equations, there is no need to check for involution of the symbol. The running time is slightly over 30sec with simplification as well as reduction flag set.

It might surprise that such a small example consisting only of ordinary differential equations should take twice as long as the Maxwell Equations. Especially if one takes into account that no symbols must be analyzed. But the reason is simply that the latter ones are linear. If we omit setting the simplification flag, i.e. `setSimpMode(0)`, the running time nearly doubles.

A closer analysis of the obtained integrability conditions and the displayed final system reveals some deficiencies in the treatment of non-linear equations. We will discuss them

[3] We count here only the non-trivial conditions. Of course there appear in addition the prolongations of the two first-order equations.

later in more detail. Here we just want to point out that in several equations the constraint was not used for simplification; in the third projection we even obtain two identical equations. Both effects are mainly due to current restrictions in the `solveFor` procedure of `JBE`.

11 Examples II — Other Applications

One of the classical examples in symmetry analysis is the Heat Equation. We will use `SYMANA` to calculate its determining system for classical Lie symmetries. We omit the further discussion of the symmetries, as it can be found in many places. The output of a corresponding AXIOM session is contained in Fig. 7 on page 35.

The first command just shows the ansatz made for the symmetry generators. If `detSys` receives only one argument, it takes automatically this most general ansatz, where every coefficients depends on all variables, i.e. in this example on x, t, u. The type of the output of `detSys` depends on the input. If one uses the general ansatz as in Fig. 7, one obtains a list of functions of type `JBE` over a new jet bundle in which the coefficients of the ansatz are the dependent variables. Thus the result can be used directly to generate a differential equation and then be further analyzed.

Since we are only looking for classical symmetries, we know that the determining system must be linear. Hence before generating a differential equation the individual equations are retracted to a new type, namely `JBLF`. Then the completion procedure is started. It turns out that we must proceed to third order and that four projections are necessary. The resulting equation can easily be solved explicitly (see e.g. Refs. [5, 30]).

Here we are only interested in calculating the size of the symmetry algebra. From the displayed Cartan characters we see that the general solution of the determining system contains two functions of one argument. But this is due to the linearity of the equation: solutions can be superposed. Indeed, we see that one of the equations in the determining system is just the Heat Equation itself.

In general it is difficult to deduce from the knowledge of the Cartan characters and of the dimension of the differential equation the number of arbitrary constants in the general solution, as one must take the lower order part of the arbitrary functions into account. These are usually not independent and thus difficult to count. The only way to do it is to construct a formally well-posed initial value problem. Here we can proceed simpler. The Heat Equation yields a seven-dimensional submanifold in a third-order jet bundle with two independent and one dependent variables. Hence its general solution contains seven arbitrary coefficients of order less than or equal to 3. Since the determining system is 13-dimensional, there remain 6 arbitrary constants for the finite-dimensional part of the symmetry algebra. Thus we find besides the superposition 6 linearly independent symmetry generators.

Setting up the determining system needs about 12 sec. The completion is, however, fairly slow and takes a bit longer than 90 sec. This is mainly due to the prolongation to third order. One should mention that the Heat Equation is in some respect a "lucky" example. Fairly often determining systems turn out to be in not δ-regular coordinate system. Then one cannot simply call `complete` but must start working with the k-tableaux.

Finally, we consider an example of a coordinate transformation. It is well-known that the Burgers Equation is related to the Heat Equation via the Cole-Hopf transformation. We will perform this transformation using the AXIOM package `JCT` starting with the Burgers Equation in potential form. The result can be seen in Fig. 8 on page 36.

The timing of the first transformation also contains the time needed for the precomputation of an inverse Jacobian. Since its result is stored, subsequent transformations will be faster. In this example this effect is neclectable, as the independent variables remain unchanged and the mentioned Jacobian measures their dependence of the old independent variables. `JCT` further keeps as hash table of already treated jet variables. Hence the second call with the same argument is much faster.

```
jb:=JB(['x,'y,'z],['u])
jbx:=JBX jb
jbl:=JBLF(jb,jbx)
de:=DE(jb,jbl)
ck:=CK(jb,jbl)

eq1 := D('u,['z,'z])$jb::jbl + 'y::jb::jbl * D('u,['x,'x])$jb::jbl
eq2 := D('u,['y,'y])$jb::jbl
janet:de := generate [eq1,eq2]

setOutMode(14)$ck
setRedMode(1)$ck
complete janet
```

Symbol M_2 not involutive! Dimension: 4

Symbol M_3 involutive! Dimension: 4

Equation R_3 not involutive! Dimension: 12

======= 1. Projection =======

Integrability condition(s)

$$u_{y,\,x,\,x} = 0$$

==========

Symbol $M_3^{(1)}$ not involutive! Dimension: 3

Symbol $M_4^{(1)}$ involutive! Dimension: 2

Equation $R_4^{(1)}$ not involutive! Dimension: 13

======= 2. Projection =======

Integrability condition(s)

$$u_{x,\,x,\,x,\,x} = 0$$

==========

Symbol $M_4^{(2)}$ not involutive! Dimension: 1

Symbol $M_5^{(2)}$ involutive! Dimension: 0

********** Final Result **********

Equation $R_5^{(2)}$ involutive!

System without prolonged equations. Dimension: 12

$$u_{x,\,x,\,x,\,x} = 0,$$
$$u_{y,\,x,\,x} = 0,$$
$$u_{z,\,z} + y\ u_{x,\,x} = 0,$$
$$u_{y,\,y} = 0$$

System of finite type.

Figure 4: In- and output for the Janet example (22).

```
jb:=IJB('x,'u,'p,4,3)
jbe:=JBE jb
de:=DE(jb,jbe)
ck:=CK(jb,jbe)

eq1:jbe := P(1,[4]) + U(1)*P(1,[1]) + U(2)*P(1,[2]) + U(3)*P(1,[3])
eq2:jbe := P(2,[4]) + U(1)*P(2,[1]) + U(2)*P(2,[2]) + U(3)*P(2,[3])
eq3:jbe := P(3,[4]) + U(1)*P(3,[1]) + U(2)*P(3,[2]) + U(3)*P(3,[3])
eq4:jbe := P(1,[1]) + P(2,[2]) + P(3,[3])
euler:de := generate [eq1,eq2,eq3,eq4]

setOutMode(14)$ck
setSimpMode(1)$ck
complete euler
```

Symbol M_1 involutive! Dimension: 8

Equation R_1 not involutive! Dimension: 11

======= 1. Projection =======

Integrability condition(s)

$$2\,p_2^3\,p_3^2 + 2\,p_1^3\,p_3^1 + 2\,{p_2^2}^2 + 2\,p_1^1\,p_2^2 + 2\,p_1^2\,p_2^1 + 2\,{p_1^1}^2 = 0$$

==========

Symbol $M_1^{(1)}$ involutive! Dimension: 7

********** Final Result **********

Equation $R_1^{(1)}$ involutive!

