ELSEVIER

Theoretical Computer Science 187 (1997) 203–219

Theoretical Computer Science

Generating power of lazy semantics

Jerzy Karczmarczuk

Department of Computer Science, University of Caen, France

Abstract

We discuss the use of the lazy evaluation scheme as coding tool in some algebraic manipulations. We show – on several examples – how to process the infinite power series or other open-ended data structures with co-recurrent algorithms, which simplify enormously the coding of recurrence relations or solving equations in the power series domain. The important point is not the "infinite" length of the data, but the fact that the algorithms use open recursion, and the user never thinks about the truncation.

Keywords: Lazy; Functional; Haskell; Series; Feynman diagrams

1. Introduction

This article develops some applications of the functional lazy evaluation schemes to symbolic calculus. Neither the idea of *non-strict* semantics, nor its application to generate infinite, open structures such as power series, are new, see, for example [1, 6], and some books on functional programming ([3, 5]), etc. The *lazy evaluation* (or *call by need* is a protocol which delays the evaluation of the arguments of a function: while evaluating $f(x)$ the code for f is entered, but if f does not need x, nothing wrong happens, even if we demanded to calculate $f(1/a)$, where $a=0$. The code for $1/a$ is compiled to a *thunk* or a *promise*, but perhaps never executed. The function f receives a promise to deliver $1/a$ when needed. The thunk is evaluated when the code of f uses it.

The domain of lazy evaluation is very well known, it constitutes one of the bases of the modern functional programming, and a priori, it has nothing to do with algebraic manipulation, although it is obviously used therein [6, 13]. However, the superficial analogy between an algebraic formula with some symbolic indeterminates, and a function body waiting to be evaluated, is quite explicit.

* E-mail: karczma@info.unicaen.fr.

PII S0304-3975(97)00065-0

Perhaps, paradoxically, this is one of the reasons why almost the totality of the computer algebra code – both user and implementor packages – is strict, the lazy objects are usually encapsulated in specific domains treated by specialized algorithms, such as the series packages in Maple ([13]) or Axiom ([6]). The possibility to operate upon *symbolic formulae* apparently makes it less fashionable for the computer algebra community to manipulate the *computations* such as thunks, higher-order functions, etc.

In this paper we present a partial and heterogeneous, but coherent approach to the *lazy evaluation as a coding tool*, restricted to some typical problems in symbolic computations. In general, the subject is enormous: the lazy semantics is very intensively used elsewhere, e.g. in the functional I/O, parsing, all kind of monadic approach to the computation semantics, nondeterminism, etc. To present the examples we shall not use any computer algebra system, but we will show some examples in the style of a lazy polymorphic programming language Gofer [15], a dialect of Haskell [14]. Our aim is not to suggest that something can be done, but *how*. We will omit the discussion of the polymorphic overloading of standard arithmetic operators permitting to write $u \cdot v$ where u and v are lists, see, for example, [16]. The examples in the text have been edited in order to simplify the notation (some conversions required by the Haskell typechecker have been omitted), and the layout has been embellished, but they are working programs.

It seems important to clarify and to underline that the main idea behind the discussed application of the lazy evaluation is *not necessarily* the possibility to handle *infinite* structures, but the following:

- The possibility to code *effectively* the fixed point definitions: $\alpha = g(\alpha)$, where α is just a data structure, and not a recursive function (see [2]). The infinite list of 1 might be coded as

```
ones = 1 : ones
```

  where the colon is the infix "cons" operator – the list constructor. For a useful and not so trivial example see Eq. (1).
- The ability to apply effectively the co-recursion, or the *extrapolating* recursion. While "standard" recursion descends on, and *analyses* the data, the co-recursion *creates* the data.

The proof techniques of some co-recurrent identities are a little unorthodox [10, 24], as the standard induction might have nothing to induct on. Take for example the definition of a sequence of iterates: $[x, f(x), f(f(x)), f(f(f(x))), \ldots]$, (very useful for the lazy approach to the iterative equation solvers, see for example, the first program of the Section 3), and the definition of the `map` functional, which applies a function to all the elements of a list:

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

map f (a:aq) = f a : map f aq
```

We can prove that `iterate f (f x) = map f (iterate f x)` in the following way:

