Scientific Computation and Functional Programming

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

January 20, 1999

(mailto:karczma@info.unicaen.fr http://www.info.unicaen.fr/~karczma)

Abstract

We advocate the usage of modern functional programming languages, and lazy functional techniques for the description and implementation of abstract mathematical objects in Quantum Mechanics, needed both for pedagogical purposes, and for some real, not too computationally intensive, but conceptually and algorithmically difficult applications. We show how to perform simple *abstract* computations on state vectors, and we discuss the construction of some lazy algorithms, which facilitate enormously the manipulation of potentially infinite data structures, or iterative processes. Lazy functional techniques may replace in many cases the usage of symbolic computer algebra packages, and often offer additionally an interesting algorithmic complement to the manipulation of mathematical data, more efficient than blindly used symbolic algebra, and easily integrable with the numerical code.

1 Introduction to Functional Style

1.1 Elementary examples

The progress in the *active* usage of software tools by a computing physicist is accompanied by a deep polarization – on one hand we see highly tuned numerical low-level codes, efficient and illegible, and on the other – an intense exploitation of computer algebra packages which help to prepare the numerical program when the "formula preprocessing" becomes unwieldy for a human. But a physicist: researcher or teacher interested in the methodology of computations might need some tools which would bridge the gap between thinking, algorithmization and coding, which would facilitate a more abstract approach to programming, where the vectors in the Hilbert space are palpable entities, where the differential forms are geometric objects and not just symbolic formulae. And such a programming system should be easy to learn, and should avoid "polluting" the code by administrative details: verbose loop organization with dozens of exception guards, many special cases with appropriate control switches, and coercions of the mathematical objects into standard, and not too intuitive data structures. And very cumbersome synchronisation of expansion orders while coding some perturbation developments.

The aim of this paper is to present some not too well known *lazy functional programming techniques* which might be useful for a theoretician, especially in the field of Quantum Theory, where the coding is notoriously difficult, due to the high level of abstraction involved. (See [1, 2].) In general, abstractions became recently easier to implement thanks to the object-oriented languages and libraries, but the evolution of algorithms is slow.

We shall introduce and use the programming language Haskell [3], which is a *de facto* standard in this domain, although other, such as Clean [4] seem also promising. The basic idea is that functions may and should be processed as data: they may be stored, combined and transformed. They will (together with other data) constitute our world of "concrete abstractions". Modern functional languages have other specificities as well: they are strongly typed, but the types are deduced automatically by the system, and there are no type declarations. Functions (e.g. the arithmetic operators) might be overloaded, and datatypes which may be processed by a given set of functions, such as *vectors*, elements of a domain where addition and

multiplication by scalars is defined, might be declared as belonging to the **class** `VectorSpace`. Haskell has thus a strong flavour of object-oriented programming, but the class system is independent of the data representation, only the common "behaviour" is factorized. Moreover, its syntax is extremely compact, almost no keywords, elegant usage of the layout: indenting means continuation of the previous construction, and permitting the declaration of user operators. The definition of a function which computes the hyperbolic sine goes as follows:

```
sh x = (exp x - 1.0/exp x)/2.0
```

without redundant parentheses or keywords. The compiler recognizes automatically that `x` and the result must be real, because the operations involved are real. In fact, this is not true... *Any* arithmetical domain which permits the automatic conversion from reals and defines the (overloaded) function exp is eligible, so this definition holds also for complex numbers. In functional programming there are no assignments nor side-effects, but the usage of local variables is possible and encouraged. The definition below is more efficient:

```
sh x = let y=exp x in 0.5*(y+1/y)
```

One more attribute of this language will be *absolutely essential*: the "laziness", or "non-strictness", which means that the argument passed to a function is not evaluated before the function uses it. If this argument is an expression which forms a compound data structure, the receiving function gets a *thunk* – a parameterless function whose evaluation makes this data structure. This is *absolutely transparent* for the user; the only thing to remember now is the fact that if the function does not need this argument, it will never be evaluated. So this silly expression: $f(2.5/0.0)$ will not fail if $f(x)$ doesn't need x. We shall see more clever application of this strategy, and in particular the manipulation of *infinite* data structures.

