Functional Differentiation of Computer Programs

Jerzy Karczmarczuk (`karczma@info.unicaen.fr`)
Dept. of Computer Science, University of Caen,
Sciences III, Bd. Maréchal Juin, 14032 Caen, France

Abstract. We present a purely functional implementation of the *computational differentiation tools* — the well known numeric (i.e., not symbolic) techniques which permit one to compute point-wise derivatives of functions defined by computer programs economically and exactly (with machine precision). We show how the use of lazy evaluation permits a transparent and elegant construction of the entire infinite tower of derivatives of higher order for any expressions present in the program. The formalism may be useful in various problems of scientific computing which often demand a hard and ungracious human preprocessing before writing the final code. Some concrete examples are given.

Keywords: Haskell, differentiation, arithmetic, lazy semantics

1. Introduction

The aim of this paper is to show the usefulness of lazy functional techniques in the domain of scientific computing. We present a functional implementation of the *Computational Differentiation* techniques which permit an efficient computation of (point-wise, numeric) derivatives of functions defined by computer programs. A previous version of this work has been presented at the 1998 International Conference on Functional Programming [14].

A fast and accurate differentiation is essential for many problems in applied mathematics. The derivatives are needed for all kind of approximations: gradient methods of equation solving, many sorts of asymptotic expansions, etc. They are needed for optimization and for the sensitivity and stability analysis of dynamical systems. They permit the computation of geometric properties of curves and surfaces in 3D modelling, image synthesis and animation. In the domain of differential equations, they are used not only directly, but also as an analytic tool for evaluating the numerical stability of a given discrete algorithm. The construction of equations of motion is often based on variational methods, which involve differentiation. Even in discrete mathematics the differentiation is useful to compute some combinatorial factors from the appropriate partition functions, as presented in Knuth, Graham and Patashnik's textbook on concrete mathematics [9, Chapter 7].

1.1. This work

We are not only interested in the computation of first derivatives, but in implementing a general differentiation operator acting on expressions within a numerical program. We show thus how laziness can be used to define data structures which represent numerical expressions together with all their derivatives wrt. a given independent variable.

Our differentiation package is implemented in the purely functional language Haskell enriched by some generic mathematical operations. It has been tested with the interpreter Hugs[12], and relies on the overloading of arithmetic operations with the aid of type classes. Our basic tools are the *co-recursive* data structures: objects defined by open, non-terminating recursive equations which would overflow the memory if implemented naively in a strict language.

The presented approach requires the lazy evaluation strategy which states that a function evaluates its argument only when it needs it. Despite a reasonably long history, lazy functional techniques are rarely used in numerical computations. First, they remain relatively unknown to the scientific computing community. Then, there are some efficiency reasons: the delayed evaluations introduce an overhead which might be considered harmful by those for whom the computation speed is crucial.

Our implementation is not meant as a replacement of highly tuned and efficient low-level numerical programs. The Computational Differentiation packages cited later have been optimized for performance. We show that lazy techniques provide useful *coding tools*, clear, readable, and semantically very powerful, economising plenty of human time. A few problems rarely addressed by the standard Computational Differentiation texts, such as the construction of functions defined by differential recurrences, become very easy to code using our lazy approach. This is the main goal of the paper. We assume that the reader is acquainted with the lazy evaluation paradigm, and can follow the Haskell code. Elementary notions of differential calculus and of algebraic structures are also needed for the understanding of our implementation.

1.2. Overview of mechanical differentiation techniques

There are essentially three ways to compute derivatives of expressions wrt. some specific variables with the aid of computers.

– The approximation by finite differences: $df \to \Delta f = f(x+\Delta x) - f(x)$. This method may be either inaccurate if Δx is big, or introduce serious cancellation errors if it is too small, so it might be numerically unstable. Sometimes the functions must be sampled many times in order to permit the construction of a decent polynomial interpolant. The complexity of the algorithm may be substantial, and its coding is rather tedious.

– Symbolic computations. This is essentially the "manual", formal method, with a Computer Algebra package substituted for the combined tool: pencil/paper/wastebasket. The derivatives, gradients, Jacobians, Hessians, etc. are exact, but the technique is rather costly. The intermediate expression swell might be cumbersome, sometimes overflowing the memory. The generated numerical program is usually unreadable, and needs a good optimizing compiler in order to eliminate all the common subexpressions, which tend to proliferate when symbolic computations are used intensely.

  Moreover, it is not obvious how to differentiate expressions which result from an iterative process or other computations which use non-trivial control structures, so this technique is usually not entirely automatic.

– The *Computational Differentiation* (CD) known also as *Automatic* or *Algorithmic Differentiation*, which is the subject of this article. Computational Differentiation is a well-established research and engineering domain, see, e.g., [3, 4, 8, 10, 11]. George Corliss also established a comprehensive bibliography [7]. The CD algorithms are numerical, but they yield results as exact as the numerical evaluation of symbolic derivatives. Relatively little has been written about functional programming in this context, most developments appear to be carried out in Fortran, C, or C++. C++ is a natural choice if one wants to exploit the arithmetic operator overloading (see the description of ADOL-C [10]). For languages without overloading, some source code preprocessing is usually unavoidable.

1.3. COMPUTATIONAL DIFFERENTIATION

The CD idea relies on standard computer arithmetic, and has nothing to do with the symbolic manipulations of data structures representing the algebraic formulae. All complicated expressions coded in a standard programming language are composed of simple arithmetic operations and elementary, built-in functions with known differential properties. Of course, a program is not just a numerical expression. It has local variables, iterations, sometimes explicit branching, and other specific control structures, which makes it difficult to differentiate symbolically and automatically a sufficiently complicated code. A symbolic package would have to unfold the loops and to follow the branches — in fact, in general, it would have to interpret the program symbolically.

But it is always possible to compute all the needed derivatives in parallel with the main expressions in an augmented program, taking into account that for all primitive arithmetic operators the derivatives are known, and that all the compositions obey the chain rule: $d(f(g(x)) = f'(g(x))d(g(x))$. The same control structures as in the main computation are used (although not necessarily in the same way).

We shall restrict the presentation to the univariate case, and we discuss the *direct* or *forward mode* of CD. The alternative, *reverse mode* is more important for the multi-variate case. A functional implementation of the reverse mode is treated in another paper [18]. The multivariate case in a geometric framework (differentiation of tensors and exterior forms) is discussed in [17].

1.4. Overview

The rest of the paper is organized as follows: we begin with the implementation of a simplified framework for computing just the first derivatives, which does not require laziness. For simple usages this variant is more efficient than the full package.

Next we discuss some features of the Haskell class system, and the differences between our framework and the numerical classes belonging to Haskell standard libraries.

We present then a short, elementary introduction to differential algebra, and we pass to the implementation of our lazy version of it.

The implementation is followed by a collection of non-trivial examples of applications of the package. They occupy a substantial part of the paper, and they show not only how to compute derivatives in a program, but principally how to *use them* for solving complex programming tasks.

