Elaborating Karczmarczuk's Program: Calculating Generating Functons in Haskell

V. E. McHale

March 19, 2026

Abstract

Karczmarczuk lays out the elegance of lazy lists for infinite power series, including generating functions in 1997; McIlroy gives a presentation in Haskell 1999. We offer further examples and comment on the abilities and logical aspects of laziness vis-à-vis generating functions.

Contents

1 Prelude

Begin with McIlroy's prelude for power series addition, multiplication, division; representing power series as lazy lists [4]:

```
module Gf where

import Data.Ratio ((%), numerator)
import Data.List (genericReplicate)

infixl 7 *.
(*.) :: Num a => a -> [a] -> [a]
x *. (p : ps) = x * p : x *. ps

instance Num a => Num [a] where
  negate = map negate
  (+) = zipWith (+)
  (*) (p : ps) (q : qs) = p * q : (p *. qs + ps * (q : qs))
  fromInteger n = fromInteger n : repeat 0

instance (Eq a, Fractional a) => Fractional [a] where
  (/) (0 : ps) (0 : qs) = ps / qs
  (/) (p : ps) (q : qs) = let r = p / q in r : (ps - r *. qs) / (q : qs)
  fromRational q = fromRational q : repeat 0
```

Note that the recursive definitions of multiplication and division use sharing; the formulation with lazy lists would be less adroit with streams in, say, Rust.

Compare the by-index reciprocal power series $1/f = \sum_{n=0}^{\infty} b_n x^n$ of $f = \sum_{n=0}^{\infty} a_n x^n$ [8, p. 33]

$$b_n = -\frac{1}{a_0} \sum_{k=1}^{\infty} a_k b_{n-k}.$$

2 Ordinary Power Series

2.1 Fibonacci Sequence

The generating function for the Fibonacci numbers is $\sum_{n=0}^{\infty} F_n x^n = \frac{1}{1-x-x^2}$ [8, p. 10]. We can represent $1 - x - x^2$ as a power series using the Haskell prelude with `1 : (-1) : (-1) : repeat 0`, so the generating function is:

```
fibs :: [Integer]
fibs = map numerator
  (1 / (1 : (-1) : (-1) : repeat 0))
```

We can verify that this is correct immediately:

```
ghci> take 15 fibs
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
```

2.2 Stirling Numbers of the Second Kind

For fixed k, the Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$ are generated by [8, p. 21]

$$\sum_{n=0}^{\infty} \left\{ {n \atop k} \right\} x^n = \frac{x^k}{(1-x)(1-2x)\cdots(1-kx)}$$

In our scheme, x^k can be expressed as `replicate k 0 ++ 1 : repeat 0`.

```
stirling2 :: Integer → [Integer]
stirling2 k = map numerator
  ((genericReplicate k 0 ++ 1 : repeat 0) / product [1 : (-fromInteger l) : repeat 0 | l ← [1..k]])
```

`product` here is operating on (infinite!) power series by our `Num` instance.

```
ghci> take 11 (stirling2 3)
[0,0,0,1,6,25,90,301,966,3025,9330]
```

2.3 Catalan Numbers

The generating function for the Catalan numbers satisfies the functional equation [8, pp. 46-47]

$$F(x) = 1 + xF(x)^2$$

which is straightforwardly translated to Haskell, viz. [4]

```
catalan :: [Integer]
catalan = 1 : catalan ↑ 2
```

```
ghci> take 11 catalan
[1,1,2,5,14,42,132,429,1430,4862,16796]
```

2.4 Harmonic Numbers

The harmonic numbers $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ are generated by [8, p. 40]

$$\sum_{n=0}^{\infty} H_n x^n = \frac{1}{1-x} \left(\log \frac{1}{1-x} \right)$$

Recall [8, p. 55]:

$$\sum_{n=0}^{\infty} \frac{x^n}{n} = \log \frac{1}{1-x}$$

```
ilog1 :: [Rational]
ilog1 = 0 : [1 % n | n ← [1..]]
```

```
harmonic :: [Rational]
harmonic = ilog1 / (1 : (−1) : repeat 0)
```

```
ghci> take 7 harmonic
[0 % 1,1 % 1,3 % 2,11 % 6,25 % 12,137 % 60,49 % 20]
```