Final remark of this section: we shall present some abstract examples in Quantum Mechanics, but *we don't manipulate formulae nor symbolic indeterminates!*. The final realisation of all data is numeric, and the intermediate objects are lazy functional data structures and partially applied functions. The programmer who writes a Haskell program constructs his code in an abstract way, but the "main" program which is just an expression, forces the evaluation of all delayed partial applications.

In the following we shall neglect, for syntactic simplicity, some type conversions and class declarations in Haskell, which must be added manually. Not all programs are thus directly runnable, the corrections are semantically important, but cosmetic. The reader might appreciate thus a little better the compactness of our codes.

1.2 More complex definition

We repeat: a pure functional program has no side effects. A variable assigned a value identifies with it, and cannot be re-assigned. All loops are implemented through (optimized) recursion. Here is a function `mysin` which computes the sine of a real number, using the recursive triplication formula: $\sin(3x) = 3\sin(x) - 4\sin(x)^3$. This example should be considered as our little, concentrated "Haskell manual".

```
mysin eps x = msn x where              -- msn is a local function
  msn x | abs x < eps  = x
        | otherwise = let y=msn(x/3)  -- y is a variable local to msn
              in y*(3-4*y^2)
```

The function `mysin` takes two arguments, `eps` and `x` – note the absence of parentheses – and defines a *local* unary function `msn` to avoid the cluttering the recursive definition by the presence of the spectator ϵ. Instead of the classical **if-then-else**, or **case** forms (which exist also), we have used the "`|`" alternative construction, where `otherwise` is just a synonym for `True`. A typical user will freeze his precision once, and he might define: `msin x = mysin 0.00001 x`, but we know that Haskell uses the *normal order of evaluation*, the form `f a b` means `(f a) b`, and may be understood as a "curried" application: `f` is applied to `a`, and the result is a function applied to `b`. This implies naturally the possibility to abbreviate the definitions with identical last arguments on both sides, and we code finally `msin = mysin 0.00001`. The RHS of this definition is a *partial application* of the function `mysin`, a perfectly legitimate *functional* data, an abstraction. All abstractions should finally be instantiated (applied), because a function is an

opaque, compiled object. The test: `let a=2.67 in msin a - sin a` returns -4.76837e-007. We might suggest a physical analogy: if `(f x y)` is defined as a force between bodies at the positions `x` and `y`, the object `(f x)` represents a *field of force* generated by a body at `x`. It is possible to construct such forms as `(3/)` or `((-)2)`, denoting respectively a function which divides 3 by its argument, or a function which subtracts 2 from the argument.

In a language which permits the creation of dynamical functions, the creation of dynamical data is also natural, and lists: `[a,b,c]` are used more often than arrays (which exist also). The colon is the list constructor operator (Lisp `cons`), and in order to sum all the elements of a numerical list we might define a recursive function

```
lsum [] = 0              -- if empty
lsum (x:xq) = x + lsum xq
```

where we note two particularities of the language: the parameters of a function might be *patterns* like in Prolog, and not just variables, which automatizes the structural recognition of the arguments, and a function might be defined by a set of clauses discriminated by differently structured arguments, which avoids the usage of `case` or conditionals. A more experienced user might not define this sum recursively, but use instead a standard generic functional, e.g. `lsum=foldl (+) 0`, where `foldl op ini l` applies the binary operator `op` to all the elements of `l` starting with the initial value `ini`. The definition of `foldl` resembles `lsum`, but instead of summing, the operation passed as argument is applied. Such generic definitions are predefined, the standard library of Haskell contains several dozens of them, and they shorten substantially typical programs. There are also such functionals as `map` which applies a unary function to the elements of a list, transforming `[x1, x2, ...]` into `[f x1, f x2, ...]`, and `zipWith` – a functional which convolves two lists into one, applying pairwise a binary operator between corresponding elements,

```
zipWith op (x:xq) (y:yq) = op x y : zipWith op xq yq
```

etc. This last functional may be used to add or subtract series or other sequences term by term.

