Lazy Time Reversal, and Automatic Differentiation

Jerzy Karczmarczuk
Dept. d'Informatique, Université de Caen, France
karczma@info.unicaen.fr

Abstract

We show how to implement functionally the reverse, or adjoint strategy within the domain of Automatic Differentiation techniques – tools permitting to compute numerically, but exactly (i.e. up to the machine precision) the derivatives of functions coded by computer programs. The imperative coding of the reverse techniques is tedious. It requires the reversal of the control thread of the program, and it recomputes the derivatives through the *adjoint statements* beginning with the definition of the final result, and ending at the independent variables. Usually a special external data structure, known as the "tape" is used to store the adjoint statements. It is created during the "forward" stage of the program, and then interpreted backwards. We show how to construct purely functionally the equivalent of such a tape, but we present also a more interesting model based on a variant of Wadler's backward propagating State Transformer monad. Our package, written in the lazy functional language Haskell, uses the overloading of standard arithmetic operations and is very simple to use, permitting the calculation of M-dimensional gradients, and also of higher derivatives.

1 Introduction

1.1 The arrow of time

At the fundamental level almost everything in this world (strong interactions, electromagnetism and classical gravitation) is neutral with respect to time reversal. If a theory (e.g., relativistic quantum physics) predicts some acausal phenomena, such as particles propagating backward in time, they can be always reinterpreted as anti-particles behaving "normally" (but whose energy gets a 'minus' sign...). Only for composite, statistically described systems such notions as entropy, irreversibility and arrow of time enter into play in a critical way. We are used to this, and such problems are not *en vogue* anymore.

However, for a computer scientist the entropy which is *information*, is a fundamental, elementary concept, and the causality, the changing of states and chronologically sane scheduling of events constitute the basis for any "natural" programming framework. The computations are essentially irreversible, the states propagate forward in time, and the mapping between the "computation time" (the ordering of events) and the physical one is natural. (Of course, in the realm of quantum computing all those questions became again very interesting and non-trivial, but we shall not treat those issues here.)

Both in physics and in computations it is not the arrow of time itself which make the story interesting, but a speculation what would happen if *both* time directions coexisted, if the world con-

tained two *antithetic* flows of temporal entities... . In physics it is relatively simple, as mentioned above, the particles and antiparticles coexist; when two of them annihilate (or are created out of energy), it may be interpreted as the change of the propagation direction of one of them. But if the creation/erasing of information is involved, the result is a severe headache, see the science-fiction book of Philip Dick [1], where the author suggests that rising of the dead is a bit simpler than un-writing a book... .

However, the "reinterpretation" of anti-causal flows, the computation models where antithetic flows of data dependencies coexist are not so rare, although they are usually presented in a less exotic manner. In general they belong to the domain of *goal-oriented* computing.

- A bottom-up parser which transforms a program source into a syntactic tree, synthesizes the semantic attributes of nodes from their descendants. But the inherited attributes descend from the root down to the leaves! Of course, technically this is a well understood and solved problem, but conceptually it is slightly "crazy", the construction of the root belongs to the future with respect to the processing of leaves, the inherited attributes "propagate backward in time". We shall return to this model, see [2].

- A common technique in the domain of creating animations, and in robotics is the "inverse kinematics". It consists in finding the driving forces which should be applied to the articulations and intermediate moving parts (e.g. the elbow) of a physical or simulated robot, in order to achieve the goal: to construct the trajectory of the final effector (its hand). This is a typical *adjoint* problem which may be formulated as anti-causal, see [3, 4, 5].

- The back-propagation of error in multi-layer neural networks is also a classical adjoint, "counter-clock-wise" process, conceptually quite similar to the reverse differentiation procedure mentioned below, see [6, 7].

- Finally, the reverse (adjoint) mode in the automatic differentiation technology, which is the main subject of this paper, is a classical example of a non-orthodox view of processes propagating in time.

