Functional Coding of Differential Forms

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France
karczma@info.unicaen.fr

Abstract. Algebraic computations in differential geometry have usually a strong "analytic" side, and symbolic formula crunching is heavily used, even if at the end, the user needs only numbers, or graphic visualization. We show how to implement in a simple way the domain of differential forms with the p-vector algebra, Hodge "star" operator, and the differentiation. There is no explicit symbolic manipulation involved, we exploit only the "standard" mathematical operations in a generic way. Everything forms a local algebra coded in Haskell, and the differentiation algorithms heavily use the lazy evaluation. Some short examples are presented. This paper generalizes our one-dimensional algorithmic differentiation formalism in functional sauce presented elsewhere.

Keywords: Haskell, differentiation, laziness, forms, geometry.

1 Introduction

1.1 Lazy Differential Algebra

The differentiation is a "mechanical", easily algorithmisable procedure and first computer implementations of symbolic differentiation are older than FORTRAN. All computer algebra packages are equipped with the appropriate modules. Less known is the concept of the *algorithmic differentiation* ([1]) of *numerical* computer programs. No symbolic processing is involved, the program computes in parallel the value of an expression and of its derivative, beginning with constants and the "differentiation variable" – whose derivatives are trivial. All arithmetic operations can be overloaded in order to handle such compound data, and the calculus threads along the program: $(x, x') + (y, y') = (x + y, x' + y')$, $(x, x') \cdot (y, y') = (xy, xy' + x'y)$, etc. All manipulations can eventually be reduced to the chaining of the elementary operations. These operations augmented by the *derivation operator* which yields the derivative, form a local differential algebra. In [2, 3] we have shown how to augment the typical "scientific" programs, by the derivation operator using a functional version of the Algorithmic Differentiation paradigm. We exploited the nice properties of lazy framework which made it possible to implement effectively and efficiently a *closed* differential algebra in Haskell [4]. Formally the generator of the differential algebra, i. e. the "differentiation variable", is equivalent to an infinite number of algebraically independent variables: the program which manipulates an expression p should be able to construct all the entities p', p'', $\ldots$ etc., and apparently the symbolic manipulation is the only way to do this, because numerical programs dislike infinite data. But not in lazy languages! We proposed to embed the expressions p (which in the program are just numbers assigned to program variables) into infinite sequences $p = [p^{(0)}, p', p'', p^{(3)}, \ldots]$. All the constants c (possibly assigned to program variables) are lifted to $[c, 0, 0, \ldots]$, and the distinguished differentiation variable, e.g. the parameter x of a function whose derivative we compute, becomes $[x, 1, 0, 0, \ldots]$. The derivation operator is the tail of such sequences, and the algebra is closed. Having two expressions $p = [p^{(0)}, p', p'', \ldots] = (p^{(0)} : \overline{p})$ and $q = [q^{(0)}, q', q'', \ldots] = (q^{(0)} : \overline{q})$, we write

$$p \pm q = (p^{(0)} \pm q^{(0)} : \overline{q} \pm \overline{q}), \tag{1}$$

$$p \cdot q = (p^{(0)} \cdot q^{(0)} : p\overline{q} + q\overline{p}), \quad p/q = (p^{(0)}/q^{(0)} : \overline{p}/q - p\overline{q}/q^2), \tag{2}$$

$$\exp(p) = r \quad \textbf{where} \quad r = (\exp(p^{(0)}) : \overline{p}r), \tag{3}$$

$$\sqrt{p} = r \quad \textbf{where} \quad r = (\sqrt{p^{(0)}} : \frac{1}{2}\overline{p}/r), \tag{4}$$

etc. The technique can be used to compute recursively defined functions with derivatives, and to other computations where symbolic manipulation is unwieldy, and numerical approximations might be unstable.

1.2 Multi-dimensional Case

The generalization to many dimensions in principle is straightforward. We can define a new datatype: `data Scal a = Df a [Scal a]` where the type `a` will typically be `Double`. The list contains all the partial derivatives wrt. the "variables". The constants are formed by the function `sCst x = Df x []`, but the "variable" now is a vector, so in 3 dimensions the component y of the vector $\boldsymbol{x}$ will have the form `Df y [0,1,0]`. The multiplication, and other operations are more involved, for example

```
p@(Df x x')*q@(Df y y')=Df (x*y) (zipWith (+) (map (p*) y') (map (*q) x'))
```

etc. The propagation of the derivatives through function applications proceeds as in the one dimensional case, only the explicit constructor of the "variable" $\boldsymbol{x}$ is more involved. We call it `dfVar`, and its argument is a list of the base type values (Doubles). It will be defined in the section (3.2).