A reader acquainted with Lisp will find all this quite simple. But he may be disturbed by the following definitions which are perfectly correct:

```
ones = 1 : ones
integs n = n : integs (n+1)
```

The first represents an infinite list of ones, and the second applied to a concrete number `m` produces the infinite list `[m, m+1, m+2, m+3, ...]`. They are recursive generating definitions without terminal clauses, which we shall call co-recursive. Their existence is based on the *lazy semantics* of the language. The application `integs 4` creates a list whose head is equal to 4, but the tail is not reduced (computed), the system stores there the *thunk*, whose evaluation returns `integs 5`. If the program doesn't need the second element of this list, it will never get evaluated, this evaluation is forced automatically by demanding the value of a delayed object. We get then the number 5, and the thunk which will generate `integs 6` hides behind it.

Lazy lists replace loops!. We can create an infinite list of iterates `[x, f x, f (f x), f (f (f x)), ...]`:

```
iterate f x = x : iterate f (f x)
```

and in a separate piece of program we may consume this list, looking for the iteration convergence. Separating the generation from the analysis of data is possible because a *piece of data* contains the code executed only and immediately when the program looks upon it. In order to test such a program we demand to process or to display an initial *finite* segment of such infinite list. The user writes `take 4 (integs 5)`, and the program prints `[5, 6, 7, 8]`.

The following example is more intricate. What does represent the following definition?

```
thing = 0:q
  where q = zipWith (+) ones thing
```

The first element of the `thing` is zero. So, `zipWith` can at least perform the summation of the heads of its arguments, and the first element of `q` becomes 1. But this is the value of the *second* value of `thing`, which implies that the second element of `q` is equal to 2, giving 3 as the third element of `thing`!. We obtain 0, 1, 2, 3,... The co-recursive definitions may be short, but quite elaborate, and what is important for us here – **we write just a recursive *equation*, and it becomes an effective *algorithm***. This is not possible without laziness.

2 Laziness at Work

2.1 Power Series

Suppose that the list `u=(u0:uq)` represents an infinite power series $U = u_0 + xu_1 + x^2u_2 + \ldots = u_0 + x\overline{u}$, where $\overline{u}$ is the tail of the list, `uq`, and x is a conceptual variable, not present physically in the data. Adding two such series is trivial, we use our old acquaintance `zipWith (+)`. Multiplying by a constant `c` uses `map (c*)`. How do we multiply them? Easily. We see that

$$W = UV \quad \longrightarrow \quad w_0 + x\overline{w} = (u_0 + x\overline{u})(v_0 + x\overline{v}) = u_0v_0 + x(u_0\overline{v} + \overline{u}v) \tag{1}$$

which is a perfectly decent co-recursive algorithm. In order to find the reciprocal we use the formula

$$\frac{1}{u_0 + x\overline{u}} = \frac{1}{u_0} - \frac{1}{u_0}x\overline{u}\frac{1}{u_0 + x\overline{u}} \tag{2}$$

which again is a correct lazy algorithm despite its auto-referential form, the reciprocal at the right is "protected" from the recursive evaluation, we get immediately only its first element. We can easily code and check all this.

```
(u0:uq)*v@(v0:vq) = (u0*v0) : (u0*:v + uq*v)
recip (u0:uq) = let z=recip u0
                    w=z : map (negate z *) uq*w  in w
-- Now, do something concrete:
take 10 (recip (p*p)) where p=1.0 : 1.0 : repeat 0.0
```

This will give us the list 1.0, -2.0, 3.0, -4.0, etc. of length 10. The predefined function `repeat` generates an infinite list, replicating its argument. The operator `(c *:)` is the multiplication of a series by a number `c`, and it is our private shortcut for `map (c *)`. The notation `v@(v0:vq)` informs the compiler that this parameter has the name `v` and the structure `(v0:vq)`.

