Reprinted from pp. 206-217, Proc. ISSAC '88, Springer-Verlag LNCS 358, 1989.

A FIXED POINT METHOD FOR POWER SERIES COMPUTATION

Stephen M. Watt
IBM Thomas J. Watson Research Center
Box 218, Yorktown Heights, NY 10598 USA

Abstract

This paper presents a novel technique for manipulating structures which represent infinite power series.

When power series are implemented using lazy evaluation, many operations can be written as simple recursive procedures. For example, the programs to generate the series for the elementary transcendental functions are almost transliterations of the defining integral equations. However, a naive lazy algorithm provides an implementation which may be orders of magnitude slower than a method which manipulates the coefficients explicitly.

The technique described here allows a power series to be defined in a very natural but computationally inefficient way and transforms it to an equivalent, efficient form. This is achieved by using a fixed point operator on the delayed part to remove redundant calculations.

The paper describes this fixed point method and the class of problems to which it is applicable. It has been used in Scratchpad II to improve the performance of a number of operations on infinite series, including division, reversion, special functions and the solution of linear and non-linear ordinary differential equations.

A few examples are given of the method and of the speed up obtained. To illustrate, the computation of the first n terms of $\exp(u)$ for a dense, infinite series u is reduced from $O(n^4)$ to $O(n^2)$ coefficient operations, the same as required by the standard on-line algorithms.

1 Introduction

When computing with power series it is often not known in advance how many terms will be required. For this reason, some computer algebra systems provide facilities for working with "infinite" series.[1] What this means is that it is possible to create series, perform various operations on them (e.g., addition, differentiation, reversion) and then at the end ask for any number of terms of the result.

Infinite series are traditionally implemented as a list of generated terms and a remainder which can be used to produce further terms as required. The remainder is represented either as a formula for the general term or as the lazily evaluated tail of the term list. There are advantages to both approaches; the lazy evaluation approach is more general but, in the cases where it is possible, it is often useful to have a formula for the general term. Only the lazy evaluation approach is considered here.

In both of these representations it is desirable to have *on-line* algorithms for manipulating series, i.e. algorithms which produce the terms individually, in order, and which do not require the number of terms to be specified in advance. It is then possible to construct a series incrementally, computing only the new terms when they are required. Knuth [2] presents a few on-line algorithms for power series.

[1] Maple [4] and Scratchpad II [3][6][7], for example.

When infinite series are implemented using lazy evaluation, many series operations have extremely simple on-line programs which arise directly from the mathematical definition. However, the advantage of mathematical transparency can be overshadowed by hidden costs which make the programs impractical beyond low orders. The principal problem is that many series have recursive defining relations which cause the initial terms of the series to be recomputed several times. What is desired is some method of avoiding this recomputation while maintaining the simple form of the programs.

One possible solution would be to use an implementation where functions remember the values they have computed (e.g. *memo* functions in Lisp, or *option remember* in Maple). The problem with this approach is that it is pessimistic, remembering every term ever computed.

The approach taken here is to apply a fixed point operator in the construction of the series, making it self-referential. When several series have mutually recursive defining relations, a fixed point operator is applied to the entire collection. This way only those terms which will be of future use are preserved.

The outline of the remainder of the paper is as follows: Section 2 presents a few recursive lazy algorithms to illustrate their simplicity. Section 3 shows how the re-evaluation arises and examines its cost. Section 4 shows how to view the series calculation in such a way that the re-evaluation is explicit. Section 5 outlines how fixed point operators can be applied to data structure-forming operations to produce self-referential objects. The interesting effects of using fixed point operators in conjunction with lazy evaluation are then discussed briefly. Section 6 shows how to use such fixed point operators in the definition of power series. Section 7 characterizes the problems to which the method is applicable. Section 8 concludes, illustrating examples of the speed-up achieved.