2. Overloading and Differentiation; First Approach

In this section we introduce an "extended numerical" structure: a combination of a numerical value of an expression *and* the value of the derivative of the same expression at the same point. We may declare

```
type Dx = (Double, Double)
```

where only for simplicity of presentation we restrict the base type to **`Double`**. In principle we can use any number domain rich enough for our needs. This domain should be at least a ring (a field if we need division).

The elementary objects which are injected into the calculations are either explicit constants, for example **`(3.14159,0.0)`**, or the (independent) *derivation variable* which is represented as something like **`(2.71828, 1.0)`**. Since we are not doing symbolic calculations, the variable does not need to have a particular name. From the above we see that constants are objects whose derivatives vanish, and the *variable* (henceforth always referred to through the italic typesetting), has the derivative equal to 1. The value x in (x, x') will be called the “main” value.

From the mathematical point of view we have constructed a specific extension of the basic domain. All objects (e, p) with $p \neq 0$ are algebraically independent of constants $(e, 0)$. The augmented arithmetic defined below ensures this property, and shows that the subset of constants is closed under all arithmetic operations.

2.1. Overloaded arithmetic

In order to construct procedures which can use the type **Dx** we declare the following numerical operator instances:

```
(x,a)+(y,b)  = (x+y, a+b)
(x,a)-(y,b)  = (x-y, a-b)
(x,a)*(y,b)  = (x*y, x*b+a*y)
negate (x,a) = (negate x, negate a)
(x,a)/(y,b)  = (x/y, (a*y-x*b/(y*y))
```

or, for the reciprocal:

```
recip (x,a)  = (w, (negate a)*w*w) where w=recip x
```

We define also two auxiliary functions which help to construct the constants and the *variable*.

```
dCst z = (z, 0.0)
dVar z = (z, 1.0)

fromDouble z = dCst z
```
(Conv. of numeric, real constants)

Now all rational functions, e.g.,

```
f x = (z + 3.0*x)/(z - 1.0) where z=x*(2.0*x*x + x)
```

called with an appropriate argument, say, **f (dVar 2.5)** compute the main value and its derivative. The user does not need to change the definition of the function. The following properties of Haskell are essential here:

– The type inference is automatic and polymorphic. The compiler is able to deduce that **f** accepts an argument of any type which admits the multiplication, addition, etc. The same function can be used for normal floating numbers.

– The numerical constants are automatically “lifted”: 3.0 in the source is compiled as the polymorphic expression **(fromDouble 3.0)**, whose type depends on the context.

We implement also the chain rule, which for every function demands the knowledge of its derivative *form*, for example **sin** $\rightarrow$ **cos**. All elementary functions may then be easily lifted to the **Dx** domain. Here are some examples:

```
dlift f f′ (x,a) = (f x , a * f′ x)

exp  = dlift exp exp
sin  = dlift sin cos
cos  = dlift cos (negate . sin)
sqrt = dlift sqrt ((0.5 /) . sqrt)
log  = dlift log recip        (recip x means 1/x)
```

and now the following program

```
res = ch (dVar 0.5) where
   ch z = let e=exp z in  (e + 1.0/e)/2.0
```

computes automatically the hyperbolic sine together with the hyperbolic cosine for any concrete value, here for $x = 0.5$. The value of **`res`** is **`(1.12763, 0.521095)`**. The call **`ch (dCst 0.5)`** calculates the main value, but its derivative is equal to zero. The expression **`sqrt (cos (dVar 1.0))`** computes also the value $-0.572388 = -\sin x/(2\sqrt{\cos x})$ for $x = 1$. If a function is discontinuous or non-differentiable, this formalism might return an unsatisfactory answer. For example, if we define **`abs x = if x>0 then x else -x`**, the derivative at zero is equal to -1, if the test **`x>0`**, with the appropriately overloaded **`(<)`** operator, uses the main value only.

2.2. Usage of the Haskell class system

The above presentation of the arithmetic operations over pairs of numbers is simplified. In a concrete implementation in Haskell the overloading must follow the discipline of its type system, where all generic operations are declared within classes, and all datatypes which accept those operations are instances of these classes. The standard Haskell library (Prelude) specifies several arithmetic classes: **`Num`** for objects which can be added or multiplied, **`Fractional`** where the division is declared, **`Floating`** with the exponential, square root, and other elementary functions, etc. Our package does not use these classes. We found it more natural to introduce a modified "algebraic style" library, which corresponds to the classical mathematical hierarchy, and is more suitable for the definition of arithmetic operations over intricate mathematical objects.

Our modified Prelude contains such type classes as **`AddGroup`** which defines the addition and the subtraction, **`Monoid`** for multiplication, **`Group`** for division, etc. Some more involved operations are made generic within such classes as **`Ring`** for structures which can be added and multiplied, **`Field`** which adds the division to a ring, or **`Module`** which abstracts over a multiplication of a complex object by an element of an underlying basic domain (e.g. the multiplication of a vector or of a polynomial by a numeric constant). The conversion of the standard numbers: **`fromInt`**, **`fromDouble`** into constants of our differential domain is declared within a new class **`Number`**, orthogonal to the algebraic hierarchy.

Some classes in Haskell permit to specify generic operations over composite data structures independently of the type of elements of these structures. If **`Constr a`** is a compound type parameterized by the type **`a`** of its elements, then **`Constr`** alone may be an instance of a *constructor class*. A canonical example of such a class is **`Functor`**. This class declares a generic mapping functional **`fmap`** which applies some function to all the elements, and constructs a structurally equivalent compound. In particular, it transforms a list $[\ldots x_k, \ldots]$ into the list of applications $[\ldots f(x_k), \ldots]$.

The current version of our class **`Module`** is also a constructor class, and the multiplication of a compound by an elementary object uses **`fmap`**. A closely related constructor class **`VSpace`** introduces a generic division operation of a compound by an element of the basic domain. In future versions of our package these classes will probably be converted into multi-parametric type classes with dependencies (see the Hugs manual [12]).

3. Differential Algebra and Lazy Towers of Derivatives

The only language attributes really needed in the example above were:

1. the possibility to overload the arithmetic operators, and

2. the construction of data structures,

so it may have been implemented in almost any serious language, for example in C++, and of course it has been done, e.g., in such packages as ADOL-C or TADIFF [10, 3]. We can extract the derivatives from the expressions, and code some mixed type arithmetics as well, involving normal expressions and the pairs **`(z,z')`** together. However, this approach is not homogeneous, and the extensions needed to get the second derivative, etc. are a little inconvenient.

We propose thus to skip all the intermediate stages, and to define — lazily — a data structure which represents an expression e belonging to an infinitely extended domain. It contains the principal numeric value e_0, and the values of *all* its derivatives: $\{e_0, e', e'', e^{(3)} \ldots\}$, without any truncation explicitly present within the code. We construct a complete arithmetic for these structures, and we show how to lift the elementary functions and their compositions.

