Computer Algebra and Lazy Semantics

Jerzy Karczmarczuk
(University of Caen, France)

Abstract

We discuss some *lazy evaluation* techniques, and we present an attempt to implement in MuPAD a lazy evaluation package, which in our opinion is underestimated as the algorithm coding tool, useful in the construction of infinite streams, e. g. power series, but also for many other iterative structures or processes. We show how to use the class (MuPAD domain) system to overload the standard arithmetic to the domain of delayed data structures. We present a working model, but we conclude that the current internal semantics of MuPAD is not well adapted to this sort of experiments.

Reminder:

The function $f(x)$ is lazy if in certain circumstances, when *in case it does not need the value of* x *at all*, it will terminate without error even if called as $f(1/a)$ with $a = 0$, or in general any $f(\perp)$, where $\perp$ is a disaster, e. g. a non-terminating computation. This implies that the evaluation of x takes place under control of f, only when the value of x is needed. 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, and usually the result of the evaluation replaces the thunk *in situ*, which prevents further, useless evaluations, if the parameter x is used twice. This assumes the *referential transparency*, the possibility of substitution of the evaluated value for the thunk, and precludes the usage of side effects, unless the programmer is a pyromaniac. (It is difficult to rely on side-effects if you don't know when they occur...)

Ultra-classical example

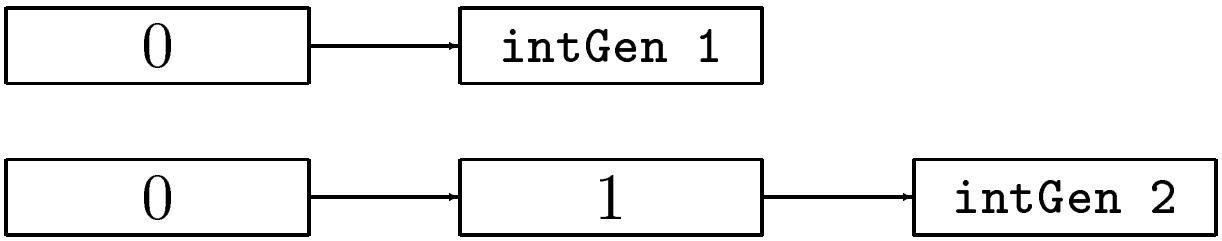
(For syntactic simplicity coded in Haskell. MuPAD examples will come later). Here is the infinite list of integers starting with 0. The colon is the infix 'cons' operator:

```
intlist = intGen 0
 where
  intGen m = m : intGen (m+1)
```

We see here that

- An open, non-terminating recursion is used.
- We may, and we must use only a part of the created data, because *the data itself, not just the process, is infinite*.
- The code is very short.

The main point, which is the leitmotif of this talk, is that *during the generation, we don't care about the truncations*. If we don't look at the element no. 132, only first 131 will be evaluated, followed by a promise.



etc.

For those who have strong connotation to memoizing functions (option: remember): this is **not** a function, but a data structure.

We repeat: our main purpose is to simplify the construction and coding of algorithms, using a functional programming style. There is no more philosophy in that. So, take a more intricate example. The same list `[0, 1, 2, 3, ...]` might be constructed as

```
intlist = 0 : (ones + intlist)
 where
  ones = 1 : ones
```

and where we have defined (omitting here some nasty details) the pair-wise summation of sequences:

```
(a:aq) + (b:bq) = (a+b) : (aq+bq)
```

(which is a shameless lie, because a Faithful Functionalist would use the `zipWith (+)` functional to make the definition more generic.)

If you wish:

```
zipWith op (a:aq) (b:bq) =
  (a `op` b) : zipWith op aq bq
```

Yes, that's it:

intlist :		0	.	.	.	.	.	.	.	.	...
ones :		1	1	1	1	1	1	1	1	1	...
intlist :	0	1	.	.	.	.	.	.	.	.	...

You may ask your 10-years old children to complete the pattern.

(Btw. such infinite list in Haskell has a standard abbreviation: `[0 ..]`.)