2 Recursive Lazy Algorithms for Power Series

Let us assume that our programming language has some support for lazy evaluation[2] and that power series are represented densely as lazy lists of coefficients. In this case, possibly the simplest on-line algorithm is to add two series:[3]

```
a + b == cons(first a + first b, delay(rest a + rest b))
```

In this definition, "+" is overloaded: the addition `first a + first b` uses the "+" from the coefficient type and the addition `rest a + rest b` is a recursive call. Since the evaluation of the second argument is delayed, this "infinite recursion" does not cause run time problems.

It is by now a classic exercise [5][7][8] to compute power series for elementary functions using this technique applied to a defining integral equation. For example, the exponential may be defined by the relation

$$e^{u(x)} = e^{u(x_0)} + \int_{u(x_0)}^{u(x)} e^{u(z)} \mathrm{d}u(z)$$

[2]If not, then lazy arguments can be simulated by passing function/environment pairs as arguments.

[3]If finite series end, rather than trail to an infinite list of zeros, then the following modification can be made:

```
a + b ==
(null a => b; null b => a; cons(first a + first b, delay(rest a + rest b)))
```

The choice of x_0 is arbitrary. Assuming u does not have a pole, taking x_0 equal to the point of expansion provides the constant coefficient as $u(x_0)$.

The corresponding program in Scratchpad II uses the series function `integrate`, which takes a constant of integration and the integrand as arguments. The evaluation of the integrand is delayed by `integrate`.

```
exp u == integrate(exp lc u, exp u * pderiv u)
```

As with “+” in the previous example, there are two different uses of `exp` here. The call `exp lc u` uses the exponential function for the coefficient domain and the second call, `exp u`, is recursive.

A number of functions on power series were initially implemented in Scratchpad II using this simple style of lazy recursion, including the arithmetic functions, composition, Lagrange inversion, the elementary functions, the hypergeometric function, elliptic functions, and the solution of certain differential equations.

Other examples of recursive lazy procedures are shown below in their naive versions, so that they may be compared with the improved versions. The trigonometric functions may be implemented in terms of:

```
tan u == integrate(tan lc u, (1 + tan(u)**2)*pderiv u)
sin u == integrate(sin lc u,  cos u * pderiv u)
cos u == integrate(cos lc u, -sin u * pderiv u)
```

The function `lde` solves linear ordinary differential equations using undetermined coefficients. The call `y := lde(la, lp)` solves the n^{th} order equation

$$\begin{aligned}&y^{(n)} + lp_{n-1}y^{(n-1)} + \cdots + lp_1 y' + lp_0 y = 0\\ &y(0) = la_0, y'(0) = la_1, ...\end{aligned}$$

The function `ldeprod` integrates the trial $y^{(n)}$ using the boundary condition for each order.

```
lde(la, lp) ==
    integrate(first la, ldeprod(rest la, lp, lde(la, lp)))

ldeprod(la, lp, y) ==
    if null la then
        -- compute  y<n> = -(lp(0)*y + lp(1)*y' + ... + lp(n-1)*y<n-1>)
        -reduce(0,_+,zip(_*,lp,generate(pderiv,y))
    else
        integrate(first la, ldeprod(rest la, lp, y))
```

Not all of the recursive functions are based on integration. Two other examples susceptible to optimization are division and Lagrange inversion.

Series division can be performed based on the identity

$$\frac{a_0 + xA}{b_0 + xB} = \frac{a_0}{b_0} + \frac{x}{b_0}(A - B * \frac{a_0 + xA}{b_0 + xB})$$

```
a/b == delay
    if null a then return 0
    if null b then error "division by zero"
    a0 := first a; A := rest a
    b0 := first b; B := rest b
    if b0 = 0 then
        if a0 = 0 then return A/B else error "division by zero"

    cons(a0/b0, 1/b0 * (A - B * (a/b)))
```

Lagrange inversion of a power series f produces the series for g satisfying $g(x) = xf(g(x))$. One method for power series reversion is based on this.

```
lagrange f == delay cons(0, compose(f, lagrange f))
```

These programs operate on whole series, rather than series coefficients so their structure can mimic rather closely the defining relationships.