System without prolonged equations. Dimension: 14

$$p_4^3 + u^2\,p_2^3 - u^3\,p_2^2 + u^1\,p_1^3 - u^3\,p_1^1 = 0,$$
$$p_4^2 - u^3\,p_1^3\,p_3^1 + \left(u^2\,p_2^2 + u^1\,p_1^2\right)p_2^3 - u^3\,{p_2^2}^2 - u^3\,p_1^1\,p_2^2 - u^3\,p_1^2\,p_2^1 - u^3\,{p_1^1}^2 = 0,$$
$$p_4^1 + u^3\,p_3^1 + u^2\,p_2^1 + u^1\,p_1^1 = 0,$$
$$p_3^3 + p_2^2 + p_1^1 = 0,$$
$$p_2^3\,p_3^2 + p_1^3\,p_3^1 + {p_2^2}^2 + p_1^1\,p_2^2 + p_1^2\,p_2^1 + {p_1^1}^2 = 0$$

Cartan characters: 3, 3, 1, 0

Figure 5: In- and output for the Euler Equation (23).

Equation R_2 not involutive! Dimension: 8

======= 1. Projection =======

Integrability condition(s)

$$2\ z\ z_t + 2\ y\ y_t + 2\ x\ x_t = 0$$

==========

Equation $R_2^{(1)}$ not involutive! Dimension: 7

======= 2. Projection =======

Integrability condition(s)

$$\left(2\ z^2 + 2\ y^2\right)\ {y_t}^2 + 4\ x\ y\ x_t\ y_t + \left(2\ z^2 + 2\ x^2\right)\ {x_t}^2 + 4\ L\ z^4 + \left(4\ L\ y^2 + 4\ L\ x^2\right)\ z^2 = 0$$

==========

Equation $R_2^{(2)}$ not involutive! Dimension: 6

======= 3. Projection =======

Integrability condition(s)

$$4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0,$$
$$4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0$$

==========

Equation $R_2^{(3)}$ not involutive! Dimension: 5

======= 4. Projection =======

Integrability condition(s)

$$\left(4\ z^2 + 4\ y^2 + 4\ x^2\right)\ L_{t,\,t} = 0$$

==========

********** Final Result **********

Equation $R_2^{(4)}$ involutive!

System without prolonged equations. Dimension: 4

$$y_{t,\,t} - 2\ L\ y = 0,$$
$$x_{t,\,t} - 2\ L\ x = 0,$$
$$\left(4\ z^2 + 4\ y^2 + 4\ x^2\right)\ L_{t,\,t} = 0,$$
$$z\ z_t + y\ y_t + x\ x_t = 0,$$
$$\left(2\ z^2 + 2\ y^2\right)\ {y_t}^2 + 4\ x\ y\ x_t\ y_t + \left(2\ z^2 + 2\ x^2\right)\ {x_t}^2 + 4\ L\ z^4 + \left(4\ L\ y^2 + 4\ L\ x^2\right)\ z^2 = 0,$$
$$\left(-z^2 - y^2\right)\ {y_t}^2 - 2\ x\ y\ x_t\ y_t + \left(-z^2 - x^2\right)\ {x_t}^2 - 2\ L\ z^4 + \left(-2\ L\ y^2 - 2\ L\ x^2\right)\ z^2 = 0,$$
$$4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0,$$
$$4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0,$$
$$z^2 + y^2 + x^2 - 1 = 0$$

System of finite type.

Figure 6: Output for the constrained system (27).

```
v:vf := ansatz()$sym

   tau D  + xi D  + eta D
        t       x        u

ds := detSys([eq])$sym

   [- 2tau , - 2tau , - xi   , - tau   , eta    - 2xi   , - 2tau    - 2xi ,
         u        x      u,u       u,u     u,u      u,x        u,x      u
    2eta    - xi    + xi , - tau    - 2xi  + tau , eta    - eta ]
        u,x     x,x     t       x,x      x      t     x,x      t

lds:List jbl2 := [retract(eq)  for eq in ds]
r2:de := generate lds
setOutMode(4)$ck
setRedMode(1)$ck
complete r2

   *************** Final Result ***************

             (4)
   Equation R     involutive!
             3
   System without prolonged equations. Dimension: 13

       tau     = 0
          t,t,t
        eta   = 0
           u,u
             1
    eta    + - xi = 0
       u,x   2   t
     eta    - eta = 0
        x,x      t
            1
   eta    + - tau   = 0
      u,t   4    t,t
         xi   = 0
           t,t
          xi = 0
            u
         tau = 0
            u
           1
     xi  - - tau = 0
       x   2    t
         tau = 0
            x

   Cartan characters: 2,0,0
```

Figure 7: Determining system for the Heat Equation.

```
burgers:jbe1 := P [2] - P [1,1] - P([1])**2

                          2
   (8)  - u    + u  - u
           x,x    t    x
                              Time: 0.33 (IN) + 0.23 (EV) + 0.80 (OT) = 1.36 sec
transform(burgers)$jct

         - v    + v
            y,y    s
   (9)  -----------
              v
                              Time: 0.22 (IN) + 2.02 (EV) + 1.54 (OT) = 3.78 sec
transform(burgers)$jct

          - v    + v
             y,y    s
   (10)  -----------
               v
                              Time: 0.01 (IN) + 0.35 (EV) + 0.05 (OT) = 0.41 sec
```

Figure 8: Cole-Hopf transformation of the Burgers Equation.

12 Outlook and Discussion

It should be clear from the discussion so far, that this environment is by far not yet finished. There are permanent changes and improvements. Most changes are needed in the domains for non-linear equations. The simplification and reduction procedures implemented in `JBE` are still too inefficient for more complicated equations. This is partly due to its current role as a substitute for a special domain for polynomial equations. Although the procedures are mainly designed for the application to polynomial equations, there had to be made concessions for more general expressions.

The most important point is reduction. In a polynomial domain one could make full use of the auto reduction algorithms proposed already by Ritt [39]. Currently `JBE` inherits the default implementation of `autoReduce` in `JBFC` which tries to solve equations for their leading derivatives and to substitute the results in the other equations. This can be very time consuming for complicated equations. Similar remarks can be made for simplification.

