Adjoint Codes in Functional Framework

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 Computational Differentiation techniques – tools permitting to compute numerically, but exactly (i.e. up to the machine precision) the derivatives of coded functions. The imperative coding of the reverse techniques is awkward. 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, 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 Haskell, uses the overloading of standard arithmetic operations and is very simple to use, permitting the calculation of M-dimensional gradients, and in principle also of higher derivatives.

1. INTRODUCTION

1.1 Computational differentiation

This paper is devoted to a specific functional implementation of *Computational Differentiation* (CD) technique – a well established solution to get accurate and fast derivatives of a numerical function represented by a program. The CD methods are based on two principles:

- any program (streamlined along the chosen decisional paths, i.e., with all the conditionals and jumps eliminated) can be seen as a composition of primitive functions whose derivatives are known,
- and can then be differentiated using the chain rule.

The computation of derivatives is performed by an augmented "main" program, not by an independent processor. They are calculated numerically, point-wise, following the control structures of the program, and it may yield directional derivatives only if, e.g., the differentiated function is defined segment-wise, but this is a general problem with calculating derivatives. In any case, the results are as precise as those obtained by symbolic methods, (no finite differences are ever used), and they are generated much faster.

There are 2 modes of CD, both having predictable complexity:

1. The forward mode in which the intermediate derivatives are computed in the same order as the program evaluates the composition of its component functions. This is the 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 multidimensional gradients of *one* function, because its complexity is (in principle) independent of the number of input variables.

The CD domain has at least 30 years, and its applications are numerous. The bibliography and the number of relevant software packages are impressive, see [1, 2, 3], and the information stored on the World-Wide-Web [4]. 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 [5] we have shown some potential advantages of lazy functional programming. We constructed in Haskell a small package implementing a *closed* local differential algebra, i.e., a domain which extended normal floating-point numbers, containing together with all standard arithmetic functions also the *derivation operator* permitting an easy computation of **all** the derivatives. Our objective was not just to differentiate programs, but to use it as an algorithmization tool, to permit to code functions defined by intricate, often implicit differential recurrences which can be found in many perturbational calculi, asymptotic expansion, etc.

We have concentrated on the one-dimensional case, where the numbers are replaced by data items structurally equivalent to infinite lists. The list $e = (e_0 : \hat{e}) = [e_0, e', e'', e^{(3)}, \ldots]$ represents the numerical expression e_0 together with all the derivatives wrt. a specified variable. This variable is anonymous – it is a numerical object of the form $[x, 1, 0, 0, \ldots]$. All explicit constants in the program have shapes $[c, 0, \ldots]$, and the construction of more complicated expressions exploits the overloaded arithmetic. For $e = (e_0 : \hat{e})$ and $f = (f_0 : \hat{f})$ we define their multiplication as: $e \cdot f = (e_0 \cdot f_0 : e\hat{f} + \hat{e}f)$. For the reciprocal we have $1/e = w$ **where** $w = (1/e_0 : -\hat{e} \cdot w^2)$, and it is easy to define the overloaded elementary functions, e.g., $\exp(e) = w$ **where** $w = (\exp(e_0) : w\hat{e})$, etc. The derivation operation is just the selector of the list tail, and its non-triviality consists in forcing the evaluation of the deferred thunks. Some well known CD packages, such as ADOL [6] also use the arithmetic operator overloading, but in a strict language it is more difficult to organize the program which needs higher derivatives of unknown *a priori* order, because of the unavoidable truncations. We do not claim that the lazy implementation can beat the efficiency of highly-tuned numerical packages, but the economy of human resources may be substantial, and some complex algorithms are much easier to code than using "classical" CD tools.

Passing to many dimensions and calculating gradients, Jacobians, Hessians, etc. within this approach is more cumbersome. Linear lazy lists become infinite N-ary trees, and the notation becomes a little complicated. Sometimes the regularity and invariance properties of the underlying mathematical domains may be exploited, in [7] we have shown how to extend our functional package to deal with the differential geometry of N-dimensional spaces, and how to construct numeric (*not* symbolic) *differential forms* – fully antisymmetric tensors with known algebraic and differential properties, extremely useful in physics and engineering. However, in differential geometry one usually constructs invariant combinations of vectors, tensors, and their derivatives. In many other branches of computational mathematics the situation is much more sparse, and less regular: the space of design parameters of a nuclear reactor has no visible geometric structure. So, much effort has been put into the development of alternative, *reverse* techniques to compute numerical, but precise (up to machine rounding errors) multi-dimensional derivatives in a way that should exploit better the sparsity, and to introduce less overhead to the program. This paper is devoted to the construction of these algorithms in a functional framework, and our ambition is oriented towards its possible practical applications. We will essentially discuss the computation of gradients, although some comments on the computation of higher order derivatives are also included in the text.

