Corecursive Processing of Scientific Data

(The art of borrowing from the future)

Jerzy Karczmarczuk

Université de Caen, Basse Normandie (France)

DSL, Bordeaux, September 8, 2011

Laziness, and in particular the construction of "infinite" lists, is too often taught as a nice curiosity (or a formal issue). We want to use it as a ***serious coding methodology***.

Laziness is the primary source of the progress of Humanity

(And you know very well that I am not joking!)

Introduction

By "scientific" we understand applied mathematics (algebra, analysis, geometry…, as practised by physicists, engineers, etc. Their programmes contain mainly iterations (often quite regular): successive approximations, solutions of differential equations (trajectories or signals generated by algorithms), construction of series (or Padé) coefficients, etc. Programmes are dominated by the handling of loops. Let's replace them by the use of ***lazy sequences*** (in Haskell).

Example (para-scientific…): A colony of rabbits evolves according to "rewriting" rules: in one unit of time a young rabbit pair `[0]` becomes a pair of adults `[1]`, and adults give birth to a couple of youngs `[1, 0]`. Since rabbits are immortal (facts known to Australians), and continue to procreate, this sequence grows indefinitely.

Exercise. Write a program which generates the population after an infinite time. Beginning with `[0]`, `[1]`, we get the following generations: `[1, 0]`, `[1, 0, 1]`, `[1, 0, 1, 1, 0]`, `[1, 0, 1, 1, 0, 1, 0, 1]`, etc. For more fun: it should be a one-liner…

The "last one" is: `[1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, ...]`
This was a real exercise during my FP (Haskell) course. My students knew the basics of corecursive constructions, the "infinite lists", such as `[0 ..] = 0, 1, 2,...`:

```
ints = intsFrom 0 where intsFrom n = n : intsFrom (n+1)
```

or: `ints = 0 : map (1+) ints`. Or:
`ints = fix ((0:) . zipWith (+) (repeat 1))` for deviants.

I tried to convey the *mental pattern* I will advocate here: **Avoid thinking in terms of individual elements of a sequence, try to consider it as a whole, and map the generators/transformers through it. Don't hesitate to lift your mathematical operations from elements to sequences.** Then, you can also write

```
ints = 0 : ones <+> ints where ones=1 : ones
```

and `(<+>) = zipWith (+)` is a lifted, element-wise addition of lists.

Such patterns form – if you wish – a ***language***, which should be specifically learnt. The application of lazy techniques is as difficult as the relational programming in Prolog, with backtracking and unification of unknown variables.

Here we have the *extrapolating recursion*, which is not so exotic: the flood-fill of contours with a colour is a classical example thereof. If the pixel – argument of the filling procedure – is unfilled, then fill it, and recur for all the neighbours. But such programming techniques are rarely taught *as methodologies*, and "infinite lists" in Haskell are presented mainly as curiosities, although everybody speaks about recursive fixpoints... If you wish to use them:

```
ints = fix ((:) 0 . map (1+))
  where fix f = f (fix f)
```

For rabbits we have a rewriting function `rwr x = 1 : [0 | x==1]` which converts 0 or 1 into a list (`[1]` or `[1, 0]`), so its application to a rabbit generation should include the concatenation of internal lists: `nxt gen = concat (map rwr gen)` which with monads may be simplified into `nxt gen = rwr =<< gen`.
We get *all* the generations by `rabbits = [0] : map nxt rabbits`

Also, in analogy with integers, we can write

```
rabbits = [0] : r1 where
  r1 = [1] : zipWith (++) r1 rabbits
```

getting thus all the Fibonacci words. *But this is not the solution of our problem.* We want "just" the "last one", the $\aleph_0$-th element of this family! And here my students gave up. Through some nice and pedagogical Socratic questions (I am happy I am still alive...) we concluded that the fixpoint of the evolution, the Ultimate Rabbit Sequence fulfills the equation `rs = nxt rs`, and we were stuck, the Holy Bottom hit us. The lesson was: **Yes, the fixpoint corecursive equation may work, but – obviously – you should provide a finite, known prefix (the seed) of your sequence**. Here the solution is simple, even if a bit ugly:

```
rs = 1:rq where _:rq = nxt rs
```

In general the solution is not so simple, and requires some experience and/or mental conditioning (my friends call it a perversion, but I believe that it is just imagination...).

