# Differential Algebra and Lazy Coding

## ("Automatic Differentiation" techniques in functional sauce. . . )

*Jerzy Karczmarczuk*

Dept. of Computer Science, University of Caen.

Differentiation (rather: derivation) is not an *analytic* procedure applicable to *functions*, but an **algebraic** operator. We implement it in Haskell. We show also how to perform the s.c. "inverse mode" differentiation in a curious computational world, in which the time goes *partly* backwards.

## What is *derivation*

A numerical program in which all reused variables have been replaced by fresh ones, and all **`if ...`** decisions taken (and loops unfold), is reducible to a functional chain, which starts with the independent variables $\{x_0, x_1, \ldots, x_p\}$, and then propagates

$$\begin{aligned} x_{p+1} &= f_{p+1}(x_0, \ldots, x_p) \\ x_{p+2} &= f_{p+2}(x_0, \ldots, x_{p+1}) \\ \ldots & \\ x_n &= f_n(x_0, \ldots, x_{n-1}) \end{aligned}$$

Thus, we shall consider it as a series of mathematically sound equations; no "**`x=x+1`**" will spoil our good humour. We have just arithmetic expressions belonging – typically – to such domain as **`Real`** or **`Complex`**. We know how to perform $e_1 \cdot e_2$, $e_1 + e_2$, $\sqrt{1/e}$, etc. What could be the **derivative** $e'$ (or $\mathsf{d}e$) of $e$?

Wait…, we “know” that the derivatives are computed out of *functions*, not expressions (values)! There must be a “differentiation variable” somewhere, and the rest has been learnt at school…

No need to underline the importance of derivatives, gradients, tangents, curvatures, etc. **All** of you needed them one day. But perhaps you missed the fact that the derivation is just a local operator acting on **any** domain with sufficiently rich algebra. It is linear and satisfies the Leibniz rule: $\mathsf{d}\,(e_1 + e_2) = \mathsf{d}e_1 + \mathsf{d}e_2$; $\mathsf{d}\,(e_1 \cdot e_2) = (\mathsf{d}e_1) \cdot e_2 + e_1\mathsf{d}e_2$. Let’s call $\mathsf{d}e$ simply $e'$, provided we defined only one $\mathsf{d}$ (only one “differentiation variable”; otherwise $\mathsf{d}_1$ etc.).

The rule for $(e_1/e_2)' = (e_1'e_2 - e_1e_2')/(e_2)^2$ comes out **automatically**. The rule for $\sqrt{e}$ as well, from $(\sqrt{e})^2 = e$. Exponentials, trigs, etc. – *idem*, e.g. from the series expansion. In general, a derivation can be cooked-up in different ways:

Take the domain of matrices (or other linear operators), and define a special multiplication through the commutator:$A \star B = [A, B] = AB - BA$. Introduce a special object $H$, and name $[A, H] = A'$. From the Jacobi identity $[A, [B, C]] + [B, [C, A]] + [C, [A, B]] = 0$ it is easy to find the translation $(A \star B)' = A \star B' + A' \star B$.

Poisson brackets in mechanics are derivations also. Lie derivatives, covariant ones, etc. in differential geometry as well, even if they are not "normal" ones. . . .

Sometimes the attempt at construction of d gives triviality. From $0 + 0 = 0; 1 \cdot 1 = 1$ we get $0' = 1' = 0$, and from $(n + 1)' = n' + 1'$ we see that for **all** integers the derivatives vanish. Idem for all rationals, so adopting some natural continuity hypothesis, we see that a *numeric* differential algebra is trivial.

So, what can we do in a numerical programs, if all expressions are ultimately numbers (and their place-holders)? Are we obliged to introduce **symbolic** entities, variables $x$, $y$, meaning “themselves”? *By no means* (ei mingil kombel; mitte mingil juhul. . . )

But one of the problems here is that in non-trivial cases, a differential algebra is **infinitely dimensional**. If we can construct $e'$, then also $e''$, $e^{(3)}$, etc., where all these entities may be (and usually are) algebraically independent!

We propose the following solution, presented for simplicity for 1 generator: one “differentiation variable”, say $x$. The generalization is trivial.

- All constants $c$: 2, $\pi$, etc. will be lifted to a compound, infinite structure equivalent to an infinite list $[c, 0, 0, \ldots]$.

- A distinguished expression, **the variable**, say, $x$, shall be converted to $[x, \mathtt{1}, \mathtt{0}, \mathtt{0}, \ldots]$.

