In Praise of Sequence (Co-)Algebra and its implementation in Haskell

Kieran Clenaghan

Abstract

What is Sequence Algebra? This is a question that any teacher or student of mathematics or computer science can engage with. Sequences are in Calculus, Combinatorics, Statistics and Computation. They are foundational, a step up from number arithmetic. Sequence operations are easy to implement from scratch (in Haskell) and afford a wide variety of testing and experimentation. When bits and pieces of sequence algebra are pulled together from the literature, there emerges a claim for status as a substantial pre-analysis topic. Here we set the stage by bringing together a variety of sequence algebra concepts for the first time in one paper. This provides a novel economical overview, intended to invite a broad mathematical audience to cast an eye over the subject. A complete, yet succinct, basic implementation of sequence operations is presented, ready to play with. The implementation also serves as a benchmark for introducing Haskell by mathematical example.

1 Introduction

Consider these titles: *Formal Power Series* [64], *Power Series, Power Serious* [57], *Elements of Stream Calculus* [71], and *Concrete Stream Calculus* [39]. The mixing of the classical and the modern in these papers is stimulating and suggests a re-telling of the elementary theory and application of sequences. Casting our net wider than the citations in those four papers, brings up a number of corroborating works, including [84, 8, 82, 83], in which the authors call attention to the intrinsic qualities and utility of an elementary calculus or algebra of sequences. We do more than re-advertise this work – we endeavour to tease out the common ground, emphasising economy of statement and notation, whilst embracing variety of approach.

Our aim is to attract those who are less well acquainted with sequence work, or those who are unfamiliar with Haskell or both. It is so that others can enjoy "messing about" (as Hayman [34] might put it) with sequences and their implementation. Elementary sequence algebra provides a good answer to the question, 'What is the smallest coherent chunk of mathematics to set undergraduates to implement, from scratch, so that they get the greatest reward?'.

First we must say that sequence algebra is an umbrella title for algebraic manipulations of finite and infinite sequences, $p = [p_0, p_1, \ldots, p_n]$, $f = [f_0, f_1, \ldots]$, over some given element set, F. It encompasses, prominently, formal power series algebra, but is not restricted to it. A finite sequence, viewed as a sequence of coefficients, can be *expressed* as a formal polynomial. Thus $[0, 1] = x$, or we may say that x is *implemented* by $[0, 1]$.

Similarly, $3x^2 - x + 4 = [4, -1, 3]$. A finite sequence can also be interpreted as an infinite sequence by appending an infinite sequence of zeros. An infinite sequence can be *expressed* as a formal power series:

$$f = \sum_{i=0}^{\infty} f_i x^i = \sum_i f_i x^i = [f_0, f_1, f_2, \ldots]$$

We write f_n or $[x^n]f$ for the element at position n, called the nth term of f; it is the coefficient of x^n in the power series view. The zeroth term can also be written $f(0)$, but otherwise the notation $f(n)$ is reserved for function application: let $p = 3x^2 + 4$, then $p_2 = 3$, but $p(2) = 16$; contrast $p_0 = p(0) = 4$. This example reveals that the symbol p is overloaded, it stands for a function of type $\mathbb{N} \to F$ and also for another of type $F \to F$, and we must be careful to distinguish them.

It seems that in spite of Niven's paper receiving the Lester R Ford award, the algebra of formal power series (FPS) has not entered the standard introductory university texts in any substantial way. Niven's paper has, however, been a key reference for the first chapters of both [36, 31], neither of which is standard undergraduate fare. In general, we find that elements of FPS/sequence algebra appear throughout a vast literature, but the algebra itself is treated minimally, perfunctorily, or is taken for granted, as might be expected in research work [21] and specialist texts [61, 2]. It also comes under *generatingfunctionlogy* [85, 32]. The term "generating function" has proven to be a bit awkward, because of the "function" part. Notice how many times Wilf [85] and others have to remind readers when convergence is not as issue. So, where some might use "generating function", we use the plain "sequence expression". The term "generating function" can be brought in when an analytic function interpretation is intended, as is done in [19, Ch. VII]. However, we freely use the standard *names* of core analytic functions for their Taylor sequences, but rather than say, $\exp(x)$, we use just exp for the sequence.

The extent of material that fits into elementary sequence algebra is perhaps under-appreciated. Our goal is to raise appreciation through a modest "survey" of examples, presented in section 3. Various notations and proof-styles appear in the literature, and an effort has been made to be inclusive and to harmonise. The one-word term "sequence" is often preferred to the three-word "formal power series", but both have their merits, the latter being preferred in the multivariate case, and when formal variable substitution is involved.

The sequence algebra we exhibit is on the same elementary level as Niven's paper [64]. Much detail has to be omitted so that we can cover more examples. Enough detail is included to convey the foundational concreteness of the topic, and omitted detail is in the literature. The Haskell implementation is given in full in section 4 so that the reader can be in no doubt about how succinct it is, and can type it all up, and "own" it. The more mathematically-inclined reader may dwell on section 3, and reflect on the potential of a sequence algebra topic in the mathematical curriculum. The more programming-inclined may dwell on sections 4, 5 and 6, and engage with the proposition that sequence algebra provides an excellent vehicle for exploring learning-mathematics-through-programming, or vice-versa. There is great scope for making a contribution to the consolidation, refinement, and application of sequence algebra as an introductory subject.

To help catch, at a glance, some of the things that come under sequence algebra, we have included a number of tables. Tables 1, 2 and 3 are instantly recognisable as belonging to a calculus text, but here, uncommonly, the objects being related are sequences, not analytic functions. Of course, they herald identities that hold for analytic functions, but in agreement with Niven [64] and Tutte [83], there is something to celebrate in the fact that the identities can be established on very elementary grounds. There is more to celebrate in tables 4 and 5, because many of the sequence expressions therein have a dual interpretation as a set-theoretic structure specification [25]. Yet more satisfaction is to be had from table 7, because the solutions to the defining differential equations for the core sequences transliterate trivially into Haskell definitions [58].

Therefore a grounding in sequence algebra and its implementation surely pays off. This claim is validated at least in the study of the two texts, *Concrete Mathematics* [32] and *Analytic Combinatorics* [25]. There we see numerous examples where sequence algebra is at play, and the implementation can be used for testing, for reinforcement, or just for fun. Some of these appear in the next section. For example, in item **W** we derive the bivariate power series $S(z,u) = \dfrac{\exp\circ uz - 1}{\exp\circ z - 1}$ that appears in [32, sect. 7.6]. This generates a sequence $\breve{S}$ such that $\breve{S}_m$ is a polynomial, and $\breve{S}_m(n) = \sum_{0\le k<n} k^m$. That is, $\breve{S} = [x, -\frac{1}{2}x + \frac{1}{2}x^2, \frac{1}{6}x - \frac{1}{2}x^2 + \frac{1}{3}x^3, \ldots]$. It is striking how accessible the mathematics behind $S(z,u)$ is, and how easy it is to define the infinite $S(z,u)$ and $\breve{S}$ in a program.

2 The basics

Two sequences f and g are equal if they are equal at all indices: $\forall n \ge 0.[x^n]f = [x^n]g$. There are many instances when it is obvious that a statement is subject to universal quantification, and in such cases we leave the $\forall$ part to be inferred by the reader. Becoming fluent with the clumsy-looking *coefficient extraction* operator, $[x^n]$, pays dividends [31, 51]. It obeys the precedence rules, $[x^n]f + g = ([x^n]f) + g$, $[x^n]fg = [x^n](fg)$, and $[x^n]f\circ g = [x^n](f\circ g)$. Moreover, it is a linear operator:

$$[x^n](f+g) = [x^n]f + [x^n]g; \quad [x^n]cf = c[x^n]f \qquad c \text{ is a constant}$$

The generalisation $[u^m z^n]$ will be used to identify a term in a bivariate sequence. Let E be the *tail* or *shift-left* operator: $E[f_0, f_1, f_2, \ldots] = [f_1, f_2, \ldots]$. In formal power series language, $E\,f = \frac{1}{x}(f - f_0)$, and $f = f_0 + xE\,f$; this is referred to as the *head-tail property* (in [71] it is the “fundamental theorem of stream calculus”, see section 3, item **P**). We also have $[x^{n+1}]f = [x^n]E\,f$ and the head-tail expansion rule, $[x^n]f = [x^0]E^n f$. We are free to mix notations – here is the definition of convolution product, $f * g$, in which, typically, the $*$ is suppressed:

$$[x^n]fg = \sum_{k=0}^{n} f_k g_{n-k}$$

Observe that, since $x_1 = 1$ is the only non-zero element of $x = [0,1]$, we have

$$[x^0]xf = 0; \quad [x^n]xf = x_1 f_{n-1} = f_{n-1}, \quad \text{for } n > 0$$

1.	$D\,a$	$=$	0	**constant**
2.	$D\,x$	$=$	1	**variable**
3.	$D\,(f+g)$	$=$	$D\,f + D\,g$	**sum**
4.	$D\,(fg)$	$=$	$(D\,f)g + f(D\,g)$	**product**
5.	$D\,(f^{-1})$	$=$	$(-D\,f)/f^2$	**reciprocal**
6.	$D\,(f/g)$	$=$	$((D\,f)g - f(D\,g))/g^2$	**quotient**
7.	$D\,(f^\circ)$	$=$	$((D\,f)\circ f^\circ)^{-1}$	**converse**
8.	$D\,(f^n)$	$=$	$n(D\,f)f^{n-1}$	**power**
9.	$D\,(a_n x^n)$	$=$	$n a_n x^{n-1}$	**monomial**
10.	$[x^n]D$	$=$	$(n+1)[x^{n+1}]$	**power-series**
11.	$D\,(f\circ g)$	$=$	$((D\,f)\circ g)(D\,g)$	**composition**
12.	$[x^n]f$	$=$	$\frac{1}{n!}[x^0]D^n f$	**Maclaurin**