1.2 Automatic differentiation

We shall present a lazy functional implementation of the *Automatic Differentiation* (AD) technique – an established solution to get accurate and fast derivatives of numerical expressions represented by programs. The AD methods are based on two principles:

- any program (streamlined along the chosen decisional paths, i.e., with all the conditionals and eventual branching resolved) can be seen as a composition of primitive functions whose derivatives are known,

- and can then be differentiated using the chain rule: $df(g(x)) = f'(g(x))dg(x)$.

The derivatives are calculated numerically by an *augmented* original program. The results are as precise as those obtained by symbolic methods, (no finite differences are ever used), and they are generated much faster. There is no need for symbolic indeterminates, term simplification, or other manipulations typical for the Computer Algebra approach to the differentiation. The augmented program may be output by a preprocessing package, or result from a modification of the original

program semantics, through the use of overloaded arithmetic operators acting upon composite data structures which contain the original numerical values, and also their derivatives. There are 2 principal modes of AD:

1. The forward mode in which the intermediate derivatives are computed in the same order as the program combines its component subexpressions. This is the most classical approach, the differentiation machinery behaves as a human who would augment the code by additional instructions computing the derivatives (and reusing the shared expressions assigned to temporary variables).

2. The reverse, or *adjoint* mode, in which the intermediate derivatives are computed in the reverse order, *from the final result down to the independent variables*. The reverse mode is better for computing multi-dimensional gradients of *one* function, because its complexity is (in principle) independent of the number of input variables.

The differentiation of numerical computer programs is a domain which is at least 30 years old, and its applications are numerous. The bibliography and the number of relevant software packages are huge, see [8, 9, 10], and the information stored on the World-Wide-Web [11]. The necessity of computing the derivatives fast and precisely is obvious for everybody active in the domain of scientific and technical computations, and the practical aspects of this research resulted in its concentration on mainly such languages as Fortran and C/C++. Functional languages from this perspective are less popular. However, in [12] we have shown some potential advantages of lazy functional programming, our package written in Haskell permitted to compute the derivatives of *any* order in a particularly easy way, almost transparent to the user.

We present now the adjoint model, and we begin with an example, for simplicity we restrict the presentation to the 1-dimensional case. Suppose that we need $z'(x)$ given by the program:

$$y = \sin(x); \quad z = y^2 - x/y \tag{1}$$

Here x is the independent variable, y is auxiliary, and z is the final, scalar result. For *all* (relevant) variables ζ (here: x, y, z) we define their *adjoints*: $\overline{\zeta} = \partial z/\partial\zeta$. In general, if the program contains an intermediate assignment $\xi = f(\zeta, \eta, \ldots)$, this implies the following assignments for the adjoints (note their imperative character; the adjoints are *updated*):

$$\overline{\zeta} \leftarrow \overline{\zeta} + \overline{\xi}\frac{\partial f}{\partial \zeta}; \qquad \overline{\eta} \leftarrow \overline{\eta} + \overline{\xi}\frac{\partial f}{\partial \eta}; \ldots \tag{2}$$

but the adjoint statements must be processed in reversed order, ξ needs ζ, but $\overline{\zeta}$ needs $\overline{\xi}$ which is *defined* when ξ is *used*. The differentiation machinery should produce the value of $\overline{x}$, easily generalizable for the case of many independent variables. We know immediately that $\overline{z} = 1$, but this data can be used when z is known, so the computational process decomposes into two antithetic phases:

First y and z are computed, during the "forward phase" of the program, and then the control thread is retraced back, beginning with the trivial initial assignments: $\overline{z} \leftarrow 1$; $\overline{x} \leftarrow 0$; $\overline{y} \leftarrow 0$:

$$\begin{aligned} z = y^2 - x/y; \quad & \text{yields} \quad \overline{x} \leftarrow \overline{x} + \overline{z}(-1/y); \\ & \qquad\quad \overline{y} \leftarrow \overline{y} + \overline{z}(2y + x/y^2); \\ y = \sin(x); \quad & \text{yields} \quad \overline{x} \leftarrow \overline{x} + \overline{y}\cos(x)\,. \end{aligned} \tag{3}$$

