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 to compute point-wise derivatives of functions defined by computer programs economically and *exactly* (with machine precision). We show how the lazy algorithm formulation 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 ambition 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 fast and accurate computation of these derivatives is essential for many problems in applied mathematics. They 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 to compute tangents to trajectories, normals to surfaces, and many other geometric entities 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 they are useful to compute some combinatorial factors from the appropriate partition functions, as presented in Knuth, Graham and Patashnik's textbook on concrete mathematics [10].

The rest of the paper is organized as follows: after an overview of the concept of Computational Differentiation, we show how to implement it in a typed functional language which permits the overloading of arithmetic operations. We show how the laziness can be used to define data structures which represent numerical expressions together with all their derivatives wrt.

a given independent variable, and finally, we give a few non-trivial application examples of our package.

1.1. Overview of 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 (and requires more iterations) 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 interpolant. The complexity of the algorithm may be quite big, and its coding is rather boring.

- Symbolic computations. This is essentially the "manual", analytic 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, and 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 symbolically an expression which results 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, and the number of theoretical and applied papers is impressive. See, e.g., [4, 5, 9, 11, 12], and especially the bibliography collected by George Corliss [8]. The CD algorithms are numerical, but they yield results as exact as the numerical evaluation of symbolic derivatives. Unfortunately relatively little has been written about functional programming in this context. It seems that the implementors concentrate their attention on Fortran, C, and C++. The last one is a natural choice if one wants to exploit the arithmetic operator overloading (see the description of ADOL-C [11]). For simpler languages, such as Fortran, some source code preprocessing is usually unavoidable.

The algorithmic differentiation idea relies on standard computer arithmetic, and has nothing to do with the symbolic manipulations of data structures representing the algebraic formulae. Even a complicated expression is composed of simple arithmetic operations and elementary, built-in functions with known differential properties. Of course, a program is not 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, taking into account that for all primitive arithmetic operators the derivation is known, and that all the compositions obey the chain rule. 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 analyse the *direct* or *forward mode* of differentiation. The alternative *reverse mode* is treated in another paper [18]; its utility is more important in the multi-variate case. Our differentiation package is implemented in almost standard (enriched by some mathematical generic classes) purely functional language Haskell, tested with the interpreter Hugs [13], and relies on the overloading of arithmetic operations. We assume that the reader can follow the Haskell code.

The presented approach requires the lazy evaluation protocol which states that a function evaluates its argument only when it needs it. Despite a reasonably long history, the lazy functional techniques are rarely used in numerical computations. First, they remain relatively unknown for 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.

The package is not meant as a replacement of highly tuned and efficient low-level numerical programs. The cited Computational Differentiation packages have been optimized for performance. We show that the 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 CD texts, such as the construction of functions defined by differential recurrences, become very easy to code. This is the main goal of the paper. Our basic tools are the *co-recursive* data structures: objects defined by open, non-terminating recursive equations which would overflow the memory if implemented in a strict (not lazy) language.

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 for simplicity only 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 the division).

The elementary objects which are injected into the calculations are either 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.

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)
```

We define also two auxiliary functions specifying the constants and the *variable*, and the reciprocal function.

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

recip (x,a) =  let ix=recip x in (ix,negate a * ix*ix)
fromDouble z = dCst z
```

(or, alternatively, the equivalent definition of the binary division).

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 doesn't need to change anything in 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.

– There are some defaults recognized by the compiler, e.g., the default definition of the division: **`x/y = x*(recip y)`**.