Table 1: differentiation rules

Thus $x[f_0, f_1, f_2, \ldots] = [0,1] * [f_0, f_1, f_2, \ldots] = [0, f_0, f_1, f_2, \ldots]$, and x can be viewed as a *right-shift* operator. The absorption law, $[x^n]x^m f = [x^{n-m}]f$, is simple but effective. The product in both $f = f_0 + xE\,f$ and $f = \sum_i f_i x^i$ is convolution product, $+$ is pointwise addition, the element f_i is automatically identified with the singleton sequence, $[f_i]$, as required, and $x = [0,1]$. A recursive equation for product is easily derived (and just as easily translated into Haskell); $E\,f$ is abbreviated to f':

$$fg = (f_0 + xf')(g_0 + xg') = f_0 g_0 + f_0 x g' + x f' g = f_0 g_0 + x(f_0 g' + f' g) \tag{1}$$

Sequences, $\mathbb{S}$, over an integral domain or field F (known from context) form an integral domain [64, 55] ($F[\![x]\!]$ is the standard notation for formal power series in x over F). We will keep $F = \mathbb{Q}$ in mind. The subsets $\mathbb{S}_0$, $\mathbb{S}_1$, and $\mathbb{S}_{\neq 0}$ comprise sequences with zeroth term 0, 1, and non-zero, respectively. A subset $\mathbb{S}_C$ of $\mathbb{S}_0$ comprises sequences f in which $f_1 \neq 0$, that is $E\,f \in \mathbb{S}_{\neq 0}$. Unique square (and nth) roots exist for sequences in $\mathbb{S}_1$. Sequence composition $f \circ g = \sum_k f_k g^k$ is defined for $g \in \mathbb{S}_0$. A unique compositional inverse, f°, called *converse*, exists for $f \in \mathbb{S}_C$. The notation follows [6] and distinguishes converse $f \circ f^\circ = x$ from multiplicative inverse $f * f^{-1} = 1$. The latter exists for $f \in \mathbb{S}_{\neq 0}$. Differentiation, D, is term-wise, as for formal polynomials: $[x^n]D\,f = (n+1)[x^{n+1}]f$. A little induction gives the Maclaurin expansion rule: $[x^n]f = \frac{1}{n!}[x^0]D^n f$. From the definition of $\int$ as a *right* inverse to D, $[x^n]D\int f = [x^n]f$, we deduce $[x^{n+1}]\int f = \frac{1}{n+1}[x^n]f$. Setting $[x^0]\int f = 0$, the fundamental theorem of sequence calculus (FTC) is immediate:

$$f = f_0 + \int Df \qquad D(\int f) = f$$

The familiar rules in tables 1 and 2 have easy sequence-algebraic proofs (the third $\int$-product rule is called the *differential Baxter axiom* in [10]). For example, here is a typical proof of the differential composition rule, or chain rule [36, Ch. 1]. A proof by coinduction is presented for contrast in item **Q** of the next section. Let $f \in \mathbb{S}$, $g \in \mathbb{S}_0$, then,

$$D\,f \circ g \;=\; f_1 + 2f_2 g + 3f_3 g^2 + \cdots + k f_k g^{k-1} + \cdots$$

1.	$\int a_n x^n$	$=$	$\frac{a_n}{n+1} x^{n+1}$	**monomial**
2.	$\int (af+bg)$	$=$	$a\int f + b\int g$	**linear**
3.	$(n+1)[x^{n+1}]\int$	$=$	$[x^n]$	**power series**
4.	$\int((D\,f)\circ g)aD\,g$	$=$	$af\circ g + c$	**composition**
5.	$\int fD\,g$	$=$	$fg - \int(D\,f)g + c$	**product (1)**
6.	$\int fg$	$=$	$f\int g - \int((D\,f)\int g)$	**product (2)**
7.	$\int(D\,f)\int(D\,h) + \int D\,(fh)$	$=$	$h\int D\,f + f\int D\,h$	**Baxter**

Table 2: integration rules

$$\begin{aligned}
(D\,f\circ g)D\,g &= f_1 D\,g + 2f_2 g D\,g + 3f_3 g^2 D\,g + \cdots + kf_k g^{k-1} D\,g + \cdots \\
&= \{\text{the power rule, } kg^{k-1}D\,g = D\,g^k\} \\
&\ f_1 D\,g + f_2 D\,g^2 + f_3 D\,g^3 + \cdots \\
[x^n](D\,f\circ g)D\,g &= \{\text{distribute } [x^n]\text{, use } [x^n]D\,g^k = (n+1)[x^{n+1}]g^k\} \\
&\ (n+1)[x^{n+1}](f_1 g^1 + f_2 g^2 + f_3 g^3 + \cdots + f_{n+1} g^{n+1}) \\
&= (n+1)[x^{n+1}]f\circ g \\
&= [x^n]D(f\circ g)
\end{aligned}$$

3 Sequence Algebra examples

The following itemization (**A-Z**) of snippets provides a brief survey of the character of sequence algebra. It is, of course, only a small fraction of the subject.

(A) Sequence algebra is foundational in the sense that it is a low-level concrete extension of arithmetic. To appreciate this, try making the following simpler. Define exp by the sequence differential equation $D\exp = \exp$; $\exp_0 = 1$. Then, by the Maclaurin rule, $\exp = 1{+}x/1!{+}x/2!{+}x/3!{+}\cdots = [1, 1, 1/2, 1/3!, \ldots]$. Define x^* by the sequence difference equation $E\,x^* = x^*$; $x^*_0 = 1$; then, by the head-tail property, $x^* = 1 + x + x^2 + \cdots = [1, 1, 1, \ldots] = 1/(1{-}x)$; the notation is based on the Kleene star [19]. Let $\log f = \mathrm{lgn}\circ(f{-}1)$ where lgn (for which there is no established name other than $\log(1 + x)$) is the converse of $\exp -1$; that is, $\mathrm{lgn} = (\exp -1)^\circ$. Observe that $(\exp -1)\circ\mathrm{lgn} = x$ implies $\exp\circ\mathrm{lgn} = 1{+}x$, so $D\,\mathrm{lgn} = D\,(\exp -1)^\circ = (\exp\circ\mathrm{lgn})^{-1} = 1/(1 + x)$. The FTC gives $\mathrm{lgn} = \int 1/(1 + x)$, and the coefficients can be calculated:

$$\begin{aligned}
[x^{n+1}]\mathrm{lgn} &= [x^{n+1}]\int \frac{1}{1+x} = \frac{1}{n+1}[x^n]\frac{1}{1+x} = \frac{1}{n+1}(-1)^n \\
\mathrm{lgn} &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots
\end{aligned}$$

$\exp\circ(\log f)$	$=$	f		exp **converse**	
$\mathrm{lgn}\circ f$	$=$	$\mathrm{lgn}\circ g$	$\Rightarrow\ f=g$	lgn **cancellation**	
$\exp\circ f$	$=$	$\exp\circ g$	$\Rightarrow\ f=g$	**exp cancellation**	
$\log f$	$=$	$\log g$	$\Rightarrow\ f=g$	**log cancellation**	
$D\,(\log g)$	$=$	$\dfrac{D\,g}{g}$		**log derivative**	
$\mathrm{lgn}\circ f$	$=$	0	$\Leftrightarrow\ f=0$	**zero** lgn	
$\log g$	$=$	0	$\Leftrightarrow\ g=1$	**zero log**	
$\log(fg)$	$=$	$\log f+\log g$		**log product**	
$\log f^r$	$=$	$r\log f$		**log rational power**	$(r\in\mathbb{Q})$
$\exp\circ(f+g)$	$=$	$(\exp\circ f)(\exp\circ g)$		**exp sum**	
$\exp^n\circ f$	$=$	$\exp\circ nf$		**power exp**	
f^r	$=$	$\exp\circ(r\log f)$		**general power defn**	$(r\in F)$
$g^r\circ h$	$=$	$(g\circ h)^r$		**general distributivity**	$(r\in F)$
$f^r f^s$	$=$	f^{r+s}		**law of exponents**	$(r,s\in F)$
$D\,f^r$	$=$	$rf^{r-1}D\,f$		**differential F-power**	$(r\in F)$

Table 3: Rules relating to log and exp

Some well-known rules relating to log and exp, derivable from the differential equation for exp using sequence algebra, appear in table 3 (preconditions are omitted to avoid clutter). Most of these are meticulously proven by Niven [64]. However, Niven does not use composition; but by using composition ($\circ$) and its associativity and distributivity laws [36], his theorem 17 and proof can be rendered as follows: let $g=1+f,\ f,h\in\mathbb{S}_0$,

$$\begin{aligned}
g^r\circ h &= \exp\circ(r\log g)\circ h=\exp\circ r((\mathrm{lgn}\circ f)\circ h)\\
&= \exp\circ r(\mathrm{lgn}\circ(f\circ h))=\exp\circ r\log((f\circ h)+1)\\
&= \exp\circ r\log((f+1)\circ h)=\exp\circ r\log(g\circ h)=(g\circ h)^r
\end{aligned}$$