So, finally $\overline{x} = -1/\sin(x) + \cos(x)\left(2\sin(x) + x/\sin(x)^2\right)$ is the desired value of $\partial z/\partial x$. In a strict language the second phase code must be physically constructed, and executed when the values of all variables are known. Our lazy solution is a "circular program", an application of the idea of Richard Bird, who shows in [13] how to code a one-pass algorithm operating upon data which will be known later.

We present now the general framework used in the implementation. We assume that $(x_1, x_2, \ldots, x_M)$ is the set of independent variables. Having streamlined the control structures, we can model a typical numerical program by a set of functional definitions:

$$\begin{aligned} x_{M+1} &= f_{M+1}(x_1, \ldots, x_M)\,, \\ x_{M+2} &= f_{M+2}(x_1, \ldots, x_{M+1})\,, \\ &\ldots \\ x_N &= f_N(x_1, \ldots, x_{N-1})\,, \end{aligned} \tag{4}$$

where for uniformity we have named "x_p" all the intermediate expressions. This set may be completed by $x_k = f_k()$, for $k \le M$. The last few of the equations (4), perhaps just the last one, determine the final outcome of the program. The functions f are typically very sparse, we can reduce everything to unary or binary operators (increasing appropriately the number of intermediate variables).

For each instruction $g \leftarrow f(e_1, e_2, \ldots, e_k)$ the adjoints of the RHS arguments are computed by

$$\overline{e}_j \leftarrow \overline{e}_j + \overline{g}\frac{\partial f}{\partial e_j}\,. \tag{5}$$

The derivatives are defined by the chain rules obeyed by the Jacobi matrices:

$$\mathbf{J}_{ik} = \frac{dx_i}{dx_k} = \delta_{ik} + \sum_{j=k}^{i-1} \frac{\partial f_i}{\partial x_j}\frac{dx_j}{dx_k}\,. \tag{6}$$

This equation gets the form $\mathbf{J} = \mathbf{I} + \mathbf{D}\mathbf{J}$, where

$$\mathbf{D}_{ik} = \frac{\partial f_i}{\partial x_k} = \begin{pmatrix} 0 & 0 & 0 & \ldots \\ \partial f_2/\partial x_1 & 0 & 0 & \ldots \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & 0 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{7}$$

By following the chain, beginning with x_{M+1}, and terminating at x_N, we may calculate $\mathbf{J}_{ik}$ iteratively, and this is the standard "forward" mode. But it is possible also to start with the *last* partial derivatives, and to follow the chain backwards. This is interesting from the efficiency point of view, if we need only the last row of the Jacobi matrix, i.e., the elements $\overline{x}_k = \mathbf{J}_{Nk} = dx_N/dx_k$. (This is the typical case for the *sensitivity analysis* (the dependence of the solutions on the set of initial conditions and system parameters) of technical or natural processes: nuclear reactor performance, meteorology and oceanography, biosphere development, etc., where *one* result depends on many inputs.)

We see immediately that $\mathbf{J}$ and $\mathbf{D}$ commute, and if we rewrite the equation for $\mathbf{J}$ in its adjoint form: $\mathbf{J} = \mathbf{I} + \mathbf{J}\mathbf{D}$, or

$$\mathbf{J}_{ik} = \delta_{ik} + \sum_{j=k+1}^{i} \mathbf{J}_{ij}\mathbf{D}_{jk}\,, \tag{8}$$

we get for its last row the equation

$$\overline{x}_k = \delta_{Nk} + \sum_{j=k+1}^{N} \overline{x}_j \mathbf{D}_{jk} \,. \tag{9}$$