The *lazy* (or **non-strict**) evaluation semantics, or **call-by-need** protocol, very well known for many years in the domain of functional programming, is usually considered as something somehow exotic by the "true programmers", including the specialists on computer algebra.

All general code in MuPAD, Maple, Axiom, Magma, Reduce etc. is *strict*, i. e. the call-by-value is used.

Why?

Probably because the non-evaluating style: operations upon symbolic data representing algebraic expressions, irreducible function calls etc., makes the programmer think much more about the *data structures* than about the *evaluation code*.

There is a slight methodological confusion not only between x as a programming variable and x as an algebraic indeterminate, but also between various "true faces" of $2\sin(x+1)-\sqrt{y}-$ is it a n-ary tree representing this *form*, or a computer *code* which evaluates it? What is the semantics behind the "love affair" within the triangle: **evaluation, substitution, simplification**?

So, it is better not to think too much about dynamic processing of computer codes, and it is better to rely on the strict semantics (*cum grano salis:* call by value protocol).

And (IMHO) it is better not to use MuPAD or Maple as the first programming language for beginners. It has been done, the effects were sometimes encouraging, because it **was interesting**, but the confusion mentioned above might be harmful.

By the way: both venerable Teams promise already for years the implementation of lexical closures... Where are we?

Some classical examples: power series manipulation.

If we represent a power series $U = u_0 + u_1x + u_2x^2 + \cdots$ as an infinite sequence (list) `[u0, u1, u2, ...]`, the lazy addition term by term is trivial. It is also easy to multiply such a list by a scalar (let's denote it by the operator `*>`) using `map`:

`s *> u = map (s *) u`, or

```
s *> (u0:uq) = (s*u0):(s*>uq)
```

The multiplication proceeds as follows. We split the series: $U = u_0 + x\bar{u}$. Then

$U \cdot V = (u_0 + x\bar{u})(v_0 + x\bar{v}) = u_0 \cdot v_0 + x(u_0\bar{v} + \bar{u}v)$.

or

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

The division $W = U/V$ is a rearrangement of the formula for $U = W \cdot V$:

$$(u_0 + x\bar{u})/(v_0 + x\bar{v}) = w_0 + x(\bar{u} - w_0\bar{v})/v$$

where $w_0 = u_0/v_0$.

How to compute $W = \exp(U)$? We know of course how to differentiate ant how to integrate the series. *Cum grano salis:*

```
diff (_:uq) = zipWith (*) uq [1..]
integ cst u = cst : zipWith (/) u [1..]
```

and knowing that $W' = U' \cdot W$, we construct

```
exp u@(u0:_) = w
 where
  w = integ (exp u0) (w * diff u)
```

We can use similar techniques to compute U^α, to invert the series, to implement the Newton algorithm for solving equations fulfilled by series, etc. We repeat: the main advantage is the extreme compactness of the code, which remains perfectly legible. Enough of Haskell...

How to do this in MuPAD?

What is **good** in MuPAD is:

- the genericity (polymorphism) of operations upon domain elements;
- the existence of the `hold` form.

We have encountered two major obstacles to implement efficiently the lazy semantics:

- lack of lexical closures and the necessity of using substitutions, and
- lack of mutable data structures (lists).

In fact, the only mutable data structure in MuPAD is the *domain*.

All the other are automatically copied, when modified by some element assignment: $A := \ldots;$ $B := A;$ $A[7] := XX;$ forces immediately the cloning of the value of B, which remains at it was before the assignment to $A[7]$. This precludes the automatic update of lazy expressions, i.e. the substitution of a value for its source thunk.