2.5 Eulerian Numbers

Fixing n, $\left\langle {n \atop k} \right\rangle$ is given by the coefficient of x^{k+1} in [7]

$$\frac{(1-x)^{n+1} \sum_{k=1}^{\infty} k^n r^k}{x}$$

This is a bit clumsy in Haskell but the reader can verify that this translates to the below in our scheme.

```
eulerian :: Integer → [Integer]
eulerian n = map numerator
    (0 : ((1 : (−1) : repeat 0) ↑ (n + 1) ∗ lin / (0 : 1 : repeat 0)))
  where
    lin :: [Rational]
    lin = 0 : [fromInteger k ↑ n | k ← [1 . .]]
```

```
ghci> take 11 (eulerian 8)
[0,1,247,4293,15619,15619,4293,247,1,0,0]
```

3 Exponential Power Series

Frequently one encounters situations in which the sequence of interest $\{a_n\}_{n=0}^{\infty}$ is the coefficient of $\frac{x^n}{n!}$ in an exponential power series

$$\sum_{n=0}^{\infty} \frac{a_n}{n!} x^n$$

To handle such situations, we define

```
expseq :: [Rational] → [Rational]
expseq ps = zipWith (λp q → p ∗ fromInteger q) ps (scanl (∗) 1 [1 . .])
```

which extracts the sequence $\{a_n\}_{n=0}^{\infty}$ from an exponential generating function.

`scanl (*) 1 [1..]` is the sequence of factorials.

3.1 Bernoulli Numbers

The Bernoulli numbers are generated by [6]

$$\frac{x}{e^x - 1}$$

In Haskell:

```
exp1 :: [Rational]
exp1 = 0 : map (1%) factorials
  where
    factorials = scanl (*) 1 [2..]

bernoulli :: [Rational]
bernoulli = expseq ebern
  where
    ebern = (0 : 1 : repeat 0) / exp1
```

and we immediately have a result with little exertion on our part:

```
ghci> take 11 bernoulli
[1 % 1,(-1) % 2,1 % 6,0 % 1,(-1) % 30,0 % 1,1 % 42,0 % 1,(-1) % 30,0 % 1,5 % 66]
```

3.2 Counting Derangements

The subfactorial $!n$ is the number of permutations of length n such that none of the elements appear in their original place (such a permutation is called a *derangement*). They are generated by [8, p. 47]

$$\sum_{n=0}^{\infty} \frac{!n x^n}{n!} = \frac{1}{e^x(1-x)}$$

```
derangements :: [Integer]
derangements = map numerator
  (expseq (invexp / (1 : (-1) : repeat 0)))

invexp :: [Rational]
invexp = zipWith (%) (cycle [1,-1]) factorials
  where
    factorials = scanl (*) 1 [1..]
```

Note the use of `zipWith ... cycle [1,-1]` to get alternating signs for coefficients, typically presented as $(-1)^n$ in $\sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}\ldots$ the Haskell code has a cute twist.

```
ghci> take 12 derangements
[1,0,1,2,9,44,265,1854,14833,133496,1334961,14684570]
```

3.3 Stirling Numbers of the First Kind

For fixed k, the Stirling numbers of the first kind $\left[{n \atop k}\right]$ are generated by [8, pp. 87-88]

$$\frac{1}{k!}\left[\log\left(\frac{1}{1-x}\right)\right]^k$$

```
stirling1 :: Integer → [Integer]
stirling1 k = [numerator a | a ← expseq (estirling1 k)]
  where
    estirling1 k = (1 % fact k) *. ilog1 ↑ k
    fact 0 = 1
    fact n = n * fact (n − 1)
```

```
ghci> take 11 (stirling1 2)
[0,0,1,3,11,50,274,1764,13068,109584,1026576]
```

3.4 Bell Numbers

The Bell numbers are given by the exponential generating function [8, pp. 23-24]

$$\sum_n \frac{B_n}{n!}x^n = e^{e^{x-1}}$$

We turn to an infinite version of Hörner's method for a solution to power series substitution in Haskell [1][4]; for $P(x)$, $Q(x)$, the power series $P(Q(x))$ can be computed by

```
infixr 9 |>

(|>) :: (Eq a, Num a) ⇒ [a] → [a] → [a]
(p : ps) |> (0 : qs) = p : qs * (ps |> (0 : qs))
```

provided that the constant term of $Q(x)$ is zero. This is Horner's method.