1.2 Reverse differentiation, and its practical usage

A natural generalization of the lazy list $e = (e_0 : \hat{e})$ for many dimensions would be a tree: $e = (e_0, [\hat{e}_1, \ldots, \hat{e}_n])$, with $\hat{e}_k = (\partial e / \partial x_k, [\ldots \text{derivatives of } e'_k \ldots])$. The propagation of such structures along the program may be quite costly in memory.

However, in several applications the number of interesting result components is much smaller than the number of independent variables. 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. Most differential equations in such context have many parameters which span a N-dimensional space, but this space has no typical "geometric" properties, and often we need just one object: the final value of the trajectory, together with its dependencies. The temperature of a reactor may depend on hundreds of design parameters, but we need only their combined effect. The same relation characterizes many optimisation problems. In such cases, as it can be read in many introductory texts belonging to the cited CD bibliography, it is not necessary to keep all the partial derivatives of intermediate expressions, it suffices that for each variable (independent or intermediate) e we store its *adjoint* $\overline{e}$ – the derivative of the *final result* with respect to this variable. It is then obvious that when a variable is injected into the evaluated expression, its adjoint cannot be calculated immediately, the final value belongs to the future. In fact, in order to compute the adjoints the computational graph of the program must be reversed, it proceeds "top-down" from the definition of the final result (after having computed it by the first, "forward" phase), down to the initial arguments.

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:

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

where for uniformity we have named "x_p" all the intermediate expressions in the program. Of course, this set may be completed by $x_k = f_k()$, for $k \le M$. The last few of the equations (2), 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

$$\overline{e}_k \leftarrow \overline{e}_k + \overline{g}\frac{\partial f}{\partial e_k}\,. \tag{2}$$

This is the adjoint statement mentioned above. For example, if we need $z'(x)$ given by the program: $y = \sin(x)$; $z = y^2 - x/y$, 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:

$\overline{z} \leftarrow 1;\ \overline{x} \leftarrow 0;\ \overline{y} \leftarrow 0:$

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

Finally $\overline{x} = -1/\sin(x) + \cos(x)\left(2\sin(x) + x/\sin(x)^2\right)$ is the desired value of dz/dx. The derivation of the general algorithm proceeds as follows. The derivatives are defined by the chain rules obeyed by the Jacobi matrices:

$$\mathbf{J}_{ik} = \frac{dx_i}{dx_k} = \delta_{ik} + \sum_{j=k}^{i-1} \frac{\partial f_i}{\partial x_j}\frac{dx_j}{dx_k}. \tag{4}$$

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

$$\mathbf{D}_{ik} = \frac{\partial f_i}{\partial x_k} = \begin{pmatrix} 0 & 0 & 0 & \dots \\ \partial f_2/\partial x_1 & 0 & 0 & \dots \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{5}$$

By following the chain, beginning with x_{M+1}, and terminating at x_N, we 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 $\overline{x}_k = \mathbf{J}_{Nk} = dx_N/dx_k$. We see immediately that $\mathbf{J}$ and $\mathbf{D}$ commute, and if we rewrite the equation for $\mathbf{J}$ in its adjoint form: $\mathbf{J} = \mathbf{I} + \mathbf{JD}$, or

$$\mathbf{J}_{ik} = \delta_{ik} + \sum_{j=k+1}^{i} \mathbf{J}_{ij}\mathbf{D}_{jk}, \tag{6}$$

we get for its last row the equation

$$\overline{x}_k = \delta_{Nk} + \sum_{j=k+1}^{N} \overline{x}_j \mathbf{D}_{jk}. \tag{7}$$