This effect is clearly visible in the example of the Euler Equation. Although the completion algorithm terminates after just one iteration, the computing time is much higher than in the Janet examples with its many prolongations and projections. This is especially remarkable, as a closer trace reveals that no Gröbner basis had to be calculated. If one looks at the second equation of the final result in Fig. 5 on page 33, one remarks that it has become considerably more complicated. This stems from "simplification"!

Even worse things are happening in the completion of the constrained system (27) where not only the constraint was several times not used for apparently easy simplifications and in one projection two identical integrability conditions were shown. The first effect will disappear as soon as a truly polynomial domain will be implemented. The current simplification and reduction routines rely heavily on the ability of `solveFor`. It in turn cannot treat products, if several factors contain jet variables. Since it is possible that one factor might be zero, `solveFor` returns in such cases `"failed"` to omit wrong results in the simplification.

Another point which will have to be addressed in further versions is regularity. As soon as we treat non-linear equations ranks and characters may differ on different points of a differential equation. If this happens, case distinctions are necessary. It is probably not efficient to treat automatically all occurring cases, as branching can occur in each step of the completion. But the output of a procedure like `complete` should at least show where problems can arise. Currently it follows just one branch without specifying the conditions.

To detect non-regularity one must inspect pivots. If a prolonged equation is not a manifold, then its Jacobian has different ranks on different parts of it. The determination of a row echelon form, say for a symbol, makes usually assumptions that certain expressions, namely the used pivots, are non-vanishing. The procedures for row echelon forms in `SEM` always return the pivots; however, the higher level domains and packages like `DE` or `CK` make currently no use of this information.

Sit [54] developed an algorithm which determines all necessary case distinctions in determining the rank of a symbolic matrix. It is based on the primary decomposition of algebraic varieties. He argued that an analysis based on pivots may lead to many unnecessary branches. While this is surely correct from a theoretical point of view, his method is probably too expensive to be applied automatically in an already expensive completion algorithm. Imagine all matrices occurring in the completion of the modified Janet example in Section 10 would be analyzed this way!

Besides simplification and reduction δ-regularity provides the most trouble. Currently `complete` in `CartanKuranishi` can handle only equations given in a δ-regular coordinate

system. Otherwise it will prolong and prolong and prolong without ever finding an involutive symbol. In principle one could implement the method presented in Ref. [50] using k-tableaux. We have refrained from this approach, because it becomes quickly computationally very demanding.

Computing row echelon forms of symbolic matrices is in general very expensive. In computer algebra a matrix is considered as large, if it has more than, say, 20 rows and columns (of course this depends on the complexity of the entries). For the symbol we could partially remedy this by using a special domain for sparse matrices. This allows to handle efficiently considerably larger matrices. In the Janet example one encounters e.g. matrices with more than 60 rows. The tableaux are, however, always dense. One reaches quickly machine limits, if several prolongations are needed or the system is large.

Often one can detect already by inspection that a coordinate system is not δ-regular. One possible strategy is to use the tableaux to find a non-systatic basis of T^*X [50] and an associated coordinate system. Then one rewrites the equation in this system and starts the completion algorithm. Of course it can always happen that the coordinate system will become again singular after some projections, but often this approach succeeds.

Our symmetry package is the first one developed in AXIOM. The REDUCE package SPDE [44] was translated [46] to its predecessor system Scratchpad II. It became, however, soon obsolete as new versions of Scratchpad II and later AXIOM appeared. `SYMANA` is still in a very early and experimental stage. The evaluation of the condition for symmetry generators is primitive compared with more sophisticated approaches, as they are e.g. described in Ref. [9].

Nevertheless, it demonstrates clearly the advantages of implementing a general environment: the basic procedure for setting up the determining equations consists of about 15 lines! Everything else is essentially provided by other domains. The many simplification and reduction routines originally implemented to support the completion procedure in `CK` are very useful to simplify determining systems. By using different domains in `JBFC` different simplification strategies can be applied.

We believe that, although there exist meanwhile so many symmetry packages, it is still possible to make significant progress in this field, even if one restricts ones attention to classical Lie symmetries (the extensions to more general symmetries are fairly straightforward anyway). We have already mentioned the possibility to handle equations which cannot be solved for some derivatives with Gröbner bases. One will have to wait whether it will be possible to provide an efficient implementation of this approach.

Another possibility for improvements is to make use of theoretical results on the form of admissible symmetry generators. Bluman [3] showed that at least for semi-linear equations one can simplify the ansatz for the generators considerably, because the coefficients' dependency of the dependent variable is very often trivial. This can save considerable computation time. First the prolongation of the generator is simpler and will lead to fewer terms, hence the resulting determining system is simpler. A lot of the work done by the packages to solve determining systems is just to rediscover through lengthy computations these simplifications!

Acknowledgments

A first version of this environment was implemented by J. Schü in his diploma thesis [42]. I am indepted to M. Bronstein for many helpful comments about AXIOM und to M.B. Monagan for turning my attention to the problem of treating sparse matrices and for explaining his and Berchtold's result. Part of this work was done at the Centre de recherches

mathématiques in Montréal and at the School of Physics and Materials in Lancaster. I am grateful to P. Winternitz and R.W. Tucker, respectively, for their hospitality. This work was partially supported by grants of Studienstiftung des deutschen Volkes, Deutsche Forschungsgemeinschaft and School of Physics and Materials, Lancaster University.

References

[1] Th. Becker and V. Weispfenning. *Gröbner Bases*. Springer-Verlag, New York, 1993.

[2] I. Berchtold. Sparse matrix determinants over integral domains. Master's thesis, ETH Zürich, 1993.

[3] G.W. Bluman. Simplifying the form of Lie groups admitted by a given differential equation. *J. Math. Anal. Appl.*, 145:52–62, 1990.

[4] G.W. Bluman and J.D. Cole. The general similarity solution of the heat equation. *J. Math. Mech.*, 18:1025–1042, 1969.

[5] G.W. Bluman and S. Kumei. *Symmetries and Differential Equations*. Applied Mathematical Sciences 81. Springer-Verlag, New York, 1989.

[6] R.L. Bryant, S.S. Chern, R.B. Gardner, H.L. Goldschmidt, and P.A. Griffiths. *Exterior Differential Systems*. Mathematical Sciences Research Institute Publications 18. Springer-Verlag, New York, 1991.

[7] W.H. Burge and S.M. Watt. Infinite structures in Scratchpad II. In J.H. Davenport, editor, *Proc. EUROCAL '87*, Lecture Notes in Computer Science 378, pages 138–148. Springer-Verlag, Berlin 1987.