The presentation below is severely massacred. Lazy lists are just domains in which linking uses the domain indices:

```
LZ:=domain(): LZ::NLZ:=0: #(first index)#

makel:=proc(x) local el; begin
  el:=new(LZ,x,(LZ::NLZ:=LZ::NLZ+1));
  domattr(LZ,el):="BUG! SHOULD NOT HAPPEN!";
  el
  end_proc:

hd:=proc(x)
  begin op(x,1) end_proc:
```

```
# A non-evaluating tl #
ltl:=proc(x) begin domattr(LZ,x) end_proc:

tl:=proc(x) local p;
  begin p:=domattr(LZ,x);
  if type(p)=DOM_PROC then p:=p();
   domattr(LZ,x):=p end_if; p
  end_proc:

# Transformation into a normal list #
take:=proc(n,l) local buf,el,i; begin
  buf:=[hd(l)]; el:=l;
  for i from 1 to n-1 do
    el:=tl(el); if type(el)=DOM_NULL
                then break end_if;
    buf:=append(buf,hd(el)); end_for;
  buf end_proc:
```

```
# Lazy 'cons' #
css:=proc(x,y) option hold; local el;
  begin el:=makel(context(x));
    domattr(LZ,el):=
     subs(proc() begin _y end_proc,_y=y);
    el
  end_proc:
```

Infinite repetitions are just trivial cyclic lists:

```
alias(cgener(vr,A,B)=
  (vr:=makel(A); domattr(LZ,vr):=B)):

lrep:=proc(x) local el;
  begin cgener(el,x,el) end_proc:
LZ::zero:=lrep(0):  LZ::one:=css(1,lzeros):
```

There are however some touchy points, and `css` is not satisfactory. Suppose we want to construct lazy mapping, or zipping of two lazy sequences.

```
lmap:=proc(f,l) begin
  if type(l)=DOM_NULL then null() else
  cssa(f(hd(l)),lmap(f,tl(l)),[f,l]) end_if
end_proc:

lzip:=proc(opr,l1,l2) begin
  if type(l1)=DOM_NULL or type(l2)=DOM_NULL
   then null() else
   cssa(opr(hd(l1),hd(l2)),
        lzip(opr,tl(l1),tl(l2)),
        [opr,l1,l2])
  end_if end_proc:

# Now we can add lists #
lplus:=proc(l1,l2)
  begin lzip(_plus,l1,l2) end_proc:
```

In order to construct the self-propagating application of the operator, we have to declare that some identifiers are global variables (`hold`ed), and must be replaced by the arguments.

```
cssb:=proc(y,vrs) local m,p,v;
  begin v:=context(vrs);
   p:=zip(vrs,v,proc(a,b)
                 begin a=b end_proc);
   m:=subs(y,op(p));
   subs(proc() begin _y end_proc,_y=m)
  end_proc:

cssa:=proc(x,y,vrs) option hold; local el;
  begin cgener(el,context(x),cssb(y,vrs));
  el end_proc:
```

There is no problem in defining the standard operations on series.

```
smult:=proc(u,v) local v0; begin
  v0:=hd(v);
  cssa(hd(u)*v0,
       (v0*tl(u)+smult(u,tl(v))),
       [u,v,v0])
  end_proc:
...

alias(sintgen(u,l)=
  cssb(lzip(proc(x,y)
            begin x/y end_proc,
            u,lnats),
       l)):

sexp:=proc(u) local up,w; begin
  up:=sdiff(u);
  cgener(w,E^hd(u),sintgen(w*up,[w,up]));
  w end_proc:
```

We see that for compactness we use macros, and that all this would be much simpler, had we closures at our disposal...

More examples

Rational approximation: Wynn process

Given a sequence $s_0, s_1, s_2, \ldots$ we can accelerate its convergence by the following elegant rationalizing recurrent process, which generalizes the Aitken procedure:

$$\epsilon_n^{(-1)} = 0; \qquad \epsilon_n^{(0)} = s_n$$

$$\epsilon_n^{(k+1)} = \epsilon_{n+1}^{(k-1)} + \frac{1}{\epsilon_{n+1}^{(k)} - \epsilon_n^{(k)}}$$

used not only for numerical sequences but also to generate the Padé approximants from a given power series. The odd Wynn iterates are auxiliary only, without any meaning.

```
wn:=proc(p,u) local du;
 begin
   du:=tl(p)+
    lmap(proc(x) begin 1/x end_proc,
         tl(u)-u);
   cssa(p,wn(u,du),[u,du])
 end_proc:
wynn:=proc(s) begin wn(LZ::zero,s) end_proc:
```

We may test the algorithm starting with a useless π approximation: $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \cdots$. After having constructed the (lazy of course) sequence of partial sums, the procedure `wynn` creates an infinite stream of infinite streams. It suffices to take the head of every second element, and we get [2.666666666, 3.133333333, 3.141391941, 3.141587301, 3.141592505, 3.141592649, 3.141592653, ...].