Proof of Euler's identity, $\exp\circ ix=\cos+i\sin$, illustrates appeal to the uniqueness of solution to certain differential equations. Let D sin $=$ cos; $\sin_0=0$, D cos $=-\sin$; $\cos_0=1$, and $i^2=-1$. Then both $\cos+i\sin$ and $\exp\circ ix$ satisfy $D\,g=i\,g;\ g_0=1$, and therefore must be equal. De Moivre's theorem follows:

$$(\cos+i\sin)^n=(\exp\circ ix)^n=\exp\circ ix\circ nx=(\cos+i\sin)\circ nx=\cos\circ nx+i(\sin\circ nx)$$

(B) A *counting* sequence, c, is either *ordinary*, $c=[c_0,c_1,c_2,c_3\ldots]$ or *exponential* $c=[c_0,c_1/1!,c_2/2!,c_3/3!,\ldots]$. In either case, c_n counts the number of objects of size n generated by some structure specification, C. In the former case $c_n=[x^n]c$, and in the latter case $c_n=n![x^n]c$. Roughly speaking, a structure is built from nodes, and its size is the number of nodes it has. The nodes may be labelled or unlabelled. Exponential sequences are used for labelled objects because, in that case, convolution product automatically counts all possible labellings in making an ordered product. Let f and g count labelled structures (generated by some F and G, respectively); then ordered pairs of such structures are counted by

$$[x^n]fg=\sum_{k=0}^{n}\frac{f_kg_{n-k}}{k!(n-k)!}=\sum_{k=0}^{n}\binom{n}{k}f_kg_{n-k}/n!$$

The ordered k-fold product, F^k, has counting sequence f^k. Ordered lists of F-objects are counted by $\mathsf{list} \circ f = x^* \circ f = f^*$. If the order does not count, then we use $\mathsf{set} \circ f = \sum_k f^k/k! = \exp \circ f$.

The fact that permutations can be written as sets of cycles can be made explicit in the definition of the counting sequence for permutations [25, 11]:

$$\mathsf{perm} = \mathsf{set} \circ \mathsf{cycle}$$

Just put $\mathsf{set} = \exp$ and $\mathsf{cycle} = \log x^*$. The sequence perm is x^* regarded as an *exponential sequence*, $x^* = [1, 1!/1!, 2!/2!, 3!/3! \ldots]$. Removal of the factorial divisors is performed by Λ, and $\Lambda\mathsf{perm} = [1, 1, 2, 6, 24, 120, \ldots]$, $[x^n]\Lambda\mathsf{perm} = n![x^n]\mathsf{perm} = n!$, the number of permutations of n symbols. There are $(n-1)!$ cyclic permutations of n symbols, and the exponential counting sequence for these is:

$$\mathsf{cycle} = \sum_{n>0} (n-1)!\frac{x^n}{n!} = \sum_{n>0} \frac{1}{n}x^n = -\mathrm{lgn} \circ (-x) = -\log(1-x) = \log x^*$$

(C) The number of ways, s_n, of inserting brackets into a list of n symbols, subject to well-formedness, is counted by the Hipparchus-Schröder sequence [75, 78], $s = \frac{1}{4}(1 + x - \sqrt{1 - 6x + x^2}\,) = [0, 1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, \ldots]$. This can be derived from the specification of a bivariate counting sequence for Schröder trees:

$$\mathsf{schroeder(z,u)} = z + u * (\mathsf{pluralList} \circ \mathsf{schroeder(z,u)}) \tag{2}$$

Here $\mathsf{pluralList} = \mathsf{list} - x - 1$. The coefficient of $u^k z^n$ in $\mathsf{schroeder(z,u)}$ gives the number of Schröder bracketings of n symbols using k pairs of brackets. Observe that equation (2) can also be read [25] as a set-theoretic specification of Schröder trees, with z and u naming different kinds of nodes. Using $\mathsf{pluralList} = x^* - x - 1$, the above definition of s derives, via the quadratic formula, from the equation $s = \mathsf{schroeder(x,1)} = x + (x^* - x - 1) \circ s = x + s^* - s - 1 = x + \dfrac{1}{1-s} - s - 1$. Here is a foretaste of computing the Schröder numbers in Haskell, which is explained in section 4:

```
schroeder   =  z + u*(pluralList `o` schroeder)
> select [1..6] (unDiag schroeder)
[[0],[1,0],[0,1,0],[0,1,2,0],[0,1,5,5,0],[0,1,9,21,14,0]]
> takeW 11 ((1+x-sqroot(1-6*x+x^2))/4)
[0,1,1,3,11,45,197,903,4279,20793,103049]
```

This reveals, for example, $[u^2 z^5]\mathsf{schroeder} = 9$, that is 9 bracketings of 5 symbols with 2 pairs of brackets. One can see that the elements of the second sequence are totals of corresponding elements of the first.

(D) The dual interpretation of sequence expressions as set-theoretic structure specifications, is exploited in [24, 25] (influenced by [44]). One might write the set-theoretic counterpart of say, $\mathsf{perm} = \mathsf{set} \circ \mathsf{cycle}$, as $\mathsf{Perm} \cong \mathsf{Set} \circ \mathsf{Cycle}$, indicating by the initial capital letters a set-theoretic interpretation. Essentially this is done in [25], but for brevity

$$
\begin{aligned}
\mathsf{emptySet} &= 1\\
\mathsf{singletonSet} &= x\\
\mathsf{singletonList} &= x\\
\mathsf{nonEmptyList} &= \mathsf{list} - 1\\
\mathsf{pluralList} &= \mathsf{list} - \mathsf{singletonList} - 1\\
\mathsf{ordPair} &= x^2\\
\mathsf{fibonacci} &= \mathsf{list} \circ (\mathsf{singletonList} + \mathsf{ordPair})\\
\mathsf{cycle} &= \log x^*\\
\mathsf{oneOrTwoCycle} &= x + x^2/2\\
\mathsf{involution} &= \mathsf{set} \circ \mathsf{oneOrTwoCycle}\\
\mathsf{nonLoopCycle} &= \mathsf{cycle} - \mathsf{singletonSet}\\
\mathsf{derangement} &= \mathsf{set} \circ \mathsf{nonLoopCycle}\\
\mathsf{permutation} &= \mathsf{derangement} * \mathsf{set}\\
\mathsf{nonEmptySet} &= \mathsf{set} - \mathsf{empty}\\
\mathsf{pluralSet} &= \mathsf{nonEmptySet} - \mathsf{singletonSet}\\
\mathsf{setPartition} &= \mathsf{set} \circ \mathsf{nonEmptySet}\\
\mathsf{oddNumberOfParts} &= \sinh \circ \mathsf{nonEmptySet}\\
\mathsf{evenSizedParts} &= \mathsf{set} \circ (\cosh - 1)\\
\mathsf{catalanTree} &= x(\mathsf{list} \circ \mathsf{catalanTree})\\
\mathsf{cayleyTree} &= x(\mathsf{set} \circ \mathsf{cayleyTree})\\
\mathsf{motzkinTree} &= x(1 + \mathsf{motzkinTree} + \mathsf{motzkinTree}^2)\\
\mathsf{connectedMapping} &= \mathsf{cycle} \circ \mathsf{cayleyTree}\\
\mathsf{mapping} &= \mathsf{set} \circ \mathsf{connectedMapping}\\
\mathsf{fixedPointFree} &= \mathsf{set} \circ \mathsf{nonLoopCycle} \circ \mathsf{cayleyTree}\\
\mathsf{idempotent} &= \mathsf{set} \circ (x * \mathsf{set})\\
\mathsf{partialMapping} &= \mathsf{mapping} * (\mathsf{set} \circ \mathsf{cayleyTree})\\
\mathsf{surjection} &= \mathsf{list} \circ \mathsf{nonEmptySet}\\
\mathsf{connectedAcyclicGraph} &= \mathsf{cayleyTree} - \tfrac{1}{2}\mathsf{cayleyTree}^2\\
\mathsf{acyclicGraph} &= \mathsf{set} \circ \mathsf{connectedAcyclicGraph}\\
\mathsf{zigzag} &= 2(\tan + \sec)
\end{aligned}
$$

Table 4: Some counting sequences

we only give the equations defining the counting sequences. Table 4 lists some univariate examples, and table 5 some bivariate ones. These can be typed more-or-less verbatim into Haskell, as illustrated in section 5. Many more could be lifted from chapters I-III of [25], from the appendices of [4], and from [15, 31, 78, 75].

