J. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom

FUNCTIONAL PEARLS
Power Series, Power Serious

M. Douglas McIlroy
Dartmouth College, Hanover, New Hampshire 03755∗
doug@cs.dartmouth.edu

Abstract

Power series and stream processing were made for each other. Stream algorithms for power series are short, sweet, and compositional. Their neatness shines through in Haskell, thanks to pattern-matching, lazy lists, and operator overloading. In a short compass one can build working code from ground zero (scalar operations) up to exact calculation of generating functions and solutions of differential equations.

I opened the serious here and beat them easy.
— Ring Lardner, *You know me Al*

1 Introduction

Pitching baseballs for the White Sox, Ring Lardner's unlettered hero, Jack Keefe, mastered the Cubs in the opening game of the Chicago series. Pitching heads and tails, I intend here to master power series by opening them one term at a time.

A power series, like that for $\cos x$,

$$1 - x^2/2! + x^4/4! - x^6/6! + \cdots,$$

is characterized by an infinite sequence of coefficients, in this case 1, 0, $-1/2$, 0, $1/24$, 0, $-1/720$, ... It is ideal for implementing as a data stream, a source of elements (the coefficients of the series) that can be obtained one at a time in order. And data streams are at home in Haskell, realized as lazy lists.

List-processing style–treat the head and recur on the tail–fits the mathematics of power series very well. While list-processing style benefits the math, operator overloading carries the clarity of the math over into programs. A glance at the collected code in the appendix will confirm the tidiness of the approach. One-liners, or nearly so, define the usual arithmetic operations, functional composition, functional inversion, integration, differentiation, and the generation of some Taylor series. With the mechanism in place, we shall consider some easily specified, yet stressful, tests of the implementation and an elegant application to generating functions.

∗ This paper was begun at Bell Laboratories, Murray Hill, NJ 07974.

1.1 Conventions

In the stream approach a power series F in variable x,

$$F(x) = f_0 + xf_1 + x^2 f_2 + \cdots,$$

is considered as consisting of head terms, $x^i f_i$, plus a tail power series, F_n, multiplied by an appropriate power of x:

$$\begin{aligned} F(x) &= F_0(x) \\ &= f_0 + xF_1(x) \\ &= f_0 + x(f_1 + xF_2(x)) \end{aligned}$$

and so on. When the dummy variable is literally x, we may use F as an abbreviation for $F(x)$.

The head/tail decomposition of power series maps naturally into the head/tail decomposition of lists. The mathematical formula

$$F = f_0 + xF_1$$

transliterates quite directly to Haskell:

```
fs = f0 : f1s
```

(Since names of variables cannot be capitalized in Haskell, we use the popular convention of appending `s` to indicate a sequence variable.)

In practice, the algorithms usually refer explicitly to only one coefficient of each power series involved. Moreover, the Haskell formulations usually refer to only one tail (including the 0-tail). Then we may dispense with the subscripts, since they no longer serve a distinguishing purpose. With these simplifications, a grimly pedantic rendering of a copy function,

```
copy (f0:f1s) = f0 : copy f1s
```

reduces to everyday Haskell:

```
copy (f:fs) = f : copy fs
```

For definiteness, we may think of the head term as a rational number. But thanks to polymorphism the programs that follow work on other number types as well. Series are treated formally; convergence is not an issue. However, when series do converge, the expected analytic relations hold. The programs give exact answers: any output will be expressed correctly in unbounded-precision rationals whenever the input is so expressed.

1.2 Overloading

While the methods of this paper work in any language that supports data streams, they gain clarity when expressed with overloaded operators. To set up overloading, we need some peculiar Haskell syntax that shows up as `instance` clauses scattered through the code. If the following brief explanation doesn't enlighten, you may safely ignore the `instance` clauses. Like picture frames, they are necessary to support a work of art, but are irrelevant to its enjoyment.

A data type in Haskell may be declared to be an instance of one or more type classes. Each type class is equipped with functions and operators that have consistent signatures across every type in the class. Among several standard type classes, the most important for our purposes are `Num` and `Fractional`. Class `Num` has operators suitable for the integers or other mathematical rings: negation, addition, subtraction, multiplication and nonnegative integer power. Class `Fractional` has further operations suitable to rationals and other mathematical fields: reciprocal and division. Of the other operations in these classes (for printing, comparison, etc.) only one will concern us, namely `fromInteger`, a type-conversion function discussed in Section 2.4.