```
iterate f (f x)
   = f x : iterate f (f (f x))
   = f x : map f (iterate f (f x))   —  Ex hypothesi!
   = map f (x : iterate f (f x)) = map f (iterate f x)
```

Note the right-to-left reduction. Such "bootstrap" is esssential in the co-inductive proofs, and we shall see that it has an enormous generative power as well. See, Section 3 for a non-trivial usage of `iterate`. In principle, it is not necessary to have the unlimited data definitions, without terminating clauses. We do not even need such academic examples as above: a typical co-recursion case, known to almost all readers, and not demanding any kind of lazy evaluation, is the construction of a transitive closure, for example, the flood-filling algorithm in graphics. In order to paint a region starting from the pixel (x, y) either do nothing if the pixel is already painted, or paint it, and do the same to all the neighbours. It is obviously an extrapolating recursion scheme, which is *guaranteed to progress*, but terminates eventually only because the universe is finite.

Of course, with lazy streams it is easy to create potentially infinite data structures such as series, continuous fractions, etc. not necessarily in the context of computer algebra [20], or to construct the non-deterministic algorithms, but there are more universal arguments for the lazy functional programming: thanks to the deferred evaluation and higher-order functions, it is easier to formulate some quite orthodox algorithms in a static, declarative manner, without polluting them with countless **for**/**while** loops and other imperative constructs, which hide sometimes the clarity of the underlying strategy.

One serious warning seems appropriate here: while standard recursive schemes consume (reversibly) the system stack while storing the contexts of the recursive calls, the lazy constructions, such as `iterate`, or `ones` fill-up the dynamic heap of the system with anonymous functional closures created ad hoc. This is time consuming and requires a very good memory management, adapted to laziness. The lazy adds on to a strict language, such as the macros `delay` or `cons-stream` in Scheme are not very efficient [1].

2. Power series generation and manipulation

In our approach, a univariate power series $U(x) = u_0 + u_1 x + u_2 x^2 + u_3 x^3 \ldots$ will be represented by the lazy list $[u_0, u_1, \ldots]$. The series coefficients may, in principle, belong to any algebraic domain. An *effective and simple* coding of an arbitrary algorithm dealing with such series is not entirely trivial. The algorithms are usually dominated by the administration of the truncation trivia. In fact, if one implements the algorithms discussed in [17] or [27] using indexed vectors, one sees mainly summing loops and the evaluation of the bounds of these loops, which becomes quite boring. Here the

addition and the subtraction term by term is given by the "zip" functional. From now on, we change the layout of our programs, to suggest visually their mathematical flavour

$u + v = \text{zip With } (+)\ u\ v,$

where

$\text{zip With op } (a : \bar{a})\ (b : \bar{b}) = \text{op}\, a\ b : \text{zip With op}\, \bar{a}\, \bar{b}$

But the multiplication and the division are equally short

$(u_0 : \bar{u}) \cdot v@(v_0 : \bar{v}) = (u_0 \cdot v_0) : (u_0 \cdot \bar{v} + \bar{u} \cdot v)$

$(u_0 : \bar{u})/v@(v_0 : \bar{v}) = (w_0 : \bar{w})$ **where**
$w_0 = u_0/v_0$
$\bar{w} = (\bar{u} - w_0 \cdot \bar{v})/v$

(where the construct $u@A$ is a way to inform the compiler that the parameter is called u and has the structure A.)

The differentiation and integration are obvious

$\text{integ}\, c\ u = c : \text{zip With } (/)\ u\ [1..]$
$\text{diff}(u_0 : \bar{u}) = \text{zip With } (\times)\ \bar{u}\ [1..],$

where $[1..]$ denotes the infinite sequence $1, 2, 3, 4, \ldots$ The integration is a *lazy operation*, permitting the construction of self-referring objects. It takes some time to master this technique and to appreciate the fact that the definition: $W = \text{Const} + \int f(W)$ is not just a specification, or an equation, but an *algorithm*. It suffices to know the constant term in order to be able to generate the next one and the whole series. The definition above is equivalent to the obvious identity for any series $f : f_n = f'_{n-1}/n$.

The integration gives thus the *direct* solution to the classical trick which constructs the transcendental functions on the series domain as the solutions of simple differential equations, see [17, 27] or [18]. For example, if $w = \exp(u)$, then $w' = u' \exp(u) = u'w$, and

$$w = \int w \cdot u' \, \mathrm{d}x. \tag{1}$$

We code thus, knowing that the integration constant is equal to e^{u_0}

$\text{serExp}\, u@(u_0 : \bar{u}) = w$ **where**
$w = \text{integ}(\exp u_0)\ (w \cdot \text{diff}\, u).$

In the same way, we construct a (real) power. If $w = u^\alpha$, then $w' \cdot u = \alpha u' \cdot u^\alpha$, or $w = u_0^\alpha + \alpha \int w \cdot u'/u$ which can be coded again in two lines. And the logarithm is: $\log u = w$, where $w = \log u_0 + \int u'/u$, which is not even self-referring.

Sometimes one has to be careful. If we take the reduced Bessel equation

$$u'' + \frac{2v+1}{x} u' + u = 0, \tag{2}$$

we see that the first two terms have the same expansion order, and the lazy integration is cumbersome. But, knowing the parity properties of the Bessel function, we introduce an auxiliary function $w : w(x^2) = u(x)$, and we integrate

$$w = 1 - \frac{1}{\nu + 1} \int \left(\tfrac{1}{4} w + x^2 w''\right), \tag{3}$$

where the reader shall note the perversion: one does not integrate w'', but w' in order to obtain w, whose second derivative is reinjected into the formula. This derivative is "protected" by the integration and the multiplication by x^2, which together add three known items in front of it. Lazy techniques might be quite laborious, and one-line procedures do not come for free... The lazy approach does no miracles, it just replaces the iterative coding of the equivalent recurrence relations. But we had to massage a little our program, exactly as somebody would manipulate a symbolic formula.

Another nice application of the co-recurrent schemes is the reversion of power series. The reverse of a given series is the solution of the following problem: Given

$$z = t + V_2 t^2 + V_3 t^3 + \cdots, \quad \text{find } t = z + W_2 z^2 + W_3 z^3 + \cdots. \tag{4}$$

Among several possible approaches to this challenge, one consists in reducing it to a composition of series. This is readily done if we note that an auxiliary series p defined by $t = z(1 - zp)$ fulfils the identity:

$$p = (1 - zp)^2 (V_2 + V_3 z(1 - zp) + V_4 z^2 (1 - zp)^2 + \cdots), \tag{5}$$

and the task is recursively solvable. The composition is very simple. We want to find $W(x) = U(V(x))$, where the series V is free from the 0th term, otherwise a full numerical series would have to be summed. The solution is nothing more than the ordinary, but infinite Horner scheme:

$$U(V) = U_0 + x(V_1 + V_2 x + \cdots) \cdot (U_1 + x(V_1 + V_2 x + \cdots) \cdot (U2 + x(\cdots))), \tag{6}$$

or, horribly enough

scomp u $(0 : \bar{v}) =$ cmv u **where**
  cmv$(u_0 : \bar{u}) = u_0 : \bar{v} \cdot$ (cmv $\bar{u}$)

and for the reverse we get

serrev$(0 : 1 : v) = t$ **where**
  $t = 0 : m$
  $m = 1 : (-m^2) \cdot$ scomp v t

Other approaches are also practical. One might code in three lines the Lagrange reversion algorithm (see [17]), or use the Newton method to solve iteratively the equation $f(t) = t + v_2 t^2 + \cdots - z = 0$, and obtain t as a function of z (see [4]). But in this case one should first read the next section.

3. Iterative approximation which pretends to be exact

If a series satisfies a more complicated, non-linear equation $f(U)=0$, the lazy approach may influence also the construction of the Newton algorithm. The idea of using Newton algorithm in the series domain is not new, see [4, 19]. Again, instead of coding a loop broken by some convergence criteria, we construct shamelessly an infinite list of infinite iterates. For example, if $W=\sqrt{U}$, then we get $[W^{(0)}, W^{(1)}, \ldots, W^{(n)}\ldots]$, where $W^{(n+1)}=\frac{1}{2}(W^{(n)}+U/W^{(n)})$. The construction of this stream is quite simple, the standard prelude function `iterate` does the job:

$$\text{sqrtS } y@(y_0 : \_) = \text{iterate}\,(\lambda x \to (1/2)\cdot(x + y/x))\ (\text{sqrt}\, y_0).$$

We should note that the starting value in this formula is not a number, the constant $\sqrt{y_0}$ is promoted into the series: $[\sqrt{y_0}, 0, 0, \ldots]$. But now comes the main point: suppose we need 7 terms of the solution. Knowing the quadratic convergence of the algorithm we take the 3rd iterate, as we know that its 8 terms are correct. If we change our mind and take another 2 terms, we have to generate the next iterate. The lazy dæmons will do all this clumsy administration, and will not permit the users to fall into their bad habits, and claim that the number of terms wanted must be explicitly given.

So, we choose *well* the 0th (initial) iterate, whose constant term must agree with the constant term of the exact solution, otherwise an arbitrary number of iterations would be necessary to construct even this.

The *final* answer is a lazily constructed series which takes 1 term from the 0th iterate, 1 (the second) from the first approximation, 2 (the third and the fourth) from the next one, 4 (from 5 to 8) from the third iterate, then 8, 16, etc. All these segments are (lazily) concatenated, and the end user will see the initial segment of the *exact* solution and will not even think about the approximation order. The correct choice of the starting value is of utmost importance, otherwise the lazy development would propagate the error through all the terms.

Here is the code of the lazy flattening. The function `segc` drops `ndrop` elements from a list and concatenates the following `ntake` items with `rst`:

```
flatn((s0 : _) : v) = s0 : aux 1 v where
  aux nd(u0 : ū) = segc u nd nd (aux m ū)
    where m = 2 · nd
  segc u@(u0 : ū)ndrop ntake rst
    | ndrop > 0 = segc ū (ndrop − 1) ntake rst
    | ntake > 0 = u0 : segc ū ndrop (ntake − 1)rst
    |otherwise = rst
```

The lazy treadmill does not free us from the necessity of analyzing special cases such as the degeneration of series, or non-trivial analyticity properties. Moritsugu et al. [21] discussed the development of the function $p=\text{ptan}(s)$ which is the solution of $p-\tan(p)=s$, and finds its applications in the analysis of the Josephson junction.

We see that this function is not regular, but it is rather a Puiseux series beginning with $p = -(3s)^{1/3} + \cdots$. If we want to use the reversion method, it should be intelligent. Here is the solution in the form of an entire series in $x = (3s)^{1/3}$:

```
ptan =
  let x = 0 : 1                  – Series: U(x) = x
      p = serpow(3 · v) (1/3) where
          (_ : _ : v) = x − tan x
  in serrev(0 : p)
```

The authors of [21] discuss the application of the Newton algorithm for the equation $m(p) = \tan(p) - p - \frac{1}{3}x^3 = 0$, observing that the derivative $m'(p) = \tan^2 p$ has no free term, so additional work is needed. The relatively simplistic techniques presented in our paper should be somehow extended if we want to generate lazily the Laurent expansions, to calculate the residues, etc., but everything can be done. In particular, a suggestive generalization of our lazy series which imposes itself, is the sparse representation, where the items are not just the coefficients, but pairs (*coefficient, exponent*). We found it useful also to include, where possible, a special object (in fact the empty list; the lazy semantics does not preclude the existence of finite objects) to denote 0. In such a way the standard polynomial packages realized in a lazy language might be lifted to the series domain.

The regular solution for $x = \sqrt[3]{3s}$ of the discussed equation is

$$\mathrm{p\,tan}(x) = -x + \frac{2}{15}x^3 - \frac{3}{175}x^5 + \frac{2}{1575}x^7 + \frac{16}{202125}x^9 - \frac{362}{9384375}x^{11} + \frac{49711}{12415528125}x^{13} + \frac{13952}{27918515625}x^{15} - \frac{574406627}{2573221666640625}x^{17}. \quad (7)$$

(The last term in [21] is erroneous, quite probably because of some bad truncation, a mistake which we could not have committed.)

4. Continued fractions and Padé approximants

The power series are not the only "infinite" data structures which can be processed by lazy algorithms, although here the co-recursion is particularly simple. But already in 1972 Gosper [11] (see also [17, 26]) has shown that the arithmetic of continued fractions can be very elegantly realized through incremental stream processing. We could give here a particularly simple realization of such arithmetic package, but for algebraic manipulation it might be more interesting to work with series than with numbers. It is quite simple to construct from a given series an infinite continued fraction. We give here a particular, simplistic form which breaks down in the presence of vanishing coefficients, but its generalisations are relatively simple, see, the comments at the end of

the previous section.

$$u_0 + u_1 x + u_2 x^2 + \cdots = g_0 + \cfrac{g_1 x}{1 + \cfrac{g_2 x}{1 + \cfrac{g_3 x}{1 + \ddots}}} \tag{8}$$

We can forget about the 0th term which is trivial. The rest of the expansion is a 2-liner:

$\mathrm{cnf}\, u@(u_1 : \bar{u}) = u_1 : \bar{g}$ **where**
$\bar{g} = \mathrm{cnf}(\mathrm{tail}(u_1/u))$

where `tail` removes the first element of the list (it is always 1), and u_1 in the division u_1/u should be promoted to a series. We do not discuss the degenerate cases when the series U is, in fact, a finite rational function, which stops the expansion, and requires a more intelligent treatment. But if we truncate the continued fraction after $2m$ terms, and if we reconvolute it back, we obtain just the $[m/m]$ diagonal Padé approximant without solving any equations. This is the reconvolution program:

$\mathrm{dpad}\, 0\ g_0\ (g_1 : \_) = (c, 1)$
$\mathrm{dpad}\, m\ g_0\ (g_1 : \bar{g}) = (c \cdot p + (0 : g_1 \cdot q), p)$
where$(p, q) = \mathrm{dpad}(m-1)\ 1\ \bar{g}$

The continuant sequence for the exponential function is equal to $[1, 1, \frac{-1}{2}, \frac{1}{6}, \frac{-1}{6}, \frac{1}{10}, \frac{-1}{10}, \frac{1}{14}, \frac{-1}{14}, \frac{1}{18}, \cdots]$, and this is a good testing exercise. The generation of the continued fractions from the $1/n!$ series is not very stable, the cancellations are important, and floating calculations behave badly. The example above served the author to discover (unwillingly!) a bug in one infinite precision rational package.