[8] G. Carrà Ferro. Gröbner bases and differential algebra. In L. Huguet and A. Poli, editors, *AAECC-5*, Lecture Notes in Computer Science 350, pages 129–140, Berlin, 1987. Springer-Verlag.

[9] B. Champagne, W. Hereman, and P. Winternitz. The computer calculation of Lie point symmetries of large systems of differential equations. *Comp. Phys. Comm.*, 66:319–340, 1991.

[10] J.H. Davenport. The AXIOM system. AXIOM Technical Report 3, NAG, 1992.

[11] J.H. Davenport. How does one program in the AXIOM system. AXIOM Technical Report 4, NAG, 1992.

[12] J.H. Davenport and B.M. Trager. Scratchpad's view of algebra I: Basic commutative algebra. In A.M. Miola, editor, *Proc. DISCO '90*, Lecture Notes in Computer Science 429, pages 40–54. Springer-Verlag, Berlin 1990.

[13] P.A.M. Dirac. *Lectures on Quantum Mechanics*. Belfer Graduate School Monograph Series 3. Yeshiva University, New York, 1964.

[14] J. Della Dora and E. Tournier. Formal solutions of differential equations in the neighbourhood of singular points. In P.S. Wang, editor, *Proc. SYMSAC*, pages 25–29. ACM, New York 1981.

[15] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Dordrecht, 1992.

[16] K. Gottheil. Axioms, categories and domains. *mathPAD*, 4:24–29, 1994.

[17] D. Gruntz and M.B. Monagan. Introduction to GAUSS. *MapleTech*, 9:23–35, 1993.

[18] D. Hartley and R.W. Tucker. A constructive implementation of the Cartan-Kähler theory of exterior differential systems. *J. Symb. Comp.*, 12:655–667, 1991.

[19] A.K. Head. Lie, a PC program for Lie analysis of differential equations. *Comp. Phys. Comm.*, 77:241–248, 1993.

[20] W. Hereman. Review of symbolic software for the computation of Lie symmetries of differential equations. *Euromath Bull.*, 2:45–82, 1994.

[21] W. Hereman. Symbolic software for Lie symmetry analysis. In N.H. Ibragimov, editor, *CRC Handbook of Lie Group Analysis of Differential Equations, Volume 3: New Trends in Theoretical Development and Computational Methods*, chapter 13. CRC Press, Boca Raton, Florida, to appear (1995).

[22] E.L. Ince. *Ordinary Differential Equations*. Dover, New York, 1956.

[23] M. Janet. Sur les Systèmes d'Équations aux Dérivées Partielles. *J. Math. Pure Appl.*, 3:65–151, 1920.

[24] D. Jenks and R.S. Sutor. *AXIOM – The Scientific Computation System*. Springer-Verlag, New York, 1992.

[25] E.R. Kolchin. *Differential Algebra and Algebraic Groups*. Academic Press, New York, 1973.

[26] H. Kredel and V. Weispfenning. Computing dimension and independent sets for polynomial ideals. *J. Symb. Comp.*, 6:231–247, 1988.

[27] M. Kuranishi. On E. Cartan's prolongation theorem of exterior differential systems. *Amer. J. Math.*, 79:1–47, 1957.

[28] D. Levi and P. Winternitz. Non-classical symmetry reduction: Example of the Boussinesq equation. *J. Phys. A: Math. Gen.*, 22:2915–2924, 1989.

[29] E. Mansfield. *Differential Gröbner Bases*. PhD thesis, Macquarie University, Sydney, 1991.

[30] P.J. Olver. *Applications of Lie Groups to Differential Equations*. Graduate Texts in Mathematics 107. Springer-Verlag, New York, 1986.

[31] J.F. Pommaret. *Systems of Partial Differential Equations and Lie Pseudogroups*. Gordon & Breach, London, 1978.

[32] J.F. Pommaret. *Differential Galois Theory*. Gordon & Breach, London, 1983.

[33] J.F. Pommaret and A. Haddak. Effective methods for systems of algebraic partial differential equations. In T. Mora and C. Traverso, editors, *Proc. MEGA '90*, pages 411–426. Birkhäuser, Boston 1991.

[34] W.H. Preuss, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[35] D.W. Rand and P. Winternitz. ODEPAINLEVE — a Macsyma package for Painlevé analysis of ordinary differential equations. *Comp. Phys. Comm.*, 42:359–383, 1986.

[36] G.J. Reid. Algorithms for reducing a system of PDEs to standard form, determining the dimension of its solution space and calculating its Taylor series solution. *Eur. J. Appl. Math.*, 2:293–318, 1991.

[37] G.J. Reid. Finding abstract Lie symmetry algebras of differential equations without integrating determining equations. *Eur. J. Appl. Math.*, 2:319–340, 1991.

[38] G.J. Reid, I.G. Lisle, A. Boulton, and A.D. Wittkopf. Algorithmic determination of commutation relations for Lie symmetry algebras of PDEs. In B.M. Trager and D. Lazard, editors, *Proc. ISSAC '92*, pages 63–68. ACM, New York 1992.

[39] J.F. Ritt. *Differential Algebra*. Dover, New York, 1966. (Original: AMS Colloquium Publications, Vol. XXXIII, 1950).

[40] B.D. Saunders. An implementation of Kovačic's algorithm for solving second order linear homogeneous differential equations. In P.S. Wang, editor, *Proc. SYMSAC*, pages 105–108. ACM, New York 1981.

[41] D.J. Saunders. *The Geometry of Jet Bundles*. London Mathematical Society Lecture Notes Series 142. Cambridge University Press, Cambridge, 1989.

[42] J. Schü. Implementierung des Cartan-Kuranishi-Theorems in AXIOM. Master's thesis, Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, 1992.

[43] 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, Acireale (Italy), 1992. Kluwer, Dordrecht 1993.

[44] F. Schwarz. Automatically determining symmetries of partial differential equations. *Comp.*, 34:91–106, 1985.

[45] F. Schwarz. Symmetries and involution systems: Some experiments in computer algebra. In M. Ablowitz, B. Fuchssteiner, and M. Kruskal, editors, *Nonlinear Evolution Equations, Solitons and the Inverse Scattering Transform*, Oberwolfach, 1986. World Science Press, Singapore 1987.

[46] F. Schwarz. Programming with abstract data types: The symmetry package SPDE in Scratchpad. In R. Janßen, editor, *Trends in Computer Algebra*, Lecture Notes in Computer Science 296. Springer-Verlag, Heidelberg, 1987.