In our package we used a **modified** numerical Prelude with such classes as `AddGroup` which declares the addition, `Monoid` with multiplication, `Group` which augments `Monoid` by division, `Transcen` where the overloaded exponential, logarithm, etc. are declared (and where we have squeezed also such functions as the square root. . .), and some other generic classes appropriate for the development of general arithmetic system. In particular, our `Eq` instance is ill-defined – we compare only the "main" values and not the derivatives, which would overflow. We don't feel particularly guilty about that, because in typical numerical computations one rarely applies the equality for real numbers; what is needed is some topology, all numbers are approximations. Moreover, expressions with *all* their derivatives are non-local objects. Comparing them is like comparing functions: an effective algorithm for this is more than difficult.

We have used such a package written in Clean [5] for generic manipulations of 3D surface models ([6]). But a more ambitious look at those manipulations **is** necessary. The triple (x, y, z) is not just a record, but a vector, a *geometric* object with its algebra, all the invariance properties of the scalar product, and all the transformation paradigms. Now we shall implement a more generic geometric formalism.

1.3 What for?

This paper belongs to our crusade against the abuse of Computer Algebra packages by some computing physicists and other similar dangerous species. One does not need to be a relativistic physicist [1] in order

[1] Relativistic Physicist: a researcher who travels from conference to conference at the speed of light; do not confound with Quantum Physicist who prefers teleporting.

to work in curved spaces, or to need some complicated tensor formulae. We have seen many times that complicated tensor computations needed in statistical physics (field theory, crystallography, etc.) or in engineering (mechanics, robotics) performed by the symbolic packages, serve only to generate computer programs in FORTRAN or "C" attached to a number-crunching application. The formulae themselves are unwieldy, offering no insight for the humans. But the standard "scientific" languages are far too poor to handle really complicated mathematical objects; the evolution of some object-oriented mathematical libraries is still quite slow. Our objective is to show on some simple examples how to throw a direct bridge between the conceptual, *simple and compact* initial formulae such as the Laplacian $\Delta\phi$ in *any* coordinate system, and their final numeric implementation. This can be easily done in many modern languages; our functional option arose from the conviction that a good representation of mathematical objects and algorithms should be as static as possible, without side effects. The usage of laziness not only simplifies some iterative processes, but bridges the gap between our computer codes and the Infinity. But *nothing* in our algorithms is specifically numeric. They are universal, and if somebody constructs a symbolic datatype whose elements belong to a commutative algebra: can be added, multiplied, etc., our code is directly reusable.

2 Polyvectors and Tensors

2.1 Vectors and Skew Products

We want to put into a common datatype all p-vectors, including scalars. We declare thus

```
data PV a = VZ | S a | V (PV a) (PV a)
```

where `VZ` is a *formal* zero vector, which acts also as the terminating item of the p-vector tree. `(S x)` is a scalar: x may belong to `Double`, or, later on, to `Scal Double`. A simple algebraic vector: $x = 1$, $y = 3.5$, $z = 2$ is represented by `V (S 1.0) (V (S 3.5) (V (S 2.0) VZ))`. The constructor `PV` is a natural `Functor`, with the generalized map functional (`fmap`) defined in a most straightforward way

```
instance Functor PV where
 fmap _ VZ = VZ
 fmap f (S x) = S (f x)
 fmap f (V p q) = V (fmap f p) (fmap f q)
```

The construction of the `Eq` and `AddGroup` instances of `(PV a)` is immediate. The tensor product which creates nested sequences is defined by

```
instance (AddGroup a,Monoid a) => Monoid (PV a) where
 VZ*_=VZ;            _*VZ=VZ
 S a * p = a *> p;  p * S a = a *> p
 (S x) * (S y) = S (x*y)
 a@(S _) * V p q = V (a*p) (a*q)
 V p q * r = V (p*r) (q*r)
```

where `x*>v = fmap (x *) v` is the implementation of the "external" multiplication of `a` by the structure `PV a` defined within the class `Module`. (The differential scalars are also instances of a `Module`: the multiplication of such a scalar by a constant propagates this operation through all the derivatives.)

We see that the recursive chaining replaces the usage of indices. The component $(u_2 v_3)$ of the product $A = (\boldsymbol{uv})$ is the third component of the second element of A. Such representation of matrices has been used extensively in Lisp, and criticized. The usage of vectors and indexing is more compact while writing formulae on paper. But we use a functional language, where the recursive list processing is natural, and *we don't want to use indices in our codes!*

- We want to work mainly with *invariant* operations, where an object preserves its geometric identity. Even the selection of a component may be implemented as a scalar product of a vector with an appropriate projector, say, $[0, 0, 1, 0]$. But the extraction of *one* component is a rare operation anyway.
- The formalism should not depend on the dimension of the space, and it should be natural, without the need of keeping separately the information about this dimension.
- We shall need not only rectangular "generalized matrices", but also antisymmetric tensors (Pfaffians, exterior forms), whose shape resembles more a simplex than a parallelepiped. This is the principal argument for our choice.
- Most symbolic manipulations of tensor formulae use the indices (free or repeated) as dummy items: placeholders, and most formulae are invariant anyway. The computer algebra package constructors tried to adapt the syntax to the established manual formula processing, but this superficial tribute paid to Cartan, Einstein, Schouten etc. need not be followed by everybody. Especially when – as here – we shall *not* do any symbolic manipulations.

