Lazy Differential Algebra and its Applications

Jerzy Karczmarczuk
Dept. of Computer Science, University of Caen, France

(mailto:karczma@info.unicaen.fr)

Abstract

We discuss the implementation of the direct Computational Differentiation paradigms in Haskell, and we show several examples of its application to some simple, but non-trivial algorithmic problems related to the scientific computing. Our examples are numerical, but the formalism is more general. We show how to *practically* extend any Field, or Ring for which an adequate arithmetic operations have been implemented, into a differential domain, by adding an infinite lazy ladder of derived expressions. We define algebraically the derivation operator, and this makes it possible to manipulate expressions containing derivatives as easily as any standard numerical code, without having to engage any symbolic manipulations of formulæ. 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. This paper may be considered as an advanced tutorial on lazy programming techniques in a specific, practically interesting context.

1 Introduction

This talk is devoted to the implementation of the differential calculus within the framework of the lazy functional programming, and might be considered as a lightweight "crusade" against the blind usage of Computer Algebra systems. We show not only how to compute derivatives of expressions coded in a programming language (Haskell, to be precise), but – and this is our principal aim – how to code, legibly and efficiently, some *algorithms which use derivatives in their definitions.* This demands that the algebra of the processed mathematical objects be closed, i. e. that we can compute the derivative of any expression, including these which are defined themselves with the aid of derivatives.

The technique is **not** symbolic, nor relies upon the finite difference approximations, but exploits the *Computational Differentiation* (or *Automatic Differentiation*)

paradigms, well known practical tools, used in many scientific and technical applications [1, 2, 3, 4]. It is based on the fact that *any* numerical expression coded in any programming language may be decomposed into a sequence of primitive steps, and for each such step: addition, multiplication, etc., the rules of differentiation are fully automatic if the program knows how to identify the *variable* of differentiation. The chain rule is also automatic, and in such a way any program can be differentiated in parallel with the main computation, independently of the intricacy of its control structures. What we need is just some overloading of the arithmetic operators to a new data domain which combines the value of an expression with the value of its derivative *at a given point*. The technique is simple to implement, and works at the machine precision, exactly as the "main" computation.

This overloading goes as follows (an over-simplified example): if a pair `(x,x')` represents the *value* of an expression together with its derivative, we define the multiplication in this domain by

```
(x,x')*(y,y') = (x*y,x*y'+x'*y)
```

which is hardly astonishing. We define the "variable" to be the structure `(x,1)`, and all the "constants" have the form `(x,0)`. The reader (or his children) may easily write other overloaded operations implementing for example `exp (x,x') = (w,w*x') where w=exp x`, etc. but we are going to be more ambitious.

Why do we need such operations? First, many numerical algorithms need derivatives: the Newton algorithm or other gradient methods, used e. g. in optimization, the stability analysis of differential equations, expansion of functions in Taylor series, asymptotic developments, etc. Too often then the derivatives are computed analytically with the aid of some Computer Algebra packages, although only the numeric code (usually horrible) is *really* needed. Many functions, for example some generating functions in combinatorics, are defined through some intricate differential equations or identities. We need derivatives in order to implement the sensibility analysis (dependence of some final result on the variations of initial conditions). In geometric modelling we need tangents to trajectories, and normals to surfaces, and in the theory of dynamical systems we need Jacobians to compute the normal forms, Hessians to find the minima, etc. All the variational problems are based on differentiation. The number of scientific problems which require the computation of gradients, Laplacians, etc. is enormous. So for a casual programmer this *should* be as inexpensive as possible.

2 Differential Algebra and its Implementation

We repeat for the last time: we do not plan to do any symbolic, formal manipulations of expressions, we are interested in a *minimalistic* extension of the standard

arithmetic, permitting to compute the derivatives at a given point. For simplicity we shall consider here the univariate problem, the generalizations are straightforward, although clumsy.