For the [4/4] approximant of the exponential function we immediately get:

$$\frac{1 + \frac{1}{2}x + \frac{3}{28}x^2 + \frac{1}{84}x^3 + \frac{1}{1680}x^4}{1 - \frac{1}{2}x + \frac{3}{28}x^2 - \frac{1}{84}x^3 + \frac{1}{1680}x^4}. \tag{9}$$

Of course, the claim that we got the Padé approximant "without solving any equations" is just a magic incantation. In fact, the reconvolution procedure *is* an equation solver by backward substitution. In the next section we present another equation solver in a Byzantine style.

A critical reader should note that the last algorithm *is not lazy*, although it uses an infinite stream. This is just a standard recursive formula. Can we do it lazily? Of course, the extrapolating recurrence relations for the continuous fraction *convergents* are well known, see [27], in our case, they take the following

form:

$$g_0 + \cfrac{g_1 x}{1 + \cfrac{g_2 x}{1 + \cfrac{g_3 x}{1 + \ddots}}} = \frac{g_0}{1}, \frac{g_0 + g_1 x}{1}, \frac{g_0 + g_1 x + g_0 g_2 x}{1 + g_2 x}, \tag{10}$$

$$\frac{g_0 + g_1 x + g_0 g_2 x + g_0 g_3 x + g_0 g_1 g_3 x^2}{1 + g_2 x + g_3 x}, \ldots, \frac{P_n(x)}{Q_n(x)}, \ldots$$

where the convergents fulfil the recurrence

$$\frac{P_{n+1}(x)}{Q_{n+1}(x)} = \frac{g_{n+1} x P_{n-1}(x) + P_n(x)}{g_{n+1} x Q_{n-1}(x) + Q_n(x)} \tag{11}$$

which gives the program below. Now we do not have to recalculate backwards another approximant if we need the next term

cnvg$(g_0 : g_1 : \bar{g}) =$ cnx$(g_0, 1)$ $(g_0 : g_1, 1)$ **where**
  cnx $r@(p_p, q_p)s@(p_m, q_m)(a_0 : \bar{a}) = r :$ cnx $s\ t\ \bar{a}$ **where**
    $t = ((0 : a_0 \cdot p_p) + p_m,$
    $(0 : a_0 \cdot q_p) + q_m))$

5. Asymptotic expansions

Some asymptotic developments are ideally well adapted to the lazy treatment. Take a typical series obtained by the iteration of the integration by parts, for example the generalized erfc function

$$\int_x^\infty \frac{\mathrm{e}^{-t^2/2}}{t^m}\,\mathrm{d}t = \frac{\mathrm{e}^{-x^2/2}}{x^{m+1}} - (m+1)\int_x^\infty \frac{\mathrm{e}^{-t^2/2}}{t^{m+2}}\,\mathrm{d}t. \tag{12}$$

This is an extremely simple open recurrence for the series in $1/x$:

erfg $m = 1 : 0 : -(m+1) \cdot$ erfg$(m+2)$.

Here the result is trivially known, but the same technique is applicable in more intricate cases.

We present here another example, suggested in the wonderful book [12]. This example is sufficiently archetypical to be useful, and sufficiently crazy to be interesting. We will show how the *perturbation* of the Stirling asymptotic series for the factorial will *generate* this series. Asymptotically $n! \simeq \sqrt{2\pi n}(n/\mathrm{e})^n S(n)$, where the series $S(n) = (1 + a_1/n + a_2/n^2 + \cdots)$ is known, but we shall not unveil the mystery yet. What we assume is that if the formula above holds, it should agree with the recurrence

$n! = n \cdot (n-1)!$, from which we deduce

$$S(n-1) = \frac{1}{\mathrm{e}} \left(1 - \frac{1}{n}\right)^{-(n-1/2)} S(n), \tag{13}$$

or, after introducing $x \equiv 1/n$

$$S\left(\frac{x}{1-x}\right) = G(x)S(x), \tag{14}$$

where

$$G(x) = \exp\left(-1 - \left(\frac{1}{x} - \frac{1}{2}\right)\log(1-x)\right). \tag{15}$$

The correcting factor is easily computable by our package. We get

$$G(x) \equiv 1 + x^2 f(x) = 1 + \frac{x^2}{12} + \frac{x^3}{12} + \frac{113}{1440}x^4 + \frac{53}{720}x^5 + \frac{25163}{362880}x^6 + \cdots. \tag{16}$$

This fixes the 0th term of S, it must be 1. We write $S(x)$ as $1 + x \cdot A(x)$ (whose first term we call A_1, and not A_0), and we realize with dismay that the formula

$$\frac{1}{1-x} A\left(\frac{x}{1-x}\right) = A(x) + x \cdot f(x) + x^2 f(x) A(x) \tag{17}$$

is not an algorithm, but a system of equations, with the unknowns having the same order on both sides. However, after the subtraction of $A(x)$ from both sides we obtain

$$A_1 \frac{1}{x}\left(\frac{1}{1-x} - 1\right) + xA_2 \frac{1}{x}\left(\frac{1}{(1-x)^2} - 1\right) + x^2 A_3 \frac{1}{x}\left(\frac{1}{(1-x)^3} - 1\right) + \cdots$$
$$= f(x)(1 + xA(x)), \tag{18}$$

where each factor $1/x(1/(1-x)^m - 1)$ is a regular series. Now the formula looks "sufficiently lazy", but it continues to be a system of equations for the coefficients of A. We propose thus a lazy approach to backward substitution. Suppose we try to find the series u obeying the equation

$$\frac{u_0}{g^{(0)}(x)} + x\frac{u_1}{g^{(1)}(x)} + x^2\frac{u_2}{g^{(2)}(x)} + \cdots = b(x), \tag{19}$$

where g and b are known. Obviously $u_0 = g_0^{(0)} \cdot b_0$, and

$$\frac{u_1}{h^{(1)}(x)} + x\frac{u_2}{h^{(2)}(x)} + \cdots = \frac{1}{x}(b(x)g^{(0)}(x) - u_0), \tag{20}$$

where $h^{(k)} = g^{(k)}/g^{(0)}$. The problem is solved. We construct the list of coefficient functions g, and we recklessly apply the schema (20) to the Eq. (18), "forgetting" that the

right-hand side is *not* known, but involves A.