We will not discuss the standard tensor products, nor general tensors constructed from 1-vectors by multiplication and addition. The scalar sector of p-vectors belongs to a normal commutative algebra, and we define for it the division, algebraic and transcendental functions, etc. We pass to something much more interesting, to the higher p-vectors resulting from the application of antisymmetric, skew product $u \wedge v$. The result will be *implicitly* antisymmetric. A 2-vector will be a nested sequence containing $n(n-1)/2$ elements, where n is the dimension of the underlying space. For example, the elements of a 2-vector for $n = 3$ form a tree $[[a_{12}, a_{13}], [a_{23}]]$, and a 3-vector in a 5-dimensional space has the components $[[[a_{123}, a_{124}, a_{125}]$, $[a_{134}, a_{135}]$, $[a_{145}]]$, $[[a_{234}, a_{235}]$, $[a_{245}]]$, $[[a_{345}]]]$. Every sub-matrix is triangular, only the items whose index set is ordered will be kept. Formally the skew product $A = u^{(1)} \wedge u^{(2)} \wedge \cdots \wedge u^{(p)}$ is defined as

$$A_{i_1 i_2 \ldots i_p} = \sum_{j_1 \cdots j_p} \delta^{j_1 \cdots j_p}_{i_1 \ldots i_p} u^{(1)}_{j_1} u^{(2)}_{j_2} \ldots u^{(p)}_{j_p}, \tag{5}$$

where the generalized Kronecker δ is equal to 1 when the upper index sequence is an even permutation of the lower set, -1 in the odd case, and 0 otherwise. Implementing this using indices and loops (and a cascade of conditionals) is dull. The recursive code is much more compact, and less error-prone. Curiously enough, it is not easy at all to find the code given below in the literature, even Lisp implementations of some early computer algebra packages are polluted by indices. . . We shall use the super-commutativity properties of the skew product. If u is a p-vector, and v – a q-vector, their product obeys $u \wedge v = (-1)^{(\mathrm{ord}_{uv})} v \wedge a$, where ord_{uv} is equal to -1 when both p and q are odd, and 1 otherwise. The order of a p-vector is immediately established by looking upon the depth of the `V ...` data structure. Omitting the trivial `VZ` and scalar sector we get a remarkably compact code (where `ordv2` computes the sign)

```
u@(V u1 uq) /\ v@(V v1 vq)
 | uq==VZ || vq==VZ = VZ           --No antisym. in 1 dim.
 | otherwise = V (u1/\vq + (ordv2 u v)*>(v1/\uq)) (uq/\vq)
```

In order to prove this, it suffices to note that for the first component of the result – the scalar or vector which contains the index 1, the structure **must** be like that; the recursion does the rest. All elements which contain the index 1 have the form: $u_{1\alpha} \pm u_{\alpha 1}$, where α is the remaining index set. The choice of sign depends on the cardinality of α: the number of transpositions necessary to push the index 1 to the right through the sequence α. In our convention we do not divide the p product by $p!$.

We can construct now the generalized vector products in n-dimensional space. In 3 dimensions a 2-product is structurally equivalent to a normal 1-vector, the known vector product. The 3-product has only one component, it is equivalent to a scalar. We might notice that in fact, in order to establish this equivalence we should be able to compute *dual* p-vectors, and for this we must have also defined the *scalar product* of vectors (the metric), otherwise there cannot be any equivalence between contra- and covariant vectors. The dualization is performed by the Levi-Civitta tensor which transposes the covariance of its argument.

2.2 Hodge "star" Operator

We will discuss here only the Euclidean case, with the scalar product of two 1-vectors defined as the sum of the component-wise products (or with the metric tensor being the identity matrix). Some (important) generalizations will be mentioned later. We don't need to, and we won't distinguish here between the contra- and covariant vectors. The dualization operation known as the Hodge "star" operator is defined by

$$(*u)_{i_1 i_2 \ldots i_q} = \sum_{j_1 \cdots j_p} \epsilon_{i_1 i_2 \ldots i_q j_1 \ldots j_p} u_{j_1 \ldots j_p}, \tag{6}$$

where ϵ is the fully antisymmetric n-tensor in n dimensional space, whose only component is equal to 1 when the set of indices is an even permutation of $\{1, 2, \ldots, n\}$, -1 for the odd configuration, and 0 in the remaining cases. We will see later that in more general coordinate systems ϵ is not equal to ± 1; it is a *density* proportional to the Jacobian of the transformation between the Cartesian and the general basis.