We see that the reverse mode generates a very non-trivial modification of the program control flow. The AD packages which operate in reverse mode, [14, 15] and many others, usually perform an involved source-to-source transformation. This machinery may be heavy; the execution of resulting program stores the intermediate expressions evaluated during the "forward sweep", and needed for the main computation, and for the evaluation of $\mathbf{D}$, on a sequential (internal or external) data structure nicknamed as the "tape". After having computed the final value, the tape is read backward, and the program reconstructs all the adjoints. This strategy may be dangerous: if the program executes an iterative loop, each new re-assignment of an intermediate variable is functionally equivalent to the creation of a new instance of it and of its adjoint. Even if the variable is reused (i.e., in a functional, tail-recursive procedure: it is replaced by the new instance), its adjoint, or rather *their* adjoints, cannot. Each instance generates a new adjoint statement, and a new entry on the tape, which may become very long. The time "transmutes into space". Many essential optimisation strategies have been proposed, see e.g. [16, 17], but the automatization of these techniques is not easy.

At any rate, if we consider the adjoints as entities which belong to the *state* of the system, we see that this state propagates backwards with respect to the "natural" control flow of the program. In view of the importance of the reverse differentiation techniques for the scientific community, we observe that the remark of Philip Wadler in his paper [18], which will be exploited and commented in the section 2.1: "*To make this change in an impure language is left as an exercise for masochistic readers*", might be considered a little too cruel... .

2 Doing It Functionally

Our ambition is to use some lazy functional techniques to make the life simpler for a casual, non-masochist, scientifically oriented user, who would like to compute the adjoints in his program *easily*, without passing through external pre/post-processing packages, and without having to introduce manually some substantial modifications to his code. At the first glance, the problem doesn't seem too easy. We have not only a "perverse" control flow to implement, but we see that the adjoints are computed incrementally; they gather additive contributions from each expression containing the referred variable, and such constructs seem to be *par excellence* imperative.

2.1 'State' Monads in Haskell

The standard State Transformer monad, currently used to model some IO and other imperative constructs is based on the lifting of all expressions of type `a` to the domain of "computations" `(\s -> (a,s))`, where `s` describes the type of the state. Every expression becomes a function which *acts* on the current state, and produces a resulting value and a new state. A functional call $k(x)$ is replaced by the form `m >>= k`, where `m` is the lifted computation which delivers the value `x` upon acting on some state. The function `k` acts on x and on a new state, and yields another lifted object, like in the explicit definition below:

```
m >>= k = \s_initial -> let (x, s_middle) = m s_initial
                            (y, s_final)  = k x s_middle
                        in  (y, s_final)
```

Philip Wadler in his article [18] demonstrated the possibility to define this monad in such a way that the computation acts on the final state, and when the program delivers the final value, one retrieves the initial state. The modification of the "bind" operator **(>>=)** is formally simple:

```
m >>= k = \s_final -> let (x, s_initial) = m s_middle
                          (y, s_middle)  = k x s_final
                      in  (y, s_initial)
```

but its meaning a bit less. The function **k**, which acts on the final state, and produces an intermediate one needs the value provided by the computation **m** which acts on this intermediate state. *The data dependencies obey the "forward" time arrow*, the two antithetic flows coexist, and it is implementable only in a lazy language, since the two internal **let** clauses are mutually recursive.

Wadler remarks that this monad appeared while analysing the cross-referencing dependencies during some intelligent text processing, but general applications of this monad seem to remain unexploited. We leave the inverse kinematics and neural networks to some future work, and here we propose to define the final state as the value of the adjoint of the final value, i.e., 1 (again, it is simpler to present the one-dimensional case), and the initial state is the adjoint of the input (differentiation) variable. When the program begins, we obviously have to inject into it this variable and its adjoint. However, the adjoint is never really needed until the end of the program. As already mentioned, our "time vehicle" belongs essentially to the same category of lazy tricks as those presented in the article of Bird [13], and reformulated many times since.