```
stirl = a where
  xm = 1 : (-1)                    -(1 - x; completed with zeros)
  a = bksub(f · (1 : a)) glist
  glist = iterg xm where
        iterg p = - p/(tail p) : iterg(p · xm)

  bksub b(g(0) : ḡ) = z0 : z̄ where
    (z0 : z̄q) = b · g(0)
    z̄ = bksub z̄q(map(/g(0)) ḡ)
```

which produces the result:

$$A = 1 + \frac{1}{12}x + \frac{1}{288}x^2 + \frac{-139}{51840}x^3 + \frac{-571}{2488320}x^4 + \frac{163879}{209018880}x^5 + \cdots \tag{21}$$

to any precision you wish, which is not too easy to find in the popular textbooks.

6. Some partition functions

We present here two more examples which show the generating power of the co-recurrent algorithms.

The generator of the unlabelled, rooted Cayley trees has the form

$$\tau(x) = x \exp\left(\tau(x) + \frac{\tau(x^2)}{2} + \frac{\tau(x^3)}{3} + \cdots + \frac{\tau(x^m)}{m} + \cdots\right). \tag{22}$$

There is no closed expression known for the coefficients of τ. Such formulæ might be interesting for people working in the theory of complexity [25], or for physicists using the diagrammatic expansions in perturbation theory, and computing several combinatorial factors [8]. The expression above seems not to be computable because of the infinite sum in the exponent. But if we introduce ψ such that $\tau = x\psi$, we see that the exponent satisfies in fact a "decent" recurrence relation, and we may immediately code