More lazy manipulations in the series domain: integration, algebraic and transcendental functions, series composition and reversal, and some algorithms dealing with other infinite data structures, e.g. continuous fractions, may be found in [6, 7]. For example, the differentiation is just a `(zipWith (*))` of the series and the natural numbers sequence, and the integration is the analogous division. But integration is structurally lazy, it needs an additional parameter – the integration constant at the beginning, which pushes the remaining elements to the tail of the result. This tail may be generated by autoreferrent recurrence, which makes it possible to define the exponential by the following contraption. Suppose that for $U = (u_0, \ldots)$ we have $W = \exp(U)$. Then, $W' = U' \cdot W$, and $W = \exp(u_0) : \int U'W$. This is an *algorithm.* The reader may find it e. g. in the second volume of Knuth [5], but the code presented therein will be 10 times longer than ours.

In [6] we have shown how to use the lazy development of the Dyson equation to generate all Feynman diagrams in a zero-dimensional "field theory" (which is quite simplistic: the diagrams are just combinatorial factors, but the algorithmic structure of the perturbative expansion is sufficiently horrible to recognize the usefulness of lazy techniques). The reader might find the discussion of lazy series elsewhere, e.g. in [7], but here our aim is to show how they may be coded, and not to play with them.

2.2 Other Lazy Data, and Algorithmic Differentiation

How to compute **exactly**, i. e. with the machine precision, the derivative of *any* (for simplicity: scalar, univariate) expression $f(x)$ given by a coded function f? Usually this is considered as an analytic problem,

which needs some symbolic computations. But it is known for years that the differentiation formally is an *algebraic* operation, and we shall show how to implement it in Haskell in an easy and efficient way. We take a domain, e. g. all real numbers, which form together the field of (differential) constants, and we augment this domain by a special object, the *generator of a differential algebra*, which we may identify with the differentiation variable (it doesn't have to possess a name, but is has a numerical value). Our domain contains thus the numbers, our abstract "x", the arithmetic operators, and we close the algebra by defining a special operator `df` which should compute the derivatives, mapping the domain into itself, as any arithmetic operation.

We will do it in an apparently completely insane way, *extensionally*. The new datatype is an infinite sequence which contains the value of an expression, the value of its first derivative, the second, third, etc.... A constant number is represented by $[c, 0, 0, \ldots]$, and the "variable" with *value* x by $[x, 1, 0, 0, \ldots]$. (In practice we will optimize this, the constants will be separated into differently tagged data items.) The differentiation operator is trivial, it is just the tail of such a sequence. And now comes in the miracle: we can close the algebra of such sequences upon the arithmetic operations, exactly as we have done with power series. We could use normal Haskell lists, but we will introduce a special data defined as follows:

```
data Dif = C Double | D Double Dif
```

which means that the *type* `Dif` is a record which might be a constant (tagged with the symbol `C`), or a general expression (tagged by a symbol `D`) with two fields. The first is numeric, and the other is – naturally – a sequence starting with the first derivative, which is again an expression of the same type. The differentiation is *defined* as:

```
df (C _) = C 0          -- The value of the const. is irrelevant
df (D _ q) = q          -- trivial.
```

And here we have some definitions (omitting trivial cases with constants only, such as `(C x)+(C y)=(C (x+y))`):

```
p@(C x)*(D y y') = D (x*y) (p*y')   -- Mult. by a constant

(D x x') + (D y y') = D (x+y) (x'+y')            -- Linearity
p@(D x x')*q@(D y y') = D (x*y) (x'*q + p*y')    -- Leibniz

recip p@(D x x') = ip where         -- Reciprocal: Auto-referencing!
    ip = D (recip x) ((negate x')*ip*ip)

exp p@(D x x') = r where r = D (exp x) (x'*r)
log p@(D x x') = D (log x) (x'/p)

-- General chain rule:
lift (f:fderiv) p@(D x x') = D (f x) (x' * lft fderiv p)
sin z=lift fz z where            --  fz=sin, sin', sin'', sin''', ...
      fz=sin:cos:(negate . sin):(negate . cos):fz

p@(D x x') `power` a = D (x**a) ((C a)*x' * p `power` (a-1))
```

etc. We repeat: our algebra contains the differentiation operator at the same footing as other arithmetic manipulations, and we don't answer the question: "how to differentiate a product", but we define the appropriate multiplication operator for this algebra. The function `lift` permits to "plug-in" in the algebra all "black-box" functions, whose formal derivatives are known, as exemplified by the definition of sine. The dot is the composition operator: `(f . g) x = f (g x)`.