3 The Cost of Naive Computation

While correct, these definitions are not the most efficient for producing the indicated series — the initial terms in the series are recomputed for each new higher order term. Examining the case of $exp(u)$ illustrates this. It is seen that the call to `exp` is exactly the same at each recursive level.

How costly is this re-evaluation? Assume u has already been computed to the required order. (In practice, u is lazy and will not normally be pre-computed so the cost of extending u may be incurred as $exp(u)$ is extended.) When `exp` is called, `integrate` produces a series with some leading coefficient and a delayed tail. When the second term is needed, the delayed part is evaluated. This causes the function `exp` to be called again, as well as the functions `pderiv` and "*". To produce the n^{th} term, the $(n-1)^{st}$ term of the delayed series must be produced, along with the $(n-2)^{nd}$ term of its delayed series, and so on. The cost of producing the n^{th} term may be written as

$$T(n) = \sum_{i=0}^{n-1} T(i) + P(n) + M(n)$$

$P(n)$, the cost to compute the partial derivative, is $O(n)$ coefficient operations. $M(n)$, the cost to perform the multiplication, is $O(n^2)$. Therefore $T(n)$ is $O(n^3)$ and the cost to compute the first n terms of $exp(u)$, given a pre-computed series u, is $O(n^4)$ coefficient operations.

An on-line algorithm presented by Knuth [2] has cost $O(n^2)$ and a semi-on-line $O(n \log n)$ can be achieved using Newton's method. So, although the above program wins in terms of conciseness, it looses badly in terms of efficiency. In section 6, the fixed point method is used to reduce the cost of our program to that of the Knuth's, while maintaining the simple, direct definition.

4 Recasting the Problem

We have seen in our examples that a recursive call often has exactly the same arguments as the original. This is the source of the re-evaluation we wish to avoid.

In these cases, the series $f(x)$ satisfies a relation

$$f(x) = F(f(x))$$

for some F and we have used this fact to compute $f(x)$.[4] While the use of this relation leads to re-evaluation in a naive implementation, it allows the redundant calculations to be identified. Once they have been identified, they can be removed.

With this in mind, we can try to express series calculations so that when such a relation is satisfied, the functional equation is explicit. For example, rather than writing

$$L[y] = 0$$

for a differential equation, we try to solve

$$\hat{L}[y] = y$$

It is the cases where we make the functional equation explicit which can be optimized by computing a fixed point.

5 Fixed Point Operators

A fixed point p of a map F is a point in the domain of F such that $p = F(p)$. Given a function which operates on a recursively defined data type, it is often possible to compute a useful fixed point. Specific effects can be obtained by tailoring a particular structure-forming or structure-transforming function of which to take the fixed point. As well as providing a functional mechanism for manufacturing self-referential structures, a combination of lazy evaluation and self-reference may be achieved.

Consider a recursively defined data type T and the class of functions mapping $T \to T$. Certain functions in this class have trivial fixed points: the identity and constant valued functions. Some functions in the class may have no fixed point. Other functions may have a fixed point which it is impossible to compute effectively.

Let us restrict our attention to functions which do not perform operations on their argument but rather simply include it in a new structure which is returned as the value. Then we may always compute a fixed point as follows:

[4]In fact, the existence of this functional relation may be the reason the series is interesting in the first place.

```
fixedPoint(F) ==
    arg := generateUnique()
    ret := F(arg)
    if arg = ret then
        -- F is the identity
        return arb. element from the domain of F
    else
        ret := subs(arg = ret in ret)
        return ret
```

Here `generateUnique` is a function which returns a unique system-wide value. Since F does not perform any operations on its argument, it is safe to pass it this generated unique value, which strictly speaking does not lie in its domain. A common notation for this fixed point is YF.

From the definition of `fixedPoint` above we see that, for functions in our restricted class, the set of fixed points will be one of

- a single constant (for functions which ignore their argument),
- the entire domain (for the identity function), or
- a single self-referential structure

As an example, an infinite repeating list can be obtained as follows:

```
cons1234(lst) == cons(1,cons(2,cons(3,cons(4,lst))))
repeating1234 := fixedPoint cons1234
```

When the use of the fixed point operator is combined with lazy evaluation, the result is more than simply a self-repeating data structure. In this case, the self-reference can occur as a value in an environment for a delayed function evaluation. When that part of the data structure is evaluated, the result can be some interesting transformation of the self-reference.