Simple consistency might be a powerful generating tool

We present here another example, suggested in the book *Concrete Mathematics* of Graham, Knuth and Patashnik. We will show how the *perturbation* of the Stirling asymptotic series for the factorial will *generate* this series. Asymptotically

$$n! \simeq \sqrt{2\pi n}(n/e)^n S(n),$$

where the series $S(n) = (1 + a_1/n + a_2/n^2 + \ldots)$ is searched for. If the formula above holds, it should agree with the recurrence $n! = n \cdot (n-1)!$, from which we deduce

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

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

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

where

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

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

$$G(x) \equiv 1 + x^2 f(x) = 1 + \frac{x^2}{12} + \frac{x^3}{12} + \frac{113}{1440}x^4 +$$

$$\frac{53}{720}x^5 + \frac{25163}{362880}x^6 + \dots \ .$$

This fixes the 0-th 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

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

is not an algorithm, but a system of equations, with the unknowns having the same order on both sides. However, having subtracted $A(x)$ from both sides we obtain something still ugly, but more "algorithmic", at least at the first glance:

$$A_1\frac{1}{x}\left(\frac{1}{1-x}-1\right)+x\cdot A_2\frac{1}{x}\left(\frac{1}{(1-x)^2}-1\right)+$$

$$x^2\cdot A_3\frac{1}{x}\left(\frac{1}{(1-x)^3}-1\right)+\ldots=$$

$$=f(x)(1+x\cdot A(x)),$$

where each factor $\frac{1}{x}(1/(1-x)^m-1)$ 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

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

where g and b are known.

Obviously $u_0=g_0^{(0)}\cdot b_0$, and

$$\frac{u_1}{h^{(1)}(x)}+x\frac{u_2}{h^{(2)}(x)}+\ldots=\frac{1}{x}\left(b(x)\cdot g^{(0)}(x)-u_0\right),$$

where $h^{(k)}=g^{(k)}/g^{(0)}$. The problem is solved.

We construct the list of coefficient functions g, and we recklessly apply the above schema to the equation, "forgetting" that the right-hand side is *not* known, but involves A.

We omit the coding details, showing just the results:

$$A = 1 + \frac{1}{12}x + \frac{1}{288}x^2 + \frac{-139}{51840}x^3 +$$

$$\frac{-571}{2488320}x^4 + \frac{163879}{209018880}x^5 + \cdots$$

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

Schröder function for a power series

In several branches of physics or biology simulation we investigate the behaviour of iterative processes, which may lead to chaos, present some universality properties, etc.

Knowing the basic iteration $x_n \to x_{n+1} = U(x_n)$, we want to study the n-fold compositions of the function U:

$$x_n = U^{(n)}(x_0) = U(U(\ldots U(x_0)\ldots)).$$

Ernst Schröder introduced for a given function $U(x)$ and a constant w the function $V(x)$ which solves the following intricate functional equation

$$V(U(x)) = wV(x)$$

We see immediately that

$$V(U^{(2)}(x)) \equiv V(U(U(x))) =$$

$$= wV(U(x)) = w^2V(x)$$

which by analytic extension gives:

$$V(U^{(\alpha)}(x)) = w^\alpha V(x)$$

for any real α (if you know what does it mean for irrational α). It suffices to find the reverse of $V(x)$ in order to solve a general iteration problem.

Suppose that U is a series with 0 as its fixed point: $U(x) = U_1x + U_2x^2 + \cdots$. We want to construct $V(x) = x + V_2x^2 + \cdots$. Obviously the equation above implies $w = U_1$.