The usage of the system is transparent for the user. Taking our definition of hyperbolic sine without any modifications, and applying to the "variable" `(D x (C 1.0))`, where `x` has some numerical value, produces the infinite sequence: `sh x`, `ch x`, `sh x`, ...

This is the lazy variant of the technique known as the Algorithmic (or Computational) Differentiation (see [8, 9], and the references in [10]). It should be noted that any decent programming language which

allows the overloading of arithmetic operators into the domain of user data structures, as C++, may be used to compute the first or second derivatives as shown above, the Computational Differentiation is an established, practical field, known and implemented. But in this limited case the domain of expressions is not closed from the point of view of the Differential Algebra, and the code is *much* more complicated than ours.

3 Functional Abstraction and Quantum Mechanics

3.1 The Notorious Oscillator

In this section we will develop the abstract, "theoretical" approach to classical quantum problems implemented almost directly in Haskell. We shall avoid all trivialities, but there is a number of conceptual questions. How do we represent our Hilbert space? We don't want to manipulate *formulae*, but rather mathematical objects.

If we use the standard Fock space basis $|n>$, with integer $n \geq 0$, what we *really* know is the definition of $<m|n> = \delta_{mn}$. We need thus to define the state vectors as vectors, and impose the orthogonality condition for the basis. But $|n>$ is not the only basis, perhaps we would like to solve the Schrödinger equation for $|x>$ with x real, or use the momentum space, or coherent states parametrized by complex values. We shall define a fairly universal datatype

```
data V = N Integer | X Double | P Double | C Complex | R
```

which becomes useful when we define some operations over it. The signification of the special tag `R` will be clarified below. The object `(N 6)` represents "somehow" the basis vector $<6|$ (ket, if you wish; we will not insist upon the differences between the bases and their duals, although we will keep the mathematics sane). The `N` alone is a function which represents the *entire* basis.

The first definition is not the addition of two vectors, because anyway we don't know what to do with, say, $<5| + <2| \ldots$ But we can define the scalar product

```
brk (N n) (N m) | n==m  && (n>=0) && (m>=0) = 1  -- Under vacuum ...
                | otherwise                 = 0  -- Orthogonality
```

Trivial? Yes. Less trivial is the abstraction `(brk (N m))` which can be considered as the representation of $|m>$. This is *a function* which "awaits" a tagged number `(N k)` to produce a scalar. We declare now for these functions a linear structure. (For technical reasons it is difficult to apply in standard Haskell the operators "`+, *`" etc. for functions, and our package used special operators "`<+>`", but we will simplify the presentation.) So, within the **class** of objects of type `v = brk (N n)` we may define

```
(v1 + v2) nk = v1 nk + v2 nk   -- Adding two functions; nk=N k
```

Analogously we may define the multiplication by a scalar

```
(x *> v) nk = x * v nk  -- (or: (x *> b) nk nm = x * b nk nm )
```

Our minimalistic approach says only that $<k|(|m> + |n>) = <k|m> + <k|n>$, etc. We are not too ambitious yet.

Here are the annihilation and creation operators `an` and `cr`, and their addition and multiplication:

```
an b 0 m = 0                -- Annihilation of the vacuum
an b (N n) (N m) | (n<0) || (m<0) = 0  -- We might need it later
                 | otherwise = sqrt n * b (N (n-1)) (N m)
cr b (N n) (N m) = sqrt(n+1) * b (N (n+1)) (N m)

(op1 + op2) b nk nm = op1 b nk nm + op2 b nk nm  -- Linearity

op1 * op2 = op2 . op1                       -- Composition
```

where the reader should note the interesting fact that the a, a^+ multiplication is a "contravariant" composition. Indeed, the operator $N = a^+a$ is given by:

```
(cr * an) brk (N n) (N m) = an (cr brk) (N n) (N m)
  = sqrt n * (cr brk (N (n-1)) (N m)) = n * brk (N n) (N m)
```

This is a typical proof of the correctness of an abstract functional program.