2.2 Lazy time reversal, and associated data structures

We shall keep one global state, the value of the adjoint, which belongs (in one dimension) to the same type as all other numerical expressions, say, **a = Double**. The corresponding monadic data type would be **a -> (a,a)**, but we define a new datatype **Ldif**, since it is then easier to define the overloaded operations for it. Its declaration, and the lifting of "constants" (explicit numeric values, or expressions which do not depend actively on the input variable), and of the variable itself go as follows:

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

lCnst c = Ld (\z -> (c, 0.0))
lDvar x = Ld (\z -> (x, z))
```

While the Wadler's monad is one of our sources of inspiration, and the composition of expressions *is* essentially a monadic chaining, our ambition is to augment the semantics of a typical numerical program without changing too much its form. A casual user should be able to write normal arithmetic expression with standard operators and elementary function calls, and never see explicit 'binds'. This means that the **Ldif** datatype will be an instance of the **Num** class, and the standard numeric conversion function, e.g. **fromInteger** may be specified as **lCnst**, which, with the aid of the compiler simplifies the coding of a program containing explicit numeric constants.

The *generic* lifting of unary and binary functions follows the anti-causal monadic chaining shown above. A function **f** acting on a numerical value is lifted into **llift f f'**, where **f'** is

$\frac{df}{dx}(x)$. A binary operator needs for its lifting two partial derivatives, denoted as **f1'** and **f2'**. If there are no special shortcuts for particular operators, we use the generic forms applicable to **Ldif** arguments:

```
cos=llift cos (negate.sin); log=llift log recip
```

etc., where

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

dllift f f1' f2' (Ld pp) (Ld qq) =
  Ld (\n->let (p,pb)=pp ep;    (q,qb)=qq eq
              ep=(f1' p q)*n;  eq    =(f2' p q)*n
          in (f p q, pb+qb) )
```

However, standard numerical operations are optimised, and quite short, although they are not so easy to grasp by an unprepared reader.

```
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) )

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

etc. Notice that the summing of the derivative members in the resulting tuples provides the "imperative", accumulating updates. In order to apply practically our techniques it suffices to construct numerical functions which are sufficiently generic, and can be overloaded to the **Ldif** domain, for example **cosh z = let e=exp z in (e + recip e)/2**, and to apply them to,

say, **`lDvar 1.3`**. The result is a functional object which should be applied to 1.0 in order to give the main value (the hyperbolic cosine) and the derivative (the hyperbolic sine), absolutely "for free".

Of course it is possible to compute second and higher derivatives with the presented techniques, it suffices to define some objects belonging to the type **`Ldif (Ldif Double)`**, etc., and to extract the appropriate final values. Such generalisation exploits the polymorphism of Haskell, and the possibility to compose the types recursively; the recursive composition of AD techniques in standard imperative languages are more difficult to implement, even if the language permits to overload the arithmetic operations.

The package has been generalized to many dimensions, and we could compute with it not only the gradients, but also the Hessians: $\partial^2 x_N / \partial x_i \partial x_k$, but this generalisation is postponed to the section 4.2.

3 Relation to Attribute Grammars

We have mentioned already that there is nothing *really* new in the presented strategy. The backward propagation of a state is a phenomenon known for many years in the domain of compilation (syntax-driven semantic analysis), and corresponds to the propagation of inherited attributes during the bottom-up parsing. A syntactic rule:

$E ::= E_1$ **Op** E_2

drives the synthesis of the attributes of E, but it is also here that the inherited attributes of E_1 and E_2 are assigned. Within the top-down parsing strategy the non-terminal E becomes a parsing function, and we might parameterise it by the inherited attributes of its RHS components. But the ascending algorithm, which can be treated as a kind of symbolic but "natural" (from leaves up to the root) construction of the parsing tree, gets into trouble, because while the evaluation proceeds from the leaves upwards, the inherited attributes descend from the root.

This problem viewed from the perspective of lazy functional programming has been analysed by Johnsson, [2]. If we denote by E_S a synthesized attribute, e.g., the value of the expression E, and by E_k_I an inherited attribute (e.g., some contextual information, environments, etc.), then the set of semantic decorations (assignments of the attributes) can be replaced by the creation of one synthesized attribute E_f which is a functional object defined by the following program (assuming that each variable has two synthesized and one inherited attribute):

$$
\begin{array}{lll}
E\_f = \lambda & E\_I \rightarrow & \\
 & \textbf{let} & (E_1\_S_1, E_1\_S_2) = E_1\_f\ E_1\_I \\
 & & (E_2\_S_1, E_2\_S_2) = E_2\_f\ E_2\_I \\
 & & \{\ldots \text{ attribute definitions } \ldots\} \\
 & \textbf{in} & (E\_S_1, E\_S_2)
\end{array}
$$

where we see that typically E_S will depend on E_k_S, and since E_k_I depend on attributes of E, the definitions are entangled. But this is more or less a kind of formula we apply, compare this with our definition of **`dllift`**!

Johnsson exploits the lazy attribute grammar paradigm to re-derive with a suggestive simplicity some circular programs discussed in the Bird's paper [13], and notices that this grammatical approach has been discovered *ex post* while trying to find a regular description of the lambda-lifting module within the LML compiler. It is amusing to find out that the perverted ST monad suggests a similar programming style, but even more amusing is the discovery that Fortran programmers may

need it, and that they simulate this style already for many years, using very painful programming tricks.

4 Variations

4.1 Optimization: forward chaining of "promises"

In our case both items of the final output: the main value and the adjoint, are deferred. But this is necessary only for the adjoints, the main values can be constructed during the forward phase of the process, and there is no need to clog the memory by postponed thunks. (Johnsson discusses also similar optimizations possible in the treatment of attributes.) The modifications of the framework are in major part straightforward. We introduce another datatype:

```
data Rdif a = Rd a (a->a)
```

where the first member of the **Rd** tuple is the main value of the expression, and the second is a "promise": a function which will yield the appropriate adjoint when the adjoint of the LHS (see the eq. 5) is known and this function can be applied to it. We define such operations as

```
rCnst c = Rd c (\_->0.0)
rDvar x = Rd x id

rlift f f' (Rd p pr) = Rd (f p) (\r->pr(r*f' p))
-- ...

-- ... within instances of Num and its subclasses (a fragment):

negate (Rd e _) = Rd (negate e) (\r->(negate r))
(Rd p pr) + (Rd q qr)=Rd (p+q) (\r->pr(r)+qr(r))

(Rd p pr)*(Rd q qr)=Rd (p*q) (\r->pr(r*q)+qr(r*p))
recip (Rd p pr)=Rd w (\r->pr(negate r*w*w))
                where w=recip p

sqrt (Rd e pr) = Rd w (\r->pr(0.5*r/w))
                 where w=sqrt e
cos = rlift cos (negate . sin)
```

etc., analogous to the corresponding functions on the **Ldif** domain, and inspired directly by them. The usage of the augmented funtion is as simple as in the previous model, the final promise should be applied to 1 in order to retrieve the adjoint of the input variable.

It turns out that the chain of promises is just a realization — particularly easy and transparent for the user — of the notorious "tape", which stores the information gathered through the forward sweep, and which permits to execute the adjoint statements during the "time-reversed" phase. In this model the laziness is not important, and the efficiency is improved considerably. In order to prevent the creation of exorbitantly long promises, e.g., in case of long loops (say, in solving differential equations), other optimisation must be considered, but this is a feature pertinent to the reverse mode of AD in general, and is beyond the scope of this text.

4.2 Multi-dimensional Case

For many input variables all adjoints will be kept together in one data structure, equivalent to a list. We introduce:

```
newtype Adj a = Adj [a]
```

and we define a natural set of operations on it, such as the addition element by element (overloaded as **(+)**), and the multiplication by a scalar denoted by **(*>)**.

We have to specify the dimension **nDim** of the independent variable space (the number of intermediate variables remains arbitrary). Our "computations" belong now to the type

```
newtype NLdif a = NLd (Adj a->(a,Adj a))
```

(In the lazy model; of course it can be easily repeated using "promises".)The definitions of constants and of variables become a little more complex. For each variable we must specify its index **k** e.g. starting at zero. This is the lifting of the primitive values:

```
unitA n k c =  -- produces [0,0,...,0,c,0,...,0]
  Adj ((replicate k 0 ++ (c : replicate (n-k-1) 0)))

nlCnst c = NLd (\_->(c,Adj (replicate nDim 0)))
nlDvar k x = NLd (\(Adj z)->(x,unitA nDim k (z!!k)))
```

All other changes are rather cosmetic. An expression **s*n** where **s** is a scalar, and **n** – the adjoint vector, is replaced by **s*>n**.

We construct our final result as an arbitrary expression containing the objects x_k: **xk = nlDvar k some_value**.

In order to get its full gradient $[\overline{x}_0, \ldots, \overline{x}_M]$ we apply the final promise to **Adj [1,1, ...,1]**, and we select the second element of the resulting pair. Obviously, if only one gradient component is needed, there is no need to consider other independent variables as *differentiation variables*, they may be constants, and the dimension of the adjoint space is reduced.

As it is, the model is not adequate for computing *directly* the higher-order derivatives in many dimensions, the algorithms become very clumsy, although the difficulties are not fundamental. It was trivial in one-dimensional case because the adjoint vector was a scalar belonging to the same type as the main expression, and a result obtained from, say, **z=f(lDvar (lDvar (lDvar xxx)))** contained the second and third derivatives of **z**. The reverse differentiation in general is not well adapted to this sort of computation, and we share this difficulty with other reverse AD packages. In general case the forward approach is cleaner, and if one wants to do it functionally in a regular, geometric framework, we would suggest the techniques elaborated in our paper [19].

5 Conclusions

This paper belongs to a longer series of texts advocating the use of lazy functional methods in the domain of scientific, mainly numeric and semi-numeric computing (geometry, power series manipulation, etc.). The ambition of *this* paper is twofold.

- We wanted to show that the scientific programming realm needs sometimes complicated, non-orthodox views on the essence of the computational processes, and that lazy functional

paradigms offer a marvelous opportunity to treat some non-trivial problems in a nice, readable way. We tried to develop some concrete, and useful examples which are much more difficult to implement using standard imperative methods, especially when the path between a mathematical formula and a computer program passes through the necessity of disentangling some recurrent, implicit definitions.

- We wanted to signal that some of that issues exist in many areas under different disguises, and that the universality of functional programming helps to find their common denominators.

Technically-oriented programmers begin slowly to discover the advantages of lazy programming in a context where it enables the transforming — often automatic — of a fixed-point equation $x = f(x)$ into an effective algorithm, if x belongs to some co-recursively defined domain, e.g. a power series, or other expression resulting from a perturbational expansion which is often formulated as an open, co-recursive equation (see e.g. our article [20] which shows how to apply laziness to a classical, but nasty computational problem in Quantum Mechanics).

We see that the applicability of lazy techniques is larger than that, being able not only to deal with entangled data dependencies, but also with some non-classical control flows. Of course, there is nothing intrinsically numeric in the code organization, and we think that non-strict semantics may considerably augment the power of the Computer Algebra packages, provided that the CA implementors liberate themselves one day from the imperative tradition.