```
tau = (0 : psi) where
  psi = serExp(exsum psi 1)
  exsum u m = 0 : ((1/m) · (compow m u) + exsum u(m + 1)),
```

where `compow m` is a simple function which separates the elements of its argument by `m` zeros. We get immediately

1, 1, 2, 4, 9, 20, 48, 115, 286, 719, 1842, 4766, 12486, 32973, 87811, 235381, 634847, 1721159, 4688676, 12826228, 35221832, 97055181...

Another case study is the generating function for the partitions of an integer. There are several ways of representing and for computing it, but we are particularly interested

by the infinite product representation:

$$Z(x) = \prod_{n=1}^{\infty} \frac{1}{1-x^n}. \tag{23}$$

Computing a finite approximation to it by standard iterative methods is rather unwieldy, so other representations are used. You may find the solution for $Z(x)$ in [6], using logarithms and the Lambert function, but we can rewrite this as an open recurrence

$$Z(x) = Z_1(x), \quad \text{where } Z_m(x) = \frac{1}{1-x^m} Z_{m+1}(x). \tag{24}$$

This is a runaway, Mephistophelean perversion rather than an algorithm, and the lazy programming will not help us *directly* here. But after having rewritten it as $Z_m(x) = Z_{m+1}(x) + x^n Z_m(x)$, and after introducing $B_m(x)$ such that $Z_m(x) = 1 + x^m B_m(x)$, we have the final recipe

$$B_m(x) = 1 + x(B_{m+1} + x^{m-1} B_m(x)), \tag{25}$$

which gives us the following effective, and quite efficient program:

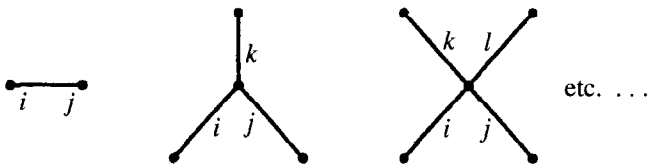
```
partgen = 1 : B 1 where
  B n = p where
    p = 1 : B(n + 1) + byxn(n - 1) p
```

where byxn is a function which multiplies a series by x^n (adds n zeros at the beginning). And here is the result: 1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77, 101, 135, 176, 231, 297, 385, 490, ..., which starts to scroll immediately through the screen, although after having generated some dozens of terms the process begins to slow down, because the dynamically created thunks become bigger.

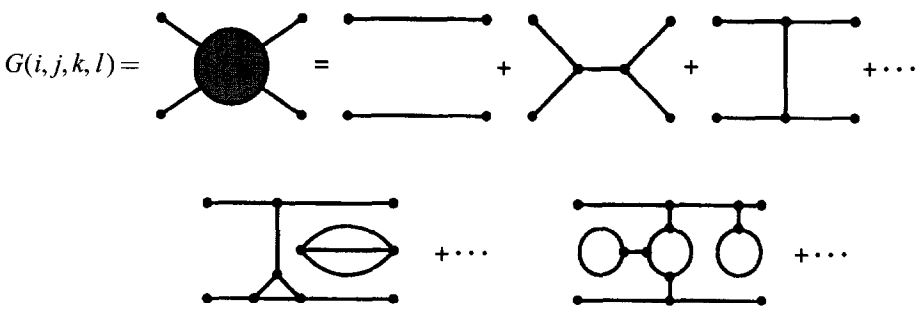
7. Diagram generation

This is just a simplistic, "toy" example of algorithmic generation of open recursive structures: representations of Feynman diagrams, using the Dyson–Schwinger equation. The same or similar techniques can be used for the Mayer graphs in statistical mechanics, geometric models of solidification, or other cases in physics, where the theory is nice enough to tell us how to expand a structure, but not how to stop the expansion. The structures: full sums of graphs, or amorphous solids, are fixed points of infinite growing processes. We restrict the presentation to a 0-dimensional scalar φ^3 theory, see for example [7]. In a zero-dimensional theory there are no spatial coordinates, so all objects are reduced to pure numbers. For us this is irrelevant, as we are just interested in the generating algorithms, but even, in general, such models are not completely useless – they provide a reasonable way to calculate combinatorial weights in a serious theory.

A particle – the field quantum – can propagate, or interact through a triple, quadruple, etc. vertex

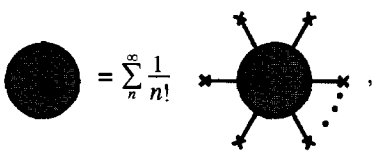


Restricting the discussion to the scalar φ^3 case means that there are no other types of vertices than triple and that the propagators have no internal (spinorial, etc.) structure. The line represents the propagator, a function Δ_{ij}, where i,j denote the attributes of the particle in the initial and the final state. The vertex, or the primitive interaction is a function γ_{ijk} which is considered small and which will be used as the perturbation parameter. The aim of the theory is to obtain the transition amplitude $G(i_1, i_2, \ldots, i_n)$ between two arbitrary states: one subset of $\{i_k\}$ denotes the incoming particles, and the remaining indices – outgoing, taking into account all possible interactions: emissions and absorptions of particles in the vertices. From these atoms one can construct all kind of composite behaviour. For example, the amplitude (or Green function) for a binary interaction (scattering) has the following graph expansion:



The exact theory requires the summation of all the graphs. If the vertex corresponds to a small coupling constant, the perturbation theory can be used (with all usual caveats). We introduce the generating functional:

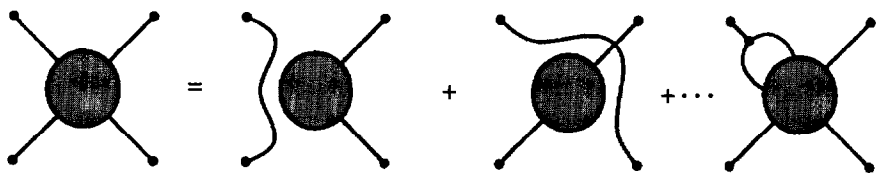
$$Z[J] = \sum_{n=0}^{\infty} \frac{1}{n!} \sum_{i_1 \ldots i_n} G(i_1, i_2, \ldots, i_n) J_{i_1} J_{i_2} \cdots J_{i_n}, \quad \text{or}$$



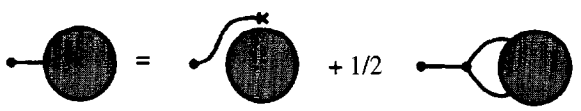
where the crosses denote the fictitious sources J. The Green functions are given by the functional derivatives

$$G(i_1, i_2, \ldots, i_n) = \frac{\partial^n}{\partial J_{i_1} \ldots \partial J_{i_n}} Z[J] \tag{26}$$

for $J = 0$. If the theory is closed, each particle either passes through as a spectator, or interacts at least once. The recursive reduction of the amplitudes becomes clear



from which we can deduce the recursive representation of $Z[J]$:



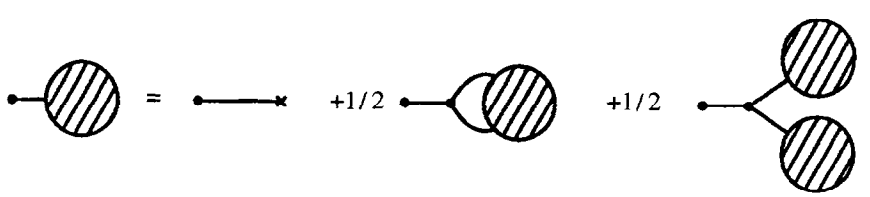
or

$$\frac{\partial}{\partial J_i} Z[J] = \Delta_{ij} J_j Z[J] + \frac{1}{2} \Delta_{ij} \gamma_{jkl} \frac{\partial}{\partial J_k} \frac{\partial}{\partial J_l} Z[J]. \tag{27}$$

$Z[J]$ generates all the graphs, including those with disconnected spectators. But from the general graph theory it is well known that $W[J] = \ln Z[J]$ is the generator of all the connected components. It satisfies the equation

$$\frac{\partial}{\partial J_i} W[J] = \Delta_{ij} \left\{ J_j + \frac{1}{2} \gamma_{jkl} \left(\frac{\partial^2 W}{\partial J_k \partial J_l} + \frac{\partial W}{\partial J_k} \frac{\partial W}{\partial J_l} \right) \right\}. \tag{28}$$

which corresponds to:



The Dyson–Schwinger equations (27) and (28) are so elegant, that one can find them in any book on Quantum Field Theory. Sometimes the authors remark casually that these equations are not very practical. For the actual Feynman diagram generation other frameworks are used, see for example [22] and references therein.

One reason for this disfavour is clear, the D–S equations are *open recursive formulæ*. However, from the lazy semantics standpoint they are not just recurrences but

algorithms! Of course, for a full-fledged theory we would need the spinor/tensor algebra, multidimensional integration, etc. They are *extremely* important, but from the generational point of view – almost irrelevant. In the zero-dimensional space the vertex and the propagator are just scalars. We can normalize the propagator, taking $\Delta = 1$.

We introduce now an auxiliary variable $\varphi = \mathrm{d}W/\mathrm{d}J$. It obeys the equation

$$\varphi = J + \tfrac{1}{2}\gamma(\varphi' + \varphi^2). \tag{29}$$

This is a derivative of W – a series in J representing the "full theory": each term is a series in γ. The equation for φ is differential in J, but algebraic in γ. Disentangling this by hand is very clumsy (this is a suggestion for particularly sadistic teachers of Quantum Field Theory).

We base our strategy on the following: φ will be considered *first* as a series in γ, whose elements are series in J. The first term is equal to $\varphi_0(J) = J$, the unit series. We define the derivative of such a compound as a *map* over its elements. Its coding in Haskell is: `indiff = map diff`. We code thus