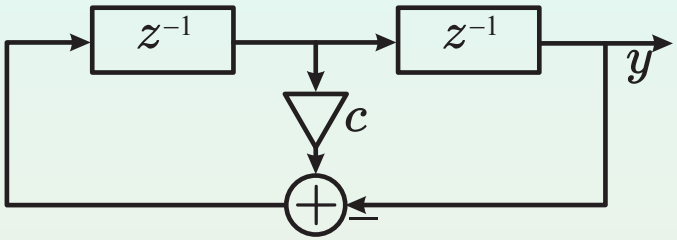
You are aware that there are thousands of articles and Web pages devoted to the corecursion. The theory is quite old. (And Jeremy Gibbons with Graham Hutton show how to prove programs based on corecursion...) But the examples *everywhere* are "standard", often a bit boring, and even less useful than the Rabbit Sequence, for the "outer world".

We shall see now some other corecursive data, which might be really useful for the applied mathematics. We begin with something more direct than the infinite power series, etc.: digital acoustic signals, and the simulation of musical instruments, which really works.

A signal will be just a sequence of (appropriately normalized) numbers representing the sound amplitude.

Signals and music

Let's generate a sinusoid. Mapping the sine over $[x_0, x_0 + h, x_0 + 2h, \ldots]$ is not natural, real generators do not "count the time".



A more natural algorithm is the recurrence, whose solution is a sinusoid, e.g., the equation below, which corresponds to the data flow diagram at the left.

$$\sin(nh) = 2\cos(h) \cdot \sin((n-1)h) - \sin((n-2)h)$$

The corecursive formulation of the corresponding sequence is

```
y = sin h : (2*cos h)*>y <-> (0:y)
```

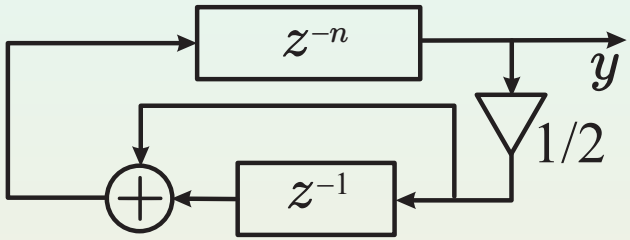
where `x *> l = map (x*) l` multiplies sequences by scalars. The block z^{-1} is a one step delay (one-element prefix). Two starting elements, 0 and $\sin(h)$, must be provided.

Exercise. A sinusoid is *also* the solution of the differential equation $y'' = -y$, and the modified Euler algorithm reads: $y_{n+1} = y_n + h \cdot y'_n$; $y'_{n+1} = y'_n - h \cdot y_{n+1}$. Write a one-line ‣ corecurrent code for it.

Karplus-Strong string model

The additive technique of sound synthesis demands some dozens of oscillators and very complicated filters. But a realistic sound of a plucked string can be obtained from a discretized string equation containing a longer delay line. Here this is just a list containing an initial excitation – some random noise.

The signal moves along the delay line, and the filter attenuates it, high frequencies first; spectral components of period comparable to the medium length – last. The code is

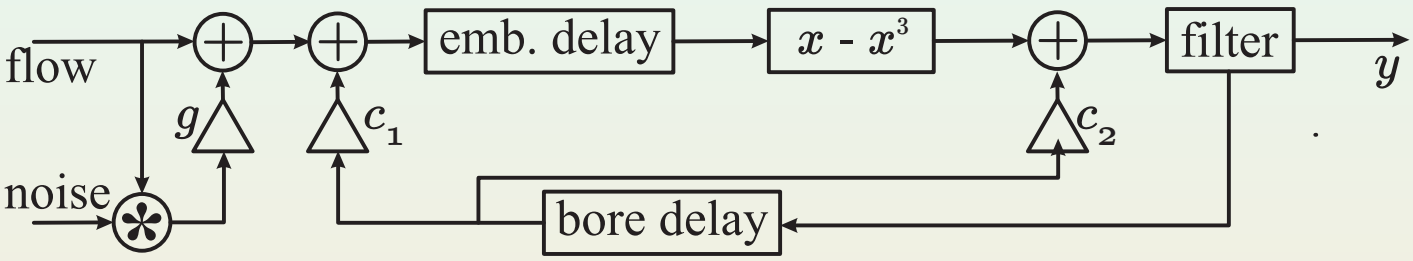



```
y = prefx ++ 0.5*>(y <+> (0:y))
  where prefx = take n randomStream
```

where n is chosen in function of the frequency and of the sampling ratio. The sound is not bad, but replacing the arithmetic mean by better filters (also in the corecursive style) we may get this or even this, when two such strings are coupled together, in order to resonate. (Stop!)