The construction of the star operator is simple and quite amusing. Take for example a 2-vector: $[[u_{12}, u_{13}, \ldots, u_{1n}]$, $[u_{23}, \ldots u_{2n}]$, $\ldots$, $[u_{(n-1)n}]]$. It is obvious from the construction (6) that the first component of $(*u)$, whose first index is 1, cannot have any items from the first (compound) element of u. Only the tail of u contributes to it. On the other hand the whole sub-vector $[u_{12}, u_{13}, \ldots, u_{1n}]$ will contribute to the tail of $(*u)$, because the index 1 *must* be present therein. We should also take into account the signs. For any p-vector u the following identity holds: $*(*u) = (-1)^{p(n-p)}u$ (if the signature of the metric tensor is trivial; in pseudo-Euclidean case it is more complicated).

We define first some generic auxiliary functions: `vdpth u` computes the depth p of a p-vector, `dsgn n` is equal to 1 for n even and -1 for n odd, `vhd` and `vtl` which retrieve the `head` and the `tail` from `V head tail`, and `nlist n x` which produces from x a nested `V` structure $[[\ldots[x]\ldots]$ of depth n. The predicates `vnull` and `scalr` verify if the vector is equal to VZ, or it is a scalar. We have then

```
hodge x = hdgn (ndim x) x
hdgn n x
 | n==0 = x
 | scalr x = (nlist n x)
 | otherwise = let s=fromInt (dsgn (n - vdpth x))
                    h=vhd x; t=vtl x
                in if vnull t then hdgn (n-1) (s*>h)
                   else V (hdgn (n-1) t) (hdgn (n-1) (s*>h))
```

The vector product of any two 1-vectors is $u \times v = *(u \wedge v)$. This is a 1-vector only for $n = 3$.

2.3 Generalized Scalar Product

The scalar product may be constructed as

$$u \cdot v = *(u \wedge *v), \tag{7}$$

and this holds for any p-vectors. If the depth of u and v is equal, the result is a scalar, because the skew product of a p- by a $(n-p)$-vector produces a n-vector which has only one component. If $A = u^{(1)} \wedge u^{(2)} \wedge \ldots \wedge u^{(n)}$, its value is the determinant $|u_k^{(i)}|$. In fact, we have constructed and coded a variant of the Laplace expansion for determinants. (The reader who fears that it is an awful algorithm, whose complexity is factorial, shouldn't worry. In practical cases the dimension of the space is never too big, and for $n = 3$ this algorithm is more efficient than the Gaussian elimination. The numerical stability of the classical Laplace expansion is another issue. . .). If $A = u^{(1)} \wedge \ldots \wedge u^{(p)}$, and $B = v^{(1)} \wedge \ldots \wedge v^{(p)}$, the generalized scalar product $A \cdot B$ is the determinant of the Gramm matrix: $|(u^{(i)} \cdot v^{(j)}|$. Eq. (7) defines a tensor contraction more general than a classical scalar product, not neccessarily symmetric. For example, in 4 dimensions such scalar product of a 2- by a 3-vector gives a 1-vector, while the reverse order product vanishes. It is useful to define a more straightforward full contraction of two p-vectors of the same order by

```
VZ <.> VZ = fromDouble 0.0
x@(S _) <.> y@(S _) = x*y
V u uq <.> V v vq = u <.> v + uq <.> vq
```

which is more efficient than the formula using the Hodge operator. (It could be represented in a more generic way using folds, but we shall not insist upon that.) All classical vector algebra identities, as the reduction of $(\boldsymbol{u} \times \boldsymbol{v})^2$, or $(\boldsymbol{u} \times \boldsymbol{v}) \times \boldsymbol{w}$ take simpler forms, whose derivation we leave for the reader.