To make the arithmetic operations of class `Num` applicable to power series, we must declare power series (i.e. lists) to be an instance of class `Num`. Arithmetic must already be defined on the list elements. The code looks like

```
instance Num a => Num [a] where
    negate (f:fs) = (negate f) : (negate fs)
    -- and definitions of other operations
```

The part before `where` may be read, 'If type `a` is in class `Num`, then lists of type-`a` elements are in class `Num`.' After `where` come definitions for the class-`Num` operators pertinent to such lists. The function `negate` and others will be described below; the full set is gathered in the appendix. The types of the functions are all instances of type schemas that have been given once for the class. In particular `negate` is predeclared to be a function from some type in class `Num` to the same type.

1.3 Numeric constants

There is one more bit of Haskell-speak to consider before we address arithmetic. Because we are interested in exact series, we wish to resolve the inherently ambiguous type of numeric constants in favor of multiple-precision integers and rationals. To do so, we override Haskell's rule for interpreting constants, namely

```
default (Int, Double)
```

and replace it with

```
default (Integer, Rational, Double)
```

Now integer constants in expressions will be interpreted as the first acceptable type in this default list. We choose to convert constants to `Integer` (unbounded precision) rather than `Int` (machine precision) to avoid overflow. In `Fractional` context constants become `Rational`s, whose precision is also unbounded. Thus the numerator of `1/f` will be taken to be `Rational`.

2 Arithmetic

2.1 Additive operations

We have seen the definition of the simplest operation, negation:

```
negate (f:fs) = (negate f) : (negate fs)
```

The argument pattern `(f:fs)` shows that `negate` is being defined on lists, and supplies names for the head and tail parts. The right side defines power-series negation in terms of scalar negation (`negate f`), which is predefined, and recurrence on the tail (`negate fs`). The definition depends crucially on lazy evaluation. While defined recursively, `negate` runs effectively by induction on prefixes of the infinite answer. Starting from an empty output it builds an ever bigger initial segment of that answer.

Having defined `negate`, we can largely forget the word and instead use unary `-`, which Haskell treats as syntactic sugar for `negate`.

Addition is equally easy. The mathematical specification,

$$F + G = (f + xF_1) + (g + xG_1) = (f + g) + x(F_1 + G_1),$$

becomes

```
(f:fs) + (g:gs) = f+g : fs+gs
```

2.2 Multiplication

Here the virtue of the stream approach becomes vivid. First we address multiplication by a scalar, using a new operator. The left-associative infix operator `(.*)` has the same precedence as multiplication:

```
infixl 7 .*                      -- same precedence as *
(.*):: Num a => a->[a]->[a]      -- type declaration for .*
c .* (f:fs) = c*f : c.*fs        -- definition of .*
```

The parentheses around `.*` in the type declaration allow it to be used as a free-standing identifier. The declaration says that `(.*)` is a function of two arguments, one a value of some numeric type `a` and the other a list whose elements have that type. The result is a list of the same type.

From the general multiplication formula,

$$F \times G = (f + xF_1) \times (g + xG_1) = fg + x(fG_1 + F_1 \times G),$$

we obtain this code:

```
(f:fs) * (g:gs) = f*g : (f.*gs + fs*(g:gs))
```

The cleanness of the stream formulation is now apparent. Gone is all the finicky indexing of the usual convolution formula,

$$(\sum_{i=0}^{\infty} f_i x^i)(\sum_{i=0}^{\infty} g_i x^i) = \sum_{i=0}^{\infty} x^i \sum_{j=0}^{j=i} f_j g_{i-j}.$$

The complexity is hidden in an unseen tangle of streams. Gone, too, is overt concern with storage allocation. The convolution formula shows that, although we may receive terms one at a time, n terms of each series must be kept at hand in order to compute the nth term of their product. With lazy lists this needed information is retained automatically behind the scenes.