We shall need an *algebraic* definition of the derivative, since all computer manipulations are algebraic and local by nature. This is known, we need a linear mapping which obeys the Leibniz rule, the *derivation*, some regularity properties, and this is all. The easiest way to proceed is to construct a Differential Field as extension of a numeric Field by a differential indeterminate, the "differentiation variable". The original domain remains as its field of constants. However, adjoining one differential indeterminate corresponds to adjoining infinitely many *algebraic indeterminates*, $(x, x', x'', \ldots)$, as one must introduce all its derivatives as additional, algebraically independent variables. For a practical programmer: a physicist or an engineer, this is simply foolish and useless. The only practical way of dealing with this infinite amount of information is to destroy the locality, and to define an extended algebra over *structural forms*. This means that an expression "knows" that it is an exponential or a product of two simpler expressions. It stores inside itself all the structural information, and we end up with a full fledged computer algebra, where the derivation is defined by its intensional semantics, and the algorithms exploit the pattern matching in order to recognize the structure of a symbolic expression.

But for us, the True Lazy Functionalists, there is another road open. We construct by *fiat*, and extensionally, a Differential Field element as an infinite chain structurally equivalent to a list `[x, x', x'', ...]` *ad infinitum*, and we define over it a complete local arithmetic algebra, including the derivation. Of course, the algorithms which generate and manipulate these structures will be lazy. Instead of manipulate symbolic forms we shall operate upon lazy data, i. e. thunks.

We prefer to replace lists by a new recursive datatype

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

where the internal type `a` will usually be `Double`, but we might need also other objects belonging to `Num`: parametric series, continued fractions, etc., or even some functional objects which for historical reasons cannot belong to `Num`, as this class requires `Eq`, but in this talk we do not plan to modify the standard Haskell numerical hierarchies. The variant `(C x)` represents constants, but these may appear in computations under the form `(D x (D 0 (D 0 (D 0)...)))` due to some cancellations. The *variable* is obviously `(D x (C 1))`, it is the only object whose derivative is a constant equal to 1. We shall refer to it in the text putting the word *variable* in italic. Again, this form may be ambiguous and contain an arbitrarily long chain of zeros. This means that we cannot define a sound and complete equality in this domain but we shall never really use it. (For people who are *very* unhappy about this we have a rhetoric question: do physicists or other numeric code creators often use the floating point equality?...) We define the *variable* through a global function

(dVar x). The derivation operator is algebraically trivial, but semantically not-so: it forces (once) the reduction of the chain:

```
df (D _ q) = q
df (C _) = C 0
```

We see that from the derivation point of view the algebra is closed. We don't even need the C field, the mentioned already above construction

```
dZero = D 0 dZero

dconst x = D x dZero
```

is co-recursively sane [5], and may be used in practice, although for efficiency reasons we shall keep the finite representation of constants.

We may now instantiate the type L as Num, Rational, and Floating, although we would surely prefer to have in Haskell some generic classes as Ring, Field, TranscendentalExtension, etc. The Eq instance is handicapped – it is possible to find in a finite number of steps that two expressions are unequal, but the equality is defined only for (finite) constants. The type L is obviously a Functor, with a natural definition of the function map which linearly follows the chain. We leave for the reader the interesting question whether it is a Monad...

Here is the definition of the extended arithmetic, presented for simplicity without the explicit class assignments. First we define L as Module over the embedded Ring, and then the extended arithmetic.

```
k*>x | k==0 = C 0                -- Optimization
     | k==1 = x
     | otherwise = map (k*) x

(C x) + (C y) = C (x+y)
(C x) + (D y y') = D (x+y) y'     -- the same for D+C
(D x x') + (D y y') = D (x+y) (x'+y')
negate = map negate               -- Subtraction is trivial
(C x) * z = x *> z                -- The same for z*C
p@(D x x') * q@(D y y') = D (x*y) (x'*q + p*y')
```

The last rule is the only interesting one. We see that it trivially *proves* that df is a derivation. The remaining definitions are equally easy

```
fromDouble  = C . fromDouble
recip (C x) = C (recip x)
recip p@(D x x') = d where
```


tional Workshop on Computational Differentiation, Philadelphia (1996), Proceedings in Applied Mathematics 89.