The remaining of this section is structured as follows: we propose first an easy formal introduction to differential algebras, then we define our lazy data structures, and we construct the appropriate overloaded arithmetic operations, defining thus a particular instance of differential algebra. We show how to use these operations, and we discuss some less evident properties of the system.

3.1. What is a differential algebra?

The theory of the domain called Differential Algebra was developed mainly by Ritt [22] and Kolchin, see also a more recent book by Kaplanski [13]. This term often denotes the branch of mathematics devoted to the algebraic analysis of differential equations, but here it is the name of a *mathematical structure*.

For the moment let us forget that the concrete computer representation of numbers is necessarily truncated, that the operations may be inexact, etc. The meaning of the arithmetic operations (the correctness of the division, of the square root, etc.) in the extended domain is inherited from the basic domain.

We begin with some field equipped with standard arithmetic operations $(+, \times, /)$. To this set of operations we add one more, the *derivation*: an internal mapping $a \to a'$ which is linear: $(a+b)' = a' + b'$, obeys the Leibniz rule: $(ab)' = a'b + ab'$, and some continuity properties. It is straightforward to prove that for the field of rational numbers the derivation is trivial, the result is always zero. Indeed, the linearity and the Leibniz rule prove immediately that for the ring of integers $0' = 1' = 0$, and from $(a^{-1}a) = 1$ it follows purely algebraically that $(a^{-1})' = -a'/a^2$. For a computer scientist it means that all numbers are *constants*. This basic field must be extended in order to generate non-trivial derivatives.

Calculating derivatives within a simple polynomial extension: $A[x]$ of any field A is well known and described in many books on algebra, e. g., Bourbaki's [5]. We know also how to compute derivatives in the rational extension $A(x)$. These extensions can be considered as based on adjoining of an algebraic indeterminate, some "x" which may be represented symbolically in the program. This is usually the way the interactive Computer Algebra packages proceed. However, it is obvious that if we know the mathematical structure of the manipulated expressions, often no symbols are needed: a polynomial may be represented just as a list of its coefficients, a rational expression as a pair of polynomials, and the construction of a commutative algebra on such data structures is a school exercise.

A practical computer program may apply all algebraic and transcendental functions to its data, and the construction of an appropriate extension is more involved. The symbolic extensions are possible, but they are costly, and much more powerful than usually needed in a numerical program: a polynomial data structure permits to compute the value of the represented polynomial for any value of the "variable" x; it behaves as a symbolic functional (non-local) object. The derivation becomes a structural operation on data objects representing the expressions.

Our approach is minimalistic, as local as possible. We just want to compute the numeric values of the expressions for a given input, and the values of some derivatives. We do not know *a priori* how many derivatives might be needed, so we require that our differential algebra is closed, in the sense that the derivation becomes an internal operation in the domain of expressions.

For any new element x introduced into the domain we have to provide x', x'', $x^{(3)}$, etc. — in fact, the possibility of an infinite number of algebraically independent objects, as there is *a priori* no reason that an x' should be algebraically dependent on x (although it might be true in some cases).

We propose thus that to any expression e (a numerical value) the program adjoins explicitly its derivative e', and by necessity *all* the higher derivatives as well. Kaplanski in [13] discusses the model where the basic domain is extended by an infinite number of indeterminates. Every item e (renamed as e_0) of the basic domain is accompanied by $e_1 \equiv e'$, $e_2 \equiv e''$, etc. The derivation operator is just the mapping $e_n \rightarrow e_{n+1}$. Our model is conceptually similar to this one in the sense that we add explicitly an infinite number of independent entities, but we do not use indeterminates. Structurally the program operates on infinite, lazy lists whose elements are *a priori* independent.

3.2. The data and basic manipulations

The data type we shall work with belongs to an infinite co-recursive domain **`Dif a`** parameterized by any basic type **`a`**, which is an instance of all needed arithmetic classes, normally it should be a field. Usually it will be **`Double`**, but rationals or complex numbers are also possible.

```
data Dif a = C a | D a (Dif a)
```

In this data the **`C`** variant represents constants. It is redundant, and **`(C x)`** could be represented by **`(D x (D 0 (D 0 ...)))`** — a purely co-recursive structure without terminating clause, but adding explicit constants is much more efficient. The first field is the value of the numerical expression itself, and the second is the tower of all its derivatives, beginning with the first.

Here are the numeric conversion functions, and the definition of constants and of the *variable*. The “**`=>`**” construct below means that the embedding of the type **`a`** onto **`Dif a`** belongs to the class of numbers only if the type **`a`** itself belongs to this class.

```
instance Number a => Number (Dif a) where
 fromDouble x = C (fromDouble x)         (etc.)

dCst x = C x
dVar x = D x 1.0
```

(The compiler should lift automatically the numeric constants, so **`D x 1.0`** should be treated as **`D x (C 1.0)`**).

The derivation operator is declared within the class **`Diff`**:

```
class Diff a where          ("a" is a type of differentiable objects)
 df :: a->a                 (the derivation operator)
```

```
instance Diff Double where           (numbers are constants)
 df _ = 0.0
instance Number a=>Diff (Dif a) where      (lifting proc.)
 df (C _) = C 0.0
 df (D _ p) = p                           (just a selector)
```

The equality (instance of the class **Eq**) for our data is semi-defined. The inequality can be in principle discovered after a finite number of comparisons, but the **(==)** operator may loop forever, as always with infinite lists. We define it only for "main" values. This is unavoidable, the equality of symbolic expressions is ill-defined as well, and Computer Algebra has to cope with this primeval sin. (The equality of floating-point numbers is also somewhat dubious and may lead to non-portability of programs, but those issues cannot be discussed here.)

3.3. Arithmetic

The definitions below construct the overloaded arithmetic operations for the **Dif** objects. The presentation is simplified. The subtraction is almost a clone of the addition, the lifting of operators to the constant subfield is routine. The **Dif** data type being a list-like structure, is a natural **Functor**, with the generalized (**fmap**) functional defined almost trivially. From the multiplication rule it follows that the operation **df** is a derivation. The algorithm for the reciprocal shows the power of the lazy semantics — the corresponding (truncated) strict algorithm would be much longer.

```
instance Functor Dif where     (‘‘mappable’’ composite types)
 fmap f (C x) = C (f x)
 fmap f (D x x′) = D (f x) (fmap f x′)
instance Module Dif where
 x *> s = fmap (x*) s
instance VSpace Dif where
 s >/ x = fmap (/x) s

instance AddGroup a => AddGroup (Dif a) where
 C x + C y = C (x+y)
 C x + D y y′ = D (x+y) y′     (and symmetrically D + C)
 D x x′ + D y y′ = D (x+y) (x′+y′)
 neg = fmap neg

instance (Monoid a, AddGroup a) => Monoid (Dif a) where
 C x * C y = C (x*y)
 C x * p = x*>p                    (and symmetrically ...)
 p@(D x x′) * q@(D y y′) = D (x*y)(x′*q+p*y′)  (Leibniz rule)
```

```
instance (Eq a, Monoid a, Group a, AddGroup a) =>
 Group (Dif a) where
  recip (C x) = C (recip x)
  recip (D x x') = ip where
   ip = D (recip x) (neg x' * ip*ip)
  C x / C y = C (x/y)
  p / C y = p>/y
  C x / p = x *> recip p
  p@(D x x') / q@(D y y')
   | x==0.0 && y==0.0 = x'/y'      (de l'Hôpital!)
   | otherwise = D (x/y) (x'/q - p*y'/(q*q))
```

(we have used the de l'Hôpital rule, which may not be what the user wishes.)