- In general, an expression $e$ **semantically** and structurally is lifted to something like $[e, e', e'', \ldots]$. The derivative of such expression is its tail.

We will work with *expressions*, but one may consider that $x$ is the parameter of the function whose derivative we compute. Now, can we *really* work with infinite sequences? How shall we define the operations upon them? The answer proposed is simple an effective, thanks to the **laziness** of the used language – Haskell. Although we have defined some *special* datatypes, in order not to confound them with other entities, in this talk we shall use lists: $[a, b, c, d, \ldots]$ with the operator **(:)** which adds a head to a list: $(a : [b, c]) = [a, b, c]$.

Haskell permits to construct infinite cyclic lists composed of a repeated constant, and it makes it possible to combine iteratively such infinite streams:

```
ones = 1 : ones

zipWith oper (x:xq)  (y:yq) =
   (oper x y) : zipWith oper xq yq
instance Num  => Num [a] where      -- never mind...
   xl + yl = zipWith (+) xl yl     -- overloading
  ...
```

It enables the **co-recursive** algorithms, such as one which computes $[0, 1, 2, 3, \ldots]$ in the following way:

```
integs = 0 : (ones + integs)
```

Yes, it works, for **`integs=[0,a,b,c,d,e,f,...]`**:

```
    1 1 1 1 1 1 1...
  + 0 a b c d e f...
  ------------------
  0 a b c d e f g...
```

You see that `a` is computable directly, from which we get `b`, etc. The magic is that it works automatically, it suffices to traverse the list from its beginning until the last element needed. The remaining are never computed.

Please note that we haven't defined a recursive function, which is also possible:

```
integs = ints 0 where
  ints n = n : ints (n+1)
```

but **recursive data**. This will be now used for the manipulation of the expressions (numbers) lifted to the stream domain, and constituting a closed differential algebra. We know already that adding or subtracting infinite streams does not present any difficulty. What about the multiplication? We use the following syntactic contraption in Haskell. A pattern (function argument) which has the structure `s` and we want that it be *named* `n`, is written as `n@s`.

Here you are the multiplication with $f = [e, e', e'', \ldots] = (e : \bar{e})$, where, naturally, $\bar{e} = [e', e'', \ldots]$.

$$f_1@(e_1 : \bar{e}_1) \cdot f_2@(e_2 : \bar{e}_2) = (e_1 \cdot e_2 : \bar{e}_1 \cdot f_2 + f_1 \cdot \bar{e}_2)$$

Division:

$$f_1@(e_1 : \bar{e}_1)/f_2@(e_2 : \bar{e}_2) = (e_1/e_2 : (\bar{e}_1 \cdot f_2 - f_1 \cdot \bar{e}_2)/f_2^2)$$

All the algebraic and transcedental expressions can thus be differentiated using standard, scholar calculus. **Note that these formulae involve infinite, non-terminating recursion!**

We underline that we operate upon *structures*, compound data, not symbols. In order to make all it practical, the language should permit the **overloading of arithmetic operations**. (In Haskell this is possible thanks to the *type classes*.) Here you are some more:

$$\sqrt{f@(e : \bar{e})} = \left(\sqrt{e} : \frac{1}{2}\frac{\bar{e}}{\sqrt{f}}\right)$$

or, in a bit optimized form:

$$\begin{aligned}
\sqrt{(e:\bar{e})} &= g \quad \textbf{where} \quad g = (\sqrt{e} : \frac{\bar{e}}{2g}) \\
\exp((e:\bar{e})) &= g \quad \textbf{where} \quad g = (\exp(e) : \bar{e}\cdot g) \\
\sin(f@(e:\bar{e})) &= (\sin(e) : \bar{e}\cdot\cos(f)) \\
\cos(f@(e:\bar{e})) &= (\cos(e) : -\bar{e}\cdot\sin(f))
\end{aligned}$$

etc. Now we can write any “standard” function definition, say

```
sinh x = (exp x - exp (negate x))/2
```

and its application to, say: **`sinh (1.5 :  1.0 :  zeros)`** gives the hyperbolic sine *and* cosine, followed by an infinite number of higher derivatives: 2.12928, 2.35241, 2.12928, 2.35241,... This is faciliteted because Haskell **overloads automatically** the numerical constants, no need to write **`(1.5 :  zeros)`** for the constant 1.5.

## Some applications

A closed algebra permits not only to compute derivatives, but to **use** them in algorithms, e.g. to solve differential recurrences, or to compute some functions as their Taylor series (not necessarily convergent, below we have $[0, 1, -2, 9, -64, 625, -7776, 117649 \ldots]$).