This combination of lazy evaluation and self-reference is what we want for our power series calculations.

6 Using Fixed Point Operators For Series

It is now shown how computing a fixed point avoids the redundant calculations seen in section 3. Let us continue with the example of the exponential:

```
exp u == integrate(exp lc u, exp u * pderiv u)
```

In this case, the lazy recursive call is `exp u`.[5] Therefore, given `u`, we take the fixed point of the unary function

[5] Recall that `integrate` delays the evaluation of the integrand.

```
e +-> integrate(exp lc u, e * pderiv u)
```

and the exponential is given by

```
exp u == fixedPoint(e +-> integrate(exp lc u, e * pderiv u))
```

Here, `a +-> b` is Scratchpad II notation for the anonymous function $a \mapsto b$.[6]

Now the recursive calls to `integrate` are able to access the leading terms of the series computed so far. When defined this way, the exponential function evaluates each term only once. The dominant cost is that of the single delayed multiplication. See 8.

Sometimes large subexpressions arise and it is better to compute them prior to determining the fixed point. This may be done by first creating a function of higher arity, with the extra parameters to specify information about the argument series.

As a trivial example, one would write:

```
expre(e0, e, du)  ==  integrate(e0, e*du)
```

in the exponential case. This function can be curried with arguments pertaining to the particular series, `u`, to produce a unary function:

```
e +-> expre(exp lc u, e, pderiv u)
```

The fixed point of this unary function is now the exponential of `u`:

```
exp u == fixedPoint(e +-> expre(exp lc u, e, pderiv u))
```

7 Obtaining f = F(f)

This method is applicable to any function in which the lazy recursive call has the same arguments as the original call.

Sometimes functions may benefit by the judicious use of an identity. For example, the tangent was defined as

```
tan u == integrate(tan lc u, (1 + tan(u)**2)*pderiv u)
```

rather than

[6]The old Scratchpad II compiler uses a different notation: e.g., `3*#1 + 1` for `n +-> 3*n + 1`

```
tan u == integrate(tan lc u, sec(u)**2 * pderiv u)
```

to make it suitable for the fixed point method. An arithmetic identity was used for division (see section 2).

When a set of functions is mutually recursive, they may be written as a single system. For example, *sin* and *cos* become

$$\begin{bmatrix} sin \\ cos \end{bmatrix} u(x) = \begin{bmatrix} sin \\ cos \end{bmatrix} u(x_0) + \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \int_{u(x_0)}^{u(x)} \begin{bmatrix} sin \\ cos \end{bmatrix} u(z) \mathrm{d}u(z)$$

```
sincosre(s0,c0,sc,du) == [integrate(s0,sc.1*du),integrate(c0,-sc.0*du)]

sincos u == fixedPoint(sc +-> sincosre(sin lc u, cos lc u, sc, pderiv u), 2)

sin u == sincos(u).0
cos z == sincos(u).1
```

The argument 2 to `fixedPoint` indicates the size of the system.

With this in mind, applying the fixed point optimization method is quite straightforward. Applying it to the other examples of sections 2 gives:

```
tan u == fixedPoint(t +-> integrate(tan lc u, (1 + t**2)*pderiv u))

ldere(la,lp,y) == integrate(first la, ldeprod(rest la, lp, y))
lde(la,lp)     == fixedPoint(y +-> ldere(la, lp, y))

divre(a0, ib0, A, B, adivb) == delay cons(a0*ib0, ib0*(A-B*adivb))

a/b ==
    if null a then return 0
    if null b then error "division by zero"
    a0 := first a; A := rest a
    b0 := first b; B := rest b
    if b0 = 0 then
        if a0 = 0 then return A/B else error "division by zero"

    fixedPoint(adivb +-> divre(a0, 1/b0, A, B, adivb))

lagrangere(f,l) == delay cons(0, compose(f, l))
lagrange f      == fixedPoint(l +-> lagrangere(f, l))
```

Once we know we want to use fixed points in computations, this programming style can be used from the start. An example of a package created this way is `PowerSeriesODESolver`, displayed in 8., with examples shown in 8.