The generalized expressions belong to a differential field. One can add, divide or multiply them, one can calculate the derivatives, which costs "nothing" to the programmer, because they are calculated (lazily) anyway, but if used, they do consume the processor time, since they force the evaluation of the deferred thunks.

One can also define the elementary algebraic and transcendental functions acting on such expressions. We begin with a general lifting functional. Then we propose some optimizations for the standard transcendental functions, $\exp$, $\sin$, etc. (They are declared within a new class **`Transcen`**. For simplicity, we have defined there the square root as well.) We omit trivial clauses, like **`exp (C x) = C (exp x)`**.

```
dlift (f:fq) p@(D x x') =     (univariate function lifting)
  D (f x) (x' * dlift fq p)

instance (Number a, Monoid a, AddGroup a, Group a,
 Transcen a, Group (Dif a)) => Transcen (Dif a) where
  exp (D x x') = r where r = D (exp x) (x'*r)
  log p@(D x x') = D (log x) (x'/p)
  sqrt (D x x') = r where
   r = D (sqrt x) ((fromDouble 0.5*>x')/r)
     (sin/cos: Use generic lifting, (for instruction))
  sin = dlift (cycle[sin,cos,(neg . sin),(neg . cos)])
  cos = dlift (cycle[cos,(neg . sin),(neg . cos),sin])
```

The function **`dlift`** lifts any univariate function to the **`Dif`** domain, provided the list of all its formal derivatives is given, for example $(\exp, \exp, \ldots)$ for the exponential, or $(\sin, \cos, -\sin, -\cos, \sin\ldots)$ for the sine. The definitions of the exponent and of the logarithm have been optimized, although the function **`dlift`** could have been used. The self-generating lazy sequences are coded in an extremely compact way. Such definitions in TADIFF [3], where a more classical approach is presented, are much longer. In [15, 16] we have shown how the lazy formulation simplifies the coding of infinite power series arithmetic as compared to the commonly used vector style, (see for example Knuth [20]). We see here a similar shortening of algorithms.

Our definition of the hyperbolic cosine still works, and gives an infinite sequence beginning with **`ch`** and followed by all its derivatives at a given point. The following function, applicable for small z:

```
lnga z = (z-0.5)*log z - z + 0.9189385 + 0.0833333/
 (z + 0.033333/(z + 0.2523809/(z + 0.525606/
 (z + 1.0115231/(z + 1.517474/(z + 2.26949/z))))))
```

called as, say, **`lnga (dVar 1.8)`** produces the logarithm of the Euler Γ function, together with the digamma ψ, trigamma, etc., needed sometimes in the same program: -0.071084, 0.284991, 0.736975, -0.523871, 0.722494, -1.45697, 3.83453,... One should not exaggerate: the errors in higher derivatives will increase, because the original continuous fraction expansion taken from the Handbook of Mathematical Functions [1], is an approximation only, and this formula has not been specifically designed to express the derivatives. We get the same error as if we had differentiated symbolically the expression, and constructed the numerical program thereof. The value of $\psi^{(3)}(x)$ has still several digits of precision.

3.4. SOME FORMAL REMARKS

We cannot include here the proofs of the correctness of the overloaded arithmetic, but some formal observations may be useful.

– We have mentioned that Kaplanski in [13] constructed a formal differential algebra by explicit adjoining of an infinite sequence of independent indeterminates to the basic domain. In our case, if the n-th derivative of an expression is a list $e^{(n)} = [p_n, p_{n+1}, \ldots]$, where the elements of the list are some numerical values, it is obvious that $e^{(n)}$ is independent of $e^{(n-1)}$; the latter adds another independent value in front of the list.

– It can be shown that our definitions are co-recursively sane. Such definitions as the exponential are presented for efficiency as self-referring data structures, but we see that $\exp(D\ x\ x') = D\ (\exp x)\ (x' \cdot \exp(D\ x\ x'))$ is a generalized *unfold*. All proofs that, e.g. (e/f) defines the inverse of multiplication, that $\log(\exp(e)) = e$, etc., are almost trivial.

3.5. PRACTICAL OBSERVATIONS

We recapitulate here the basic properties of the presented computational framework.

– If the definition of a function is autonomous, without external black-box entities, the computation of *all* derivatives is fully automatic, without any extra programming effort. It suffices to call this function with appropriately overloaded arguments.

– The derivatives are computed exactly, i.e., up to machine precision. There is no propagation of instabilities other than the standard error propagation through normalization, truncation after multiplication etc. The roundoff errors might grow a little faster than those of the "main" computation, since usually more arithmetic operations are needed for the derivative than for the main expression (unless it is a polynomial). If the main numerical outcome of the program is an approximation, e.g., the result of an iterative process, the error of the derivative depends on the behaviour of the iterated expression in the neighbourhood of the solution.

– The generalization to vector or tensor objects depending on scalar variables is straightforward, it fact nothing new is needed, provided the standard commutative algebra has been implemented.

– The efficiency of the method is good. The manual, analytic, highly tuned differentiation may be faster because a human may recognize the possibility of some global simplification, but the automatic symbolic differentiation techniques are far behind: symbolic differentiation of graph-like structures, simplification, shared sub-expression handling — these operations increase the computational complexity considerably. Obviously, a symbolic formula may be differentiated only once, and then evaluated numerically for many arguments, but even then, the CD techniques may be competitive, because treating independently the "main" formula and its derivative may inhibit the optimization of shared sub-expressions.

– A few words on control structures are needed. The computation of the derivatives follow the normal control thread. But sometimes the decisions are based on numerical relations: **`if a==b then ... else ...`**, and if **`a`** and **`b`** are lifted, we have to define the arithmetic relations of equality, inferiority etc., even if they are imperfect. In our package the standard operators **`==, <, >=`** etc. check only the main values, and ignore the derivatives. To handle them the user has to write his own procedures.

However, all deferred numerical operations generate closures or *thunks*, functional objects whose evaluation produces eventually (upon demand) a numerical answer. The space leaks induced by these deferred closures might be dangerous. The reader should not think that he can compute 1000 derivatives of a complex expression using our lazy towers, unless the program finds some specific shortcuts preventing the proliferation of thunks, since a closure keeps references to all global values used in its definition, and the lazy towers grow with the order of the derivative. We know that symbolic algebraic manipulations suffer from the intermediate expression swell that may render it impossible to calculate too high order derivatives of complicated expressions. In our framework we have "just" lists of numbers, but a similar difficulty exists.