Take the definition of the Lambert function $W(z)$, which is nasty, given implicitly by a non-invertible identity: $W(z)e^{W(z)} = z$. We can write

$$\frac{dz}{dW} = e^{W}(1 + W) \qquad \text{and} \qquad \frac{dW}{dz} = \frac{e^{-W}}{1 + W}$$

which gives us the McLaurin expansion of $W$ in one line:

```
wl = 0.0 : exp (negate wl / (1.0 + wl))
```

A differential equation $Ay''(x) + By'(x) + Cy(x)$ is usually solved for $y''$, and integrated twice to get $y$. But all acquainted with Bessel functions know that this is difficult in view of singularities for $x = 0$. The function $u(x)$ defined through $u(x^2) = x^{-\nu} J_\nu(x)$ obeys

$$u'(x) = -\frac{1}{\nu + 1}\left(x^2 u''(x) + \frac{1}{4}u(x)\right) .$$

The only way to find the power series for $u$ is perverted a bit, requiring $u$ (locally) as well as $u''$, and to compute $u'$. This is doable, because the unknown terms of $u''$ are "protected" by $x^2$, the series $x^2u''$ starts as $[0, 0, a, \ldots]$. We code, with $\zeta(f) = z \cdot f$ for $z = 0$:

```
ubes = 1.0 : fp where
 fp = negate (0.25*ubes + zeta (zeta (df fp))
 zeta f = 0.0 (f + zeta (df f))
 df (_ : eb) = eb
```

We could show you some dozen of examples where the laziness is essential in the structuration of the algorithms, not only for the representation of ‘augmented expressions’. After all, lazy data is a **representation of a process** rather than a static entity. We shall pass thus to another way of seing things, to the **reverse mode** of algorithmic differentiation, where expressions transmute to explicit, lazily evaluated **functional objects**. Prepare yourselves to one of the craziest programming tricks you ever saw, and yet perfectly natural, and – in a sense – implemented even in Fortran and used by engineers.

The technique proved its utility mainly in $N$-dimensional case, for the solution of sensitivity problems (reactivity of a meteorologic or industrial process to small changes of initial conditions), but for simplicity we treat a 1-dim context.

Suppose that a program yields some final result $f$. For *all* variables $x$, $y$ (initial and intermediate), we introduce their **adjoints**: $\widehat{x} = df/dx$, etc. We need the adjoints of *initial* variables. Of course they cannot be known until the final equation $f = \ldots\ (x)\ \ldots$ is processed.

It is easy (although not entirely trivial) to show that a fragment (definition) in a program:

$$y = g(x_a, x_b, \ldots, x_p)$$

permits to **specify/update the adjoints of variables at the RHS**:

$$\widehat{x}_a \leftarrow \widehat{x}_a + \widehat{y} \cdot \frac{\partial g}{\partial x_a}, \quad \widehat{x}_b \leftarrow \widehat{x}_b + \widehat{y} \cdot \frac{\partial g}{\partial x_b}, \quad \text{etc.}$$

So, in order to compute $\widehat{x}_a$ we have to have $\widehat{y}$. But this quantity will be specified **later**, when $y$ will be **used**!

So, the algorithm goes as follows: We trace the program before or during its execution, and for each definition of a variable we *construct and memorize* the equations for the adjoints of the RHS variables. They are **not** executable yet.

When we construct the final result $f$, we have automatically $\widehat{f} = 1$ (by definition). This permits to construct effectively the adjoints of all variables upon which $f$ depends directly. The initial values of these adjoints are equal to zero.

The sequence of adjoint equations is retraced from the end, down to the beginning. That's why they must be stored in a linear data structure, called usually "the tape", written forward, but executed backwards.

Exemple. Our program is

$$y = \mathsf{sin}(x); \quad z = y^2 - x/y$$

with $x$ independent. The adjoint equation set is

$$\begin{aligned} z = y^2 - x/y; \qquad & \text{yields} \qquad \widehat{x} \leftarrow \widehat{x} + \widehat{z}(-1/y); \\ & \qquad\qquad\quad\; \widehat{y} \leftarrow \widehat{y} + \widehat{z}(2y + x/y^2); \\ y = \mathsf{sin}(x); \qquad & \text{yields} \qquad \widehat{x} \leftarrow \widehat{x} + \widehat{y}\,\mathsf{cos}(x)\,. \end{aligned}$$

Finally $\widehat{x} = -1/\mathsf{sin}(x) + \mathsf{cos}(x)\left(2\,\mathsf{sin}(x) + x/\mathsf{sin}(x)^2\right)$ is the desired value of $dz/dx$, if we begin with $\widehat{x} = \widehat{y} = 0$.