[2] George F. Corliss, *Automatic Differentiation Bibliography*, originally published in the SIAM Proceedings of *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, ed. by G.G. Corliss and A. Griewank, (1991), but many times updated since then. Available from the *netlib* archives (netlib@research.att.com), or by an anoymous ftp to ftp://boris.mscs.mu.edu/pub/corliss/Autodiff

[3] Ralf Giering, Thomas Kaminski, *Recipes for Adjoint Code Construction*, Tech. Rep. 212, Max-Planck-Institut für Meteorologie, (1996), ACM TOMS in press.

[4] S.L. Campbell and R. Hollenbeck, *Automatic differentiation and implicit differential equations*. In [1], pp. 215–227.

[5] David A. Turner, *Elementary Strong Functional Programming*, Proceedings of the Symposium *Functional Programming Languages in Education*, FPLE'95, Nijmegen; Springer, LNCS **1022**, (1995), pp. 1–13.

[6] R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, D. E. Knuth, *On the Lambert W Function*, Advances in Comp. Mathematics **5** (1996), pp. 329-359. See also the documentation of the Maple SHARE Library.

[7] Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*, Theoretical Computer Science **187**, (1997), pp. 203–219.

[8] Jerzy Karczmarczuk, *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.

[9] Carl M. Bender, Steven A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill, (1978).

[10] Donald E. Knuth, *The Art of Computer Programming, Vol 2 / Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).

higher derivatives are ordered; such specific indexation is built into many Computer Algebra packages, and applied if we declare e. g. a given tensor as symmetric). The algorithms are a little heavy, and if the number of final needed expressions is small, for example if we have *one main* expression, and everything else is its derivatives, the preferred technique is the *reverse* or *adjoint* differentiation, a set of completely different algorithms extensively discussed in the cited literature, which can also be implemented functionally, but the details will be presented elsewhere.

4.2 Conclusions

In several circumstances we need analytical formulæ for expressions arising in scientific computing, to have some insight, to evaluate their non-trivial asymptotics, or just to compare them with known forms. The algorithmic, algebraic differentiation advocated in this paper is useless in such cases. However, quite frequently the analytic manipulations are needed just to generate computer codes, and the necessity to differentiate complicated expressions is a pure nuisance. We have shown that a polymorphic *lazy* language permits to implement the local differential calculus, and to solve recurrent differential identities in an *extremely* easy way. When the technique is mastered, it becomes a simple student exercice. Trying to port these algorithms into a "more efficient" imperative programming language is practically impossible.

Note that the presented formalism should not be considered as a replacement of the Computer Algebra, but rather as its algorithmic complement. We may combine the two, the structure of our algorithms does not depend on the details of the underlying "domain of constants". If we have a package which is already capable of algebraic manipulation of formal expressions, adding such lazy code as presented here, simplifies enormously all the differential computations avoiding the boring pattern-matching clauses in the typical algorithms. One should not be fascinated by the fact that lazy sequences are "infinite" and that we can compute cheaply hundreds of terms. From this point of view an equivalent strict code may be *much* more efficient, since it will not permit the creation of huge unevaluated thunks which may clobber the heap. What we have really achieved is the elimination of "administrative charges" during the programming process.

In our opinion, showing that the lazy paradigms offer *new coding tools* for the scientific community is a good way to popularize functional programming.