In order to increase the package performance, and to prevent the memory overflow it would be more efficient to use a truncated, strict variant of the method, sketched in Section 2, provided we know how many derivatives are needed. The code generated by packages written in C++ will be faster.

If a function is discontinuous, e.g., defined segment-wise, the CD algorithm does not discover it automatically, and it blindly computes one of the possible values, by following the control thread of the program. This strategy may be or may be not what the user wishes, in such circumstances the technique cannot be fully automatic. We have constructed a small experimental extension of our package, replacing normal numbers by a non-standard arithmetics which includes "infinity" and "undefined", and which permits the usage of such objects as the Heaviside step function, but this direction leads towards the *symbolic* calculus, which we tried to avoid in this work. In general, the user would have to treat limit cases as carefully as he would do it on paper.

3.6. Is laziness indispensable?

It is possible to implement the derivation in a strict language which permits overloading, but the truncation code is more complicated and error-prone, although the resulting program might be faster. (The standard CD packages are of course based on strict semantics.) A combined strategy is also possible. We have reimplemented the CD in Scheme (Rice University MzScheme [21]) using lazy streams constructed with explicit thunks. The speed of the resulting program is comparable with the fully lazy solution tested under Hugs. Some execution-time space efficiency seems to be gained, since in Scheme only these thunks which are really needed occupy the memory, and the Hugs strictness analyser is not ideal. However, the coding is much more tedious, and the code is longer.

A more thorough comparison of performances is difficult, because Scheme is a dynamically typed language. Moreover, some of our algorithms (for example the definition of the exponential) exploit self-referring *variables*; this requires that either the concerned definitions contain unreadable combinations of thunks and recursive binding constructs (**`letrec`**), or those constructs must be implemented as macros. This may not be portable. Despite the standardisation of macros in the *Revised(5) Report on the Algorithmic Lan-*

guage Scheme[19], currently used dialects of Scheme often use their own syntactic extensions. A fully lazy language, especially with a good type system is much easier to use.

4. Some Applications

The application domain covered by the cited literature on CD is very wide, ranging from nuclear reactor diagnostics, through meteorology and oceanography, up to biostatistics. The authors not only used CD packages in order to get concrete results, but they have thoroughly analyzed the behaviour of their algorithms, and several non-trivial optimisation techniques have been proposed.

The examples in this section demonstrate how the lazy semantics bridges the gap between intricate *equations* which are natural formulations of many computational problems, and *effective algorithms*. The following issues are treated:

– We show how to code the solution of differential recurrences of any order, and how to construct a function defined by these recurrences. This is a standard technique for symbolic manipulation, but rarely found in a numerical context.

– We show how to automatically differentiate functions defined implicitly. Such issues are rarely addressed by the CD literature, although the mathematics involved is rather elementary, and many scientific computations, e.g., the asymptotic expansions exploit them very intensely.

– If a function obeys a differential equation, its formal solution as series can often be obtained by iterated differentiation. If the equation is singular, a naive algorithm breaks down, and the automation of the process may be difficult. We show how to deal with such an equation by transforming it into

  using a particular implicitization thereof.

– We develop an asymptotic expansion known as the WKB approximation in quantum theory. This example shows an interplay between lazy differentiation, and lazy power series, whose terms "bootstrap" themselves in a highly co-recursive manner.

– Finally, in the last example construct the Stirling approximation of the factorial, using the Laplace (steepest descent) asymptotic expansion. This is a "torture test" of our package, which shows that sometimes a good deal of human preprocessing is necessary in order to apply lazy techniques to non-trivial cases.

4.1. Recurrently defined functions

Suppose that we teach Quantum Mechanics, and we wish to plot a high-order Hermite function, say $H_{24}(x)$ in order to show that the wave-function envelope of the oscillator corresponds to the classical distribution. But we insist on using only the fact that $H_0(x) = \exp(-x^2/2)$, and that

$$H_n(x) = \frac{1}{\sqrt{2n}} \left(xH_{n-1}(x) - \frac{dH_{n-1}(x)}{dx} \right). \tag{1}$$

We do not want to see the polynomial of degree 24, we need just numerical values to be plotted. It suffices to code

```
herm n x = cc where
 D cc _ = hr n (dVar x)
 hr 0 x = exp(neg x * x / fromDouble 2.0)
 hr n x = (x*z - df z)/(sqrt(fromInteger (2*n)))
          where z=hr (n-1) x
```

(some normalization factors are omitted here), and to launch, say, **`map (herm 24) [-10.0, -9.95 .. 10.0]`** before plotting the obtained sequence. This example is a bit contrived, we could use the Rodrigues formula, or any other recurrence, but this one works in practice without problems. The efficiency of the differential recurrences is as good as any other method. The generation of the 400 numbers in the example above takes less than 20 sec on a 400MHz/130MB PC with 6MCells of heap space allotted to Hugs, which is a Haskell *interpreter*, and thus much slower than the compiled code would be. Mapping the explicit, symbolically computed form would be much faster, but this first stage is much more costly. Maple using the equivalent procedure (and reusing all lower-order forms) chokes before $n = 24$. Other recurrence schemes are more suitable.

4.2. Lambert function

We find the Taylor expansion around zero of the Lambert function defined implicitly by the equation

$$W(z)e^{W(z)} = z, \tag{2}$$

without using any symbolic data. This function is used in many branches of computational physics and in combinatorics. Many interesting differential equations have closed solutions in terms of W. Corless *et al.* [6] discuss the existence and the analyticity properties of this function. The differentiation of (2) gives

$$\frac{dz}{dW} = e^W(1 + W) \qquad \left(= \frac{z}{W}(1 + W) \text{ for } z \neq 0 \right) \tag{3}$$

whose inverse

$$\frac{dW}{dz} = \frac{e^{-W}}{1+W} \qquad \left(= \frac{W}{z}\frac{1}{1+W}\right) \tag{4}$$

gives a one-line code for the McLaurin sequence of W, knowing that $W(0) = 0$.

```
wl = D 0.0 (exp (neg wl)/(1.0+wl))
```

producing the following numerical sequence: 0.0, 1.0, -2.0, 9.0, -64.0, 625.0, -7776.0, 117649.0, -2097152,..., which agrees with the known theoretical values: $W^{(n)}(0) = (-n)^{n-1}$.

If we insert the formula (4) into any program which calculates numerically $W(x)$ for any $x \neq 0$, (for example using the Newton or Haley approximation [6]) we obtain all its derivatives at any point.