[47] W.M. Seiler. *Analysis and Application of the Formal Theory of Partial Differential Equations*. PhD thesis, School of Physics and Materials, Lancaster University, 1994.

[48] W.M. Seiler. On the arbitrariness of the general solution of an involutive partial differential equation. *J. Math. Phys.*, 35:486–498, 1994.

[49] W.M. Seiler. Arbitrariness of the general solution and symmetries. *Acta Appl. Math.*, to appear, 1995. (Special Issue Proc. Algebraic and Geometric Structures in Differential Equations, Twente 1993, J. Krasilshik and P.H.M. Kersten, eds.).

[50] W.M. Seiler. Generalized tableaux and formally well-posed initial value problems. Preprint Lancaster University, 1995.

[51] W.M. Seiler and R.W. Tucker. Involution and constrained dynamics I: The Dirac approach. Preprint Lancaster University, 1995.

[52] M.F. Singer. Liouvillian solutions of n-th order homogeneous linear differential equations. *Am. J. Math.*, 103:661–682, 1981.

[53] M.F. Singer. Formal solutions of differential equations. *J. Symb. Comp.*, 10:59–94, 1990.

[54] W.Y. Sit. An algorithm for solving parametric linear systems. *J. Symb. Comp.*, 13:353–394, 1992.

[55] K. Sundermeyer. *Constrained Dynamics.* Lecture Notes in Physics 169. Springer-Verlag, New York, 1982.

[56] R.S. Sutor and R.D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. *SIGPLAN Notices*, 22:7–14, 1987.

[57] Ph. Tombal and A. Moussiaux. MACSYMA computation of the Dirac-Bergmann algorithm for Hamiltonian systems with constraints. *J. Symb. Comp.*, 1:419–421, 1985.

[58] V.L. Topunov. Reducing systems of linear differential equations to a passive form. *Acta Appl. Math.*, 16:191–206, 1989.

[59] S.M. Watt, R.D. Jenks, R.S. Sutor, and B.M. Trager. The Scratchpad II type system: Domains and subdomains. In A.M. Miola, editor, *Computing Tools for Scientific Problem Solving.* Academic Press, New York, 1990.

[60] J. Weiss. The Painlevé property for partial differential equations II: Bäcklund transformations, Lax pairs, and the Schwarzian derivative. *J. Math. Phys.*, 24:1405–1413, 1983.

[61] J. Weiss, M. Tabor, and G. Carnevale. The Painlevé property for partial differential equations. *J. Math. Phys.*, 24:522–526, 1983.

[62] A.Yu. Zharkov and Yu.A. Blinkov. Involution approach to solving systems of algebraic equations. In G. Jacob, N.E. Oussous, and S. Steinberg, editors, *Proc. Int. IMACS Symp. Symbolic Computation*, pages 11–17, Lille, 1993.

[63] A.Yu. Zharkov and Yu.A. Blinkov. Involutive bases of zero-dimensional ideals. *J. Symb. Comp.*, to appear, 1994.

[64] R. Zippel. *Effective Polynomial Computation.* Kluwer, Dordrecht, 1993.

A Exported Procedures

The purpose of this appendix is to provide tables of most procedures currently implemented in our environment for geometric computations with partial differential equations. They also contain brief descriptions of the tasks of the different operations. However, they do not comment on the implementation or the methods used. For this we refer to the main text. The same holds for examples of the usage of the environment.

The order of the tables corresponds to the hierarchy explained in Section 6. We always start with the categories. Tables for domains are only given, if they contain additional operations. Some procedures return objects of a fairly complicated type. These are mostly meant for the use within programs and not for interactive use. Thus these types are not given explicitly. They must be looked up in the source code or with the `)show` command of the interpreter.

In order to make the tables not larger than they already are, we use abbreviations for the types. Those of the newly implemented ones will be introduced in the next paragraphs (cf. also the table in Fig. 3); the library types include `Boolean (B)`, `Expression Integer (EI)`, `Integer (I)` with the subtypes `PositiveInteger (PI)` and `NonNegative-Integer (NNI)` and the quotient field (rational numbers) `Fraction Integer (FI)`, `List (L)`, `Matrix (M)`, `Symbol (S)`, `OutputForm (OUT)`, `UnivariatePolynomial (UP)`, and `Vector (V)`. Following the usual convention of AXIOM we denote the domain or category under consideration by `$`.

Jet Bundles: The category `JetBundleCategory` abbreviated by `JBC` was mainly introduced to facilitate the passing of arguments to other domains and categories. Otherwise it would always be necessary to repeat explicitly all parameters like number of independent and dependent variables, respectively, or their names. There exist three instances of this category, namely `JetBundle` abbreviated by `JB`, `IndexedJetBundle (IJB)` and `JetBundleSymAna (JBSA)`.

`JB` takes two lists of symbols as arguments. They represent the names of independent and dependent variables, respectively. For the derivatives repeated index notation is used. `IJB(x,u,p,n,m)` generates a jet bundle with `n` independent and `m` dependent variables. In the output they are denoted by `x` and `u`, respectively. For derivatives `p` is used. Multi-index as well as repeated index notation can be chosen. `JBSA` is essentially identical with `JB`. It only determines its parameters differently: it requires as arguments another jet bundle which provides the new independent variables and symbols for the names of the new dependent variables.

Most operations of `JBC` can be found in Fig. 9 on page 45; in addition it is a subcategory of `OrderedSet`. `IJB` does not provide additional procedures. `JB` (and hence also `JBSA`) implements two additional procedures for the easier input of jet variables, namely a `coerce` from `Symbol` and an operation `D` with two arguments, a `Symbol` and a `List` of `Symbol`, to enter derivatives.

Functions on Jet Bundles: The basic category in this layer is `JetBundleFunction-Category` abbreviated by `JBFC`. It specifies the fundamental operations like partial and formal differentiation, computation of Jacobians and simplification. A list of most operations is given in Fig. 10 on page 46. Additionally there are the operations inherited from `PartialDifferentialRing Symbol`. For a more convenient input the operations `X`, `U`, `P`, `numIndVar`, `numDepVar`, `setNotation` and `getNotation` are copied from `JBC`.

The simplification procedures can be divided into two classes. `simplify`, `simpMod` and `simpOne` use only algebraic operations. One of the main tasks of `simplify` is to exhibit integrability conditions, if any are present. Otherwise no mixing of equations of different order happens. `reduceMod` and `autoReduce` also use differential operations. They correspond to algorithms used in differential algebra.