References

[1] Martin Berz, Christian Bischof, George Corliss, Andreas Griewank, eds., *Computational Differentiation: Techniques, Applications and Tools*, Second Interna-


```
    d = D (recip x) (negate x'*d*d)
C x / C y = C (x/y)
D x x' / p@(C y) = D (x/y) (x'/p)
C x / d@(D y y') = x*>recip d
n@(D x x') / d@(D y y') = D (x/y) (x'/d-n*y'/(d*d))
```

And some Floating functions (omitting the `C` cases):

```
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') = p where p = D (sqrt x) (0.5*x'/p)


p@(D x x') `pow` a = D (x**a) (a*>x' * p `pow` (a-1))
```

Now a casual user writes an ordinary definition, for example

```
ch z = let e=exp z in       -- Our hyperbolic cosine
       (e + 1.0/e)/2.0
```

and the application `ch (dVar 0.5)` generates 1.12762597, 0.521095305, 1.12762597, 0.521095305, ... without frowning.

In the last instance we propose the overloading of the composition – the implementation of the generic chain rule. In order to apply an unknown, "normal" function `f` to an element of `L`, we must know (magically) all its formal derivatives. We pack them together with `f` in a list, and this defines the "lifting" of `f`.

```
dlift (f:fders) p@(D x x') = D (f x) (x' * dlift fders p)
sin z = dlift (cycle [sin,cos,(negate . sin),(negate . cos)]) z
```

The theory is over. We did not discuss the implementation of the reverse differentiation which is a nice programming challenge, nor the extension to many variables, which will be treated elsewhere. We pass to some applications, and – as mentioned in the Introduction – we want to show how to exploit the laziness of our formalism to treat intricate *differential identities* in a constructive way. We manipulate "just" numbers, and in such a way we may hope to avoid the intermediate expression swell which is a nuisance in symbolic manipulation, but one should be careful: higher derivatives of complicated expressions may generate enormous unevaluated thunks which also can be dangerous.

3 Applications

From the definition of the lifted exponential and other non-polynomial functions we see how to get "for free" the full Taylor expansion when the first derivative is recursively known.

3.1 The Lambert Function

A nice case is the definition of the Lambert "W" function [6] which fulfils the equation

$$W(z)e^{W(z)} = z. \tag{1}$$

Its derivative is easily obtained by inverting dz/dW:

$$\frac{dW}{dz} = \frac{e^{-W}}{1+W} = \frac{W}{z}\frac{1}{1+W}, \tag{2}$$

and that gives the following code for the McLaurin sequence of W, knowing that $W(0) = 0$.

```
wl = D 0.0 (exp (negate wl)/(1.0+wl)),
```

numerically equal to: 0, 1, -2, 9, -64, 625, -7776, 117649, -2.09715e+6, ..., which agrees with the analytical values: $W^{(n)}(0) = (-n)^{n-1}$. Can we use the second, apparently cheaper form of (2) which does not use the exponential? For $z \neq 0$ obviously yes, provided we knew W using some numerical approximation, e. g. Newton or Haley (which may use derivatives computed with the aid of our technique, although this is not needed here). In some case we can treat the singularities 0/0 with the aid of an – admittedly *cavalière* – technique which makes use of the l'Hôpital's rule to define the division:

```
n@(D x x') / d@(D y y')
  | y==0.0 && x==0.0 = x'/y'
  | otherwise = ...
```

but it will not work in our example, we will obtain a tautology, i. e. bottom. Lazy algorithms needs sometimes some intelligent pre-processing in order to transform equations in algorithms, exactly as in Computer Algebra. In the paper [7] (see also [8]) 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 an involved gymnastics. 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

$$Y' = -Y\,(Y + zY') \qquad \text{or} \qquad Y' = \frac{-Y^2}{1+zY}. \tag{3}$$

Both forms are implementable now, and the first, recursive, is faster! We have just to introduce an innocent dirty trick, a function ζ which multiplies an expression f by the *variable* z at $z = 0$.

the burden: no more explicit sums, indices, truncations, synchronisations of powers... In a sense, this style is much more close to the abstract mathematics, and more universal than a typical Computer Algebra system. The presented examples are intricate (we are far from claiming that our example is the best way 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. We admit that an *advocatus diaboli* may, and will defend the symbolic computations on the psychological basis: the concerned person *sees* better what he or she is doing. That is often true, but the symbolic debugging may be much more painful.