Let us examine a less-than-obvious expression, ascents from table 5, the origin of which illustrates the principle of *inclusion-exclusion* [25, 88]. It counts permutations according to the number of ascents. For example, the permutation $|248|3679|5|1|$ of [9] has 4 up-runs demarcated with vertical bars, 3 descents, and 5 ascents. If there are k descents then there are $k+1$ up-runs. The reversal of a permutation with k descents delivers a permutation with k ascents. The count $n![u^k][z^n]\mathsf{ascents}$ is called an *Eulerian number*, and gives the number of permutations with k ascents or $k+1$ up-runs [32]. To come up with the sequence expression, first note that an up-run with at least one ascent corresponds to a plural set. The counting sequence for such a set in which the k of u^k records the ascents is $(\mathsf{pluralSet} \circ uz)/u$. Let us specify permutations in which some *parts* of up-runs are identified

pascal	=	$(u+z)^*$
intComposition	=	$\mathsf{list} \circ (u * (\mathsf{nonEmptyList} \circ z))$
schroeder	=	$z + u * (\mathsf{pluralList} \circ \mathsf{schroeder})$
catalanLeaves	=	$u * z + z * (\mathsf{nonEmptyList} \circ \mathsf{catalanLeaves})$
cayleyLeaves	=	$u * z + z * (\mathsf{nonEmptySet} \circ \mathsf{cayleyLeaves})$
ebinom	=	$\mathsf{set} \circ (z + uz)$
cycles	=	$\mathsf{set} \circ (u * (\mathsf{cycle} \circ z))$
parts	=	$\mathsf{set} \circ (u * (\mathsf{nonEmptySet} \circ z))$
permFixedPts	=	$(\mathsf{derangement} \circ z) * (\mathsf{set} \circ uz)$
ascents	=	$\mathsf{list} \circ (z + (\mathsf{pluralSet} \circ (uz - z))/(u-1))$
valleys	=	$\dfrac{\sqrt{1-u}}{\sqrt{1-u} - \tanh \circ (z\sqrt{1-u})}$
powerSums	=	$\dfrac{\exp \circ uz - 1}{\exp \circ z - 1}$
bernoulli	=	$\dfrac{z \exp \circ uz}{\exp \circ z - 1}$
legendre	=	$(1 - 2uz + z^2)^{-1/2}$
chebyshev	=	$\dfrac{1 - uz}{z^2 - 2uz + 1}$
laguerre	=	$\dfrac{1}{1-z} \exp \circ \dfrac{-uz}{1-z}$
hermite	=	$\exp \circ (2uz - z^2)$
meixner	=	$(1+z^2)^{-1/2} \exp \circ (u \arctan \circ z)$

Table 5: Some bivariate sequences

as sets, and other elements are undistinguished: $b(z,u) = \mathsf{list} \circ (z + (\mathsf{pluralSet} \circ uz)/u)$. Now propose that $\mathsf{ascents}(z,u)$ is the exact counting sequence we are after, then the inclusion-exclusion principle says that $\mathsf{ascents}(z, u+1) = b(z,u)$ and $\mathsf{ascents}(z,u) = b(z, u-1)$, which is cited in the table.

(E) The sequence defined by $D \tan = 1 + \tan * \tan;\ \tan_0 = 0$ is the Maclaurin expansion of the tangent function (the $*$ is explicit just for emphasis). The numbers in $t = \Lambda \tan = [0, 1, 0, 2, 0, 16, 0, 272, 0, 7936, \ldots]$ are called *tangent numbers*. The tangent numbers count certain kinds of alternating permutations (or ordered binary trees) [77]. One can define t also by $E\,t = 1 + t \otimes t$, where $\otimes$ is *shuffle* (or *Hurwitz* [47]) product. Convolution product (1) and shuffle product can be defined in head-tail form (E is $'$):

$$\begin{array}{rcl|l} (st)' &=& s't + s_0 t' & (st)_0 = s_0 t_0 \\ (s \otimes t)' &=& s' \otimes t + s \otimes t' & (s \otimes t)_0 = s_0 t_0 \end{array}$$

The rule $E(s \otimes t) = E\,s \otimes t + s \otimes E\,t$ matches $D(fg) = D\,f + f D\,g$ and we have the Leibniz formulae

$$D^n(fg) = \sum_{k=0}^{n} \binom{n}{k} (D^k f) D^{n-k} g; \qquad E^n(s \otimes t) = \sum_{k=0}^{n} \binom{n}{k} E^k s \otimes E^{n-k} t$$

Applying $[x^0]$ to the latter gives the pointwise definition of $\otimes$ (note that $a \otimes b = ab$ when

a and b are scalars):

$$[x^n](s \otimes t) = \sum_{k=0}^{n} \binom{n}{k} ([x^0]E^k s) \otimes ([x^0]E^{n-k} t) = \sum_{k=0}^{n} \binom{n}{k} s_k t_{n-k}$$

A shuffle inverse, $s^{-1\otimes}$, derives from the specification $s \otimes s^{-1\otimes} = 1$ together with the shuffle product rule (in exactly the same way that the rule for $D\,f^{-1}$ is derived):

$$0 = 1' = (s \otimes s^{-1\otimes})' = s' \otimes s^{-1\otimes} + s \otimes (s^{-1\otimes})'; \quad (s^{-1\otimes})' = -s' \otimes s^{-1\otimes} \otimes s^{-1\otimes}$$

(F) Let $\mathbb{S}_F(*)$ denote the ring $(\mathbb{S}_F, +, *, 0, 1)$, of sequences over some field F (of characteristic 0) with the availability of inverses implied. Then $(\mathbb{S}_F(*), D, \int)$ is an integro-differential algebra [10], and so too is $(\mathbb{S}_F(\otimes), E, x)$, where x is the right-shift operator: $xf = x * f$. The transform (Λ, Λ^{-1}) is an isomorphism between them, and is a formal Laplace transform [67, 26]. In the previous example, the equation defining tan is transformed by Λ into the equation defining t. The sequence of factorial numbers , $x^{\otimes} = \Lambda x^*$ can be defined by applying Λ to $x^* = 1 + xx^*$ to get $x^{\otimes} = 1 + x \otimes x^{\otimes}$. From this we deduce $x^{\otimes} = (1-x)^{-1\otimes}$. Observe, for $n > 0$:

$$[x^n]x^{\otimes} = [x^n]x \otimes x^{\otimes} = \sum_{k=0}^{n} \binom{n}{k} [x^k]x[x^{n-k}]x^{\otimes} = \binom{n}{1} [x^{n-1}]x^{\otimes} = n[x^{n-1}]x^{\otimes}$$

Also, $[x^n]x^{\otimes} = n[x^{n-1}]x^{\otimes} = (n-1)[x^{n-1}]x^{\otimes} + [x^{n-1}]x^{\otimes} = [x^n]x^2 D\, x^{\otimes} + [x^n]xx^{\otimes}$ leads to the differential equation for the factorials:

$$x^{\otimes} = 1 + xx^{\otimes} + x^2 D\, x^{\otimes}$$

Furthermore,

$$(x^{\otimes})' = ((1-x)^{-1\otimes})' = (1-x)^{-2\otimes}; \quad x^{\otimes} = 1 + x(1-x)^{-2\otimes}$$

(G) We recall a classic proof [64] of the binomial theorem. Let $r \in F$, $r^{\underline{k}} = r(r-1)\cdots(r-k+1)$ (the falling factorial), and $r^{\overline{k}} = r(r+1)\cdots(r+k-1)$ (the rising factorial):

$$[x^k](1+bx)^r = \frac{1}{k!}[x^0]D^k(1+bx)^r = \frac{1}{k!}[x^0]r^{\underline{k}}b^k(1+bx)^{r-k} = b^k \binom{r}{k}$$

A corollary is $[x^k](x^*)^r = [x^k](1-x)^{-r} = (-1)^k(-r)^{\underline{k}}/k! = r^{\overline{k}}/k! = \binom{r+k-1}{r-1}$. This result, and $[x^m]\exp^n = [x^m]\exp \circ nx = n^m/m!$, are basic ingredients in the search for nth term formulas. They are applied next.

(H) A Lagrange inversion formula [76, 59, 60, 29] gives an expression for the nth term of the converse of a sequence. For example, let $g = x(r \circ g)$, then $x = g/(r \circ g) = (x/r) \circ g$, so g is the converse of x/r. Below is the Lagrange inversion formula for this case, followed by its application to the counting sequences for Catalan trees, $c = x(\mathsf{list} \circ c) = x(x^* \circ c)$, and Cayley trees, $t = x(\mathsf{set} \circ t) = x(\exp \circ t)$:

$$\begin{aligned}
[x^n]g &= \frac{1}{n}[x^{n-1}]r^n \\
[x^n]c &= \frac{1}{n}[x^{n-1}](x^*)^n = \frac{1}{n}\binom{2n-2}{n-1} \\
[x^n]t &= \frac{1}{n}[x^{n-1}]\exp^n = \frac{1}{n}\frac{n^{n-1}}{(n-1)!} = \frac{n^{n-1}}{n!}
\end{aligned}$$

For a history of the Catalan numbers, see [66]. Cayley trees are rooted connected acyclic graphs, counted by Cayley in [13]. Cayley also counted the Catalan trees in [12], and the first part of Niven [64] sets out to legitimise the sequence algebra underlying Cayley's proof (Niven cites [42], not Cayley; but Raney [70] cites both).

A slightly more general statement of Lagrange inversion is that it solves $h = g \circ f$ for g, where $h \in \mathbb{S}$, $f \in \mathbb{S}_C$. The theory of Lagrange inversion sometimes employs Laurent series – series with negative powers (or sequences with negative indicies). In the following formula for the nth term of $g = h \circ f^\circ$, the coefficient of x^{-1} (called the *residue*) is identified:

$$[x^n]g = \frac{1}{n}[x^{-1}]D\,hf^{-n}; \qquad [x^0]g = h_0$$

Let $s = x(r \circ s)$, $s = (x/r)^\circ$, and $h = g \circ (x/r)$; then Lagrange inversion gives $g = h \circ s$, and when $h = x^k$, we get $[x^n]s^k = \frac{k}{n}[x^{n-k}]r^n$. This result specialises, when $r = x^*$, to a variant of the *cycle lemma* [17], called Raney's lemma in [32], which has a history in statistics [69]. There are various Lagrange inversion formulas and many proofs; [14] takes an approach that also facilitates proof of a formula for $D^n(f \circ g)$ (Faà di Bruno's formula [43]).