These examples have been written in Clean, a lazy functional language similar to Haskell, with some low-level advantages. The idea was not only to write the shortest programmes in the world which simulated realistic musical instruments, but to convince some people that *practical teaching* of this may be fun, because of the **modularity and composability** of our blocks.

Even such complicated instruments as violin, or flute (as below)



may be coded in less than 10 lines, some just for a reasonable spectral filter... And the sound is acceptable. I cheated, the true instrument has more than 10 lines, since I included the *vibrato* transducer (of course "the shortest program in the world" (about 6 lines) which lazily resamples a signal according to a nonlinear, oscillating pattern).

Exercise. Transform into a stream the general IIR filter:

$$y_n = \sum_{k=0}^{m} b_k x_{n-k} + \sum_{k=1}^{p} a_k y_{n-k} \,.$$

Here is the (trivial) answer for the simplest, one pole case:

```
f1pole b0 a1 x = y where
  y = delay (b0*>x - a1*>y)
```

I could show you much more, e.g. the corecursively coded paradoxical Shepard-Risset sound (exploited in very bad Italian Science-Fiction movies), the reverberation combinators, etc., but we have to pass to some other applications. (Stop)

Back to math...

Power series

Several approximation schemes in applied math need power series. We often seek just the set of coefficients $u_0, u_1, u_2, \ldots$ of $U(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_n x^n + \cdots$, but since the algebra of series expressed in terms of coefficients looks horrible, the computer algebra packages are notoriously abused, and the manipulation of the symbolic "x" makes the algorithms quite heavy.

The corecursion based arithmetic is based on the decomposition $U = u_0 + x \cdot \overline{U}$ which corresponds naturally to the splitting of a sequence into its head and tail: `u = u0 :* uq` (instead of standards lists we use another algebraic datatype in order to avoid confusion). The addition is element-wise (zip-lifted), and the multiplication follows the pattern

$$U \cdot V = (u_0 + x \cdot \overline{U}) \cdot (v_0 + x \cdot \overline{V}) = u_0 \cdot v_0 + x \cdot (u_0 \overline{V} + \overline{U} \cdot V)\,,$$

or:

```
(u0 :* uq)*v@(v0 :* vq) = u0*v0 :* u0*>vq+uq*v
```

(No special names like `<+>` etc.)

The division is also easy. If $W = U \cdot V$, then $U = W \cdot U$, and we rephrase the above formula:

$$w_0 = u_0/v_0; \qquad \overline{W} = (\overline{U} - w_0 \cdot \overline{V})/V\,.$$

```
(u0 :* uq)/v@(v0 :* vq) = w0:*wq where
  {w0=u0/v0; wq=(uq - w0*>vq)/v}
```

This corecursive definition is much shorter than the classical solution with indices presented e.g., by Knuth...

We shall need also the differentiation and integration of series:

`sdiff u` : $\quad U \to U' = u_1 + 2u_2 \cdot x + 3u_3 \cdot x^2 + \cdots$

`sint u c` : $\quad U \to \int U = c + u_0 \cdot x + \frac{1}{2}u_1 \cdot x^2 + \frac{1}{3}u_2 \cdot x^3 + \cdots$

whose codes are trivial (zip-products/quotients with $[1, 2, 3, 4, \ldots]$).

Now, if we want to compute $W = \exp(U)$, it suffices to observe that $W' = U' \cdot W$, so

$$W = \int U' \cdot W + \exp(u_0)\,,$$

where the last term is the integration constant. The code is:

```
exp u@(u0:*_) = w where w = sint (exp u0) (w*sdiff u)
```