The reader might be unhappy by the fact that, say, `(brk (N 5))` is an opaque functional object, not a "normal" data item. We can apply it and combine it, but apparently it is not possible to see that it contains 5. Our special tag `R` might help us. As it belongs to the `V` datatype, it is possible to define a special scalar product just to extract the hidden value:

```
brk (N x) R = x
```

This linear algebra on functionals which the reader has probably seen many times on paper, but seldom in a computer, will help us to define and to solve *numerically* the recurrence relations for the Hermite functions. We use the standard calculus and the recursivity to compute

$$<x\,|\,n+1> = \frac{1}{\sqrt{n+1}}\left(\sqrt{n}<x\,|\,n-1> - 2\frac{d}{dx}<x\,|\,n>\right), \tag{3}$$

knowing from $<x\,|\,a\,|\,0> = 0$ that $<x\,|\,0>$ is equal to $\exp(-x^2/2)$. For simplicity we do it independently of our bra-ket algebra, but we can *easily* incorporate it into the formalism discussed above by replacing the parameter type of the basis variant `X Double` by `X Dif`.

Even better, in Haskell it is possible to define restricted parametrized types bundled in *classes*, for example all types which accept standard aritmetic operations belong to such classes as `Number`, `Fractional`, etc., and our real implementation of the differential algebra defines the data `(Dif a)` where `a` is *any type* belonging to the generalized "numbers". The type `Dif a` is also a generalized number. Our abstract basis will be parametrized by such numbers, and an object `(X x)` may contain a real, a complex, a lazy differential "tower", a power series, etc. But we cannot discuss type classes in these notes.

Here is the generating program:

```
herm n x = cc where
 D cc _ = hf n (D x (C 1.0))
 hf 0 x = exp(negate x * x / 2.0)
 hf 1 x = 2.0*x*hf 0 x
 hf n x = (C (sqrt(n-1))*hf (n-2) x
  - 2.0*df(hf (n-1) x))/(C (sqrt n))
```

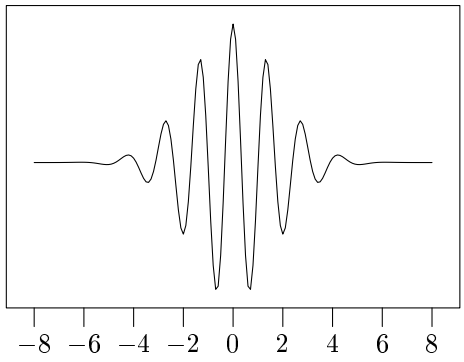

Figure 1: Hermite function of degree 16

The aim of the exercise is to prove that the method works in practice. It does, the expression `map (herm 16) [-8.0, -7.9 .. 8.0]` produces a list which can be plotted, giving on Fig. (1) what we should expect. The solution is not extremely efficient, but the plot at the right is the result of the program above, without any optimization, obtained in a few seconds. We teach our students various recurrence relations. How often did we mention that the recurrences involving the differentiation are good for symbolic manipulations, but numerically utterly useless?...

3.2 Perturbation Theory and the Anharmonic Oscillator

We shall use the annihilation and creation operators for the generation of the *full perturbation series* for the Hamiltonian $H = a^+a + \lambda x^4$. We define accessorily a "partial multiplication by a constant"`(cm c)` defined by: `cm x b = x *> b`. Our perturbed Hamiltonian is obviously given by `h'` below:

```
x = cm (1/2) * (an + cr)
x2=x*x;         h'=x2*x2
```

We construct now the perturbation series starting with the Brillouin-Wigner formulation, and we find the corrections to the ground state: $|E_0> = |0> + \lambda |\overline{E}>$, with the eigenvalue $E_0 = \epsilon_0 + \lambda \overline{E}$. It might seem preposterous to repeat here a well-known textbook derivation, but *this* presentation the reader has probably never seen before.