8 Conclusion

It has been shown how to use the fixed point operator to reduce the computational complexity of recursive lazy power series algorithms by removing redundant calculations. This allows many functions to have efficient programs which look much like their defining equations. Examples of the improvements achieved are shown in 8 and 8.

This method is particularly well suited to power series computation since recursive common subexpressions seem to be the norm, rather than the exception. The modification of programs using the fixed point is quite straightforward and is something which could in principle be done by an optimizing compiler.

Current work includes investigation of fixed points in multivariate power series and their use in solving implicit equations.

Acknowledgements

William Burge was the principal designer of infinite power series in Scratchpad II. The examples presented in section 2 are based on his programs. When the author presented him with this fixed point method to to improve existing programs, he then found other places where the method could be applied.

Bibliography

[1] H.B. Curry and R. Feys, *Combinatory Logic* North Holland, Amsterdam, 1958.

[2] D.E. Knuth, *The Art of Computer Programming Volume 2, Second Edition,* Addison-Wesley, Reading Mass, 1981.

[3] R.D. Jenks and B.M. Trager, *A Language for Computational Algebra,* Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation.

[4] B.W. Char, K.O. Geddes, G.H. Gonnet and S.M. Watt, *Maple User's Guide,* Watcom Publications, Waterloo Ontario, 1985.

[5] H. Abelson and G. Sussman (with J. Sussman), *Structure and Interpretation of Computer Programs,* The MIT Press, Cambridge Mass, 1985.

[6] R.D. Jenks, R.S. Sutor and S.M. Watt, *Scratchpad II: An Abstract Datatype System for Mathematical Computation,* IMA Volumes in Mathematics and Its Applications, Volume 14, Springer-Verlag, New York (to appear). (Also RC 12327, IBM Research 1986)

[7] W.H. Burge and S.M. Watt, *Infinite Structures in Scratchpad II,* Proc. 1987 European Conference on Computer Algebra, Leipzig, GDR, Springer Verlag Lecture Notes in Computer Science (to appear).

[8] J.P. Henry and M. Merle, *Puiseux Pairs, Resolution of Curves and Lazy Evaluation,* Preprint 1987.

	exp(x), for the monomial x						exp(u), for u dense + infinite					
	Naive Recursion			Fixed Point			Naive Recursion			Fixed Point		
Terms	+	*	/	+	*	/	+	*	/	+	*	/
10	62	135	55	11	23	10	255	275	55	63	65	10
20	227	475	210	21	43	20	1710	1750	210	228	230	20
30	492	1015	465	31	63	30	5365	5425	465	493	495	30
40	857	1755	820	41	83	40	12220	12300	820	858	860	40
50	1322	2695	1275	51	103	50	23275	23375	1275	1323	1325	50
60	1887	3835	1830	61	123	60	39530	39650	1830	1888	1890	60