Other functions follow similar patterns, e.g.,

$$W = \sqrt{U} \quad \to \quad W' = \frac{1}{2\sqrt{U}} \cdot U', \quad \text{or} \quad W = \int U'/(2 \cdot W) + \sqrt{u_0}\,,$$

etc. Other functions may be developed in such a way, but this is not a blind automaton. If we want the series for the solution of the modified Bessel equation: $x^2 w'' + w' + \frac{1}{4} w = 0$, the way is to integrate "perversely"

$$w = -\int (x^2 w'' + \frac{1}{4} w) + w_0\,.$$

The application of the lazy corecursion may be a serious *algorithmization* problem, not just the coding. Suppose we want to compute the number of partitions of an integer:
$5 = 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1,$
altogether: 7 partitions. This is the 5-th coefficient of the series – partition function

$$Z(x) = \prod_{n=1}^{\infty} \frac{1}{1-x^n}\,, \quad \text{or:}$$

$$Z(x) = Z_1(x) \qquad \text{where} \qquad Z_m(x) = \frac{1}{1-x^m} Z_{m+1}(x)\,.$$

This is a "crazy" runaway corecursion, useless for, say, Fortran programmers. But, if we define $B_m(x)$ such that $Z_m = 1 + x^m B_m$, this auxiliary function fulfills

$$B_m(x) = 1 + x\left(B_{m+1}(x) + x^{m-1} B_m(x)\right)\,,$$

which can be directly coded:

```
partgen = 1 :* b 1 where
  b m = p where p = 1 :* b (m+1) + (0:*1)^(m-1)*p
```

We get: `[1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77,...]` in no time. (This has some usage in the theory of symmetry in quantum physics).

We can functionally invert series (find: $t = W(x)$ such that $x = U(t) = t + u_2 t^2 + u_3 t^3 + \cdots$; *extremely* important in physics and astronomy), ...
compute their Schröder functions ($\Psi(x)$ such that $\Psi(U(x)) = s \cdot \Psi(x)$),
solve recurrences involving derivatives, construct and evaluate rational (Padé) approximations, etc.

We have been able to express – relatively easily – some (known and useful) asymptotic expansions arising from the perturbational calculus, which are notoriously difficult to derive (and dangerous for the mental health of students forced to do it (and then, for the physical health of the teacher)...). We shall come into it later.

Differential Algebras

One – a little similar to series, but with a different algebra – sort of sequence has been inspired by the technology called "automatic differentiation" (AD). Its theoretical basis relies on the fact that ***computing derivatives is a purely algebraic process***, no analysis, no "limits" involved. The differentiation, or the *derivation* is an operator $e \to e'$ which is linear, and fulfils the Leibniz identity: $(e \cdot f)' = e' \cdot f + e \cdot f'$.

The expression e may be "anything", and their algebra may be different from what you think. Lie/Poisson brackets in mechanics are derivations. Commutators in quantum algebra are derivations...
We shall think of e as of (almost) "normal" numerical expression in a rather ordinary program, which contain standard arithmetic operations. It contains some manifest constants, and *one* specific object called the "variable" (whose name is irrelevant).

We can lift the domain of "standard" expressions to sequences $(e_0, e_1, e_2, \ldots)$, where e_0 is the "normal" value, as computed in a typical way, and $e_1 = e', e_2 = e''$, etc., wrt. the above mentioned variable. This variable is just the sequence $(0, 1, 0, 0, \ldots)$, and all the constants have the form $(c, 0, 0, \ldots)$.
All programs start with constants, and one (here just one) or more "variables". The only thing to do is to reconstruct the arithmetic of general sequences belonging to this domain. They will be coded as algebraic data `e0 :> eq`, where `eq` is the tail of the sequence. Of course, the addition/subtraction will be element-wise, the derivation is linear. The multiplication becomes

```
e@(e0:>eq)*f@(f0:>fq) = e0*f0 :> (eq*f + e*fq)
```

and the division:

```
recip (e0:>eq) = ip where ip = recip e0 :> (-eq*ip*ip)
```

but beware, ***beware!***. The possibility to compute 1000 derivatives of a complicated expression is nice, but without simplification you will be murdered by the exponential complexity of the Leibniz rule!

The expression `(sin x)*exp(-x)` with `x` being the differentiation variable will explode rather fast (don't try 50 derivatives...) But:

```
exsn x@(x0:>_) = p where {ex=exp(-x0);
  p = sin x0*ex :> q - p;
  q = cos x0*ex :> (-p - q)}
```

works very well. No, ***lazy AD is not a free / cheap lunch!***

Please notice: `... eq*f ...` , not `eq*f0` in the Leibniz rule (this was one of the typical students' error... But the AD simple algorithms which compute just the first derivative, and are (can be) coded strictly, would use *that*).