2.3 Division

The quotient, Q, of power series F and G satisfies

$$F = Q \times G.$$

Expanding F, Q, and one instance of G gives

$$\begin{aligned} f + xF_1 &= (q + xQ_1) \times G = qG + xQ_1 \times G = q(g + xG_1) + xQ_1 \times G \\ &= qg + x(qG_1 + Q_1 \times G). \end{aligned}$$

Whence

$$\begin{aligned} q &= f/g, \\ Q_1 &= (F_1 - qG_1)/G. \end{aligned}$$

(We have rediscovered long division.) When $g = 0$, the division can succeed only if also $f = 0$. Then $Q = F_1/G_1$. The code is

```
(0:fs) / (0:gs) = fs/gs
(f:fs) / (g:gs) = let q = f/g in
    q : (fs - q.*gs)/(g:gs)
```

2.4 'Constant' series and promotion of constants

The code below defines two trivial, but useful, series. These 'constant' series are polymorphic, because literal constants like 0 and 1 can act as any of several numeric types.

```
ps0, x:: Num a => [a]          -- type declaration
ps0 = 0 : ps0                  -- power series 0
x = 0 : 1 : ps0                -- power series x
```

As a program, `ps0` is nonterminating; no matter how much of the series has been produced, there is always more. An invocation of `ps0`, as in `x`, cannot be interpreted as a customary function call that returns a complete value. Stream processing or lazy evaluation is a necessity.†

To allow the mixing of numeric constants with power series in expressions like $2F$, we arrange for scalars to be coerced to power series as needed. To do so, we supply a new meaning for `fromInteger`, a class-`Num` function that converts multiprecision

† The necessity is not always recognized in the world at large. The MS-DOS imitation of Unix pipelines has function-call rather than stream semantics. As a result, a pipeline of processes in DOS is useless for interactive computing, since no output can issue from the back end until the front end has read all its input and finished.

`Integer`s to the type of the current instance. For a number c to serve as a power series, it must be converted to the list `[c, 0, 0, 0, ... ]`:

```
instance Num a => Num [a] where
    -- definitions of other operations
    fromInteger c = fromInteger c : ps0
```

A new `fromInteger` on the left, which converts an `Integer` to a list of type-`a` elements, is defined in terms of an old `fromInteger` on the right, which converts an `Integer` to a value of type `a`. This is the only place we need to use the name `fromInteger`; it is invoked automatically when conversions are needed.

2.5 Polynomials and rational functions

Subtraction and nonnegative integer powers come for free in Haskell, having been predefined polymorphically in terms of negation, addition and multiplication. Thus we now have enough mechanism to evaluate arbitrary polynomial expressions as power series. For example, the Haskell expression `(1-2*x^2)^3` evaluates to

```
[1, 0, -6, 0, 12, 0, -8, 0, 0, 0, ... ]
```

Rational functions work, too: `1/(1-x)` evaluates to (the rational equivalent of)

```
[1, 1, 1, ... ]
```

This represents a power series, $1 + x + x^2 + x^3 + \cdots$, that sums to $1/(1-x)$ in its region of convergence. Another example, `1/(1-x)^2`, evaluates to

```
[1, 2, 3, ... ]
```

as it should, since

$$\frac{1}{(1-x)^2} = \frac{d}{dx}\frac{1}{1-x} = \frac{d}{dx}(1 + x + x^2 + x^3 + \ldots) = 1 + 2x + 3x^2 + \ldots$$

3 Functional composition

Formally carrying out the composition of power series F and G (or equivalently the substitution of G for x in $F(x)$), we find

$$F(G) = f + G \times F_1(G) = f + (g + xG_1) \times F_1(G) = (f + gF_1(G)) + xG_1 \times F_1(G).$$

This recipe is not implementable in general. The head term of the composition depends, via the term $gF_1(G)$, on all of F; it is an infinite sum. We can proceed, however, in the special case where $g = 0$:

$$F(G) = f + xG_1 \times F_1(G).$$

The code, which neatly expresses the condition $g = 0$, is

```
compose (f:fs) (0:gs) = f : gs*(compose fs (0:gs))
```

(We can't use Haskell's standard function-composition operator because we have represented power series as lists, not functions.)

3.1 Reversion

The problem of finding the functional inverse of a power series is called 'reversion'. There is considerable literature about it; Knuth (1969) devotes four pages to the subject. Head-tail decomposition, however, leads quickly to a working algorithm. Given power series F, we seek R that satisfies

$$F(R(x)) = x.$$

Expanding F, and then one occurrence of R, we find

$$F(R(x)) = f + R \times F_1(R) = f + (r + xR_1) \times F_1(R) = x.$$

As we saw above, we must take $r = 0$ for the composition $F_1(R)$ to be implementable, so

$$f + xR_1 \times F_1(R) = x.$$

Hence f must also be 0, and we have

$$R_1 = 1/F_1(R).$$

Here R_1 is defined implicitly: it appears on the right side hidden in R. Yet the formula suffices to calculate R_1, for the n-th term of R_1 depends on only the first n terms of R, which contain only the first $n-1$ terms of R_1. The code is

```
revert (0:fs) = rs where
    rs = 0 : 1/(compose fs rs)
```

Reversion illustrates an important technique in stream processing: feedback. The output `rs` formally enters into the computation of `rs`, but without infinite regress, because each output term depends only on terms that have already been calculated. Feedback is a leitmotif of Section 4.1.