The unperturbed energies are $\epsilon_k = k$, and both $|\overline{E}>$ and $\overline{E}$ should be considered as series in λ. The normalization is: $<0|E_0> = 1$, which implies $<0|\overline{E}> = 0$. We have

$$(H_0 - E_0)\,|E_0> = -\lambda H'\,|E_0>, \tag{4}$$

which gives, after bracketing it with $<0|$ and $<k|$ for $k \neq 0$ the following two equations

$$\overline{E} = <0|H'|E_0> = <0|H'|0> + \lambda <0|H'|\overline{E}> \tag{5}$$

$$<k|\overline{E}> = \frac{1}{\epsilon_k - \epsilon_0 - \overline{E}}\big(<k|H'|0> + \lambda <k|H'|\overline{E}>\big). \tag{6}$$

We need one more trivial transformation

$$<k|H'|\overline{E}> = \sum_m <k|H'|m><m|\overline{E}>, \tag{7}$$

and knowing that for a polynomial perturbation the sum over m is finite, we claim that the equations above constitute *an algorithm*. We will need to import our small series package constructed as shown in section (2.1), which does some simple arithmetics on infinite lists, and we code

```
vmat k j | (j<0) || (k<0) = 0.0      -- Matrix el. of h'
         | otherwise = (h' brk) j k  -- Redundant parentheses

ebar = vmat 0 0 : hket 0             -- This is the result
hket k = sum [vmat k j *: eket j | j <-[k-4,k-2 .. k+4] ]
-- (Sum over non-vanishing matrix elements only.)

eket 0 = zeros  where zeros = 0:zeros       -- Null series
eket k = (vmat k 0 : hket k) / (k : ebar)
```

and if we ask for the value of the energy `ebar`, we get the sequence: 0.1875, 0.164062, 0.325195, 0.942535, 3.49705, 15.6208, 81.1605, 479.862, 3179.7 23344.9, 188192.0, 1.65336e+006, etc. This is possibly one of the shortest programs performing this computation ever written. Of course, it does not affects our humble respect for the general recursive formalism elaborated in [11]. Note a nice syntactic shortcut called the *list comprehension*: the form `[f x | x<-list]` is a legible variant of `map f list`. The operator `sum` is a predefined summing functional, equivalent to our `lsum`.

3.3 Caveats

We must confess that there are some little white lies and trivial omissions which do not need to be commented here (for example the necessity to declare the precedence of new infix operators, or the usage of a particular data structure for the series, slightly different from normal lists; the overloading of arithmetic should follow some formal rules concerning the Haskell classes, etc.). However, there is also one inacceptable sin. The reader wishing to test this algorithm will obtain 6, perhaps 7 terms of the perturbation series, and the program will stop because of the memory overflow. Increasing the available heap by a factor of 10 generates one more term, perhaps two, and the disaster repeats. The complexity of this implementation is exponential, or worse.

Lazy semantics is not a free lunch! A deferred expression which waits to be evaluated remains stored in the memory as a dynamically created procedure with its code and the references to all its global variables. If a lazy program is badly written, the memory is polluted by very big thunks, and we obtain the lazy functional equivalent of the intermediate expression swell in symbolic computation. Needless to say, if we couldn't correct this fault, this paper would have never been written.

We have defined two *functions* $|\overline{E}>$ and $h'|\overline{E}>$ which *apply* recursively each other. Each application, i. e. the computation of the appropriate Dirac bracket generates a series. There is plenty of identical

applications, lazily stored in many exemplars, and the solution consists in defining these objects not as functions, but as lists, whose k-th elements return the appropriate values. Accessorily we can tabulate also the matrix elements of h'. Here is the whole truth, with `A !! k` meaning the same as `A[k]` in other languages:

```
loopN n fn = map fn [n ..]    -- [n, n+1, n+2, ...]
loop = loopN 0
mtrx = loop m where m k = loop (h' brk k)
-- uses the symmetry of the matrix element

-- Tabularized matrix elements of h'
vmatT k j | (j<0) || (k<0) = 0.0
          | otherwise = mtrx !! j !! k

ebar = vmatT 0 0 : (hketT 0)  -- no modification

hketM = loop (\k->sum [vmatT k j *: eketT j|j<-[k-4, k-2 .. k+4]])
hketT k | k<0 = 0.0
        | otherwise = hketM !! k

eketM = zeros : loopN 1 (\k->recip (k : ebar)*(vmatT k 0 : hketT k))
eketT k | k<0 = 0.0
        | otherwise = eketM !! k
```