Now, how to implement this? We want to have a superficially one-pass algorithm, with no external ‘tapes’, and purely functional, so the update operator “$\leftarrow$” may be a nuisance!

## Antitemporal State Monad

This is a digression about the ‘stateful’ programming in functional framework. How to compute things with side-effects, e.g., an application of any function should increment **one** global counter? The answer is based on the monadic approach, but we won’t speak anymore about monads. Just assume that “things” (data), say $e$, are “lifted” to a new domain of **functions** which act on some “mysterious” entity called the **state** (which can be the value of the counter). This function returns a *pair* containing the “normal” value, and, a possibly modified state (say, the incremented counter).

A piece of **static data** $e$ (say, a constant) cannot – obviously – do anything with the state, so it gets lifted into $\tilde{e} = \lambda s \rightarrow (e, s)$. In order to get anything meaningful, we must apply our lifted object to the current state, and we recover its value, together with the state.

Now, a (normal) function $f$ which acts on $e$ producing $e' = f\ e$ must be augmented **by the user** in a way that it acts on the state as well. In our silly counter model $f$ becomes $\overline{f}$ with the semantics $\overline{f}\ e\ s = (f\ e, s + 1)$. As simple as that. **Here primes are not derivatives!!**

In a more complicated state model, which may pick some information from the data and influence the data, we will have in general $\overline{g}\ e\ s = (e', s')$ some new data value, and a new state. What it *really* does depends on the computation the user wants, nobody can decide for him. **YOU define $\overline{g}$!**

But the system can help to **lift** such functions so that they can process other lifted expressions, say $\tilde{g} = \mathsf{lift}\ \overline{g}$. How such a function works?

$$\tilde{g}\ \tilde{e} = \lambda s \rightarrow \mathbf{let}\ (e, s') = \tilde{e}\ s\ \mathbf{in}\ \overline{g}\ e\ s'\ .$$

Or, perhaps more verbosely:

$$\tilde{g}\ \tilde{e} = \lambda s \rightarrow \ \begin{array}{ll} \textbf{let} & (e, s') = \tilde{e}\ s \\ & (e', s'') = \overline{g}\ e\ s' \\ \textbf{in} & (e', s'')\ . \end{array}$$

We read it as follows: $\tilde{e}$ acts on the **initial state** $s$ producing the **intermediate one** (here: $s'$) which can be equal to $s$ but not necessarily if $\tilde{e}$ is already a complex form, not a lifted static datum. It produces also the associated 'value' $e$.

Then, $\tilde{g}$ acts (through its hidden kernel $\overline{g}$) on this value, and on the intermediate state $s'$, and produces a changed value, say, $e'$, associated with the **final state** $s''$. Cool and simple, isn't it?

Now, fasten your belts. Wadler proposed (casually, without really thinking hard on its applications. . . ) a version in which the time goes backwards in the following sense. The function $\tilde{g}$ is supposed to act on the final state producing the intermediate one. The expression $\tilde{e}$ acts on the intermediate state, and recovers the initial one.

But wait: the function $\tilde{g}$ needs for its effective execution its 'principal' argument $e$, which will be produced by $\tilde{e}$. Unfortunately, $\tilde{e}$ needs the intermediate state which will be produced by $\tilde{g}$. In two words:

$$\begin{aligned} \tilde{g}\ \tilde{e} = \lambda s'' \rightarrow \quad & \mathbf{let} \quad (e, s) = \tilde{e}\ s' \\ & \phantom{\mathbf{let}} \quad (e', s') = \bar{g}\ e\ s'' \\ & \mathbf{in} \quad\ (e', s)\ . \end{aligned}$$

Note the cross-referencing between $e$ and $s'$. No headache yet? You are not a human, then. . .

But, believe or not, this is an exotic form of what is known elsewhere as the calculus of adjoints. The error back-propagation through neural networks may be represented through this form. The "inverse kinematics" in animation and robotics, which computes the forces, tensions and reactions of joints resulting from a given movement, is a variant of that. Moreover, this kind of crossed, antithetic data dependencies is **well known** in the compilation theory!