The source code of our package is available from our Web page:
`www.users.info.unicaen.fr\~karczma\Work\adjdif.hs`. It has been tested with GHCi, version 5.04. This is *not* an industrial-strength package, but rather a pedagogical essay, and it will not be very efficient in a complex, professional context, because of considerable memory consumption. A thorough analysis of the complexity of our approach, and many serious optimizations (such as periodic reduction of the accumulated lazy thunks) remain to be done.

Above all, this kind of coding may be really enjoyable in a field which is usually a little boring, and where the algorithmization process is extremely costly in human resources.

References

[1] Philip K. Dick, *Counter Clock World*, Berkley P.B., (1967). See also some other Dick's books, e.g., *The World Jones Made* (1956) where the hero, who lives simultaneously in two world time slices has to synchronise the events conditioned by his knowledge of the future, with the "natural" data dependencies.

[2] T. Johnsson, *Attribute Grammars as a Functional Programming Paradigm*, Conference on Functional programming Languages and Computer Architecture, Portland, Proceedings: Springer LNCS 274, pp. 154–173, (1987).

[3] D.E. Whithney, *Resolved Motion Rate Control of Manipulators and Human Prostheses*, IEEE Trans. on Man-Machine Systems, **0**:2, pp. 47-53, **(1969)**.

[4] J.U. Korein, N. Badler, *Techniques for Generating the Goal-Directed Motion of Articulated Structures*, IEEE Computer Graphics and Applications, pp. 71–81, **(1988)**.

[5] C. Welman, *Inverse Kinematics, and Geometric Constraints for Articulated Figure Manipulations*, Master Thesis, Simon Fraser Univ. **(1993)**.

[6] P. Werbos, *Backpropagation Through Time: What It Does and How to Do It*, Proceedings IEEE, special issue on neural networks **2**, pp. 1550–1560, (1990).

[7] E.A. Wan, F. Beaufays, *Diagrammatic Derivation of gradient Algorithms for Neural Networks*, Neural Computation, (1994), or other papers of Eric Wan and Françoise Beaufays.

[8] A. Griewank, *On automatic differentiation.* In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, Kluwer, (1989), pp 83–108.

[9] D. Juedes, *A taxonomy of automatic differentiation tools.* In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, Penn., (1991), pp 315–329.

[10] L.B. Rall, *Automatic Differentiation – Techniques and Applications*, Springer Lecture Notes in Computer Science, Vol. 120, (1981).

[11] Argonne National Laboratory Computational Differentiation site: **`http://www-unix.mcs.anl.gov/autodiff/index.html`**.

[12] J. Karczmarczuk, *Functional Differentiation of Computer Programs*, Higher-Order Symbolic Computations **14**, (2001), pp. 35–57.

[13] R.S. Bird, *Using circular programs to eliminate multiple traversals of data*, Acta Informatica **21**(4), pp. 239–250, (1984).

[14] A. Griewank, D. Juedes H. Mitev, J. Utke, O. Vogel, A. Walther, *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM TOMS, **22**(2) (1996), pp. 131–167, Alg. 755.

[15] R. Giering, T. Kaminski, *Recipes for Adjoint Code Construction*, ACM Trans. On Math. Software, **24(4)**, (1998), pp. 437–474.

[16] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, **1**, (1992), pp. 35–54.

[17] P. Hovland, C. H. Bischof, D. Spiegelman, M. Casella, *Efficient Derivative Codes through Automatic Differentiation and Interface Contraction: an Application in Biostatistics*, Mathematics and Computer Science Division, Argonne National Laboratory, Preprint MCS–P491–0195, (1995).

[18] P. Wadler, *The Essence of Functional programming*, 19'th Symposium on Principles of programming Languages, Santa Fe, (1992).

[19] J. Karczmarczuk, *Functional Coding of Differential Forms*, I-st Scottish Workshop on Functional Programming, Stirling, (September 1999).

[20] J. Karczmarczuk, *Scientific Computation and Functional Programming*, Computing in Science & Engineering, Vol. **1**(3), section: "Scientific Programming", pp. 64–72, (1999).