4 Calculus

Since $\frac{d}{dx}x^n = nx^{n-1}$, the derivative of a power series term depends on the index of the term. Thus, in computing the derivative we use an auxiliary function to keep track of the index:

```
deriv (f:fs) = (deriv1 fs 1) where
    deriv1 (g:gs) n = n*g : (deriv1 gs (n+1))
```

The definite integral, $\int_0^x F(t)dt$, can be computed similarly:

```
integral fs = 0 : (int1 fs 1) where
    int1 (g:gs) n = g/n : (int1 gs (n+1))
```

4.1 Elementary functions via differential equations

With integration and feedback we can find power-series solutions of differential equations in the manner of Picard's method of successive approximations (Pontryagin 1962). The technique may be illustrated by the exponential function, $\exp(x)$, which satisfies the differential equation

$$\frac{dy}{dx} = y$$

with initial condition $y(0) = 1$. Integrating gives

$$y = 1 + \int_0^x y(t)dt.$$

The corresponding code is

```
expx = 1 + (integral expx)
```

Evaluating `expx` gives

```
[1%1, 1%1, 1%2, 1%6, 1%24, 1%120, 1%720, ... ]
```

where `%` constructs fractions from integers. Notice that `expx` is a 'constant' series like `ps0`, not a function like `negate`. We can't call it `exp`, because Haskell normally declares `exp` to be a function.

In the same way, we can compute sine and cosine series. From the formulas

$$\begin{aligned} \tfrac{d}{dx}\sin x &= \cos x, & \sin(0) &= 0, \\ \tfrac{d}{dx}\cos x &= -\sin x, & \cos(0) &= 1, \end{aligned}$$

follows remarkable code:

```
sinx = integral cosx
cosx = 1 - (integral sinx)
```

Despite its incestuous look, the code works. The mutual recursion can get going because `integral` produces a zero term before it accesses its argument.

The square root may also be found by integration. If $Q^2 = F$, then

$$2Q\frac{dQ}{dx} = F'$$

or

$$\frac{dQ}{dx} = \frac{F'}{2Q},$$

where $F' = dF/dx$. When the head term f is nonzero, the head term of the square root is $f^{1/2}$. To avoid irrationals we take $f = 1$ and integrate to get

$$Q = 1 + \int_0^x \frac{F'(t)dt}{2Q(t)}.$$

If the first two coefficients of F vanish, i.e. if $F = x^2F_2$, then $Q = xF_2^{1/2}$. In other cases we decline to calculate the square root, though when f is the square of a rational we could do so for a little more work. The corresponding program is

```
sqrt (0:0:fs) = 0 : (sqrt fs)
sqrt (1:fs) = qs where
    qs = 1 + integral((deriv (1:fs))/(2.*qs))
```

Haskell normally places `sqrt` in type class `Floating`; the collected code in the appendix complies. Nevertheless, when the square root of a series with rational coefficients can be computed, the result will have rational coefficients.

5 Testing

The foregoing code is unusually easy to test, thanks to a ready supply of relations among analytic functions and their Taylor series. For example, checking many terms of `sinx` against `sqrt(1-cosx^2)` exercises most of the arithmetic and calculus functions. Checking $\tan x$, computed as $\sin x/\cos x$, against the functional inverse of $\arctan x$, computed as $\int dx/(1+x^2)$, further brings in composition and reversion. The checks can be carried out to 30 terms in a few seconds. The expressions below do so, using the standard Haskell function `take`. Each should produce a list of 30 zeros.

```
take 30 (sinx - sqrt(1-cosx^2))
take 30 (sinx/cosx - revert(integral(1/(1+x^2))))
```

6 Generating functions

A generating function S for a sequence of numbers, s_n, is

$$S = \sum_n x^n s_n.$$

When the s_n have suitable recursive definitions, the generating function satisfies related recursive equations (Burge, 1975). Running these equations as stream algorithms, we can directly enumerate the values of s_n. This lends concreteness to the term 'generating function': when run as a program, a generating function literally generates its sequence.

We illustrate with two familiar examples, binary trees and ordered trees.