3 How to Implement Forms

3.1 Forms as p-vectors over Differential Scalars

We cannot present here an adequate introduction to Differential Forms, see e. g. [7] or [8]. But this **is** an important topic, whose importance in physics and technical sciences became simple explosive, especially in fields where the importance of topology is primordial, e.g. in the analysis of the turbulence. Forms permit to unify all variants of the Stokes' theorem, get rid of the grad/div/curl symbols whose combinations are difficult to read and to memorize, and to formulate many profound and beautiful theorems in differential geometry in a very simple way. *There is no modern theoretical physics without differential forms.*

A 1-form may be written as a "path" element: $\omega = P\,dx + Q\,dy + R\,dz$. It is "something which can be integrated". (A 0-form is a scalar function $f(\boldsymbol{x})$; its "integration" is its evaluation at a point.) A field of directed surface elements (for $n = 3$) is a 2-form: $\alpha = A\,dxdy + B\,dxdz + C\,dydz$. Finally, a 3-form is the volume element $M\ dxdydz$. Algebraically forms span a Grassman algebra with the anticommutative generators dx, dy, etc. All products $dx_k dx_k$ vanish. Because of the antisymmetry the number of components of a p-form is equal to $\binom{n}{p}$ – exactly as for the p-vectors (or $(n-p)$-vectors).

Structurally we might say that a p-form **is** a p-vector of its coefficients, and keep the products $dx_i \ldots dx_j$ implicit. Again, a purist would prefer not to confound forms and vectors, and he would be right, our presentation is simplified. All the linear operations and the skew multiplication is identical for forms and for "standard" vectors.

Now we have to define the differentiation of these objects, so for us they will belong to the type `PV (Scal a)`. For any scalar expression f, i. e. `S (Df f [fx, fy, fz])` the list of partial derivatives converted into a 1-vector generates automatically

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz\ , \tag{8}$$

whose coefficients contitute the gradient of f. In principle we could replace lists in the definition of the `Scal` datatype by the `PV` objects, but at a deeper level the derivatives of the components are *not* vectors nor forms, and we couldn't reuse the vector operations upon them. We need thus a trivial conversion function which transforms `[a,b,c,...]` into `V (S a) (V (S b) (V (S c) ... ).`

3.2 The Exterior Differentiation of Forms

We know how to differentiate scalars. What about $d\boldsymbol{u}$? If the vector $\boldsymbol{u}$ represent the form $u_1 dx + u_2 dy + u_3 dz$, with natural generalization for the n-dimensional case, $d\boldsymbol{u}$ is a 2-form:

$$d\boldsymbol{u} = \left(\frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}\right)dxdy + \left(\frac{\partial u_z}{\partial x} - \frac{\partial u_x}{\partial z}\right)dxdz + \left(\frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z}\right)dydz, \tag{9}$$

which symbolically might be written as $d \wedge \boldsymbol{u}$. The differentiation increases the depth of the form by 1, like the multiplication by 1-vector. Here is the algorithm permitting to differentiate a p-form. First we define the constructor of a differential variable $\boldsymbol{x}$.

```
dfVar l = dV (length l) 1 l where
 dV _ _ [] = VZ
 dV n k (x:xq)=V (S(dfScal n k x)) (dV n (k+1) xq) --where
dfScal n k x = Df x (unitV n k)
unitV n k = (replicate (k-1) (fromDouble 0.0) ++
 ((fromDouble 1.0):replicate(n-k)(fromDouble 0.0)))
```

First we scan recursively all the elements of a form, replacing all scalars by the lists of their partial derivatives.

```
recd VZ = VZ
recd (S (Df _ v)) = listToVector v
recd (V p q) = V (recd p) (recd q)
```

Then, we split the result of `recd` into the derivatives wrt. the first variable, and the rest

```
dsplit VZ=(VZ,VZ)
dsplit (V x@(S _) q) = (x,q)  --The first partial
dsplit (V v q)=let (a,b)=dsplit v; (c,d) = dsplit q in (V a c,V b d)
```

Then, the following code solves the problem, performing the recursive antisymmetrization:

```
df v = asym (recd v)
asym VZ = VZ
asym (V VZ _) = VZ  -- Singleton. No asym.
asym p@(V (S _) _) = p
asym p = let asx VZ _ _ = VZ; asx p r q = V (p - asym r) (asym q)
             (V _ qq, V r1 rq) = dsplit p
         in  asx qq r1 rq
```

The easiest way to check the package is to verify that the Poincaré lemma holds, i.e. that `df (df any)` gives zero (not necessarily scalar or `VZ`. This lemma encompasses known identities, such as $\mathrm{div}\circ\mathrm{curl} = 0$, or $\mathrm{curl}\circ\mathrm{grad} = 0$). It is easy to verify also that $1/|\boldsymbol{x}|$ is a harmonic function in three dimensions. The coding of the Laplace operator for differential forms is trivial, for scalars $\Delta f = *d(*df)$. We begin with any numerical vector constructed as the "variable" $\boldsymbol{x}$:

```
x = dfVar [1.3, -2.1, 0.8]  :: PV (Scal Double)
invr = recip (sqrt (x/.\x))
res = (hodge . df . hodge . df) invr
```