There are currently three instances of this category implemented, namely the domains `JetBundleExpression (JBE)`, `JetBundleLinearFunction (JBLF)` and `JetLazyFunction` (JLF). `JetBundleXExpression (JBX)` is a kind of specialization of `JBE`. It belongs to the sub-category `BaseFunctionCategory (BFC)` for functions only defined on the base space X.

Higher Structures: The core of this layer is the domain `DifferentialEquation` abbreviated by `DE`. Its main purpose is to provide an efficient data structure to store intermediate results. In its current form it is especially adapted for computations as they are needed in the completion algorithm 1. `DE` takes two arguments: a jet bundle from `JBC` and a domain from `JBFC` to represent the equations. Fig. 11 on page 47 shows the implemented operations of `DE`.

The implementation of the differential geometric domains `VectorField` (abbreviated by `VF`) and `Differential (DIFF)` are fairly primitive. They should be considered as a temporary hack, until AXIOM contains a better environment for differential geometric calculations. The arguments for both are like the ones for `DE`: a jet bundle from `JBC` and a coefficient domain from `JBFC`. Most operations of these domains can be found in Fig. 12 and Fig. 13, respectively, on page 48. In addition they contain the operations of a `Module` over the coefficient domain denoted by `D`.

Applications and Utility Packages: The package `CartanKuranishi` is abbreviated by `CK`. It contains basically a procedure to complete a given differential equation to an involutive one. Furthermore there are some procedures for computations with the Cartan characters and the Hilbert polynomial. A full list is shown in Fig. 14 on page 49. `CK` takes two arguments: one domain `JB` of the category `JBC` and one domain `D` of the category `JBFC(JB)`. The differential equations are then from the domain `DE(JB,D)` (abbreviated by `DE` in Fig. 14). Note that if lists of characters like $\alpha_q^{(k)}$ or $\beta_q^{(k)}$ are passed as argument, they must always be ordered by increasing k.

The package `SymmetryAnalysis` is abbreviated by `SYMANA`. It is still in an experimental state. Its first argument is a jet bundle; the other two are symbols used for the names of the coefficients in the ansatz for symmetry generators. A preliminary list of its procedures is shown in Fig. 16 on page 51.

The domain `SparseEchelonMatrix` is abbreviated by `SEM`. It is especially designed for the fast determination of row echelon forms for sparse matrices which are already close to this form. It takes as argument a domain `D` from `Ring` for the entries and a domain `C` from `OrderedSet` whose elements are used to label the columns of the matrices. A complete list of the operations of `SEM` is contained in Fig. 15 on page 50.

The packages `JetCoordinateTransformation (JCT)` and `LUDecomposition (LUD)` provide only two and three procedures, respectively. `JCT` takes four arguments: two jet bundle `JB1` and `JB2` plus two vectors `Y,W` containing expressions for the old coordinates in terms of the new ones. We will write `E1,E2` for `JBE JB1` and `JBE JB2`, respectively. `LUD` expects a domain of type `Field` as argument. The operations are shown with their signatures in Fig. 17 and Fig. 18, respectively, on page 51.

`allRepeated`	`L NNI -> L L PI`	Computes all possible realizations of a given multi-index as repeated index.
`class`	`L NNI -> NNI`	Returns the class of a multi-index.
`class`	`$ -> NNI`	Returns the class of a jet variable.
`coerce`	`$ -> EI`	Coerces a jet variable into an expression.
`coerce`	`$ -> S`	Coerces a jet variable into a symbol.
`derivativeOf?`	`($, $) -> L NNI`	Checks whether the first argument is a derivative of the second one. Returns either the difference of their multi-indices, if positive, or an empty list.
`differentiate`	`($, PI) -> Union($,"0")`	Differentiates a jet variables with respect to the independent variable labeled by the second argument.
`dimJ`	`NNI -> NNI`	Returns the (fiber) dimension of the jet bundle of the given order.
`dimS`	`NNI -> NNI`	Returns the dimension of $S_qT^*X \otimes V\mathcal{E}$ for the given value of q.
`getNotation`	`() -> S`	Shows the currently used notation.
`index`	`$ -> PI`	Returns the upper index of a jet variable.
`integrate`	`($, PI) -> $`	Integrates a jet variable with respect to an independent variable. Yields an error if not possible.
`integrateIfCan`	`($, PI) -> Union($,"failed")`	Integrates a jet variable with respect to the independent variable labeled by the second argument.
`multiIndex`	`$ -> L NNI`	Returns the multi-index of a jet variable.
`m2r`	`L NNI -> L PI`	Transforms a multi-index into a repeated index.
`name`	`$ -> S`	Returns the name of a jet variables.
`numDepVar`	`() -> PI`	Returns the number of dependent variables.
`numIndVar`	`() -> PI`	Returns the number of independent variables.
`one?`	`$ -> B`	Checks whether its argument is the "jet variable" 1.
`order`	`$ -> NNI`	Order of a jet variable (0 for non-derivatives).
`P`	`(PI, L NNI) -> $`	Generates a derivative. The interpretation of the second argument depends on the used notation.
`Pm`	`(PI, L NNI) -> $`	Generates a derivative with a given multi-index.
`Pr`	`(PI, L NNI) -> $`	Generates a derivative with a given repeated index.
`repeatedIndex`	`$ -> L PI`	Returns the multi-index of a jet variable in repeated index notation.
`r2m`	`L PI -> L NNI`	Transforms a repeated index into a multi-index.
`setNotation`	`S -> S`	Chooses the notation used for derivatives. Returns the old one. Possible values are `Repeated` and `Multi`.
`type`	`$ -> S`	Yields the type of a jet variable. Possible values are `Const`, `Indep`, `Dep`, `Deriv`.
`U`	`PI -> $`	Generates a dependent variable.
`variables`	`NNI -> L $`	Returns an ordered list of all jet variables up to the given order.
`variables`	`(NNI, PI) -> L $`	Returns all jet variables of the given order whose class is greater than or equal to a given value.
`weight`	`$ -> NNI`	Computes the weight assigned to a jet variable in the used ranking.
`X`	`PI -> $`	Generates an independent variable.
`1`	`() -> $`	Generates the special "jet variable" 1.