Binary trees In the generating function T for enumerating binary trees, the coefficient of x^n is the number of trees on n nodes. A binary tree is either empty or has one root node and two binary subtrees. There is one tree with zero nodes, so the head term of T is 1. A tree of $n+1$ nodes has two subtrees with n nodes total; if one of them has i nodes, the other has $n-i$. Convolution! Convolution of T with itself is squaring, so T^2 is the generating function for the counts of n-node pairs of trees. To associate these counts with $n+1$-node trees, we multiply by x. Hence

$$T = 1 + xT^2$$

The Haskell equivalent is

```
ts = 1 : ts^2
```

(The appealing code `ts = 1 + x*ts^2` won't work. Why not? How does it differ from `expx = 1 + (integral expx)`?) Evaluating `ts` yields the Catalan numbers, as it should (Knuth 1968):

```
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ... ]
```

Ordered trees Consider next the generating function for nonempty ordered trees on n nodes. An $n+1$-node tree, for $n >= 0$, is made of a root and an n-node forest. An n-node forest is a list of trees whose sizes sum to n. A list is empty or an $n+1$-item list made of a head item and an n-item tail list. From these definitions follow relations among generating functions:

$$\begin{aligned}\mathbf{tree}(x) &= x\mathbf{forest}(x)\\ \mathbf{forest}(x) &= \mathbf{list}(\mathbf{tree}(x))\\ \mathbf{list}(x) &= 1 + x\mathbf{list}(x)\end{aligned}$$

The first and third relations are justified as before. To derive the second relation, observe that the coefficient of x^k in $\mathbf{tree^n}$ tells how many n-component forests there are with k nodes. Summing over all n tells how many k-node forests there are. But $\mathbf{list}(\mathbf{tree})$, which is $1+\mathbf{tree}+\mathbf{tree^2}+\ldots$, does exactly that summing. Composition of generating functions reflects composition of data structures.

The code

```
tree = 0 : forest
forest = compose list tree
list = 1 : list
```

yields this value for **tree**:

```
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ... ]
```

Catalan numbers again! The apparent identity between the number of binary trees on n nodes and the number of nonempty ordered trees on $n+1$ nodes is real (Knuth 1968): a little algebra confirms that $\mathbf{tree} = xT$.

7 Final remarks

Stream processing can be beaten asymptotically if the goal is to find a given number of coefficients of a given series (Knuth 1969). In particular, multiplication involves convolution, which can be done faster by FFT. Nevertheless, stream processing affords the cleanest way to manipulate power series. It has the advantage of incrementality–one can decide on the fly when to stop. And it is compositional.

While a single reversion or multiplication is not too hard to program in a standard language, the composition of such operations is a daunting task. Even deciding how much to compute is nontrivial. How many terms are required in each intermediate result in order to obtain, say, the first 10 nonzero coefficients of the final answer? When can storage occupied by intermediate terms be safely reused? Such questions don't arise in the lazy-stream approach. To get 10 terms, simply compute until 10 terms appear. No calculations are wasted along the way, and intermediate values neither hang around too long nor get discarded too soon.

In Haskell, the code for power-series primitives has a familiar mathematical look, and so do expressions in the primitives. Only one language feature blemishes the code as compared to the algebraic formulation of the algorithms. Type-class constraints that allow only limited overloading compelled us to invent a weird new operator `(.*)` and and to use nonstandard names like `expx` for standard series.

In the interest of brevity, I have stuck with a bare-list model of power series. However, the simple identification of power series with lists is a questionable programming practice. It would be wiser to make power series a distinct type. To preserve the readability of the bare-list model, we may define a power-series type, `Ps`, with an infix constructor `(:+:)` reminiscent both of the list constructor `(:)` and of addition in the head/tail decomposition $F = f + F_1$. At the same time we may introduce a special constructor, `Pz`, for power series zero. Use of the zero constructor instead of the infinite series `ps0` forestalls much bootless computation. Polynomial operations become finite. Multiplication by a promoted constant becomes linear rather than quadratic in the number of output terms.

The data-type declaration for this realization of power series is

```
infixr 5 :+:                  -- precedence like :
data Num a => Ps a = Pz | a :+: Ps a
```

Some definitions must be added or modified to deal with the zero constructor, for example

```
instance Num a => Num Ps a where
    Pz + fs = fs
    fromInteger c = fromInteger c :+: Pz
```

Definitions for other standard operations, such as printing and equality comparison, which were predefined for the list representation, must be given as well. Working code is deposited with the abstract of this paper at the journal's web site, `http://www.dcs.gla.ac.uk/jfp`.