Figure 1: Coefficient operations in computing the exponential

```
          DE1            DE2            DE3             DE4              DE5

f(a)=1 ------------------------------------------------------------------------
  5:      30/   10        23/   14        28/   22         15/   15         15/     5
 10:     110/   20        86/   29        80/   47        155/  110         55/    10
 15:     240/   30       183/   44       162/   72       1140/  255        120/    15
 20:     420/   40       321/   59       291/   97       2850/  400        210/    20
 25:     650/   50       493/   74       438/  122       5285/  545        325/    25
 30:     930/   60       706/   89       615/  147       8445/  690        465/    30
100:     .../  200       .../  299       .../  497        .../ 2720        .../   100

f(a)=1+a+a**2 -----------------------------------------------------------------
  5:      46/   17        29/   19        29/   23         15/   15         50/    20
 10:     191/   37       122/   44        94/   58        155/  110        275/    65
 15:     436/   57       274/   69       201/   93       1390/  345        800/   135
 20:     781/   77       492/   94       367/  128       3850/  590       1750/   230
 25:    1226/   97       769/  119       564/  163       7535/  835       3250/   350
 30:    1771/  117      1112/  144       804/  198      12445/ 1080       5425/   495
100:     .../  397       .../  494       .../  688        .../ 4510        .../  5150

f(a)=sin a --------------------------------------------------------------------
  5:      50/   20        30/   20        29/   23         15/   15         71/    32
 10:     275/   65       156/   65       104/   68        155/  110        466/   117
 15:     800/  135       435/  135       262/  138       1490/  405       1436/   252
 20:    1750/  230       936/  230       551/  233       5050/  950       3231/   437
 25:    3250/  350      1715/  350       974/  353      12085/ 1745       6101/   672
 30:    5425/  495      2841/  495      1574/  498      23845/ 2790      10296/   957
100:     .../ 5150       .../ 5150       .../ 5153        .../43670        .../ 10197
```

`nnn/fff` compares the number of multiplications for naive vs. fixed point
`ttt:` number of terms
DE1-DE4 solved using `lde`
DE5 solved using `ode1`

Figure 2: Coef operations in the method of undetermined coefficients

```
++ This package provides power series solutions
++ to regular linear or non-linear ordinary
++ differential equations of arbitrary order.

PowerSeriesODESolver(z: Expression, K: Field): Interface == Implementation where
    UPS ==> UnivariatePowerSeries(z, K)
    L   ==> List

    Interface ==> with
        ode1: ((UPS->UPS), K) -> UPS
           ++ ode1(f,c) is the solution to
           ++   y'=f(y) such that y(0)=c

        ode2: ((UPS,UPS) -> UPS, K,K) -> UPS
           ++ ode2(f,c0,c1) is the solution to
           ++   y''=f(y,y')
           ++ such that
           ++   y(0) = c0, y'(0) = c1

        ode:  (L UPS -> UPS, L K) -> UPS
           ++ ode(f, cl) is the solution to
           ++   y<n>=f(y,y',..,y<n-1>)
           ++ such that
           ++   y<i>(0) = cl.i, for i in 0..n-1

    Implementation ==> add
        ode1(f,c) ==
            fixedPoint(y +-> integrate(c, f y))

        ode2(f, c0, c1) ==
            fixedPoint(y +-> integrate(c0, integrate(c1, f(y, pderiv y)))

        -- Compute [y,y',..,y<n>] = [int(y'),..,int(y<n>),f(y,..,y<n-1>)]
        odeNre(f: L UPS->UPS, cl: L K, yl: L UPS): L UPS ==
            yis := [integrate(c, y) for c in cl for y in rest yl]
            append(yis, [f yis])

        ode(f, cl) ==
            fixedPoint(yl +-> odeNre(f,cl,yl), #cl+1).0
```

Figure 3: Power Series ODE Solver: Implementation

```
-- Problem:  Solve  y''' = sin(y'') exp(y) + cos(x)
--           subject to y(0)=1, y'(0)=0, y''(0)=0.

-- Allow the series to have elementary function coefs.
ups := UPS('x, EF I);

-- Define f(y) = y'''.
f(C: List ups): ups == sin(C.2)*exp(C.0)+cos x;

-- Use ode with the appropriate boundary conditions.
y := ode(f, [1, 0, 0])$PSODE('x, EF I)

                          2            3
         1  3   %e  4   %e  - 1  5   %e  - 2%e  6
(3)  1 + - x  + -- x  + ------- x  + --------- x
         6      24        120            720

         4      2
       %e  - 8%e  + 4%e + 1  7      8
     + -------------------- x  + O(x )
               5040

-- By default, only the first few terms are shown.

-- Test the solution.
yp:=pderiv y; ypp:=pderiv yp; yppp:=pderiv ypp;

yppp - f [y, yp, ypp]

        8
(5)  O(x )
```

Figure 4: PowerSeries ODE Solver: Example