Hugs gives for `res`: -3.1225e-17, (please blame the implementation of the floating-point internals. . .) and `df res` produces a vector with values of the same order. A differentiation operator which lowers the order of its argument form, the composition $\partial = *(d*\square)$: `codf u = (hodge . df . hodge) u`, is called the co-differential. The general Laplace-deRham operator for any form is $d\partial + \partial d$, which can also be coded in a straightforward manner. The use of the package needs some preparations. Suppose we want to compute the normal vector to an implicit surface, e.g. a cone anchored at the origin, whose axis is given by the vector $\boldsymbol{n}$, and the angle between the axis and the surface is θ. The equation and the code of the cone are

$$h(\boldsymbol{x}) = (\boldsymbol{x}\cdot\boldsymbol{n})^2 - c\boldsymbol{x}^2 = 0, \quad \text{where} \quad c = \cos^2(\theta)\ . \tag{10}$$

```
n :: PV Double
n = listToVector [1.0, -1.0, 2.0]
th = 0.85 :: Double
c=z*z where z=cos th
h n c x = sq (x<.>n) - c*>sq (x<.>x) where sq x =x*x
```

and we want to compute this normal for $\boldsymbol{x} = [0.5, 1.0, 2.0]$. But $\boldsymbol{n}$ belongs to `PV Double`, and `c` is a floating constant. So, without changing anything in the definition of the surface, we call it like that:

```
cVec v = fmap (\x->(sCst x)) v          -- (or: cVec = fmap sCst)
surf = h (cVec n) (sCst c) (dfVar [0.5, 1.0, 2.0])
theNormal = df surf
```

where `cVec` lifts a `Double` vector to `PV (Scal Double)`. The result is [2.4264, -16.1471, -4.29426]. Its scalar product with $\boldsymbol{x}$ does not vanish, which proves that the chosen vector does not belong to the surface.

4 Further Manipulations

4.1 Linear Transformations of p-vectors

If $\mathbf{A}$ is a linear transformation which transforms 1-vectors belonging to one space to another: $\boldsymbol{u} \to \boldsymbol{w} = \mathbf{A}\boldsymbol{u}$, the p-products transform naturally as $\mathbf{A}\boldsymbol{u_1} \wedge \mathbf{A}\boldsymbol{u_2} \wedge \ldots \mathbf{A}\boldsymbol{u_p}$. Applying this to the basis vectors, and exploiting linearity implies the existence of a unique "lifting" of $\mathbf{A}$ to the space of p-vectors. This is a hyper-matrix called the p-th compound of $\mathbf{A}$ – a matrix whose elements are all minors of order p constructed from the elements of $\mathbf{A}$. Its dimension is $\binom{n}{p}$, the dimension of p-vectors. We denote it by $\bigwedge^p \mathbf{A}$. It is easy to prove a nice (and non-trivial) property: $\bigwedge^p(\mathbf{AB}) = (\bigwedge^p \mathbf{A})(\bigwedge^p \mathbf{B})$, but how to construct it explicitly? Normal matrices will be represented as vectors of 1-vectors; a particular instance of such an operator is the standard dyadic (tensor, *not* skew) product of two 1-vectors. Our hyper-matrices should have a structural shape appropriate to handle the p-vectors, as normal matrices do with 1-vectors:

```
-- lintrf Matrix Vector -> Vector
lintrf _ VZ = VZ
lintrf (V a1 aq) u = V (a1/.\u) (lintrf aq u)
lintrf VZ _  = VZ
```

The construction of the lifted matrices is quite easy. We see that all the order p minors of our transformation matrix may be obtained by taking the skew products of p of its rows. The only problem is the recursive structuring of the hyper-matrix adapted to the construction of a generalization of `lintrf`. It should be a p-vector of p-vectors. We begin with the construction of a "deep" mapping functional `vmap n v`, not fully recursive, as `fmap`, but adapted to the `PV` sequences of a given depth:

```
vmap 1 _ VZ = VZ
vmap 1 f (V x q) = V (f x) (vmap 1 f q)
vmap n f u = vmap 1 (vmap (n-1) f) u
```

Now, the hyper-matrix constructor and the generalization of `lintrf` have the form

```
hypmat 1 m = m
hypmat n m@(V a1 aq)
 |n>vlength m = VZ
 |otherwise=V(vmap (n-1) (a1/\)(hypmat (n-1) aq))(hypmat n aq)
lintrf p m u = vmap p (<.> u) m
```

where `m` is the hyper-matrix, and `u` – the transformed p-vector. This completes our construction. However, one of the remarkable properties of Forms is their nice behaviour with respect to any coordinate transformations, and here the structural algebra does not suffice, we must return to our differential properties, and to have a look into the *dual* space (not to be confused with the $p \leftrightarrow (n-p)$-vector duality).