Knuth cites in ACP the solution of Brent and Traub. We follow them in trying to solve a more general equation:

$$V(U(x)) = W(x)V(x) + S(x)$$

for $V(x) = V_0 + V_1 x + V_2 x^2 + \cdots$, given U, W, and S as power series. We assume that $U(0) = 0$, so $U(x)$ will have the form $U(x) = x\bar{U}$. The idea is to compute the left-hand side of the generating equation up to $O(x^n)$, with n growing to infinity in a lazy iterative process.

We begin with $V_0 = S(0)/(1 - W(0))$. In the case of 0/0 we define $V_0 = 1$. Then of course, if $V(x)$ is represented as $V_0 + x\bar{V}$, we have

$$\bar{V}(U) = \frac{W}{\bar{U}}\bar{V}(U) + \left(V_0\frac{\bar{W}}{\bar{U}} + \frac{\bar{S}}{\bar{U}}\right)$$

which is again the same (open) recursive formula.

```
schroder:=proc(u,w,s)
 begin schr(1/tl(u),w,s)
 end_proc:

schr:=proc(ur,w,s) local s0,v0,w0,sn;
 begin s0:=hd(s); w0:=hd(w);
       v0:=(if s0=0 and w0=1 then 1
            else s0/(1.0-w0) end_if);
       sn:=(v0*tl(w)+tl(s))*ur;
       cssa(v0,schr(ur,w*ur,sn),[ur,sn])
 end_proc:
```

The function `schr` should be local within `schroder`, recursive, with *global* variable `ur`, and it should `hold` its own call. This might be done in MuPAD by emulating exotic function calls through special domains with the `func_call` attribute, but this would take us too far.

Using our function, it is quite easy to solve some homeworks, such as finding the functional square root of the exponential series, more precisely: find such $\phi(x)$ that $\phi(\phi(x)) = 2(\exp(x) - 1)$. The solution (cum grano salis) is given by

```
sphi:= 2.0*(exp(lzx) - 1): # lzx=0+x #
tmp:=serinv(schroder(sphi,2,0));
answ:= scompo(tmp,sqrt(2)*sphi);
```

[0, 1.414213562, 0.2928932188, 0.02137646171, 0.0005415565286, 0.00003308232283, -0.00001071513544, 0.000004055924908, -0.00000142615311 4, 0.0000004635955185, ...]

Oh, yes:

If $W(x) \equiv W_0 + x\bar{W} = U(V(x))$, and $V_0 = 0$, i. e. $V(x) = x\bar{V}$, then of course $W_0 = U_0$, and

$$\bar{W} = \bar{V} \cdot \bar{U}(V)$$

or

```
scompo:=proc(u,v)
 begin
  cssa(hd(u),tl(v)*scompo(tl(u),v),[u,v])
 end_proc:
```

From that we can construct the series inversion for $V(x) = x + V_2x^2 + \cdots$:

```
serinv:=proc(v) local t,vt,m;
 begin vt:=tl(tl(v));
  m:=makel(1); cgener(t,0,m);
  domattr(LZ,m):=cssb((-1)*m*m*scompo(vt,t)),
                          [m,vt,t]);
 t end_proc:
```

Conclusions

As you see, without lexical closures, the necessity of using the substitutions makes the lazy coding somehow delicate. Moreover, the memory administration is inefficient. Shall we sit and cry?

No. MuPAD is not a functional language, and the presented technique is somehow orthogonal to its design. We shall not suggest that the lazy techniques should be more heavily used in MuPAD (or in Maple, or Axiom), because this simply would not work. We wanted to show that

- The object-oriented flavour and coding contraptions (domains etc.) provide an excellent training field for some experiments in algorithm implementation, and evaluation semantics.

- People who plan to revamp MuPAD, or to construct a computer algebra system with some new possibilities ensured by a more elaborate virtual machine, might take into account some functional approaches to the algorithm construction, which seems to be an old, but unexploited and still promising domain.

  The very process of algorithm design in the functional sauce is very interesting and elegant. You might use lazy techniques to test something before optimizing its implementation.