Can we use the second, apparently cheaper form of (4) which does not use the exponential? For $z \neq 0$ naturally yes, provided we knew independently the value of $W(z)$. But lazy algorithms sometimes need some intelligent reformulation in order to transform equations in algorithms, and to make co-recursive definitions effective. In the example above, there is no immediate solution, passing from $Y = \exp(-W)$ to $Y = W/z$ at $z = 0$ loses some information, we do not know any more that the value of $Y(0) = 1$. We can add it by hand, and we get for the derivative

$$Y' = -Y\left(Y + zY'\right) \qquad \text{or} \qquad Y' = \frac{-Y^2}{1+zY}. \tag{5}$$

Both forms are implementable now, and the first, recursive, is faster, because the differentiation of a fraction is more complex. We just have to introduce an auxiliary function ζ which multiplies an expression f by the *variable* z at $z = 0$. The resulting "main value" vanishes, but the result is non-trivial:

```
zeta f = D 0.0 (f + zeta (df f))
yl = D 1.0 yl' where yl' = neg yl*(yl + zeta yl')
```

from which we can reconstruct the derivatives of $W = zY$ in one line.

4.3. A SINGULAR DIFFERENTIAL EQUATION

The previous example shows also how to code the Taylor expansion of any function satisfying a (sufficiently regular) differential equation. There is nothing algorithmically specific in the lazy approach, only the coding is much shorter than an approach using arrays, indices and truncations. In some cases it is possible to treat also singular equations. The function $u(x)$ defined by $u(x^2) = x^{-\nu}J_\nu(x)$ obeys the equality

$$f'(x) = -\frac{1}{\nu+1}\left(x^2 f''(x) + \frac{1}{4}f(x)\right) \quad , \tag{6}$$

which is implicit: needing f and f'' to compute f', and singular at $x = 0$ (although this singularity is not dangerous). We may apply now our $\zeta(w)$ trick. By putting for simplicity ν equal to zero, and replacing $x^2 f''(x)$ by $\zeta(\zeta(f''))$ in (6), we obtain $f = 1.0, -0.25, 0.0625, -0.140625, 0.878906, -10.7666, 218.024, \ldots$ for

```
besf = D 1.0 fp where
  fp = neg (0.25*besf + zeta (zeta (df fp)))
```

because the second derivative is protected *twice* from being touched by the reduction of the auto-referential expression **fp**. In [15] we have used a similar trick to generate the power series solution of the Bessel equation.

4.4. WKB EXPANSION

Our next exercise presents a way of generating and handling functions defined by intricate differential identities in the domain of power series in some small perturbation parameter (*not* the differentiation variable). We derive higher order terms for the Wentzel-Kramers-Brillouin approximation, as presented in the textbook [2], and useful for some quasi-classical approximation to the wave function in Quantum Mechanics. We start with a generalized wave equation

$$\epsilon^2 y'' = Q(x) y. \tag{7}$$

with ϵ very small. The essential singularity at zero prevents a regular development of y in ϵ. Within the standard WKB formalism y is represented as

$$y \approx \exp\left(\frac{1}{\epsilon}\sum_{n=0}\epsilon^n S_n(x)\right) . \tag{8}$$

Inserting (8) into (7) generates a chain of coupled recurrent equalities satisfied by S_n. The lowest approximation is $S_0' = \pm\sqrt{Q}$, (which needs an explicit integration irrelevant for our discussion), and $\exp(S_1) = 1/\sqrt{S_0'}$, which has profited from the fact that the coefficients S_{2n+1} are directly integrable.

We propose the following expansion, which separates the odd and the even powers of ϵ. The coefficients of proportionality, and the necessity to combine linearly the two solutions differing by the sign of $\sqrt{Q}$ are omitted.

$$y \approx \exp\left(\frac{1}{\epsilon}S_0 + U(x,\epsilon^2) + \epsilon V(x,\epsilon^2)\right) . \tag{9}$$

Injecting this formula into the equation (7) gives the following differential identities:

$$U' = \frac{-1}{2}\frac{S_0'' + \epsilon^2 V''}{S_0' + \epsilon^2 V'} \qquad \text{or} \qquad e^U = \frac{1}{\sqrt{S_0' + \epsilon^2 V'}} , \tag{10}$$

and

$$V' = \frac{-1}{2S_0'}\left(U'^2 + U'' + \epsilon^2 V'^2\right) \quad . \tag{11}$$

These cross-referencing definitions seem intricate, but they constitute an *effective lazy algorithm*. The aim of this section is to show how to code $U(x)$ and $V'(x)$. The last one has to be integrated using other methods.

Until now we never really needed all derivatives of a function, and the reduction of the lazy chain stopped always after a finite number of steps. Here, in order to get one numerical value of, say $V'(x)$, we need the second derivative of U, which needs the second and the third derivative of V, etc.

The point is that U and V should be treated as series in ϵ^2, and the higher derivatives of U and V appear only in higher-order terms, which make the co-recursive formulae effective.

We have thus to introduce some lazy techniques of power series manipulation. This topic has been extensively covered elsewhere, e.g., in our own work [15]. We review here the basics. The series $U(z) = u_0 + u_1 z + u_2 z^2 + \cdots$ (with the symbolic variable z implicit) is represented as a lazy list **`[u0, u1, u2,...]`**. The linear operations: term-wise addition and multiplication by a scalar are easy (**`zip`** with **`(+)`**, and **`map`**). The multiplication algorithm is a simple recurrence. If we represent $U(z) = u_0 + z\overline{u}$, then $U \cdot V = u_0 v_0 + z(u_0\overline{v} + v_0\overline{u} + z\overline{u}\overline{v}) = u_0 v_0 + z(v_0\overline{u} + U\overline{v})$. For the reciprocal $W = 1/U$ (with $u_0 \neq 0$) we have $w_0 = 1/u_0$, and $\overline{w} = -w_0\overline{u}/U$, which result from $U \cdot W = 1$. The differentiation and integration need only some multiplicative zips with factorials, and an integration constant. The elementary functions such as $W = \exp(U)$ may use the following technique: $W' = U'W$, and thus $W = \exp(u_0) + \int U'W\,dz$, which is a known algorithm, see the Knuth's book [20], although its standard presentation is not lazy.

The terms u_i need not be numbers. They may belong to the domain **`Dif`**, or on the contrary, our differential field may be an extension of the series domain, i.e., the "values" present within the **`Dif`** structure are not Doubles, but series. The first variant is used here. Hence we have a doubly lazy structure, and we need an extension of the differentiation operator over the *variable* x which is a lazy list representing a series over ϵ. In this domain it suffices to define **`df = map df`**, or, more explicitly

```
df (u0:uq) = df u0 : df uq
```

In our actual implementation series are not lists, but similar, specific data structures with **`(:>)`** as the chaining infix constructor, and a constant **`Z`** representing the zero (empty) series, more efficiently than an infinite list of zeros. Any **`Dif`** expression p may be converted into its Taylor series by

```
taylor p = tlr 1 (fromInteger 1) p where
 tlr _ f (C x) = (x*f) :> Z
 tlr m f (D x q)=(x*f):>tlr (m+1) (f/fromInteger m) q
```