Figure 9: Procedures in `JetBundleCategory`.

autoReduce	L $ -> L $	Reduces a system with respect to itself.
class	$ -> NNI	Class of an expression.
const?	$ -> B	Checks whether an expression depends on jet variables.
coerce	JB -> $	Transforms a jet variable into a function.
denominator	$ -> $	Denominator of an expression.
differentiate	($, JB) -> $	Differentiation with respect to a jet variable.
dimension	(L $, SEM, NNI) -> NNI	Dimension of a system with given Jacobian in the jet bundle of given order.
dSubst	($, JB, $) -> $	Like **subst** but takes also derivatives into account.
extractSymbol	SEM -> SEM	Extracts symbol from the Jacobian.
formalDiff	($, PI) -> $	Formal differentiation with respect to an independent variable given by its label. There exist further mode maps for systems and with more detailed output. In some versions it is also possible to pass the Jacobian as additional argument.
freeOf?	($, JB) -> B	Checks whether an expression depends on a given jet variable.
jacobiMatrix	L $ -> SEM	Computes Jacobian of a system.
jacobiMatrix	(L $, L L JB) -> SEM	Computes Jacobian of a system. For each equation the occurring jet variables are given.
jetVariables	$ -> L JB	Returns all jet variables effectively occurring in an expression.
leadingDer	$ -> JB	Leading derivative of an expression.
numerator	$ -> $	Numerator of an expression.
order	$ -> NNI	Highest order of the effectively occurring jet variables.
orderDim	(L $, SEM, NNI) -> NNI	Dimension of a given system taking only the given order into account.
reduceMod	(L $, L $) -> L $	Reduces a system with respect to another one.
simplify	(L $, SEM) -> SIMPREC	Simplifies a system with given Jacobian. Returns the simplified system and its Jacobian. Additionally it tries to trace the dependency of the returned equations of the original equations.
simpOne	$ -> $	Simplifies individual equations (removes unnecessary coefficients and signs etc.).
simpMod	(L $, L $) -> L $	Like **reduceMod**, but does not use derivatives of equations.
simpMod	(L $, SEM, L $) -> SIMPREC	Simplifies a system modulo another one. Returns also the changed Jacobian.
solveFor	($, JB) -> Union($,"failed")	Solves an expression for a given jet variables if possible.
sortLD	L $ -> L $	Sorts a list of functions according to their leading derivatives.
subst	($, JB, $) -> $	Substitutes a given expression for a jet variable in another expression.
symbol	L $ -> SEM	Computes symbol of a system.

Figure 10: Procedures in `JetBundleFunctionCategory`.

analyseSymbol	SEM -> MVREC	Computes a row echelon form of a given symbol. Counts the multiplicative variables and determines the rank of the matrix.
copy	$ -> $	Returns a copy of a differential equation.
dimension	($, NNI) -> NNI	Computes the dimension of a given differential equation considered as submanifold of a jet bundle of given order.
display	$ -> OUT	Prints most of the information stored about a differential equation: equations ordered by their order, for each order the Jacobian, whether the system is already simplified and so on.
extractSymbol	($, B) -> SEM	Extracts the symbol from the Jacobian of the highest order part of the equation. If the second argument is true, its row echelon form is computed.
generate	L D -> $	Generates a differential equation from a given system.
insert	(L D, $) -> $	Inserts additional equations into a differential equation.
jacobiMatrix	$ -> L SEM	Returns the Jacobians separately for each order.
join	($, $) -> $	Combines two differential equations into one.
order	$ -> NNI	Returns the order of a differential equation.
project	($, NNI) -> $	Projects a differential equation to a given order. Assumes the equation was previously prolonged.
prolong	$ -> SREC	Prolongs all equations in a differential equation which have not already been prolonged, i.e. especially all equations of highest order. Found integrability conditions are returned separately.
prolong	($, NNI) -> SREC	Like `prolong`. Considers, however, only equations below a given order.
prolongMV	MVREC -> MVREC	Calculates directly the multiplicative variables for a prolonged symbol. Result only valid, if the symbol is involutive.
prolongSymbol	SEM -> SEM	Prolongs directly a given symbol.
printSys	L D -> OUT	Writes a list of functions as a vector of equations (right hand side 0) and coerces the result to `OUT`.
retract	$ -> L D	Returns a list of all equations contained in a given differential equation.
setSimpMode	NNI -> NNI	Sets the simplification mode. If greater than zero, equations are simplified modulo lower order equations.
simplify	$ -> SREC	Simplifies the equations order by order. Found integrability conditions are returned separately.
tableau	(SEM, DIFF) -> SEM	Computes the tableau to a given symbol parameterized by a given one-form.
tableau	(SEM, L DIFF) -> SEM	Computes the k-tableau to a given symbol parameterized by a given list of one-form.

Figure 11: Procedures in `DifferentialEquation`.

`coefficient`	`($, JB) -> D`	Returns the coefficient in a given direction.
`coefficients`	`$ -> L D`	Returns the coefficients of a vector field.
`commutator`	`($, $) -> $`	Computes the commutator of two given vector fields.
`copy`	`$ -> $`	Returns a copy of a given vector field.
`diff`	`JB -> $`	Generates base vector field with given direction.
`diffP`	`(PI, L NNI) -> $`	Generates base vector field in direction of a derivative.
`diffU`	`PI -> $`	Generates base vector field in direction of a dependent variable.
`diffX`	`PI -> $`	Generates base vector field in direction of an independent variable.
`directions`	`$ -> L JB`	Returns a list of the directions of the base vectors where the vector fields has non-vanishing coefficients.
`eval`	`($, D) -> D`	Applies a vector field to a function.
`lie`	`($, $) -> $`	Computes the Lie derivative of a given vector field in the direction of another vector field.
`prolong`	`($, NNI) -> $`	Prolongs a vector field defined on $\mathcal{E}$ to a field on the jet bundle of the given order.
`table`	`L $ -> M $`	Computes the commutator table to a given list of vector fields.

Figure 12: Procedures in `VectorField`.

`coefficient`	`($, JB) -> D`	Returns the coefficient of a given differential.
`coefficients`	`$ -> L D`	Returns a list with the coefficients of a given one-form.
`contract`	`(VF, $) -> D`	Contracts a vector field with a one-form (interior derivative).
`d`	`JB -> $`	Generates the differential of a given jet variable.
`d`	`D -> $`	Computes the differential of a function.
`differentials`	`$ -> L JB`	Returns a list of the base differentials with non-vanishing coefficients.
`dP`	`(PI, L NNI) -> $`	Generates the differential to a derivative.
`dU`	`PI -> $`	Generates the differential to a dependent variable.
`dX`	`PI -> $`	Generates the differential to an independent variable.
`eval`	`($, VF) -> D`	Applies a one-form to a vector field.
`lie`	`(VF, $) -> $`	Computes the Lie derivative of a one-form in direction of a given vector field.

Figure 13: Procedures in `Differential`.