How to compute $\exp(x)$? Since $\exp(e)' = e' \cdot \exp(e)$, the construction of elementary functions is straightforward:

```
exp es@(e:>eq) = w where w = exp e :> eq*w
log es@(e:>eq) = log e :> (eq/es)
sqrt es@(e:>eq) = w where w = sqrt e :> ((1/2) *> (eq/w))
```

etc.

We dispose of an implementable, non-trivial algebra, where the derivation is "just" another operator acting on generalized numbers, without any symbolic manipulations.

The applications are infinite: dynamical control of the stability of the differential equations solvers, general recurrent definitions involving derivatives (some special functions are like that), multidimensional extensions (cumbersome) involving the de l'Hôpital rule, etc.

Let's compute the Maclaurin series for the Lambert function $W(z)$, defined implicitly:

$$W(z) \cdot e^{W(z)} = z \,.$$

It is useful in combinatorics, and elsewhere: solutions of several differential equations may be expressed through it. The differentiation of the formula gives

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

Its inversion:

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

may be directly coded, taking into account that $W(0) = 0$:

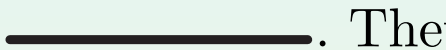
```
w = 0.0 :> (exp (-w)/(1.0+w))
```

This gives the asymptotic (divergent) series 0.0, 1.0, 2.0, 9.0, 64.0, $\ldots$, $(-n)^{n-1}$, $\ldots$, which is of course known, but the idea may be easily used in other expressions. (BTW., the fact that it diverges does not preclude its usefulness; We often need just a few terms, and there exist many convergence improvements. Some of them can be nicely coded using corecursion...)

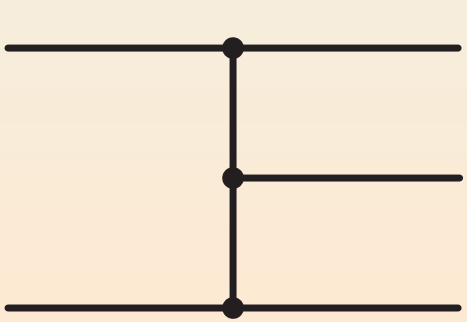
Feynman diagrams

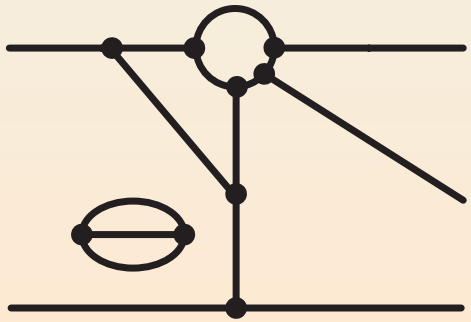
(In a 0-dimensional φ^3 field theory)

Conceptually the Quantum Field Theory of interactions is the simplest theory in the world. ***Everything happens!***

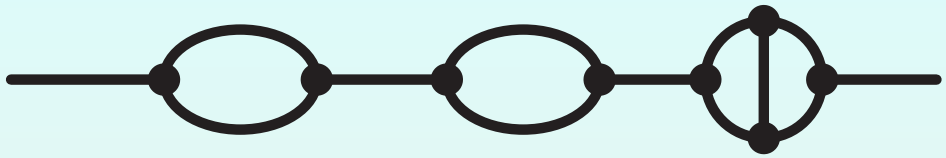
We begin with the *elementary* interaction: the emission/absorption of a particle by another one:  , and the "movement", or propagation:  . They are functions of space coordinates or momenta, charges, spins, etc. In a toy theory they will have numerical (or symbolic) values. Finally, we construct the *probability amplitude* of a process /transition. We call also this amplitude: *Green function*, G_n.

For a given initial state, e.g., two colliding particles, and some final state, say, three particles (two scattered and one new produced), we have to construct **all** the possible diagrams with the appropriate external structure, and ***anything*** inside in order to construct G_5.