We may test the WKB algorithm and generate the approximation to the Airy function which is the solution of the equation (7) for $Q(x) = x$, for some numerical values of x. We fix the value of the *variable*, e. g. **`q = dVar 1.0`**. Then we define **`s0'=sqrt q`** and **`s0'' = df s0'`**, and the equations (10) and (11) may be coded as

```
u' = (-0.5)*>(s0'' :> df v')/(s0' :> v')
v' = p where p=((-0.5)/s0') *>(u'^2 + df u' +:> p*p)
```

where a shifted addition operator **`a +:> b`** which represents $a+\epsilon^2 b$ is defined as

```
(a0 :> aq) +:> b = a0 :> (aq+b)
```

and **`(*>)`** multiplies a series by a scalar. Now **`u'`** is a series whose elements belong to the data type **`Dif`**, but we do not need the derivatives, only the main values, so we construct a function **`f`** which returns this main value from the **`Dif`** sequence. One application of **`map f`** to the series **`u'`** suffices to obtain -0.25, -0.234375, -1.65527, -28.8208, -923.858, -47242.1, -3.52963e+006, etc. while **`v'`** produces -0.15625, -0.539551, -6.31905, -152.83, -6271.45, -391094.0, -3.44924e+007, etc., and this is our final solution. The generation and exponentiation of **`u`**, and the integration of **`v'`** give for a sufficiently small ϵ a good numerical precision. This result is known. Our aim was to prove that the result can be obtained in a very few lines of user-written code, without any symbolic variables. Other asymptotic expansions, for example the saddle-point techniques which also generate unwieldy formulae may be implemented with equal ease.

4.5. Saddle-point Approximation

We want the asymptotic evaluation of

$$I(x) = \int f(t) e^{-x\varphi(t)} dt \ , \tag{12}$$

for $x \to \infty$, knowing that $\varphi(t)$ has one minimum inside the integration interval, (see [2], or any other similar book on mathematical methods for physicists). The Laplace method and its variants (saddle point, steepest descent) are extremely important in natural and technical sciences. It consists in expanding φ about the position of this minimum p: $\varphi'(p) = 0$. Then $\varphi(t) = \varphi(p) + \varphi''(p)(t-p)^2/2 + R(t)$, and evaluating the integral

$$I(x) = e^{-x\varphi(p)} \int e^{-x\varphi''(p)(t-p)^2/2} f(t) e^{-x(t-p)^3 R} \ , \tag{13}$$

considering the expansion of $f(t)\exp(-x(t-p)^3 R)$ as polynomial correction to the main Gaussian contribution around the point where the maximum is assumed. R is a series in $(t-p)$, beginning with the constant $\varphi'''/3!$. Analytically we get

$$I(x) \approx \sqrt{\frac{2\pi}{x\varphi''}}e^{-x\varphi}\left\{f + \frac{1}{x}\left(\frac{f''}{2\varphi''} - \frac{f\varphi^{(iv)}}{8(\varphi'')^2}\right.\right. \tag{14}$$
$$\left.\left. - \frac{f'\varphi'''}{2(\varphi'')^2} + \frac{5f(\varphi''')^2}{24(\varphi'')^3}\right) + \frac{1}{x^2}\left(\cdots\right)\cdots\right\}$$

where f, φ and their derivatives are taken at p. The next terms need a good dose of patience. Even an attempt to program this expansion using some Computer Algebra package is a serious task, and the resulting formula is difficult to read. These terms *are* often necessary, for example in computations in nuclear physics or quantum chemistry, where x is proportional to a finite number of particles involved. Is it possible to compute the expansion terms without analytic manipulation? The problem is that the expressions here are bivariate, and all the expansions mix the dependencies on x and t, so we obtain a series of series. We have to disentangle it, because we want the dependence on x to remain parametric: x should not appear in the expansion. We begin with computing $\varphi(t)$ as a series at p (in the **`Dif`** domain), extracting the constant $\varphi_0 = \varphi(p)$, $\varphi''/2$ and the series R with its coefficient $(t-p)^3$:

```
phi0 :> _ :> ah :> r  = taylor phi
```

Henceforth we do not care about $\exp(\varphi_0)$ nor about the normalization, we compute only the asymptotic series. Expanding the exponential and multiplying it by f: **`u = fmap (f *) exp (Z :> neg r :> Z)`** we get

$$U = \sum_{n=0}^{\infty} x^n (t-p)^{3n} U_n(t-p) \ , \tag{15}$$

where U_n is a series in $(t-p)$. It suffices to integrate (15) with a Gaussian, but this is easy: $I_m = \int \exp(-xat^2/2)t^{2m}dt$ is equal to $\sqrt{2\pi/ax}\cdot(2m-1)!!/(ax)^m$, where $(2m-1)!! = 1\cdot 3\cdot 5\cdots(2m-1)$. Here is the program which computes the Gaussian integral of a series v multiplied by $(t-p)^m$:

```
igauss a mm v@(_:>vq)
 | odd mm = igauss a (mm+1) vq
 | otherwise =
 let cf k t | k<mm = cf (k+2) ((t*fromInteger k)/a)
            | otherwise=t:>cf(k+2)((t*fromInteger k)/a)
     ig (c0:>cq) (v0 :> vq) = v0*c0 :> ig cq (stl vq)
     ig _ Z = Z
 in (mm ‘div‘ 2, ig (cf 1 (fromInteger 1)) v)
```

where **`stl`** is the series tail, **`ig`** is the internal iterator, and **`cf`** computes the series of coefficients $(2m-1)!!/a^m$. We keep with each term an additional number m_0, the least power of $1/a$ (and subsequently of $1/x$) of the resulting

Laurent series. Applying this function to our series of series U, after having restored the coefficient $(t-p)^{3n}$:

```
dseries a u = ds 0 u where
 ds n3 (u0:>uq) = igauss a n3 u0 :> ds (n3+3) uq
```

we obtain a sum of the form

$$\sum_{n=0}^{\infty} x^n G_n\left(\frac{1}{x}\right) \ , \qquad \text{where} \quad G_n\left(\frac{1}{x}\right) = \sum_{m=0} g_{nm}(1/x)^m \tag{16}$$

The resulting infinite matrix must be re-summed along all diagonals above and including the main diagonal, in order to get coefficients of $(1/x)^{m-n}$. It is easy to prove that the sum is always finite, because the factor $(t-p)^{3n}$ makes m_0 grow faster than n . The re-summation algorithm uses m_0 in order to "shift right" the next added term, and if it can prove that there is nothing more to be added, emits the partial result, and *lazily* recurs. here is the final part of the program:

```
resum ((m0,g0) :> gq) = rs m0 g0 gq where
 rs _ Z  ((m1,g1) :> grst) = rs m1 g1 grst
 rs m0 g0@(ghd :> gtl) gq@((m1,g1) :> grst)
  | m1==m0+1 = rs m1 (g0+g1) grst          (strict sum. step)
  | otherwise = ghd :> rs (m0+1) gtl gq    (lazy iteration)

finalResult = resum (dseries a u)
```

In order to test the formula we may take $\varphi = z - \log(z)$ at $z = 1$, and we obtain in less than 4 seconds the well known Stirling approximation for the factorial:

$$n! = \int t^n \exp(-t)dt \approx n^{(n+1)} \int \exp(-n(z - \log z))dz \ . \tag{17}$$

The first terms of the asymptotic sequence in $(1/n)$ are

$$1, \frac{1}{12}, \frac{1}{288}, \frac{-139}{51840}, \frac{-571}{2488320}, \frac{163879}{209018880}, \frac{5246819}{75246796800} \dots \ . \tag{18}$$

5. Conclusions

The present work belongs to a longer suite of papers in which we try to demonstrate the applicability of modern functional programming paradigms to the realm of scientific computing [15, 16, 17, 18]. This domain is usually dominated by low-level coding techniques, since the computational efficiency is considered primordial, and although one often needs here elaborate numerical methods, sophisticated *algorithmisation tools* are rare.