(I) The (forward) *difference* operator, $\Delta s = [E - 1]s = s' - s$ produces the sequence of term-to-term differences, $[x^n]\Delta s = s_{n+1} - s_n$ (assume s to be infinite). The definition of an *anti-difference* operator Σ on sequences is calculated [37] as a right identity to Δ, with $(\Sigma s)_0 = 0$:

$$\Delta\Sigma s = s \Leftrightarrow (\Sigma s)' - \Sigma s = s \Leftrightarrow (\Sigma s)' = \Sigma s + s \Leftrightarrow \Sigma s = 0 + x(\Sigma s + s) \Leftrightarrow \Sigma s = \frac{xs}{1-x}$$

Thus, $\Sigma = xx^*$ computes all the prefix sums of s (including the empty one). Applied to Δs:

$$[x^n]\Sigma\Delta s = [x^n]xx^*\Delta s = [x^{n-1}]x^*\Delta s = \sum_{i=0}^{n-1}\left([x^{i+1}]s - [x^i]s\right) = [x^n]s - [x^0]s$$

There follows the fundamental theorem of discrete calculus (FDC) on sequences: $s = s_0^\bullet + \Sigma\Delta s$; $s = \Delta\Sigma s$, where $a^\bullet = ax^*$ is the sequence with a everywhere.

(J) Here are the E to Δ translations extended to powers:

$$\begin{aligned}
E^n = (1+\Delta)^n &= \sum_{k=0}^{n} \binom{n}{k} \Delta^k \\
[x^n]s = (E^n s)_0 &= \sum_{k=0}^{n} \binom{n}{k} (\Delta^k s)_0 \qquad (3) \\
\Delta^n = (E-1)^n &= \sum_{k=0}^{n} \binom{n}{k} (-1)^{n-k} E^k \\
(\Delta^n s)_0 &= \sum_{k=0}^{n} \binom{n}{k} (-1)^{n-k} s_k = [x^n](-x)^* \otimes s \qquad (4)
\end{aligned}$$

The identity $\binom{n}{k} = [x^n]x^k/(1-x)^{k+1}$ turns equation (3) into the Euler expansion, $s = \sum_k \frac{(\Delta^k s)_0 x^k}{(1-x)^{k+1}}$. This expansion can also be derived from $s \otimes (-x)^* = \sum_k (\Delta^k s)_0 x^k$, plus the facts $(-x)^* \otimes x^* = 1$, and $x^k \otimes x^* = x^k/(1-x)^{k+1}$ (see [3] and items **K** and **P**):

$$s = s \otimes (-x)^* \otimes x^* = (s_0 + (\Delta s)_0 x + (\Delta^2 s)_0 x^2 + \cdots) \otimes x^* = \sum_k \frac{(\Delta^k s)_0 x^k}{(1-x)^{k+1}} \qquad (5)$$

Let $g = \mathrm{lgn} \circ -x$, whence $\mathrm{lgn} = g \circ -x$, and apply Euler's expansion to g,

$$\begin{aligned}
\mathrm{lgn} &= \left(\sum_k \frac{x^k}{(1-x)^{k+1}} (\Delta^k g)_0 \right) \circ -x = \sum_k \frac{(-x)^k}{(1+x)^{k+1}} (\Delta^k g)_0 \\
\mathrm{lgn}(1) &= \sum_k \frac{(-1)^k}{2^{k+1}} (\Delta^k g)_0
\end{aligned}$$

It is instructive to use this to approximate $\log(2) = \mathrm{lgn}(1)$.

(K) The sequence $\mathcal{N}s = (-x)^* \otimes s = [s_0, (\Delta s)_0, (\Delta^2 s)_0, \ldots]$ is the sequence of *Newton coefficients* [3]. It may also be specified by $(\mathcal{N}s)' = \mathcal{N}(\Delta s)$; $(\mathcal{N}s)_0 = s_0$. The operator $\mathcal{N} = ((-x)^* \otimes \_)$, called the *Newton transform* in [3], has the converse $\mathcal{N}^{-1} = (x^* \otimes \_)$, called the *Binomial transform* in [39]. The identity $x^* \otimes (-x)^* = 1$ holds because the head of $x^* \otimes (-x)^*$ is 1, and the tail is

$$(x^* \otimes (1+x)^{-1})' = x^* \otimes ((1+x)^{-1} + ((1+x)^{-1})') = 0$$

The following products introduce two new rings, the *Hadamard* ring $(S_R, +, -, \odot, 0, 1^\bullet)$, and the *infiltration* ring $(S_R, +, -, \uparrow, 0, 1)$ [3]:

$$\begin{aligned}
(s \odot t)' &= s' \odot t' & (s \odot t)_0 &= s_0 t_0 \\
(s \uparrow t)' &= s' \uparrow t + s \uparrow t' + s' \uparrow t' & (s \uparrow t)_0 &= s_0 t_0
\end{aligned}$$

1.	$\Delta(s \odot t)$	$=$	$s \odot \Delta t + \Delta s \odot t'$	**Δ-product (1)**
2.	$\Delta(s \odot t)$	$=$	$\Delta s \odot t + s \odot \Delta t + \Delta s \odot \Delta t$	**Δ-product (2)**
3.	$\Sigma(s \odot \Delta t)$	$=$	$s \odot t - \Sigma(\Delta s \odot t') - (s \odot t)_0^\bullet$	**Σ-product (1)**
4.	$\Sigma(s' \odot v)$	$=$	$s \odot (\Sigma v) - \Sigma(\Delta s \odot \Sigma v)$	**Σ-product (2)**
5.	$\Sigma\Delta s \odot \Sigma\Delta u + \Sigma\Delta(s \odot u)$	$=$	$(\Sigma\Delta s) \odot u + s \odot (\Sigma\Delta u)$	**Σ-Baxter rule**

Table 6: $\Delta - \Sigma$ rules

The rules in table 6 apply, and $(\mathcal{N}, \mathcal{N}^{-1})$ is an isomorphism between the Hadamard and infiltration rings. The following is a point-wise definition of $s \uparrow t$:

$$[x^n]s \uparrow t = [x^n]\mathcal{N}(\mathcal{N}^{-1}s \odot \mathcal{N}^{-1}t) = \sum_{i=0}^{n} \binom{n}{i} (-1)^{n-i}[x^i]((x^* \otimes s) \odot (x^* \otimes t))$$

The proof that $\mathcal{N}$ is a morphism from $\odot$ to the new product $\uparrow$ can be re-imagined as a discovery of what the definition of $\uparrow$ should be. The morphism presumption is signalled on the right below.

$$\begin{array}{lll}
(\mathcal{N}(s \odot t))' & & \\
\quad = & \mathcal{N}(\Delta(s \odot t)) & \text{defn. } \mathcal{N} \\
\quad = & \mathcal{N}(\Delta s \odot t + s \odot \Delta t + \Delta s \odot \Delta t) & \Delta\text{-product (2)} \\
\quad = & \mathcal{N}(\Delta s \odot t) + \mathcal{N}(s \odot \Delta t) + \mathcal{N}(\Delta s \odot \Delta t) & \text{morphism} \\
\quad = & \mathcal{N}\Delta s \uparrow \mathcal{N}t + \mathcal{N}s \uparrow \mathcal{N}\Delta t + \mathcal{N}\Delta s \uparrow \mathcal{N}\Delta t & \text{morphism} \\
\quad = & (\mathcal{N}s)' \uparrow \mathcal{N}t + \mathcal{N}s \uparrow (\mathcal{N}t)' + (\mathcal{N}s)' \uparrow (\mathcal{N}t)' & \text{defn. } \mathcal{N} \\
\quad = & (\mathcal{N}s \uparrow \mathcal{N}t)' & \text{defn. } \uparrow
\end{array}$$

(L) The following defines permutation cycle numbers, $\begin{bmatrix} n \\ k \end{bmatrix} = n![u^k z^n]\mathsf{cycles}$, and set partition numbers $\begin{Bmatrix} n \\ k \end{Bmatrix} = n![u^k z^n]\mathsf{parts}$. These are also called Stirling numbers of the first and second kind, respectively.

$$\begin{array}{lll}
\mathsf{cycles} & = & \mathsf{set} \circ (u * (\mathsf{cycle} \circ z)) \\
\mathsf{parts} & = & \mathsf{set} \circ (u * (\mathsf{nonEmptySet} \circ z))
\end{array}$$

The well-known recurrences [32],

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = \begin{bmatrix} n \\ k-1 \end{bmatrix} + n \begin{bmatrix} n \\ k \end{bmatrix}, \qquad \begin{Bmatrix} n+1 \\ k \end{Bmatrix} = \begin{Bmatrix} n \\ k-1 \end{Bmatrix} + k \begin{Bmatrix} n \\ k \end{Bmatrix}$$

translate, using $c = \mathsf{cycles}$ and $p = \mathsf{parts}$, into

$$D_z c = uc + zD_z c \qquad\qquad D_z p = up + uD_u p$$

where D_z and D_u are the partial differentiation operators with respect to z and u. To see this, note that $\begin{bmatrix} n+1 \\ k \end{bmatrix} = (n+1)![u^k z^{n+1}]c = n![u^k z^n]D_z c$, $\begin{bmatrix} n \\ k-1 \end{bmatrix} = n![u^{k-1} z^n]c =$

$n![u^k z^n]uc$, and so on. The recurrences can be checked: first, $D_z c = D_z \exp\circ(u \log z^*) = ucz^* = uc + uczz^* = uc + zD_z c$; and second, $up + uD_u p = up + u(\exp\circ z - 1)p = pu\exp\circ z = D_z p$. We may write $n![z^n]\exp\circ(u\ \log\circ z^*) = n![z^n](1-z)^{-u} = u^{\overline{n}}$. The cycles recurrence can also be written $[x^k]x^{\overline{n}} = [x^{k-1}]x^{\overline{n-1}} + (n-1)[x^k]x^{\overline{n-1}}$, which follows from $x^{\overline{n}} = x^{\overline{n-1}}(x+n-1) = xx^{\overline{n-1}} + (n-1)x^{\overline{n-1}}$.