We see that while the "forward" method of computing derivatives essentially needs only the facility to overload the standard arithmetic operators, the reverse mode generates a very non-trivial modification of the program control flow. The CD packages which operate in reverse mode, [6, 8] and many others, usually perform an involved source-to-source transformation. This machinery may be quite heavy. The conversion is not trivial. The execution of resulting program stores the intermediate expressions evaluated during the "forward sweep", and needed for the evaluation of $\mathbf{D}$, on a sequential (internal or external) data structure: the "tape". After having computed the final value, the tape is read backward, and the program reconstructs all the adjoints. The reverse strategy may be dangerous: if the program executes an iterative loop, each new re-assignment of an intermediate variable is functionally equivalent to a 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. Many essential optimisation strategies have been proposed, see e.g. [9, 10], but their automatization 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. Admitting the importance of the reverse differentiation techniques for the scientific computing, we observe that the remark of Philip Wadler in his known paper [11]: "*To make this change in an impure language is left as an exercise for masochistic readers*", might be considered a little too cruel ...

2. FUNCTIONAL APPROACH TO REVERSE DIFFERENTIATION

Our ambition is to use some lazy functional techniques to implement the reverse CD technique in a *simple* way. 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. In order to expose better the control flow, we begin the presentation with the treatment of the one-dimensional case, the generalization for many independent variables is conceptually easy, but technically a little involved.

The language used is Haskell, and the programs have been tested with Hugs. We show two similar, but structurally different models of computing adjoints. Both can be formulated in monadic language, and the second model was partially inspired by the Wadler's backward state propagator, but we want explicitly to cast everything into the normal expression syntax with all the arithmetic operators and elementary functions appropriately overloaded.

2.1 Distributed state and forward chaining of promises

The "tape" strategy applied in imperative implementation of adjoint codes can be reformulated as follows: while evaluating an expression, the augmented program stores on the tape (conceptually this is a stack) a *promise* to compute the local contribution to the adjoints of the RHS arguments according to the equation (2). We shall code this promise as a functional object, parameterised by the adjoint of the LHS which will be computed later. In principle we could use one global state containing the composition of these promises, but it seems more natural to distribute them, and to keep them together with the evaluated sub-expressions. The computation of the root of the computational graph, i.e., the final result, yields the final promise which should be applied to $\overline{x}_N = 1$.

Expressions (i.e. numbers, belonging to a type `a`) will be lifted to the following data structure

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

where the second field is the promise to compute the adjoints of the arguments of this expression. All numerical constants, and a distinguished object which is the differentiation *variable* (e.g. the parameter of a differentiated function) are lifted by the following constructs:

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

(we will omit in the presented programs all casts `fromDouble` etc., which in the real implementation are sometimes necessary). All unary functions and binary operators may be lifted by

```
rlift f f' (Rd p pr) = Rd (f p) (\r->pr(r*f' p))
drlift g g1' g2' (Rd p pr) (Rd q qr) =
   Rd (g p q) (\r->pr(r*g1' p q)+qr(r*g2' p q))
```

where `f'` is the formal derivative of `f` etc. The definitions above are generic, for elementary operations we will see some simplifications. Notice that for all binary operators we add blindly two contributions to the final adjoint. The lifting (and overloading) of standard operations is presented below. We will omit the details of the appropriate instance declarations, type class contexts, etc. We did not use the numerical hierarchy of classes from the standard Preludes of Haskell, but a more algebraically oriented collection of such classes as `AdditiveGroup, Monoid, Ring, Field`, etc. The overloaded definitions are simple and sufficiently readable:

```
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)+qr(negate r))

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

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

A characteristic form of the promise generated by a unary function `f`: `(\r->pr(r*f' p))` may be rewritten of course as `(pr . \r->r*f' p)`, the chaining of items on the tape is just a functional composition. This model does not require laziness.

2.2 Lazy time reversal

The standard State Transformer monad 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 acts on the current state, and produces a value and a new state. Philip Wadler in his article [11] demonstrated the possibility to redefine this monad in such a way that the computation acts on the final state, and when the program delivers the final value, one recovers the initial state. A functional expression $k(x)$ is replaced by the form `m >>= k`, where `m` is the lifted computation which delivers the value `x` upon acting on some state. The modification of the "bind" operator `(>>=)` is formally simple:

```
m >>= k = \s_fin -> let (x, s_ini) = m s_mid
                        (y, s_mid) = k x s_fin
                    in  (y, s_ini)
```

but its understanding a bit less. This is more tortous than the time reversal in physics which would not change anything observable in the world. If, as usual within the lazy protocol, we suppose that the evaluation begins by entering the function `k`, it acts on the final state, and produces an intermediate one. But it needs the value provided by the computation `m` which acts on this intermediate state. *The data dependencies obey the "forward" time arrow*, and we cannot "cheat" by redefining the time arrow. (For complementary literature, see Philip Dick's book [12], where the author shows that it is conceptually easier to resuscitate the dead than to unwrite a book ...). It is obvious that the implementation of this monad needs a lazy language, since the two internal `let` clauses are mutually recursive.

We propose to define the final state as the value of the adjoint of the final result, i.e., 1 (again, this is simpler to present in one-dimensional case), and the initial state is the adjoint of the differentiation variable. The idea is obviously that this initial adjoint is never really needed until the end of the program. Our "time vehicle" belongs essentially to the same category of lazy tricks as those which have been presented in the known article of Bird [13], and reformulated many times since.

This time we shall keep one global state, the value of the adjoint, which belongs (in one dimension) to the same type as all other expressions. The declaration of the lifted data type, and the lifting of constants and of the variable go as follows:

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

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

The numeric conversion functions, e.g. `fromDouble` are specified as `lCnst`. The generic lifting of unary and binary functions follows the anti-causal monadic chaining shown above.

```
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) )
```

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

If there are no special shortcuts, we use the generic forms, i.e., `cos=llift cos (negate.sin); log=llift log recip`, etc. In order to apply practically our techniques it suffices to construct 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)/2.0`, 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". In the previous model the main value was computed immediately.

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. If the package is generalized to many dimensions, we can compute with it not only the gradients, but also the Hessians: $\partial^2 x_N/\partial x_i \partial x_k$, but this generalisation does not belong to objectives of this paper.

Such generalisations exploit the polymorphism of Haskell, and the possibility to compose the types recursively; the recursive composition of CD techniques in standard imperative languages are much more difficult to implement, even if the language permits to overload the arithmetic operations.

3. RELATION TO ATTRIBUTE GRAMMARS

We are now prepared to confess 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:

$E ::= E_1 \ \mathbf{Op} \ E_2$

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" evaluation of the parsing tree, gets into trouble, because the evaluation proceeds from the leaves upwards, and the inherited attributes descend from the root.

This problem viewed from the perspective of lazy functional programming has been analysed by Johnsson, [14] (see also the compiler construction language Elegant [15]). 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 (usually this is some contextual information: relative position, 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):

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

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 some years, using very painful programming tricks.

4. MANY-DIMENSIONAL CASE

The formalism should be usable in a context when the independent variables do not span a regular, "geometric" manifold, but they are more or less heterogeneous, and the expressions show no particular invariance properties. The variables do not obey any particular naming nor structural protocol, but their adjoints will be kept together in one data structure, equivalent to a list. We introduce:

```
newtype Adj a = Ad [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 `(*>)` (this is a member of the instance of the algebraic Left Module class, and it is defined simply through `map (*)`).

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

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 k x = Ad ((replicate k 0.0 ++
               (x : replicate (nDim-k-1) 0.0)))