`alpha`	`(NNI, L NNI) -> L NNI`	Computes the Cartan characters for a differential equation of given order from the $\beta_q^{(k)}$.
`alphaHilbert`	`UP("r",FI) -> L NNI`	Compute the Cartan characters for a given Hilbert polynomial.
`arbFunctions`	`(NNI, I, L NNI) -> L I`	Uses the Cartan characters to compute the number of arbitrary functions of a fixed differentiation order for a differential equation of given order.
`bound`	`(NNI, NNI, NNI) -> NNI`	Computes a bound $\tilde{q}(n, m, q)$ for the number of prolongations needed to render a symbol involutive.
`complete`	`DE -> Void`	Completes a given differential equation to an involutive one. No result is returned, but information on the completion process is displayed. The amount of it can be controlled using `setOutput`.
`complete2`	`DE -> CREC`	Analog to `complete`. Returns a record containing the involutive equation, a generating set of equations, the order of the involutive equation, the number of projections needed, the dimension of the equation and its Cartan characters.
`gauge`	`(NNI, I, L NNI) -> L I`	Computes the gauge corrections to the number of arbitrary functions for given values of γ_l. The other arguments are as for `arbFunctions`.
`gaugeHilbert`	`(NNI, L NNI) -> UP("r",FI)`	Computes analog to `hilbert` the gauge correction to the Hilbert polynomial.
`hilbert`	`L NNI -> UP("r",FI)`	Computes the Hilbert polynomial for given Cartan characters.
`setOutMode`	`NNI -> NNI`	Controls the amount of output generated by `complete`. The following values are possible for the argument: $0 \rightarrow$ no output $1 \rightarrow$ result is displayed $2 \rightarrow$ Cartan characters are displayed $3 \rightarrow$ integrability conditions are traced $4 \rightarrow$ intermediate dimensions are traced $5 \rightarrow$ intermediate systems are traced $6 \rightarrow$ intermediate symbols are traced Addition of 10 yields output in TEX.
`setRedMode`	`NNI -> NNI`	Sets reduction mode. If value greater than zero, then integrability conditions are reduced.
`setSimpMode`	`NNI -> NNI`	See `DifferentialEquation`.

Figure 14: Procedures in `CartanKuranishi`.

*	(M D, $) -> $	Left multiplication of a sparse matrix with a usual matrix.
*	(M F D, $) -> $	Left multiplication of a sparse matrix with a usual matrix over the quotient field of D. Available only if D belongs to IntegralDomain.
allIndices	$ -> L C	Yields a list of all indices used to label columns.
appendRow!	($, ROWREC) -> Void	Adds a new row as last row.
elimZeroCols!	$ -> Void	Removes columns containing only zeros. This effects, however, basically only the value of allIndices.
coerce	$ -> M D	Coerces a matrix from SEM to the usual matrix type.
consRow!	($, ROWREC) -> Void	Adds a new row as first row. Argument is changed destructively.
copy	$ -> $	Yields a copy of a matrix.
deleteRow!	($, I) -> Void	Deletes the indicated row.
elt	($, I, C) -> D	Returns the entry in the given row and the column labeled by the given index.
extract	($, I, I) -> $	Returns a sub-matrix consisting of the indicated range of rows.
horizJoin	($, $) -> $	Concatenates horizontally two matrices. It is assumed that all indices of the second one are smaller than those of the first one.
horizSplit	($, C) -> Record(Left:$, Right:$)	Splits a matrix into two at the indicated column.
join	($, $) -> $	Concatenates vertically two matrices.
ncols	$ -> NNI	Returns the number of columns of a matrix.
new	(L C, I) -> $	Generates a new matrix with a given number of rows. The first argument contains the indices.
nrows	$ -> NNI	Returns the number of rows of a matrix.
pivot	($, I) -> Record(Index:C, Entry:D)	Returns the pivot of the indicated row (plus label for column).
pivots	$ -> ROWREC	Returns all pivots (plus labels for columns).
primitiveRow-Echelon	$ -> Record(...)	Like rowEchelon. Uses a primitive fraction-free method. The returned transformation matrix contains fractions. Only available, if D belongs to GcdDomain.
purge!	($, C->B) -> Void	Eliminates all columns whose label satisfy the given criterion function.
row	($, I) -> ROWREC	Yields a row of a matrix
rowEchelon	$ -> Record(...)	Computes the row echelon form of a matrix using fraction-free elimination. Returns furthermore the transformation matrix, the used pivots, and the rank.
setElt!	($, I, C, D) -> Void	Sets an entry to a given value. Row and column are indicated as in elt.
setGcdMode	S -> S	Sets flag for method to compute GCDs for lists.
setRow!	($, I, L C, L D) -> Void	Sets a whole row given as lists of entries and labels, respectively.
setRow!	($, I, ROWREC) -> Void	Sets a whole row given as a record.
sortedPurge!	($, C->B) -> Void	Like purge!. Assumes that the criterion respects the ordering of the labels.

Figure 15: Procedures in SparseEchelonMatrix.

`ansatz`	`() -> VF`	Yields the most general ansatz for a symmetry generator.
`detSys`	`(L JBE1, L JB, VF) -> L JBE1`	Computes the determining system for a given system. The second argument contains a list of derivatives for which the equations can be solved. If it is omitted the leading derivatives are used. The third argument contains an ansatz for the generators. It can also be omitted. `ansatz()` is then used.
`linearize`	`L JBE2 -> L JBL2`	Retracts equations to linear ones, if possible.
`ncDetSys`	`(L JBE1, L JB, VF) -> L JBE1`	Computes the determining system for conditional symmetries. The meaning of the arguments is as in `detSys`.
`transform`	`JBE1 -> JBE2`	Transforms between the different jet bundles involved.

Figure 16: Procedures in `SymmetryAnalysis`.

`transform`	`JB1 -> E2`	Transforms a jet variable in the old coordinates into an expression in the new ones.
`transform`	`E1 -> E2`	Transforms an expression in the old coordinates into the new ones.

Figure 17: Procedures in `JetCoordinateTransformation`.

`LUDecomp`	`M D -> Record(...)`	Computes the LU decomposition of a given matrix. Returns a record with a matrix containing the two triangular ones, the permutation needed for partial pivoting, and the used pivots.
`LUSolve`	`(M D, V I, V D) -> V D`	Expects a matrix in LU decomposition, the permutation used for pivoting, and a right hand side. Returns the solution vector.
`LUInverse`	`M D -> Record(...)`	Computes the inverse of a matrix using an LU decomposition. Besides the inverse the used pivots are returned.

Figure 18: Procedures in `LUDecomposition`.