It is known that during the parsing process, the "inherited attributes" (coerced types, contextual and positional information, etc.), go from the root of the parsing tree towards its leaves. But a LR bottom-up parsing algorithm marches from the leaves, and it constructs the root at the end!

We have "two time arrows" in the system, one of which goes against the other one. . .

The implementation of such concepts is usually done *ad hoc*, but there is nothing bizarre in it. The crux of the matter is that the "future data" are normally not needed immediately, it suffices to have a "promise" that they will be delivered when the time comes. You can pay with a doubtful cheque before the end of the month, can't you?

In the classical treatment of the reverse mode, the program is executed, and the adjoint statements are constructed, but not executed, stored on a "tape". In our model the adjoints constitute the "state".

We construct **lazy** forms, which wait until the adjoints of their children will be available, when the children procreate their own. Finally, the last generation child, the final result, has the adjoint equal to one, and the evaluation process may be finally effectively executed. It will take some time, sometimes a lot of time, but for the user this is transparent.

## The model

We define a special Haskell type

```
newtype Ldif a = Ld (a->(a,a))
```

This is the "state monad" tagged by a symbol **`Ld`**. We introduce two banal lifting functions for static data, constants and *the* variable:

```
lCnst c = Ld (\z->(c,0))
lDvar x = Ld (\z->(x,z))
```

The last form says intuitively that if $z = x$, then $\hat{x} \leftarrow \hat{x} + \hat{z}$. We shall define now some lifted forms for the standard functions and binary operations. Don't try to memorize them, the construction was a painful translation of the general idea.

```
exp (Ld pp)=Ld(\n->let (p,pb)=pp(w*n);w=exp p
                   in  (w,pb))
recip (Ld pp) = Ld(\n ->               --  "1/pp"
  let (p,pb)=pp eb; w=recip p
      eb=negate (w*w)*n
  in  (w,pb))
sqrt (Ld pp) = Ld(\n ->
  let (p,pb)=pp eb; w=sqrt p
      eb=(0.5/w)*n
  in  (w,pb))
```

and for an arbitrary function **`f`** whose *formal* (structural) derivative **`f’`** is known (e.g., sin → cos), we have a generic lifting operator

```
llift f f’ (Ld pp) = Ld (\n ->
  let (p,pb)=pp eb; eb=(f’ p)*n  in (f p,pb)
```

It suffices thus to define **`log = llift log recip`** under the Haskell system of class instances, i.e. for the overloading of log: lifted, LHS, expressed through the standard logarithm at the RHS.

What about binary functions? This isn't too difficult neither, just requires that the programmer know what he is doing. We remind that an adjoint equation is a *modification* of a variable: $\widehat{x} \leftarrow \widehat{x} + \widehat{y} \cdot (\ldots)$. In our construction the lifted operators gather the contributions from *all* adjoint equations involving $\widehat{x}$; they might be numerous, since $x$ could have been *used* many times. We won't discuss the details for humanitarian reasons…

Finally we get the following definitions, easy to interpret when you try to construct the adjoint equations from, say, $z = x+y$, or $z = x\cdot y$. At the end of the program one has to *apply it* to 1.

```
negate (Ld pp)=Ld (\n ->
  let (p,pb)=pp (negate n) in (negate p,pb))

(Ld pp)+(Ld qq) = Ld (\n ->
  let (p,pb)=pp n;  (q,qb)=qq n
  in  (p+q, pb+qb) )
(Ld pp)-(Ld qq) = Ld (\n ->
  let (p,pb)=pp n;  (q,qb)=qq (negate n)
  in  (p-q, pb+qb) )
(Ld pp)*(Ld qq) = Ld (\n ->
  let (p,pb)=pp (n*q);  (q,qb)=qq (p*n)
  in  (p*q, pb+qb) )
(Ld pp)/(Ld qq) = Ld (\n ->
  let (p,pb)=pp (recip q*n);  (q,qb)=qq eq
      eq    =negate (p/(q*q))*n
  in  (p/q, pb+qb) )
```

## Conclusions

All this works, although cannot be considered as an industrial-strength package, this wasn't our objective. We have shown two very different faces of the power of lazy codes.

Lazy functional programming is an immense fun, inspiring and profound, giving the opportunity to see statically and mathematically the intricacies of the dynamics, of processes, usually treated imperatively. It is an ambitious enterprize if you really want to do something new, but you might have a chance to discover some common features of many, apparently unrelated branches of science. The reason is that functional programming relies on strong, generic mathematic properties, and when you start exploiting them through concrete implementations, it is easier to See the Light.

Thank you. Suur tänu.