4.2 Transformations of Forms

It is the general transformation issue which shows clearly that forms and vectors, such as $\boldsymbol{x}$ belong to different spaces. Until now we have abused the language, here we will try to be *a little* more precise, but this section is a very superficial introduction to the transformation of forms. A zero-form, which is a scalar function f: $f(\boldsymbol{x}) \in \mathcal{R}$ for $\boldsymbol{x} \in \mathcal{P}$ belongs to the **dual** space $\mathbf{F}^0(\mathcal{P})$. If $\boldsymbol{x} \in \mathcal{P}$ is transformed in $\boldsymbol{y} \in \mathcal{Q}$: $\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x}) = \mathbf{T}\boldsymbol{x}$, then the zero-forms undergo the transformation $\mathbf{T}^*$ defined by

$$(\mathbf{T}^* f)(\boldsymbol{x}) = f(\mathbf{T}\boldsymbol{x}) \qquad \text{or} \qquad \mathbf{T}^* \circ f = f \circ \mathbf{T} \ . \tag{11}$$

We see that $\mathbf{T}^* \in (\mathbf{F}^0(\mathcal{Q}) \to \mathbf{F}^0(\mathcal{P}))$. For the 1-forms the situation is a little more complex. A typical term $a_i(\boldsymbol{y})dy_i$ transforms into

$$\sum a_i(\boldsymbol{y}(\boldsymbol{x}))\frac{\partial y_i}{\partial x_j}dx_j \ . \tag{12}$$

The coding is relatively easy, and once we manage to transform 1-forms, the technique used in (4.1) will permit the lifting of those transformations to any p-forms. The implementation of the Eq. $\mathbf{F}^0$, uses (11): `vtrf t f x = f (t x)`. (For those readers who are combinator maniacs: `vtrf = flip (.)`, apart from the type checking.) It may be interesting to note that the abandon of the standard Haskell `Num` classes made it easier to implement directly a partial algebraic structure on forms treated as functions, for example

```
type Form0 a = (PV a -> a)   -- 0 forms
instance AddGroup a => AddGroup (Form0 a) where
 f + g = \x-> f x + g x
 neg f = \x-> neg (f x)
instance Module ((->) (PV a)) where
 a *> f = \x -> a*(f x)
```

etc. (Current versions of Haskell will not accept the instance declaration `Module Form0`.) Now, even if structurally we keep the "normal vector expressions": the values of 1-forms: $\omega = a_1(\boldsymbol{y})dy_1 + \cdots + a_n(\boldsymbol{y})dy_n$ in standard vectors $[a_1, \ldots, a_n]$ belonging e.g. to `PV Double`, or to `PV (Scal Double)`, in order to transform them we must pass to the dual space, to their functional abstraction $\mathbf{F}^1$, as in the case of $\mathbf{F}^0$. In fact, we might abandon the previous version of $\mathbf{F}^0$ altogether: the scalar function f will belong not to $\mathcal{P} \to \mathcal{R}$, but its result will be injected into the PV domain using the S tag. Then we can declare this functional domain as Monoid with the composition playing the role of multiplication. The AddGroup declaration remains identical. *This* representation of the dual space is not vectorial; in order to facilitate the transformation of the coefficients in (12) we might construct another space: `PV (PV a -> a)` of vectors whose components belong to our previous $\mathbf{F}^0$. The conversion between these two possible representations is not difficult, a vector function f should be converted into the vector $[\mathrm{vhd}.f, \mathrm{vhd}.\mathrm{vtl}.f, \ldots, \mathrm{vhd}.\mathrm{vtl}.\ldots.\mathrm{vtl}.f]$, where vhd and vtl are the selectors of the PV fields, but its usage may be very inefficient. Suppose that the function h transforms the coordinates, for example defines the transformation between spherical and Cartesian bases:

```
h (V r (V theta (V phi VZ))) =          -- [x,y,z]
 let rsth = r*sin theta
 in V (rsth*cos phi) (V (rsth*sin phi) (V (r*cos theta) VZ)
```

How to compute the Jacobian determinant of such a transformation for a given value of the argument? Here is the answer, for, say, `arg = dfVar [3.0, pi/6.0, pi/4.0]`:

```
jMatrix = recd (h arg)
jDet = hodge (hypmat 3 jMatrix)         -- = 4.5
```

The matrix `jMatrix` before its application to the coefficient vector a in (12) should be transposed. This is a standard list-processing algorithm which we omit.