The application of streams to power series calculations is a worthy addition to our stock of canonical programming examples. It makes a good benchmark for stream processing–simple to program and test, complicated in the actual running. Pedagogically, it well illustrates the intellectual clarity that streams can bring to software design. Above all, the method is powerful in its own right; it deserves to be taken serious.

8 Sources

Kahn used a stream-processing system (Kahn and MacQueen 1977) for power-series algorithms like those given here; the work was not published. Abelson and Sussman (1985) gave examples in Scheme. McIlroy (1990) covered most of the ground in a less perspicuous stream-processing language. Hehner (1993) demonstrated the technique in a formal setting. Burge (1975) gave structural derivations for generating functions, including the examples given here, but did not conceive of them as executable code. Knuth (1969) and McIlroy (1990) gave operation counts. Karczmarczuk (1997) showed applications in analysis ranging from Padé approximations to Feynman diagrams.

I am grateful to Phil Wadler for much generous and wise advice, and to Jon Bentley for critical reading.

References

Abelson, H. and Sussman, G. J. 1976. *The Structure and Interpretation of Computer Programs.* MIT Press.

Burge, W. H. 1975. *Recursive Programming Techniques.* Addison-Wesley

Hehner, E. C. R. 1993. *A Practical Theory of Programming.* Springer-Verlag.

Kahn, G. and MacQueen, D. B. 1977. Coroutines and networks of parallel processes, in Gilchrist, B. (Ed.), *Information Processing 77*, 993–998. North Holland. Volume 1, 2.3.4.4. Addison-Wesley.

Karczmarczuk, J. 1997. Generating power of lazy semantics. *Theoretical Computer Science*, 187: 203–219.

Knuth, D. E. 1968. *The Art of Computer Programming*, Volume 1, 2.3.4.4. Addison-Wesley.

Knuth, D. E. 1969. *The Art of Computer Programming*, Volume 2. Addison-Wesley.

McIlroy, M. D. 1990. Squinting at power series. *Software–Practice and Experience*, 20: 661–683.

Pontryagin, L. S. 1962. *Ordinary Differential equations.* Addison-Wesley.

A Collected code

Source code is deposited with the abstract of this paper at `http://www.dcs.gla.ac.uk/jfp`.

```
import Ratio
infixl 7 .*
default (Integer, Rational, Double)

    -- constant series
ps0, x:: Num a => [a]
ps0 = 0 : ps0
x = 0 : 1 : ps0

    -- arithmetic
(.*):: Num a => a->[a]->[a]
c .* (f:fs) = c*f : c.*fs

instance Num a => Num [a] where
    negate (f:fs) = (negate f) : (negate fs)
    (f:fs) + (g:gs) = f+g : fs+gs
    (f:fs) * (g:gs) = f*g : (f.*gs + fs*(g:gs))
    fromInteger c = fromInteger c : ps0

instance Fractional a => Fractional [a] where
    recip fs = 1/fs
    (0:fs) / (0:gs) = fs/gs
    (f:fs) / (g:gs) = let q = f/g in
        q : (fs - q.*gs)/(g:gs)

    -- functional composition
```

```
compose:: Num a => [a]->[a]->[a]
compose (f:fs) (0:gs) = f : gs*(compose fs (0:gs))

revert::Fractional a => [a]->[a]
revert (0:fs) = rs where
    rs = 0 : 1/(compose fs rs)

    -- calculus
deriv:: Num a => [a]->[a]
deriv (f:fs) = (deriv1 fs 1) where
    deriv1 (g:gs) n = n*g : (deriv1 gs (n+1))

integral:: Fractional a => [a]->[a]
integral fs = 0 : (int1 fs 1) where
    int1 (g:gs) n = g/n : (int1 gs (n+1))

expx, cosx, sinx:: Fractional a => [a]
expx = 1 + (integral expx)
sinx = integral cosx
cosx = 1 - (integral sinx)

instance Fractional a => Floating [a] where
    sqrt (0:0:fs) = 0 : sqrt fs
    sqrt (1:fs) = qs where
        qs = 1 + integral((deriv (1:fs))/(2.*qs))

    -- tests
test1 = sinx - sqrt(1-cosx^2)
test2 = sinx/cosx - revert(integral(1/(1+x^2)))
iszero n fs = (take n fs) == (take n ps0)
main = (iszero 30 test1) && (iszero 30 test2)
```