Now everything works much better, and the generation finally terminates by floating overflows, the series is asymptotic only. Of course, all this computation can be repeated using big floats, or infinite precision rationals (extended by square roots), or any other arithmetic domain we wish. The lazy formulation of the generating algorithm is orthogonal to this choice. The obtained answer may be Padéized, or resummed using some convergence improving tricks, and all this may also be performed lazily in an elegant manner, but these techniques cannot be discussed here.

4 Conclusions

This paper could be much longer, e. g. in [12] we have coded in a few lines the infinite Wentzel-Kramers-Brillouin (WKB) expansion.

We propose to use the functional framework to define "implementable abstractions" making it possible to *manipulate procedures* (e. g. to combine thunks) and not only symbolic data. Such tools in various forms exist in computer algebra systems. In Maple any expression can be "unevaluated", and transformed into a function of one of its indeterminates. But the functional layer of Maple is too weak, and the confusion between the *programming variable* and *symbolic indeterminate* might be harmful, despite its superficial usefulness.

Unfortunately, even if the computer algebra package is aware of the mathematical properties of manipulated objects, it will not help us to organize the resulting code, nor suggest how to implement abstractions, unless something has been prepared by the package creators. The path from a textbook describing the general idea, to a debugged program may be very long. The symbolic packages might also generate some abominable code which should never be shown to students, because they laugh loudly at it, as noted by the authors of [13].

Our examples show how the lazy semantics simplifies the coding, we suggest thus that a non-strict functional programming be more often used in computational physics, at least for pedagogical purposes. We don't claim that lazy functional techniques should replace highly optimized imperative codes in contexts where the efficiency is primordial, but even then they might be selectively used to check the correctness of the final programs: lazy code is so short that it is difficult to introduce obscure bugs! We are convinced that lazy languages provide an adequate computing tool for a theoretical physicist who prefers elegance over a brutal force. The results are encouraging, and some of the algorithms seem sufficiently crazy to be interesting.

References

[1] S. Brandt, H.-D. Dahmen, *Quantum Mechanics on the Personal Computer*, Springer, (1994).

[2] Peter Gorik, Peter Carr, *Quantum Science Across Disciplines*, University of Boston Internet Site, `http://qsad.bu.edu/main.html`

[3] John Peterson *et al.*, *Haskell 1.4 report*, Yale University, available together with the software and all relevant references at `http://haskell.org`.

[4] Rinus Plasmeijer, Marko van Eekelen, *Concurrent Clean – Language Report*, HiLT B. V. and University of Nijmegen, (1998). Software and documentation available from `http://www.cs.kun.nl/~clean/`.

[5] Donald E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).

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

[7] Jerzy Karczmarczuk, *Functional Programming and Mathematical Objects*, Functional Programming Languages in Education, FPLE'95, Lecture Notes in Computer Science, vol. **1022**, Springer, (1995), pp. 121–137.

[8] Martin Bert, Christian Bischof, George Corliss, Andreas Griewank, eds., *Computational Differentiation: Techniques, Applications and Tools*, Second International Workshop on Computational Differentiation, (1996), Proceedings in Applied Mathematics 89.

[9] 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`

[10] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Proceedings of the III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203.

[11] M. Born, W. Heisenberg, P. Jordan, Zeitschrift f. Physik **35**, (1925), p. 565.

[12] Jerzy Karczmarczuk, *Lazy Differential Algebra and its Applications*, Workshop, III International Summer School on Advanced Functional Programming, Braga, Portugal, 12–18 September, 1998.

[13] R. M. Corless, D. J. Jeffrey, M. B. Monagan, Pratibha, *Two Perturbation Calculations in Fluid Mechanics using Large-expression Management*, J. Symbolic Computation **23**, (1997), pp. 427-443.