The manipulation of Forms in any coordinate system might exploit the fundamental invariance property: $\mathbf{T}^*(d\omega) = d(\mathbf{T}^*\omega)$. We terminate this section by showing that $1/r$ is a harmonic function *in the spherical coordinate system*. The blind application of the previously defined $*d(*df)$ operation to $1/r$ will fail, but **not** because the differential formulae are more complex! In fact, what we need to do is to modify the Hodge star operator: its result should be modified by the intrinsic density element in the new system, i.e. by the Jacobian determinant. This is the entire additional code:

```
ovr = recip (cvector [1.0,0.0,0.0] <.> arg)   -- pick up 1/r invariantly
xhodge u = jDet * (hodge u)
xlapl x = (xhodge . df . xhodge . df) x
xlapl ovr               -- Ugh! Of course -2.498e-16
```

5 Conclusions

The slogan "doing mathematics on a computer" becomes methodologically more serious, if we choose a programming language which permits a decent implementation of hierarchically structured mathematical objects obeing formal properties independently of the underlying data constructors. Ideally the same algorithm should be used both for numerical and symbolic manipulations (unless we need some specifically numeric approximations, which is not the case here). The modern functional languages with their type/class systems offer this possibility. The class system of Haskell is far from the ideal (classes are not categories, types are not domains, neither extensions nor algebraic subsumptions are directly implementable...), but we managed to implement in a fairly universal way a Differential Algebra package, including simple-minded tensors and differential forms. The package can be used directly for numerical computations, avoiding the "formula crunching" stage, so they might be useful for computational physicists [2].

[2] Computational Physicist: a researcher for whom the World is composed of computations, sometimes disguised in palpable things like electrons or galaxies. This is different from Computing Physicist who believes in Reality, but in order to understand it, he must model it on a computer; he has often serious problems with his love affairs.

The differentiation layer *needs* the lazy semantics, the algorithms which compute the (primitive) derivatives are co-recursive, although the construction of the differentials in the domain of forms may be coded strictly. (In an N-dimensional space all differentials must be of order $\leq N$.) The code is very compact, in accord with the basic psycho-philosophy of the Form-alist church, that *all* formulae should be short, otherwise they are the work of Devil. We have shown accessorily that all typical *regular* indicial manipulations are very well representable by recursive iterators. The code would be even more concise, if we used generalized `zips` and `folds`, but it would make the algorithms rather cryptic, impressive, but difficult to read. This work will of course continue, it is possible to implement more tensor manipulations, Lie derivatives, connections, etc. What is essential is to find some concrete, useful examples.

The possible applications are very numerous, ranging from the algorithms for the adaptive sampling of curved surfaces in the 3D modelling, or the light reflection off some anisotropically textured surfaces in image synthesis, to the search of the canonical representation of some thermodynamical or mechanical systems (verification of the order of some forms, which may suggest the existence of turbulence, irreversibility, etc.)

It goes without any doubt that we all need symbolic manipulations, and the computer algebra packages *are* indispensable. Such packages as ours can augment their power in the following way:

- There is *nothing* inherently numeric in our code. Some new symbolic algorithms can be implemented, provided an adequate basic symbolic data are designed, and equipped with the appropriate algebraic properties. The advantage of the functional approach is its genericity and compactness.
- Computer algebra packages are not Oracles. They give the answer sometimes in a badly simplified form requiring some manual post-processing, or they need to be enriched by user-defined procedures, which might contain errors. Direct computations with some random numeric data are an *excellent* way to find bugs in unwieldy formal expressions.

We believe very strongly that the best way to popularize the functional programming among the scientific computing community is by showing how to implement the structures and the algorithms *they* need.

References

1. G. F. Corliss, *Automatic Differentiation Bibliography*, originally published in the SIAM Proc. of *Automatic Differentiation of Algorithms: Theory, Implementation and Application* (1991), but many times updated. Available from the *netlib* archives (`netlib@research.att.com`), or from `ftp://boris.mscs.mu.edu/pub/corliss/Autodiff`
2. Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Proceedings of the III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203.
3. Jerzy Karczmarczuk, *Lazy Differential Algebra and its Applications*, Workshop, III International Summer School on Advanced Functional Programming, Braga, Portugal, 12–18 September, 1998.
4. John Peterson et al., *Haskell 1.4 Report*, Yale University, available from `http://haskell.org/report`.
5. Rinus Plasmeijer, Marko van Eekelen, *Concurrent Clean – Language Report, v. 1.3*, HILT – High Level Software Tools B. V., and University of Nijmegen, (1998).
6. Jerzy Karczmarczuk, *Geometric Modelling in Functional Style*, Proc. of the III Latino-American Workshop on Functional Programming, CLAPF'99, Recife, Brazil, 8-9 March 1999.
7. H. Flanders, *Differential Forms with Applications to the Physical Sciences*, Acad. Press, NY, (1963).
8. C. von Westenholz, *Differential Forms in Mathematical Physics*, North Holland, Amsterdam, (1978).