(M) A *factorial polynomial* uses falling factorials instead of powers. For example, let $p = 1 + 2x + x^2$, then the falling factorial counterpart is $\underline{p} = 1 + 3x^{\underline{1}} + x^{\underline{2}}$. Coefficients in $\underline{p}$ are identified by $[x^{\underline{k}}]\underline{p}$, for example $[x^{\underline{1}}]\underline{p} = 3$.

The symbols Σ and Δ are overloaded as operators on factorial polynomials and obey rules identical to those for D and $\int$ on polynomials: let $\underline{p}$ denote a polynomial in falling factorials, then $[x^{\underline{k}}]\Delta\underline{p} = (k+1)[x^{\underline{k+1}}]\underline{p}$ and $[x^{\underline{n+1}}]\Sigma\underline{p} = \dfrac{1}{n+1}[x^{\underline{n}}]\underline{p}$. The fundamental theorem of the discrete calculus on factorial polynomials is immediate: $\underline{p} = \underline{p}_0 + \Sigma\Delta\underline{p}$; $\underline{p} = \Delta\Sigma\underline{p}$. In [32], the theorem provides one of seven ways of deducing the polynomial for summing squares: given $\Delta\underline{p} = (1+x)^2 = 1 + 3x^{\underline{1}} + x^{\underline{2}}$, apply Σ to both sides,

$$\underline{p} - \underline{p}_0 = 1x^{\underline{1}} + 3/2\,x^{\underline{2}} + 1/3\,x^{\underline{3}};\ p = 1/6\,(x + 3x^2 + 2x^3) \tag{6}$$

Note that if p is a polynomial nth term formula for sequence s, $p(n) = s_n$, then $(\Delta\underline{p})(n) = (\Delta s)_n$ and $(\Sigma\Delta\underline{p})(n) = p(n) - p(0) = s_n - s_0 = (\Sigma\Delta s)_n$.

(N) An analogue of the Maclaurin rule holds: $[x^{\underline{n}}]\underline{p} = \dfrac{1}{n!}[x^{\underline{0}}]\Delta^n\underline{p} = \dfrac{1}{n!}(\Delta^n p)(0)$. The latter equality involves yet another interpretation of Δ: $(\Delta p)(n) = p(n+1) - p(n)$. Gregory-Newton formulas for p of degree m follow; the second (see also (3)) uses $n^{\underline{k}}/k! = \dbinom{n}{k}$:

$$\begin{aligned}\underline{p} &= p(0)x^{\underline{0}} + (\Delta p)(0)x^{\underline{1}} + \frac{(\Delta^2 p)(0)}{2!}x^{\underline{2}} + \cdots + \frac{(\Delta^m p)(0)}{m!}x^{\underline{m}} && (7)\\ p(n) &= p(0)\binom{n}{0} + (\Delta p)(0)\binom{n}{1} + (\Delta^2 p)(0)\binom{n}{2} + \cdots + (\Delta^m p)(0)\binom{n}{m} && (8)\end{aligned}$$

Let $s = [0, 1, 5, 14, 30, 55, \ldots]$ be the sequence $s_n = 1^2 + 2^2 + \cdots + n^2$, for which we seek the polynomial p such that $p(n) = s_n$. Then the above expansions produce the polynomials in (6). By contrast, the Euler expansion (5) produces the sequence expression $s = (x + x^2)/(1-x)^4$.

(O) We have seen $[x^k]x^{\overline{n}} = \begin{bmatrix} n \\ k \end{bmatrix}$, so we can express the polynomial for $x^{\overline{n}}$ in terms of cycle numbers, and by change of signs, also the polynomial for $x^{\underline{n}}$:

$$x^{\overline{n}} = \sum_{k=1}^{n} \begin{bmatrix} n \\ k \end{bmatrix} x^k; \qquad x^{\underline{n}} = \sum_{k=1}^{n} (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k;\ \ x^{\overline{0}} = x^{\underline{0}} = 1$$

This shows how to translate falling factorials into powers. The converse is

$$x^n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} x^{\underline{k}} \qquad\qquad [x^{\underline{k}}]x^n = \left\{ {n \atop k} \right\}$$

and for two different proofs see [9] and [32].

(P) Infinite sequences are called *streams* when they are identified with the final object in a category of head-tail coalgebras [41, 71]. This accounts for the name "stream" in the following:

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{Fundamental theorem of (sequence) calculus (FTC)}} \\
f = f_0 + \int D\,f & & f = D \int f \\
\multicolumn{3}{l}{\text{Fundamental theorem of stream calculus (FSC)}} \\
f = f_0 + x(Ef) & & f = E(xf) \\
\multicolumn{3}{l}{\text{Fundamental theorem of discrete calculus (FDC)}} \\
f = f_0^{\bullet} + \Sigma\Delta\,f & & f = \Delta\,\Sigma f
\end{array}$$

The co-algebraic stream calculus [71] introduces a proof principle called *coinduction.* For example, the identity $x^k \otimes x^* = x^k/(1-x)^{k+1} = x^*(xx^*)^k$ used in the proof of (5) can be proved using coinduction (it can also be proved from the point-wise definition of $\otimes$ and the binomial theorem). Here is the gist of the coinductive proof. Propose the relation $x^k \otimes x^* \sim x^*(xx^*)^k$. This is used as a coinductive hypothesis. Head-equality holds, $(x^k \otimes x^*)_0 = (x^*(xx^*)^k)_0$. The proof is completed by showing that the tails are equal under the hypothesis, signified below by the use of ($\sim$):

$$\begin{array}{rcll}
(x^*(xx^*)^k)' & = & (x^*)'(xx^*)^k + 1((xx^*)^k)' & \text{conv. product rule} \\
& = & x^*(xx^*)^k + x^*(xx^*)^{k-1} & (x^*)' = x^* \text{ and } ((xf)^n)' = f(xf)^{n-1} \\
& \sim & x^k \otimes x^* + x^{k-1} \otimes x^* & \text{coinduction hyp.} \\
& = & (x^k \otimes x^*)' & \text{shuffle product rule}
\end{array}$$

(Q) One can check from the pointwise definitions of D and $\otimes$ that $D\,f = (x \otimes f')'$. Alternatively, equality can be proved by showing that these expressions satisfy the same head-tail equations. The head-tail equation for $D\,f$ is calculated:

$$D\,f = D(f_0 + xf') = f' + xD\,f' = f_0' + xf'' + xD\,f' = f_1 + x(f'' + D\,f')$$

Hence, $(D\,f)_0 = f_1$ and $(D\,f)' = f'' + D\,f'$. Now let $F\,f = (x \otimes f')'$. We find $(F\,f)_0 = f_1$, and $(F\,f)' = (f' + (x \otimes f''))' = f'' + F\,f'$. Thus, $D\,f = F\,f$ since they satisfy the same head-tail equations.

To give a little more feeling for the coinduction game, let us prove $D(f \circ g) = (D\,f \circ g)D\,g$ using head-tail properties, such as $(f \circ g)_0 = f_0$; $(f \circ g)' = (f' \circ g)g'$. We will also make use of $(Dh)_0 = h_0'$. Head equality is confirmed:

$$(D(f \circ g))_0 = (f \circ g)_0' = (f' \circ g)_0 g_0' = f_0' g_0' = (Df \circ g)_0 (Dg)_0 = ((Df \circ g)Dg)_0$$

Tails are proved equal under the coinductive hypothesis, $D(f \circ g) \sim (Df \circ g)Dg$:

$$\begin{array}{lll}
\multicolumn{2}{l}{((Df \circ g)Dg)'} & \\
= & (Df \circ g)'Dg + (Df \circ g)_0(Dg)' & (')\text{-product} \\
= & ((Df)' \circ g)g'Dg + (f' \circ g)_0(Dg)' & (')\text{-composition} \\
= & (f'' \circ g)g'Dg + (Df' \circ g)g'Dg + (f' \circ g)_0(Dg)' & D\text{-defn and } \circ\text{-distr} \\
\sim & (f' \circ g)'Dg + D(f' \circ g)g' + (f' \circ g)_0(Dg)' & (')\text{-comp., coinduction} \\
= & (f' \circ g)'(g' + xDg') + D(f' \circ g)g' + (f' \circ g)_0(g'' + Dg') & \text{expand } D\,g \text{ and } D\,g' \\
= & D(f' \circ g)g' + (f' \circ g)'xDg' + (f' \circ g)_0Dg' + (f' \circ g)'g' + (f' \circ g)_0g'' & \\
= & D(f' \circ g)g' + (f' \circ g)Dg' + ((f' \circ g)g')' & \text{head-tail, } h = h_0 + xh' \\
= & D((f' \circ g)g') + ((f' \circ g)g')' & D\text{-product} \\
= & D(f \circ g)' + (f \circ g)'' & (')\text{-composition, twice} \\
= & (D(f \circ g))' & D\text{-defn}
\end{array}$$

The bracketed (co-) in the paper's title indicates that we only touch lightly on co-algebraic concepts. The survey [33] provides background.