4 Last comments

4.1 Generalization

The recursive overloading has permitted us to operate on bivariate expressions (series) without any special mechanisms, because the roles of the two variables were very different. This cannot be pushed further, especially if many *variables* coexist at the same footing, because we might produce ambiguities difficult to resolve: producing a series in x whose coefficients are series in y whose coefficients depend again on x is a wide road to hell. Anyway the solutions for the series and for the L-expressions might be different.

The general multivariate case of our implementation of the differential algebra introduces just some technical complications, but these influence very badly the overall complexity of the algorithms. We have to label the *variables*, and the generalized expressions become infinite trees rather than linear chains. The *value* of the second field of an expression is not a scalar, but a sequence (list or vector) whose elements are partial derivatives over the indexed variables. A constant contains a sequence of zeros, and a *variable* – 1 in exactly one position. A variant more economic in space, but costly in time does not allocate zeros, but constructs a list of associations between the label (index) of a variable, and the appropriate partial derivative. In both cases the chain rule becomes arborescent, but in practice this tree is sparse, so it is better to avoid useless zeros.

All this can be easily implemented, and in such a way we may compute gradients or Jacobians, but there is one obvious source of inefficiency which should be dealt with if we plan to generate higher derivatives – the combinatorial explosion due to the fact that such simplistic implementation will not recognize that $\partial^2 f/\partial x\partial y = \partial^2 f/\partial y\partial x$, and a program may then compute the same derivative twice. Of course, the optimization is simple, but slow. The indexing machinery should ensure that the n-th derivatives do not span a n-dimensional hypercube, but a simplex (the indices of

where the result is a pair (n, s) with n being the leading negative power of x in our left-truncated Laurent series s. This is our most complicated program…

The series `u` may be map-multiplied by any series `f` as in (5), but the factors $(t-p)^{3n}$ in u are implicit only, and accessorily the result if `igausX` is not a series, so while applying this last operation to `f*>u` we construct a normal list `asyml a 0 0 u`

```
asyml a n k (u0:>uq) =
  let (m,s)=igausX a n u0
  in (m-k,s) : asyml a (n+3) (k+1) uq
```

representing $\sum_{n=0} x^n \sum_{k=m} x^{-k} g_{nm}$, where m is a function of n, and we can trivially prove that $m > n$ ($m-k$ in the program above is always positive, excepted the first zero, and ascending). The final chord is the resummation of this series. This is an infinite sum, but it terms are shifted more and more to the right by the presence of $x^{(-m)}$, and the operation is co-recursively effective.

```
shift u = map (\(m,x) -> (m-1,x)) u

resum (_,Zs) (s0:sq) = resum s0 sq
resum  (m0,u@(u0:>uq)) s@((m1,v):sq)
 | m0>0 = 0 :> resum (m0-1,u) (shift s)
 | m1>0 = u0 :> resum (0,uq) (shift s)
 | otherwise = resum (0,u+v) sq        -- One strict step

finally = resum g0 gq where
 (g0:gq) = asyml 0 0 u
```

In order to test the formula we may take $\varphi = z - \log(z)$ at $z = 1$, and we obtain in no time the well known Stirling approximation for the factorial: $n! = \int t^n \exp(-t)dt$ which should be first transformed into $n^{(n+1)} \int \exp(-n(z - \log z))dz$. We leave for the reader some minor adjustments permitting to calculate the exponential series over the rationals (when $u_0 = 0$), and to write down the code which produces the sequence

$$\left(1, \frac{1}{12}, \frac{1}{288}, \frac{-139}{51840}, \frac{-571}{2488320}, \ldots\right) \tag{9}$$

The lesson from this exercice is the following: if instead of playing with indeterminate symbols we concentrate upon the algebraic properties of the operations on our expressions, we can avoid the creation of complicated analytical formulæ which we don't need otherwise than coded. This does not comes 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


```
zeta f = D 0.0 (f + zeta (df f))

yl = D 1.0 yl' where yl' = negate yl*(yl + zeta yl')
```

from which we can reconstruct the derivatives of W. We leave to the reader the exercise of defining W through Y in *one very short* (as usual) line of code.