lCnst c = Ld (\_->(c,Ad (replicate nDim 0.0)))
lDvar k x = Ld (\(Ad z)->(x,unitA nDim k (z!!k)))
```

(The function `unitA` produces a list $[0, 0, \ldots, x, \ldots 0]$ and can be defined in many other ways.) All other changes are purely 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 objets x_k: `xk = lDvar k x_val`.

In order to get its full gradient $[\overline{x}_0, \ldots, \overline{x}_M]$ we apply the final promise to `Ad [1,1, ...,1]`, where the "1" should be overloaded (see the next paragraph), 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 *can be used*, although it is not friendly for computing higher-order derivatives. 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 x_val)))` contained the second and third derivatives of `z`. Here we would have to specify the variables as, say, `x0=lDvar 0 (lDvar 0 x_val)`, etc., and to disentangle the Hessians from a relatively complex result obtained by applying the resulted promise to the overloaded adjoint constant: a list of "1" belonging to `Adj (Ldif a)`. In general case the forward approach is cleaner, and if one wants to do it functionally, we would suggest the techniques elaborated in our Differential Form paper [7]. At any rate we can do it here very simply, while the standard imperative CD packages which use the reverse techniques, have to apply some *ad hoc* coding, and this strongly increases their volume and complexity.

4.1 A few words on complexity

As mentioned in the section (1.2), the reverse CD technique "converts the time into space" – each step of an iterative procedure generates a new adjoint statement, and in our case: a new promise, composed with the existing one, so the resulting memory consumption may be substantial. Several papers on CD, e.g. [9, 10] address this issue, and the search for optimisation strategies goes in many directions. One of them consists in avoiding the storing of long intermediate data sequences which are the result of the fact that the reversed control flow comes *after* the main calculation. The functional solution does not suffer from that, and the data sharing between the "main" and adjoint computation goes as far as possible. We cannot eliminate the composition of promises, but they are just thunks which do not occupy much memory.

There are some obvious optimisations of the fixed point calculations. If $y = y(x)$ is the solution of the equation $y = f(x, y)$ obtained as the limit of $y_{n+1} = f(x, y_n)$, there is no point in keeping all the intermediate chain of promises in view of the algebraic identity: $y' = f'_x/(1 - f'_y)$.

The situation is different if the iteration is progressive, e.g., if we solve a discretized equation for $y(t)$: $t(t + \Delta t) = h(y(t))$, for $t = 0, \Delta t, 2\Delta t, \ldots, T$, and if we need ultimately $\partial y(T)/\partial y(0$. Some imperative optimisation techniques use the "snapshots" of the state of the system, and its retrieval, whose implementation, as presented in [9], is *very* complex. The functional equivalent strategy is cleaner; the promise chain may be broken by the evaluation of the adjoints for some intermediate times, and the final derivatives are reconstructed from the chaining of the "macro-slices".

The main optimisation trick which might be very efficient, but difficult to implement is based on a conscious use of sparsity. If the program is composed of many internal functions, there is no necessity to keep the same set of independent variables for each function. The dimensions of each separate adjoint vectors may be much smaller. On the other hand their recombination might be cumbersome.

We are not yet ready to present a thorough analysis of the complexity of our approach, the optimisation techniques in CD based on C and Fortran have been elaborated during 10 years. But the simplicity and the elegance of the lazy functional formulation of adjoint code is simply incomparable with known imperative packages.

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.). We try 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.

Technically-oriented programmers begin slowly to discover the advantages of lazy programing in a context where it permits to transform automatically 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 [16] which shows how to apply laziness to a standard, 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.

We end this section on a (slightly) lighter tone. Yet another example of application of laziness to some "anti-causal" computations in a numeric setting is presented in our article [17], where we elaborated an arithmetic package on infinite (decimal, hexadecimal, etc.) fractional numbers, e.g., 3.14159 ... represented as lazy lists of digits. Strict algorithms are helpless here, because the carry propagates to the left, and we cannot even begin the process of adding such numbers (the top-down approach would overflow the recursive stack immediately). But we can proceed rightward lazily, *borrowing* the carry through a peephole look ahead. Again, the computation goes against the flow of propagating carries from the added digits. If the algorithm locally fails, e.g. on the second digit of 0.23XYZ ... + 0.96PQR ... , we increase the look ahead depth, combining a strict (possibly runaway) recursion with a co-recursion. We define the multiplication, division, etc. on our fractions, and although these methods are co-recursively unsafe (and for theatrical reasons a little insane ...), the package is able to compute some thousands of digits of π using the hexadecimal expansion of Bailey, Borwein and Plouffe.

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

6. REFERENCES

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

[2] 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.

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

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

[5] J. Karczmarczuk, *Functional Differentiation of Computer Programs*, Proceedings, III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203. A modified version is submitted to the Journal of Higher-Order Symbolic Computations.

[6] 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.

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

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

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

[10] 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).

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

[12] Philip K. Dick, *Counter Clock World*, Berkley P.B., (1967). See also 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.

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

[14] 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).

[15] P. Jansen, L. Augusteijn, H. Munk, *An Introduction to Elegant*, 2nd ed., Philips Research (1993).

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

[17] J. Karczmarczuk, *The Most Unreliable Technique in the World to Compute* π, Workshop at the IIIrd Summer School on Advanced Functional Programming, Braga, Portugal, (1998).