```
phi = j : (1/2) · (phi^2 +indiff phi) where
  j = 0 : 1                    – The unit series
```

In order to compute the scattering amplitudes, the propagators, etc., we have to *transpose* φ. It will be treated as a series in J, whose elements are functions of γ. The propagator is equal to $W'' = \varphi'|_{J=0}$, so, it is enough to collect the second elements of the internal items of φ

```
d2 = map(head .tail) phi
```

The final formula for the propagator is

$$d_2 = 1 + \gamma^2 + \frac{25}{8}\gamma^4 + 15\gamma^6 + \frac{12155}{128}\gamma^8 + \frac{11865}{16}\gamma^{10} + \cdots, \tag{30}$$

(which corrects a small mistake in the Cvitanovič's book).

8. Conclusions

One may observe that the presented examples do not belong to the domain called usually "computer algebra", as there are no symbolic indeterminates in the results. (We do not cheat: a univariate polynomial or series does not need to include explicitly the indeterminate. As we know, Knuth calls this domain "seminumerical".) We want to stress upon the following:

The co-recurrent approach to the construction of lazy data structures does not depend on the underlying mathematical domain. We have voluntarily used a universal functional language in order to keep the examples simple, but the series, etc. could have symbolic coefficients as well, which would require the use of some symbolic package *just to manipulate these coefficients*. We tried to suggest that the manipulation of

programs – co-recursive arrangement of evaluations, auto- and cross-referring (lazy) data, application of higher-order combinators (maps and zips), etc. provides an elegant and practical alternative to some *symbolic data* manipulations. The lazy formulation of algorithms permits to

- deal *directly* with some extrapolating recursive problems found in sciences;
- replace the chain of recurrence formulæ by a compact representation of the *full solution* of these recurrences;
- liberate the user from the curse of controlling explicitly the truncation orders in all sorts of iterative processes;
- formulate in an extremely compact way the solution of a system of equations adapted to the back-substitution mechanism.

The potential of non-strict evaluation is not restricted to "infinite" streams, but constitutes a reasonable coding tool in many other cases, it has been used to construct animation packages, or solve numerical problems using finite elements. It would be very useful to have a full-fledged lazy algebraic package, but it seems that for efficiency reasons it must be built upon a lazy evaluation kernel, as adding it ad hoc to an existing strict systems makes it difficult to exploit its full power. So, those who would like to implement immediately their lazy algorithms should use lazy languages such as Haskell, Hope [23], or commercial, superbly distributed and documented Miranda of Research Software Ltd. The programs are, in general, as efficient as their strict equivalents, but the comparison is difficult, as often there are no equivalents... In all of the presented examples, the results started appearing on the screen immediately, even if the last term could take a few minutes. The suspended evaluations might save much work, but the dynamic creation of thunks is costly, and the unevaluated closures occupy the storage which must be reclaimed by the garbage collector after the evaluation. This is one of the reasons why the lazy functional languages are considered to be not very efficient. We are mostly interested in saving *human* work, and here the lazy approach clearly wins.

The author implemented a toy lazy package in MuPAD using its powerful and user-friendly object-oriented subsystem, but neither MuPAD [9] nor Maple are suitable for this purpose, due to the fact that the lexical closures (local, dynamically constructed functions) must be simulated by substitutions.

Unfortunately, the industrial strength functional lazy languages are relatively new, and the work has just begun. For the time being, the reader who is mainly interested in computer algebra, is encouraged to do some experiments in Axiom or, perhaps, in Mathematica, which, being partially a rewriting system, might be better adapted to lazy manipulations than a procedural language such as Maple.

References

[1] H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1984.

[2] L. Allison, Circular programs and self-referential structures, Software Practice and Experienc ···· 19(2) (1989) 99–109.

[3] R.S. Bird, P. Wadler, Introduction to Functional Programming, Prentice-Hall, Englewood cliffs, NJ, 1988.
[4] R.P. Brent, H.T. Kung, Fast algorithms for manipulating formal power series, J. ACM 25 (1978) 581–595.
[5] W.H. Burge, Recursive Programming Techniques, Addison-Wesley, Reading, MA, 1975.
[6] W.H. Burge, S.M. Watt, Infinite Structures in Scratchpad II, EUROCAL'87, Lecture Notes in Computer Science, vol. 378, pp. 138–148.
[7] P. Cvitanovič, Field theory, Nordita Lecture Notes, 1983.
[8] C. Domb, Graph theory and embeddings, in: C. Domb, M.S. Green, (Eds.), Phase Transitions and Critical Phenomena, Academic Press, New York, 1974, pp. 1–92.
[9] B. Fuchssteiner et al., MuPAD Manual, Birkhäuser, Basel, 1995.
[10] A.D. Gordon, A tutorial on co-induction and functional programming, Proc. 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, September, 1994.
[11] R.W. Gosper, MIT AI Laboratory Memo 239, Hack 101, February, 1972, pp. 37–44.
[12] R.E. Graham, Donald E. Knuth, Oren Patashnik, Concrete Mathematics, Addison-Wesley, Reading, MA, 1989.
[13] D. Guntz, M. Monagan, Introduction to Gauss, Sigsam Bull. 28 (2) (1994) 3–19.
[14] P. Hudak, S.P. Jones, P. Wadler et al., Report on the programming language Haskell (Version 1.3), Tech. Report Yale University/Glasgow University, 1996.
[15] M.P. Jones, Gofer, Functional Programming Environment (1991). Unpublished documentation of the Gofer language. Available from: `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`.
[16] J. Karczmarczuk, Functional programming and mathematical objects, Functional Programming Languages in Education, FPLE '95, Lecture Notes in Computer Science, vol. 1022, Springer, Berlin, 1995, pp. 121–137.
[17] D.E. Knuth, The art of computer programming, Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, MA, 1981.
[18] H.T. Kung, J.F. Traub, J. ACM 25 (1978) 245–260.
[19] J.D. Lipson, Newton's method: a great algebraic algorithm, Proc. ACM Symp. on Symbolic and Algebraic Computation 1976, pp. 260–270.
[20] M.D. McIlroy, Squinting at power series, Software – Practice and Experience 20 (1990) 661–683.
[21] S. Moritsugu, N. Inada, E. Goto, Symbolic Newton iteration and its applications, Symbolic and Algebraic Computations by Computers, World Scientific, Singapore, 1985, pp. 105–117.
[22] P. Nogueira, Automatic feynman graph generation, J. Comput. Phys. 105 (1993) 279–289.
[23] N. Perry, Hope+, Technical Report IC/FPR/LANG/2.5.1/7, Imperial College, 1987.
[24] D.A. Turner, Elementary strong functional programming, Proc. Functional Programming Languages in Education (FPLE '95), Nijmegen, Lecture Notes in Computer Science, vol. 1022, Springer, Berlin, pp. 1–13.
[25] J.S. Vitter, P. Flajolet, Average-case analysis of algorithms and data structures, in: Jan Van Leeuven, (Ed.), Algorithms and Complexity, Elsevier, Amsterdam, 1990, pp. 431–520.
[26] J. Vuillemin, Exact real computer arithmetic with continued fractions, IEEE Trans. Comput. 39(8) 1990 1087–1105.
[27] R. Zippel, Effective Polynomial Computation, Kluwer Academic Publishers, Boston, 1993.