The power series for e^x is typically expressed by-index as $\sum_{n=0}^{\infty} \frac{x^n}{n!}$ but we invite the reader to mull the equivalent

```
exps :: [Rational]
exps = 1 : zipWith (*) exps [1 % n | n ← [1..]]
```

which expresses the reuse one would employ in calculating $n!$ for each term.

Somewhat miraculously, we immediately have:

```
bell :: [Integer]
bell = map numerator
  (expseq (exps |> exp1))
```

```
ghci> take 11 bell
[1,1,2,5,15,52,203,877,4140,21147,115975]
```

4 Counting Trees

4.1 Structurally Different Oriented Trees

The generating function $A(z) = \sum_{n \geqslant 0} a_n z^n$ for the number of structurally different (i.e. unlabeled) oriented trees satisfies the functional equation [3, p. 386]

$$A(z) = \frac{z}{(1-z)^{a_1}(1-z^2)^{a_2}(1-z^3)^{a_3} + \cdots}$$

Taking logarithms, we arrive at Pólya's [3, p. 395]

$$A(z) = z \cdot \exp\left(A(z) + \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots\right)$$

Following Karczarmaczuk [1], we can express this in our program as:

```
a :: [Rational]
a = (0 : a1) where
  a1 = exps |> s 1 (0 : 1 : repeat 0)
  s k sp = 0 : (1 % k) *. a1 |> sp + s (k + 1) (0 : sp)
```

```
ghci> take 15 (map numerator a)
[0,1,1,2,4,9,20,48,115,286,719,1842,4766,12486,32973]
```

4.2 Structurally Different Free Trees

The generating function for the number of structurally different free trees is [**?**, p. 388]knuth]

$$F(z) = A(z) - \frac{1}{2}A(z)^2 + \frac{1}{2}A(z^2)$$

whence

```
f :: [Integer]
f = map numerator
  (a - (1 % 2) *. a ↑ 2 + (1 % 2) *. a |> (0 : 0 : 1 : repeat 0))
```

```
ghci> take 18 f
[0,1,1,1,2,3,6,11,23,47,106,235,551,1301,3159,7741,19320,48629]
```

5 Commentary

The novelty offered by laziness is that one can defer curtailing the sequence until the final step [1]—the situation is actually not irredeemable in a strict language, but the difference is significant from a logical perspective: laziness' ability to deal with infinite data structures as arguments and return values [5]

means that the desired sequence length `n` need not be supplied on the left. We "pull" computations as dependencies rather than pushing through the desired table size `n`.

Thus laziness gives us a glimpse of something beyond structured programming, in which one proceeds by stepped reductions of multiple arguments to one return value—information can flow "up" in a program [2].

References

[1] Jerzy Karczmarczuk. Generating power of lazy semantics. *Theoretical Computer Science*, 187(1):203–219, 1997.

[2] Jerzy Karczmarczuk. Lazy time reversal , and automatic differentiation. 2002.

[3] Donald E. Knuth. *Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1998.

[4] M. Douglas McIlroy. Power series, power serious. *J. Funct. Program.*, 9(3):325–337, May 1999.

[5] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[6] Eric W. Weisstein. Bernoulli number. `https://mathworld.wolfram.com/BernoulliNumber.html`.

[7] Eric W. Weisstein. Eulerian number. `https://mathworld.wolfram.com/EulerianNumber.html`.

[8] Herbert S. Wilf. *generatingfunctionology*. CRC Press, 3rd edition, 2006.