3.2 Asymptotic Evaluation of Integrals; Laplace method

We must begin with a digression. We have said that we have the Taylor expansion "for free". But the datatype `L` does **not** represent power series. We propose thus to introduce another well known domain and another lazy structure `u0 :> u1 :> u2 :> ...` which will represent a formal (parametric) series $U = u_0 + xu_1 + x^2u_2 + \ldots$ with x implicit, and u belonging to any sufficiently rich arithmetic domain. For efficiency we may introduce a finite constructor for the null series which we denote by `Zs`, and `sTl` denotes the tail which returns `Zs` for the null series. The series inherit Ring from u, and if $u_0 \neq 0$, they inherit also the Field property. A comprehensive discussion of lazy algorithms over series has been published in [7, 8], we repeat here just the concentrate of the basic formulæ.

The `Series` datatype is an obvious Functor, as any other structural list, and thus we can multiply a series by an element of the u domain exactly like in the case of `L` (and we will denote it by the same operator `(*>)`). We add series element-wise, and if we adopt the following notation: $U = u_0 + x\overline{u}$, then the multiplication $W = UV$ is given by $w_0 = u_0v_0$, and $\overline{w} = u_0\overline{v} + \overline{u}v$. The reciprocal $W = 1/U$ is defined as $w_0 = 1/u_0$, and $\overline{w} = -w_0\overline{u}W$. Other functions are based on the technique of formal differential equations discussed e. g. in [10], but implemented by us differently. It is easy to calculate incrementally the derivative over x of a series. If it were a normal list we would `zipWith (*)` its tail with `[1..]`, and if we introduce an additional parameter – the integration constant, the formal integration is equally easy, it suffices to zip-divide the whole series by `[1 ..]` and to insert the integration constant at the head. The generalization of the `zip` relators to `L` or to `Series` is straightforward.

Suppose we need to compute $W = \exp(U)$. We see immediately that $W = \exp(u_0) + \int U'W$, the logarithm $W = \log U = \log(u_0) + \int U'/U$, the power $W = U^\alpha = u_0^\alpha + \alpha \int U'W/U$, etc. In the cited papers we have shown several other formulæ, e. g. how to compose series and how to reverse them incrementally, and thus the only missing ingredient is the conversion of `L` into `Series` through zip-dividing the chain by factorials. We omit for simplicity the "`C`" clauses, and we get:

```
taylor f = tlr f 1 where
  tlr (D z z') n = z :> tlr (z'/C (fromInt n)) (n+1)
```

or something equivalent. The reader should not confound the *variable* of the differential domain with the formal parameter of the series, and he should understand the following relation between the two objects: the transformed L-expression `f: s = taylor f` is a sequence $f^{(n)}(z)/n!$, so, as a series in x it represents the object $f(z+x)$, where we have conceptually considered `f` as a function f of the *variable* z, although obviously it is not a function. What matters is the quasi-locality of this object. If we could explicitly (exactly if truncated, or approximatively), for a given concrete value of `a`, compute the sum equivalent to the Horner scheme

```
ssum (s0:>sq) = s0 + a * ssum sq      -- Termination clause missing
```

we would obtain the value of $f(z+a)$, such is the magic of the infinite information hidden in the differential tower. But the program will not answer any question about the convergence of this sum.