(R) Coinduction is also used in [72] to show how continued fractions can be obtained from combinatorially-inspired automata. For example, the tangent sequence can be defined by $t = xu_1$, where $u_k = 1/(1 - k(k+1)x^2u_{k+1})$. Thus t can be displayed as a continued fraction:

$$t = \cfrac{x}{1 - \cfrac{1 * 2x^2}{1 - \cfrac{2 * 3x^2}{1 - \cfrac{3 * 4x^2}{\ddots}}}}$$

More combinatorially-inspired continued fractions appear in [23, 31, 72]. Below is one for $\Lambda_z\mathsf{cycles} = (1 - z)^{-u_\otimes}$ (Λ_z removes the factorial divisors of powers of z).

$$(1 - z)^{-u_\otimes} = \cfrac{1}{1 - uz - \cfrac{1uz^2}{1 - (u+2)z - \cfrac{2(u+1)z^2}{1 - (u+4)z - \cfrac{3(u+2)z^2}{\ddots}}}}$$

Setting $u = 1$ and $z = x$ gives a continued fraction for the factorials, $(1 - x)^{-1_\otimes} = x^\otimes$.

(S) A kth-order linear ordinary homogeneous differential equation,

$$b_kD^kf + b_{k-1}D^{k-1}f + \cdots + b_0f = 0$$

can be written $b(D)f = 0$. Similarly, a difference equation (also called a recurrence equation) can be written $b(E)s = 0$. Let $\grave{b} = b_k + b_{k-1}x^1 + \cdots + b_1x^{k-1} + b_0x^k$ be the *reverse* of $b = b_0 + b_1x + b_2x^2 + \cdots + b_kx^k$. Klarner [48] presents this fact: the solution s to $b(E)s = 0$ is

$$s = \frac{(\grave{b} * \text{inits})[0..k-1]}{\grave{b}}$$

where inits$= s_0 + s_1x + \cdots + s_{k-1}x^{k-1} = s[0..k-1]$ are the initial k elements of s. Also

$$b(E)s = 0 \Leftrightarrow b(D)\Lambda^{-1}s = 0$$

For example $E^2s + s = 0;\ s_0 = 0,\ s_1 = 1$ has solution $x/(1+x^2)$, and $\Lambda^{-1}s = \sin$, the Maclaurin expansion for sin, that is, the solution to $D^2s + s = 0;\ s_0 = 0,\ s_1 = 1$. Another example is $[z^n]C - 2u[z^{n-1}]C + [z^{n-2}]C = 0;\ C_0 = 1,\ C_1 = u$, where $[z^n]C$ is a Chebyshev polynomial [9] in u. Then, $b = 1 - 2uz + z^2 = \grave{b}$, and

$$C(z,u) = \frac{(\grave{b} * (1+uz))[0..1]}{\grave{b}} = \frac{1-uz}{1-2uz+z^2}$$

By the translation rules of item **J**, difference equations can be written using either E or Δ. The equation $b(E)s = 0$ transforms into $b(1+\Delta)s = 0$, or $\hat{b}(\Delta)s = 0$, where $\hat{b} = b \circ (1+x)$. The converse is $b = \hat{b} \circ (x-1)$, reflecting $\hat{b}(\Delta) = \hat{b}(E-1)$. Clearly, $b = b \circ (1+x) \circ (x-1)$.

(T) A sequence s is called *rational* if it is the quotient, $s = a/b$, of polynomials; it is called a *LODE solution*, written LODE(s), if it is a solution of a linear ordinary homogeneous difference equation; and it is called *recognizable* if it is the behaviour of a finite automaton. Then,

$$\text{rational}(s) \Leftrightarrow \text{LODE}(s) \Leftrightarrow \text{recognizable}(s)$$

Following [19], a finite automaton can be modelled as a system of linear equations over sequences, $E\,S = AS;\ S(0) = v$. Here, matrix A records transition labels connecting pairs of states, and S is a vector of sequences, one for each state, with initial values $S_i(0) = v_i$. The solution is $S = (Ax)^*v$ where

$$(Ax)^* = I + Ax + A^2x^2 + A^3x^3 + \cdots = \sum_{i \geq 0} A^i x^i$$

is the Kleene star. Notice that $(Ax)^*$ can be viewed as a matrix of sequences or as a sequence of matrices. Using $(Ax)^* = (I - Ax)^{-1}$, we get $(I - Ax)S = v$ and Cramer's rule applies: $S_i = \dfrac{\det(I - Ax)[i \leftarrow v]}{\det(I - Ax)}$ (column i replaced by v). Thus S_i is rational. Justification of the above equivalences is completed by noting that a LODE can be transformed into a system of linear equations. We remark that a quotient of polynomials, a/b, can also be written as the solution to a system of linear equations [71].

(U) Let $b = \det(xI - A) = b_0 + b_1x^1 + \cdots + b_{n-1}x^{n-1} + x^n$ be the characteristic polynomial of matrix A. The Cayley-Hamilton theorem [19, 49] can be stated as $b(A) = 0$, or as $b(E)(Ax)^* = 0$. This will hold if, taking the sequence of matrix powers, $(Ax)^*$, now as a matrix of sequences, we have $b(E)((Ax)^*)_{ij} = 0$, which in turn holds if $((Ax)^*)_{ij} = \dfrac{a}{\grave{b}}$. We have $\grave{b} = x^n b(1/x) = \det(I - xA)$. So we are done if we come up with an a such that

$$((Ax)^*)_{ij} = \frac{a}{\det(I - Ax)}$$

Let $M = Ax$, and $J = M^*\underline{j} = \underline{j} + MJ$, where $\underline{j}$ is the vector with 1 at position j and zero elsewhere. Then, $(I - M)J = \underline{j}$ and Cramer's rule delivers the a we are looking for:

$$(M^*)_{ij} = J_i = \frac{\det(I - M)[i \leftarrow \underline{j}]}{\det(I - M)}$$

(V) Consider the matrix exponential, $\exp \circ Ax = \Lambda^{-1}(Ax)^*$ as a matrix of sequences. We know that $(Ax)^*$ solves $E\,S = AS$, $S(0) = I$, and $(Ax)^*[0..k-1] = [I, A, A^2, \dots, A^{k-1}]$. Cayley-Hamilton says that $b(E)(Ax)^* = 0$, where b is the characteristic polynomial of A, of degree k, say. By uniqueness of solution, we have [54, 56, 53] $\phi = (Ax)^*$ if $b(E)\phi = 0$ and $\phi[0..k-1] = [I, A, A^2, \dots, A^{k-1}]$. Let S be a vector of sequences such that $b(E)S_i = 0$ and $S_i[0..k-1] = x^i$. Set

$$\phi = S_0 I + S_1 A + \cdots S_{k-1}A^{k-1} = \sum_{i=0}^{k-1} S_i A^i$$

Clearly $\phi[0..k-1] = [I, A, A^2, \dots, A^{k-1}]$, and $b(E)\phi = 0$ because

$$\sum_{j=0}^{k} b_j E^j \phi = \sum_{j=0}^{k} b_j \left(\sum_{i=0}^{k-1} E^j S_i A^i \right) = \sum_{i=0}^{k-1} \left(\sum_{j=0}^{k} b_j E^j S_i \right) A^i = \sum_{i=0}^{k-1} (b(E)S_i) A^i = 0$$

(W) The elements of the sequence $B = x/(\exp - 1) = [1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, \dots]$ are called *Bernoulli numbers*. The corresponding recurrence is calculated from $B \exp = B + x$:

$$n![x^n]B \exp = n![x^n](B + x); \qquad \sum_{k=0}^{n} \binom{n}{k} B_k = B_n + [n = 1]$$

Bernoulli numbers are used by Graham *et al* [32] for the most impressive of their deductions of the polynomial that sums squares – impressive because it defines the formulas for all powers at once. Let $S_{(n)}$ be the sequence such that $m![x^m]S_{(n)}$ is the sum of the mth powers of the naturals to $n - 1$. Then

$$\begin{aligned}
m![x^m]S_{(n)} &= \sum_{k=0}^{n-1} k^m = \sum_{k=0}^{n-1} m![x^m] \exp \circ kx \\
S_{(n)} &= \sum_{k=0}^{n-1} \exp^k = \frac{\exp^n - 1}{\exp - 1} = \frac{\exp \circ nx - 1}{\exp - 1} = B\,\frac{\exp \circ nx - 1}{x} \\
m![x^m]S_{(n)} &= \sum_{k=0}^{m} \frac{B_{m-k}}{(m-k)!} \frac{n^{k+1}}{(k+1)!} = \sum_{k=0}^{m} \binom{m}{k} B_{m-k} \frac{n^{k+1}}{k+1}
\end{aligned}$$

Now replace n by u and x by z to get the expression advertised in the introduction:

$$S = \frac{\exp \circ uz - 1}{\exp \circ z - 1}$$

Then $m![z^m]S$ is a polynomial of degree $m+1$ in u, and $(m![z^m]S)(n) = m![x^m]S_{(n)}$.