This is partly due to the lack of sufficiently powerful abstraction mechanisms in standard languages used for numerical computations, such as C or Fortran. The path between an analytical formula and its implementation in a numerical context is often long. Human time is precious, and Computer Algebra packages are often exploited. Symbolic computations are often needed for insight, people like to see the analytical form of their numerical formulae. However, it is not unfrequent that the symbolic algebra is applied in despair, just to generate some huge expressions consumed by the Fortran or C compiler only, and never looked upon by a human. For many years it was typical of many computations involving differentiation.

The development of Computational Differentiation tools changed that. We know now how to compute efficiently and exactly the numerical derivatives of expressions without passing through the symbolic stage. Several highly tuned packages adapted to C, C++ and Fortran exist, and their popularity steadily increases, although they do not always integrate smoothly with existing numerical software.

It was not our aim to propose a replacement for these packages. However, on the methodological side our ambition was a little bigger. The specificity of our contribution may be summarized as follows:

– The derivation operation exists in the program at the same footing as all standard arithmetic procedures. It can be applied an arbitrary (*a priori* unknown) number of times. This makes it possible and easy to code functions defined by differential recurrences. No explicit truncation of the derivation orders, synchronisation of powers, etc. are needed. Laziness liberates the user from the major part of the algorithmisation burden.

– The usage of our package is extremely simple and straightforward. It suffices to load a very short library of overloaded, extended arithmetic methods, and to declare in a few places in the program that a given identitifier corresponds to the differentiation *variable*. Polymorphism, and the automatic type inference of Haskell does the rest.

– Thanks to the Haskell class system, the extended arithmetics remains valid for any basic domain, not only for floating-point reals. No changes are needed in order to compute complex derivatives. If the user constructs some specific arithmetic operations for polynomials or ratios of polynomials, the lifting of these operations to the differential ring or field becomes almost automatic.

The exercise of the lazy style of programming needs some experience, and the conceptual work involved may be substantial. The efficiency of current implementations of functional languages is far from ideal. The examples we have presented are intricate (there are easier ways to compute the Stirling formula), but they are *generic*, presented modularly, and their discussion is fairly complete. To us there is plenty of evidence that lazy functional languages, which permit better than many others to concentrate upon the algebraic properties of operations in complicated mathematical domains, have a nice future in the area of applied mathematics.

6. Acknowledgements

I would like to express my gratitude to the Referees, and to the Editor, whose efforts to render this paper more readable were considerable. Of course, I am responsible for all remaining faults.

References

1. Abramowitz Milton, Stegun Irene, eds. *Handbook of Mathematical Functions*, Dover Publications, (1970).
2. Bender Carl, Orszag Steven, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill, (1978).
3. Bendtsen Claus, Stauning Ole, *TADIFF, a flexible C++ package for automatic differentiation*, Tech. Rep. IMM-REP-1997-07, Dept. of Mathematical Modelling, Technical University of Denmark, Lyngby, (1997).
4. Berz Martin, Bischof Christian, Corliss George, Griewank Andreas, eds., *Computational differentiation: techniques, applications and tools*, Second SIAM International Workshop on Computational Differentiation, Proceedings in Applied Mathematics **89**, (1996).
5. Bourbaki Nicolas, *Algebra*, Springer (1989).
6. Corless Robert, Gonnet Gaston, Hare D.E.G., Jeffrey D.J., Knuth Donald, *On the Lambert W function*, Advances in Computational Mathematics **5** (1996), pp. 329–359. See also the documentation of the Maple SHARE Library.
7. Corliss George, *Automatic differentiation bibliography*, originally published in the SIAM Proceedings of *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, ed. by G. Corliss and A. Griewank, (1991), but many times updated since then. Available from the *netlib* archives (**`www.netlib.org/bib/all_brec.bib`**), and in other places, e.g. **`liinwww.ira.uka.de/bibliography/Math/auto.diff.html`** . See also [4]
8. Giering Ralf, Kaminski Thomas, *Recipes for adjoint code construction*, Tech. Rep. **212**, Max-Planck-Institut für Meteorologie, (1996), ACM TOMS in press.
9. Graham Ronald, Knuth Donald, Patashnik Oren, *Concrete Mathematics*, Addison-Wesley, Reading, MA, (1989).
10. Griewank Andreas, Juedes David, Mitev Hristo, Utke Jean, Vogel Olaf, Walther Andrea, *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM TOMS, **22(2)** (1996), pp. 131–167, Algorithm 755.
11. Hovland Paul, Bischof Christian, Spiegelman Donna, Cosella Mario, *Efficient derivative codes through automatic differentiation and interface contraction: an application in biostatistics*, SIAM J. on Sci. Comp. **18**, (1997), pp. 1056–1066.

12. Jones Mark P., *The Hugs 98 User Manual*, available from the Web site `http://www.haskell.org/hugs` together with the full distribution of Hugs.
13. Kaplansky Irving, *An Introduction to Differential Algebra*, Hermann, Paris (1957).
14. Karczmarczuk Jerzy, *Functional Differentiation of Computer Programs*, Proceedings, III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203.
15. Karczmarczuk Jerzy, *Generating power of lazy semantics*, Theoretical Computer Science **187**, (1997), pp. 203–219.
16. Karczmarczuk Jerzy, *Functional programming and mathematical objects*, Proceedings, *Functional Programming Languages in Education*, FPLE'95, Lecture Notes in Computer Science, vol. **1022**, Springer, (1995), pp 121–137.
17. Karczmarczuk Jerzy, *Functional coding of differential forms*, talk at the 1-st Scottish Workshop on Functional Programming, Stirling, (September 1999).
18. Karczmarczuk Jerzy, *Adjoint Codes in Functional Framework*, informal presentation at the Haskell Workshop, Colloquium PLI 2000, Montreal, (September 2000), available from the author: `www.info.unicaen.fr/~karczma/arpap/revdiff.pdf`
19. Kelsey Richard, Clinger William Rees Jonathan (editors), *Revised(5) Report on the Algorithmic Language Scheme*, available from the Scheme Repository: `www.cs.indiana.edu/scheme-repository/`.
20. Knuth Donald, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).
21. Rice University PLT software site, `http://www.cs.rice.edu/CS/PLT`.
22. Ritt Joseph, *Differential Algebra*, Dover, N.Y., (1966).