Even a simple movement of the particle may create some “bubbles”

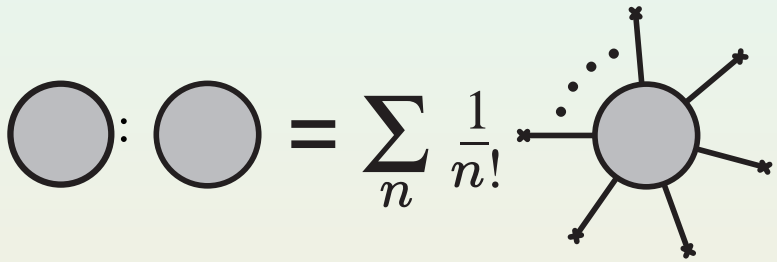


and this makes a difference between a “bare” propagator, and the “dressed” one, which depends on the interaction (coupling) constant attributed to the vertex  .

Exercise. (Admittedly, difficult, difficult also for PhD students in Physics...) Suppose that each vertex contributes one factor proportional to the coupling constant γ. Assume the value 1 for the “bare” propagator. Compute the “dressed” propagator G_2^c, which contains all possible **connected** bubbles:  as a power series in γ. (So, the diagram above will contribute γ^8). But if you draw all the diagrams for a given order, just counting them is not enough, there is a catch... The fact that quantum particles are ***identical***, truly indistinguishable, means that for a diagram which admits N labellings of lines giving the same topology we have to introduce the factor $1/N!$.

Don't shoot yourself yet...
Computing G_2 is too complicated, so instead we will compute ***all*** G_n together through the generating functional

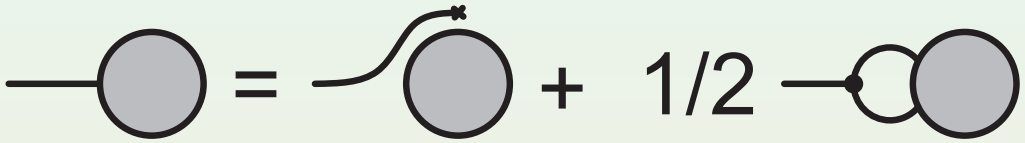
$$Z(J) = \sum_{n=0}^{\infty} \frac{1}{n!} G_n J^n \,.$$

This $Z(J)$ can be depicted as 

where each cross denotes one occurrence of the auxiliary "current" J, and the n-legged beastie is G_n. And inside the gray bubble *anything* happens. By differentiating Z you get back the Green functions. For example $dZ(J)/dJ =$ 

And then you pronounce a magic truth: ***The World is Recursive!***

(Actually, you know already the Real, True Truth: it is ***corecursive***...
There is no final reduction to the "base case", the inside-extrapolating, down-the-rabbit-hole "bubbling" goes *ad infinitum*.)
And miracle happens, all the theory *within the perturbation framework* may be expressed by the so called Dyson-Schwinger equation:



If you detach one particle from all that magma, either it interacts or not. If not, the propagator must end on a current. If yes, then it ends up on a vertex, and then ...
And then anything may happen.
Exercise. Write down the analytical form of the D-S equation.

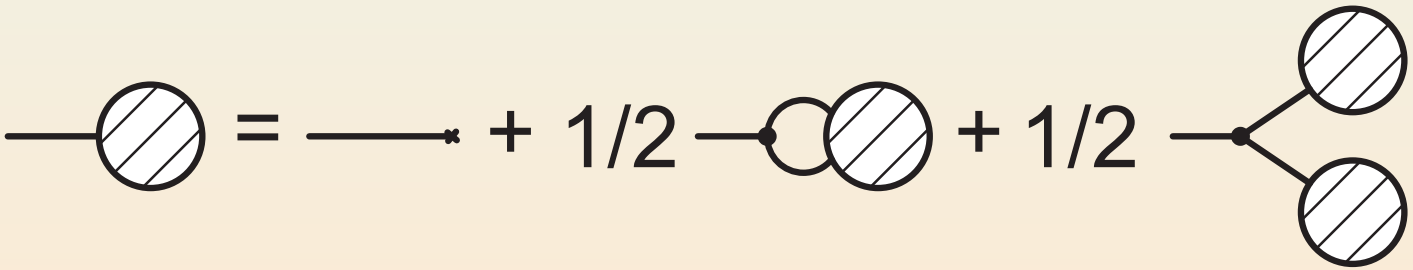
There is nothing to be afraid:

$$\frac{d}{dJ}Z(J]) = J \cdot Z[J] + \frac{1}{2}\gamma\frac{d^2}{dJ^2}Z(J)\,.$$

If you have learnt the theory of graphs, you might remember that the generating functional for a complete family of graphs is the exponential of the generating functional of the *connected* beasties. This means that if we introduce $W = \log Z$, we have:

$$\frac{d}{dJ}W = J + \frac{1}{2}\gamma\left(\frac{d^2}{dJ^2}W + \left(\frac{dW}{dJ}\right)^2\right)\,,$$

or:



Exercise. Write a program which solves the DS equation and computes $\varphi = dW/dJ$. Let's repeat the equation:

$$\varphi = J + \frac{1}{2}\gamma\left(\phi' + \varphi^2\right)$$

φ is a double series, in J and in γ. It is easier to begin with φ as a series in γ, whose coefficients are series in J. The first element of the result, $\varphi_0(J) = J$ is the identity. The differentiation φ' propagates to the coefficients, and is coded as `fmap sdiff`. Here is the solution:

```
phi :: Series (Series Rational)
phi = j :* (1/2)*>(fmap sdif phi + phi*phi)
  where j = 0:*1
```

Exercice. Find G_2^c by taking the second term of each element of φ.

```
g2 = fmap (sHead . sTail) phi
```

The result is:

$$G_2 = 1 + \gamma^2 + \frac{25}{8}\gamma^4 + 15\gamma^6 + \frac{12155}{128}\gamma^8 + \frac{11865}{16}\gamma^{10} + \frac{7040125}{1024}\gamma^{12} + \cdots$$

Conclusions

Lazy sequences are dynamical processes disguised in data ; one can transport them, and launch them by looking at their components. Such programming demands a different vision of what a piece of data is, and thinking that people will read the manual of Haskell, and start programming using corecursive algorithms, is too optimistic. Some people dislike them. I believe that people who don't like laziness simply never needed it. I did...

Teaching corecursion is difficult... There are some patterns to learn, e.g., instead of `iterate f x = x : iterate f (f x)` we *rather* say: `iterate f x = w where w = x : map f w`,
or: `fix ((x :) . (map f))` (is it a reforestation?...). We have seen the trick which replaced
`w = f w` by `w = (1:q) where _:q = f w`.

The "circular" programs of Richard Bird, and the usage of folding / unfolding transformations should be studied, but **this takes time!** And, convincing anybody that something *may be* useful, does not belong to science, but to politics (or to a religious activity).

The corecursion is a Real World pattern. A child learning to walk applies corecursion. Bank loans are quite often corecursive, and the financing of the social security in France is *really* corecursive. (And don't ask me how all that will terminate...) Some cosmologists speculate that our quantum Universe is a corecursive bubble, a quantum fluctuation of Nothing (not even the vacuum), which "lives on credit", virtually, and one day the Creditor may wake-up, and then ...

So, learn to *use* corecursion, you will be better prepared when the Day comes.

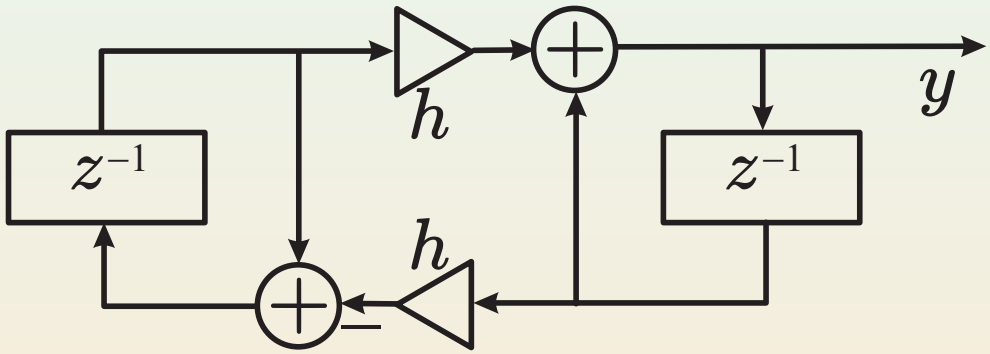
Solutions

Corecursive harmonic oscillator

Here it is, the one-liner:

```
y=0:w where {w=y+h*>u ; u=1:u-h*>w}
```

Well... a nice fellow would write it in TWO lines. It corresponds to the diagram



Where on the diagram are `u` and `w`? Identify the prefixes (the initial contents of the delay blocks, 0 and 1).

◂ (go back)