– 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 have yet to implement the chain rule, which for every unary 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 standard 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
```

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, but always coherent. 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`** uses the main value only.

Our package does not use the standard arithmetic classes of Haskell: **`Num`**, **`Floating`** etc. for the definition of overloaded arithmetic operations. We found it more natural to introduce a modified "algebraic style" library (Prelude), which corresponds to the classical mathematical hierarchy. Our 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`**, **`Field`** or **`Module`**. The lifting of the standard numbers: **`fromInt`**, **`fromDouble`** into the field of constants of our differential domain is declared within a new class **`Number`**, orthogonal to the algebraic hierarchy.

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 [11, 4]. 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 possible 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, we show how to lift the elementary functions and their compositions, and we give some simple application examples of the constructed framework, but we propose first an easy formal introduction.

3.1. What is a differential algebra?

The theory of the domain called Differential Algebra was developed mainly by Ritt [21] and Kolchin, see also a more recent book by Kaplanski [14]. 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 we 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 legality 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. To this set of operations we add one more, the *derivation*: an endomorphism $a \to a'$ which is linear, obeys the Leibniz rule: $(ab)' = a'b + ab'$, and some regularity 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.

The derivation within a simple polynomial extension: $A[x]$ of any field A is well known and described in many books on algebra, e. g. [6]. We know also how to compute derivatives in the rational extension $A(x)$. These extensions might be considered as based on adjoining of an *algebraic indeterminate*, some "x" which may be represented symbolically in the program, and 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, and the construction of a commutative algebra on such data structures is a school exercice. Some *typed* Computer Algebra programs such as Axiom [2] exploit the mathematical knowledge, in order to optimize the compilation of well specified procedures.

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

Our approach is minimalistic, as local as possible. We want just to compute the numeric values of the expressions (for a given input), and the values of some derivatives. We don't know how many, so we require that our differential algebra is closed, in the sense that the derivation becomes its internal operation.

For any new e introduced into the domain we have to provide e', e'', $e^{(3)}$, etc. – in fact, the possibility of an infinite number of algebraically independent objects, as there is *a priori* no reason that an e' should be algebraically dependent on e (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. The computational differentiation technique – as sketched in the previous section – augmented by the lazy evaluation protocol gives to the framework a simple, precise and efficient operational semantics. Structurally the program operates upon infinite lists which might be assimilated to the Taylor series of the manipulated expressions, although the arithmetic rules are different in both cases.

3.2. THE DATA AND BASIC MANIPULATIONS

The data type we shall work with belongs to an infinite co-recursive domain **`Dif`**. The derivation operator **`df`** is defined generically, parameterized by any

basic type (usually **Double**, but it might work with rationals or complex numbers as well)

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

class Diff a where
 df :: a->a

instance Diff Double where
 df _ = 0.0
instance Number a=>Diff (Dif a) where
 df (C _) = C 0.0
 df (D _ p) = p
```

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 is the definition of constants and of the *variable*: **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)**):

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

The instance of **Eq** for our data is semi-defined. The inequality can be discovered after a finite number of comparisons, but the **(==)** operator may loop forever, as always with infinite lists, it is defined only for constants. This is normal, the equality of symbolic expressions is ill-defined as well, and the 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 the program.)

3.3. ARITHMETIC

The definitions below are straightforward, and the presentation of the arithmetic is simplified. The subtraction is almost a clone of the addition, the lifting of operators to the constant subfield is routine. The full package is available from the author. The **Dif** data type being a list-like structure, is a natural **Functor**, with the generalized **map** (**fmap**) functional defined almost trivially. This functional transforms a list $[\ldots x_k, \ldots]$ into the list of applications $[\ldots f(x_k), \ldots]$. It is useful for the definition of the **Module** class, where we define the multiplication of a composite object by an instance of its base type. From the multiplication rule it follows that the operation **df** is a derivation. The algorithm for the reciprocal shows the power of the lazy protocol – the corresponding (truncated) strict algorithm would be much longer.

```
instance Functor Dif where
 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')

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' -- caveat emptor!
   | otherwise = D (x/y) (x'/q - p*y'/(q*q))
```

(we have used the de l'Hôpital rule, which is not what the user might wish.)

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 define also the elementary – algebraic and transcendental functions for 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 special 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') =
  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 p@(D x x') = r where r = D (exp x) (x'*r)
  log d@(D x x') = D (log x) (x'/d)
  sqrt (D x x') = p where
   p = D (sqrt x) ((fromDouble 0.5*>x')/p)
  -- 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 [4], 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 vector style found anywhere, for example in Knuth [19]. 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 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.

A few words on control structures are needed. The computation of the derivatives follow the normal control thread, nothing changes. 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.

3.4. DISCUSSION

We recapitulate here the basic properties of our 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.

- 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 possible difficulties are exactly the same as with the "main" computation, perhaps *a little* more important, as usually more arithmetic operations are needed for the derivative than for the main expression. 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 considerably the computational complexity.

However, all deferred numerical operations generate closures or *thunks*, functional objects whose evaluation produce eventually (upon demand) a numerical answer. The space leaks induced by these deferred closures might be dangerous, the reader should not think that he might compute 1000 derivatives of a complex expression using our lazy towers, unless the program finds some specific shortcuts preventing the proliferation of closures. A closure is a function which keeps references to all 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" numbers, but a similar difficulty exists, although thunks are usually more compact than symbolic formulae.

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*.

If a function is discontinuous or defined segment-wise, the CD algorithm does not discover it, and it blindly computes one of the possible values, by following the control threads of the program. This strategy may be or may be not what the user wishes, in such circumstances the differentiation cannot be fully automatic. This problem has been addressed several times by the Computational Differentiation community. 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.

4. Some Applications

The application domain covered by the cited literature on CD is very wide, ranging from the nuclear reactor diagnostics, through meteorology and oceanography, up to biostatistics. The authors not only used the 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 objectives of this paper are different, they are mainly methodological. Lazy formulation of a computational problem helps sometimes to transform *automatically* intricate *equations* into *algorithms*. We show now how the formalism presented may be practically used in this context.