We pass to the main exercice, the asymptotic evaluation of

$$I(x) = \int f(t) e^{-x\varphi(t)} dt, \tag{4}$$

for $x \to \infty$, knowing that $\varphi(t)$ has one minimum inside the integration interval, see for example [9], or any other similar book. The standard Laplace method consists in developing 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

$$I(x) = e^{-x\varphi(p)} \int e^{-x\varphi''(p)(t-p)^2/2} f(t) e^{-xR}, \tag{5}$$

considering $f(t)$ and $\exp(-xR)$ as polynomial corrections to the main Gaussian contribution around the maximum. R begins naturally with $\varphi'''(p)(t-p)^3/3!$. We get

$$I(x) \approx \sqrt{\frac{2\pi}{x\varphi''}} e^{-x\varphi} \left\{ f + \frac{1}{x} \left(\frac{f''}{2\varphi''} - \frac{f\varphi^{(iv)}}{8(\varphi'')^2} - \frac{f'\varphi'''}{2(\varphi'')^2} + \frac{5f(\varphi''')^2}{24(\varphi'')^3} \right) + \frac{1}{x^2} \left(\cdots \right) \ldots \right\} \tag{6}$$

where f, φ and their derivatives are taken at p. The next terms need a *good* dose of patience. But even an attempt to program this in Maple or some other Computer Algebra package is a serious torture (i.e. an excellent student exercice...), unless somebody included a magical package which knows all the tricks of the art of asymptotic expansions. The resulting formula is awful, and nobody needs its analytical form, although these terms **are** sometimes necessary, for example in computing the multiparticle 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. In general, the Laplace method and its variants

(saddle point, steepest descent) are *extremely important* in natural and technical sciences.

Is it possible to compute all this without analytic manipulation? The problem is that the expressions here are bivariate, and all the expansions will mix the dependencies on x and t. But we want that all the dependence on x remain parametric, so the procedure must be careful. The main engine is the Gaussian integral

$$\int e^{-xat^2/2} U(t) dt, \tag{7}$$

where $U(t)$ is given by its Taylor series (for simplicity we put the extremum of the Gaussian at zero), and a denotes φ''. The integral $\int \exp(-xat^2/2)t^{2m}dt$ is known and equal to $\sqrt{2\pi/ax} \cdot (1 \cdot 3 \cdot 5 \cdots (2m-1)/(ax)^m)$. Neglecting the square root factor in front of the resulting series, the asymptotic expansion in $1/x$ is simply coded as

```
igauss a u = gs 1 u where
  gs mm (u0:>_:>uq) = u0 :> gs (mm+2) (mm/a *> uq)
```

We have now to convert (5) into a form which disentangles the x and t dependencies so that we can apply (7), and then reduce the positive and negative powers of x. The procedure `igauss` should be generalized a little. In fact, if after having computed the saddle-point p and the corresponding L-expression φ we compute

```
(phi0 :> _ :> ah :> r) = taylor phi
a=2*ah                             -- a = phi''
u = exp (Zs :> negate r :> Zs)    -- Yes! This a series of series
```

the expression `u` is a series

$$\sum_{n=0}^{\infty} x^n (t-p)^{3n} U_n(t-p), \tag{8}$$

where again U_n is a series in $(t-p)$. Now each term in (8) can be integrated by (7) generalized to a form which explicitly contains the leading power of t in front of U. The integral $\int \exp\left(-xat^2/2\right) t^m U(t) dt$ is coded as

```
igausX _ _ Zs = (0,Zs)
igausX a m u@(_:>uq)
 | odd m = igausX a (m+1) uq
 | otherwise =
    let gs k f u@(u0:>ur)
         | k<m = gs (k+2) (f*fromInt k/a) u
         | otherwise = (f*u0) :> gs (k+2) (f*fromInt k/a) (sTl ur)
        gs _ _ Zs = Zs
    in (m `div` 2,gs 1 1 u)
```