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

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

We don't 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. The 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 that 20 sec on a 400MHz/130MB PC with 6MB of heap space allotted to Hugs, which is a rather slow Haskell interpreter. 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 [7] given 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. The differentiation of (2) gives

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

whose inverse

$$\frac{dW}{dz} = \frac{e^{-W}}{1+W} = \frac{W}{z}\frac{1}{1+W} \tag{4}$$

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 [7]) 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 $W(z)$. Lazy algorithms sometimes need some intelligent re-formulation in order to transform equations in algorithms, and to make co-recursive definitions effective. In the papers [15, 16] we have presented a different approach to similar questions – a complete incremental arithmetic on lazy power series, the domain where the derivation is also defined trivially, but differently. Some examples therein need a little work as well. In the example above, there is *no* immediate solution, passing from $\exp(-W)$ to W/z

at $z = 0$ loses some information, we don't know any more that its value at 0 is equal to 1. We can add it by hand introducing a function Y defined by $W = zY$, and putting $Y(0) = 1$. Then $Y = \exp(-zY)$, 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 have just 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 new 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, see the trick used in [15] to produce the Bessel function. Here it would be different, but based on the same principle: 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,... 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**.

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 book [3], 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

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

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 paper [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 $U \cdot V = u_0v_0 + z(u_0\overline{v} + v_0\overline{u} + z\overline{uv}) = u_0v_0 + z(v_0\overline{u} + U\overline{v})$. 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 book of Knuth [19], 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. We shall have thus 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 don't 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 [3], 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 corrections 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

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

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, the resulting formula is difficult to read, although these terms *are* sometimes necessary, for example in computing the multi-particle phase-space volume in high energy physics, or in some other 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 again 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 iter.

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

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

5. Conclusions

We see that if instead of playing with indeterminate symbols we concentrate upon the algebraic properties of the operations in complex mathematical expressions, we can avoid the creation of complicated analytical formulae. This simplification does not come for free. The conceptual work involved may be substantial, and requires some experience in the exercise of this style, but the lazy functional programming liberates us from the major part of the *coding* burden: no more explicit sums, indices, truncations, synchronisations of powers... In a sense, this style is more close to the abstract mathematics than a typical Computer Algebra program. 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 complete. The interested reader may apply such code to many other problems. Certainly, symbolic computations are often needed for insight, and sometimes just for psychological reasons, people prefer to *see* the analytical form of their numerical formulae. Then, the usage of the extended arithmetic does not suffice, but often 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.

If we are interested not only in the computation of some derivatives, but we need also to implement complex differential identities which define our data,

the derivation should coexist on the same footing with other numeric operations. Here the laziness is invaluable. 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. A combined strategy is also possible. We have reimplemented the CD and the lazy power series packages in Scheme (Rice University MzScheme [20]) using explicit thunks. The speed of the resulting program is comparable with the fully lazy solution tested under Hugs. Some 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. A more thorough comparison of performances is difficult, because Scheme is a dynamically typed language. Moreover, some of our algorithms (for example the exponential in the domain of power series) exploit self-referring *variables*; this requires that the integration of power series must be implemented as a macro, which may not be portable. In general, the code is longer.

We think that at the present stage, a fully lazy language, especially with a good type system is preferable for the programmer.

The generalization of our framework to multi-dimensional case is simple, although the memory consuption may be quite substantial. The value of each expression is accompanied by a vector of all the partial derivatives, so instead of a linear list we generate an infinite lazy tree. Elsewhere [17] we show how to construct specific geometric objects – the Differential Forms in N-dimensional space, using our lazy Differential Algebra framework.

References

1. Abramowitz Milton, Stegun Irene, eds. *Handbook of Mathematical Functions*, Dover Publications, (1970).
2. NAG, *The AXIOM System*, Web site: `www.nag.co.uk/symbolic_software.asp` .
3. Bender Carl, Orszag Steven, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill, (1978).
4. 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).
5. 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).
6. Bourbaki Nicolas, *Algebra*, Springer (1989).
7. 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.
8. Corliss George, *Automatic differentiation bibliography*, originally published in the SIAM Proceedings of *Automatic Differentiation of Algorithms: The-*

ory, 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 [5]

9. Giering Ralf, Kaminski Thomas, *Recipes for adjoint code construction*, Tech. Rep. **212**, Max-Planck-Institut für Meteorologie, (1996), ACM TOMS in press.
10. Graham Ronald, Knuth Donald, Patashnik Oren, *Concrete Mathematics*, Addison-Wesley, Reading, MA, (1989).
11. 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.
12. 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.
13. The package and all documentation can be obtained from **`http://www.haskell.org/hugs`**
14. Kaplansky Irving, *An Introduction to Differential Algebra*, Hermann, Paris (1957).
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*, to be published, available from the author: **`www.info.unicaen.fr/~karczma/arpap/revdiff.pdf`**
19. Knuth Donald, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).
20. Rice University PLT software site, **`http://www.cs.rice.edu/CS/PLT`**.
21. Ritt Joseph, *Differential Algebra*, Dover, N.Y., (1966).