(X) Observe that $B_1 = -1/2$ is non-zero whilst all the other odd-degree coefficients of B appear to be zero. Perhaps if we make B_1 zero then we will have a sequence which can be *proved* to be even (i.e. with zeros at odd positions). Adding $\frac{1}{2}x$ to B cancels B_1:

$$C = B + \frac{x}{2} = \frac{x}{\exp -1} + \frac{x}{2} = \frac{x}{2}\frac{\exp +1}{\exp -1}$$

Recall $\coth = \dfrac{\exp + \exp \circ -x}{\exp - \exp \circ -x}$, so $C = \frac{x}{2}(\coth \circ \frac{x}{2})$, from which eveness, $C \circ -x = C$, can be deduced. Thus,

$$C = B + \frac{x}{2} = \sum_k \frac{B_{2k}}{(2k)!} x^{2k}$$

Now $C \circ 2x = x \coth$, and, using $x \cot = ix(\coth \circ ix)$ and $\tan = \cot -2\cot \circ 2x$, we get

$$x\cot = C \circ 2ix = \sum_k (-1)^k 2^{2k} \frac{B_{2k}}{(2k)!} x^{2k} \quad (9)$$

$$\tan = \sum_k (-1)^{k-1} 4^k (4^k - 1) \frac{B_{2k}}{(2k)!} x^{2k-1} \quad (10)$$

With a bit of analysis (reals, $\cot(x)$ an analytic function with period π, and uniqueness of series expansion), one can deduce another series for $x\cot$, due to Euler. The omitted analysis [50, 1] is hidden in the first equals sign:

$$x\cot(x) = 1 - 2\sum_{n=1}^{\infty} \frac{x^2}{n^2\pi^2 - x^2} = 1 - 2\sum_{n=1}^{\infty} \frac{x^2}{n^2\pi^2}\left(n^2\pi^2\right)^* = 1 - 2\sum_{k=1}^{\infty} \frac{x^{2k}}{\pi^{2k}} \sum_{n=1}^{\infty} \frac{1}{n^{2k}}$$

Equating coefficients with those in the expansion (9) yields, for $k > 0$,

$$[x^{2k}]x\cot = \frac{-2}{\pi^{2k}} \sum_{n=1}^{\infty} \frac{1}{n^{2k}} = (-1)^k 2^{2k} \frac{B_{2k}}{(2k)!}$$

Therefore, the values of Riemann's $\zeta(s) = \sum_{n \geq 1} \frac{1}{n^s}$ at even positive integers is given by

$$\zeta(2k) = (-1)^{k-1} 2^{2k-1} \frac{B_{2k}}{(2k)!} \pi^{2k}$$

(Y) The Formal Taylor Theorem may be expressed:

$$f \circ (u + z) = f \circ u + ((D\,f) \circ u)z + \frac{(D^2 f) \circ u}{2!} z^2 + \frac{(D^3 f) \circ u}{3!} z^3 + \cdots$$

Write $f \circ (u+z) = g_0 + g_1 z + g_2 z^2 + g_3 z^3 + \cdots$. Let $z = 0$, then $g_0 = f(u)$. Differentiate with respect to z: $D_z(f \circ (u+z)) = g_1 + 2g_2 z + 3g_3 z^2 + \cdots$. Note that $D_z(f \circ (u+z)) = f_1 + 2f_2(u+z) + 3f_3(u+z)^2 + \cdots = (D\,f) \circ (u+z)$. Let $z = 0$, then $g_1 = (D\,f) \circ u$. Differentiate again: $D^2(f \circ (u+z)) = 2g_2 + 3!g_3 z + \cdots$. Let $z = 0$, then $g_2 = ((D^2 f) \circ u)/2$. And so on. The Maclaurin expansion is the special case with $u = 0$, and the Taylor expansion of $x^n \circ (u+z)$ is an instance of the binomial theorem. Lipson [55] uses the theorem in the application of Newton's iterative root-finding algorithm to polynomial equations over sequences.

(Z) The following manipulations, originating with Lagrange, have a captivating charm. We adapt them from [32] to show that elementary sequence algebra plays a role through to the final chapter of that book (where, however things become more demanding). In the Formal Taylor theorem, let f be a polynomial, $z = 1$, $u = x$, and employ an operator style:

$$\begin{aligned} Ef &= f \circ (x+1) = f + (D\,f) + \frac{D^2 f}{2!} + \frac{D^3 f}{3!} + \cdots = [1 + D + \frac{D^2}{2!} + \frac{D^3}{3!} + \cdots]f \\ &= \exp(D) f \end{aligned}$$

Putting $\Delta = E - 1 = \exp(D) - 1$ together with $\Delta\Sigma f = f$, suggests $\Sigma = (\exp(D) - 1)^{-1}$. Then $B = x/(\exp - 1)$ applied to D is $D\Sigma$, so $\Sigma = D^{-1}B(D)$. Expanding this, and writing $\int$ for the first term D^{-1} (since $B_0 = 1$), gives a "template" version of the Euler-Maclaurin summation formula [32, 9].

$$\sum = \int + \sum_{k \geq 1} \frac{B_k}{k!} D^{k-1}$$

Now introduce limits:

$$\textstyle\sum_a^b f = \displaystyle\int_a^b f + \sum_{k \geq 1} \frac{B_k}{k!} D^{k-1} f \Bigg|_a^b \tag{11}$$

An application to x^2 gives yet another derivation [32] of the sum-of-squares formula:

$$\begin{aligned} \textstyle\sum_0^n x^2 &= \frac{1}{3}n^3 + \left(-\frac{1}{2}x^2 + \frac{1}{12}2x\right)\Bigg|_0^n \\ &= \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n \end{aligned}$$

The way the limits appear on the summation sign has significance: $\sum_0^n x^2 = \displaystyle\sum_{x=0}^{n-1} x^2$. The definite summation symbol follows the pattern of definite integration:

$$\begin{aligned} g = Df \ &\Rightarrow\ \int_a^b g \ = \ f\Big|_a^b \ = \ f(b) - f(a) \\ g = \Delta f \ &\Rightarrow\ \textstyle\sum_a^b g \ = \ f\Big|_a^b \ = \ f(b) - f(a) \ = \ \displaystyle\sum_{x=a}^{b-1} f(x+1) - f(x) \\ &\qquad\qquad = \ \sum_{x=a}^{b-1} \Delta f(x) \ = \ \sum_{x=a}^{b-1} g(x) \end{aligned}$$

Let's add $f(b)$ to both sides of (11) and separate out $B_1 = -1/2$:

$$\sum_{x=a}^{b} f = \int_a^b f + \frac{1}{2}(f(b) + f(a)) + \sum_{k\geq 2} \frac{B_k}{k!} D^{k-1} f \Bigg|_a^b \tag{12}$$

The Euler-Maclaurin formula can also be applied to non-polynomial functions. Let us illustrate this, without justification. To compute $\sum_{x=1}^{\infty} 1/x^2$, set $S_9 = \sum_{x=1}^{9} \frac{1}{x^2} = 1.5397677310$ and the apply (12) to $g = 1/(x+10)^2$:

$$\sum_{x=1}^{\infty} \frac{1}{x^2} = S_9 + \sum_{x=0}^{\infty} g(x) = S_9 + \int_0^{\infty} g + \frac{1}{2} g(0) + \sum_{k\geq 2} \frac{B_k}{k!} D^{k-1} g \Bigg|_0^{\infty}$$

Applying the formula up to B_4 gives $\zeta(2) = \pi^2/6 \approx 1.64493407$.

4 A programming delight

McIlroy [57, 58] has gifted us some "tiny gems" of program definitions for implementing sequence manipulations. The definitions are written in Haskell, and are effortlessly derived by mathematical reasoning. A textbook introduction appears in [18], and related definitions are presented in [45]. We want to entice the reader to type up and experiment with the Haskell code (a file containing the definitions is available on request from the author). The code has been tested in the Haskell GHCi system, and also in the Hugs98 system (an older system, but well-suited to beginners). Both GHCi and Hugs98 are freely available on the web at www.Haskell.org.

In this section we present all of the definitions, thus duplicating some of the contents of [57, 58]; however, there are some modifications and additions. The fact that the definitions have no pre-requisites, other than the standard Haskell Prelude, means that one can take "deep" ownership, building things from the ground up. This contrasts to using a sophisticated computer algebra system – something perhaps for the newcomer to move on to with greater appreciation.

Haskell [68] has evolved to be a fairly large and sophisticated language, but we shall stick to a modest subset. It is expected that the reader can comprehend Haskell from examples. The language gives types to objects and variables, and within context, the most general type is used. Haskell's lists represent sequences; in Haskell, head-tail decomposition $s_0 + xs'$ becomes `s0:s'`. Here are some list-processing functions:

```
take n _ | n<=0     = []
take _ []           = []
take n (s0:s')      = s0: take (n-1) s'
map f []            = []
map f (s0:s')       = f s0 : map f s'
iterate f z         = z: iterate f (f z)
foldr f z []        = z
foldr f z (s0:s')   = f s0 (foldr f z s')
scanl op q s        = q: (case s of
                            []    -> []
                            s0:s' -> scanl op (op q s0) s')
zip (s0:s') (t0:t') = (s0, t0): zip s' t'
zip _ _             = []
zipWith op s t      = [op sn tn | (sn,tn) <- zip s t]
```

These definitions implement the following functions which feature in the algebra of program calculation [7, 6]:

$$\begin{array}{lcl}
\mathsf{take}\ n\ s & = & [s_0, s_1, \ldots s_{n-1}] \\
\mathsf{map}\ f\ s & = & [fs_0, fs_1, fs_2, \ldots] \\
\mathsf{iterate}\ f\ z & = & [z, f\ z, f(fz), f^3 z, \ldots] \\
\mathsf{foldr}\ f\ z\ s & = & f\ s0\ (f\ s1\ (f\ s2\ (\ldots\ z)\ldots) \\
\mathsf{scanl} \oplus q\ s & = & [q, q \oplus s_0, (q \oplus s_0) \oplus s_1, ((q \oplus s_0) \oplus s_1) \oplus s_2, \ldots] \\
\mathsf{zip}\ s\ t & = & [(s_0, t_0), (s_1, t_1), (s_2, t_2), \ldots] \\
\mathsf{zipWith} \odot s\ t & = & [s_0 \odot t_0, s_1 \odot t_1, s_2 \odot t_2, \ldots]
\end{array}$$

The types deduced are

```
take    :: Int -> [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
iterate :: (a -> a) -> a -> [a]
foldr   :: (a -> b -> b) -> b -> [a] -> b
scanl   :: (a -> b -> a) -> a -> [b] -> [a]
zip     :: [a] -> [b] -> [(a,b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Type expressions are built from type names such as `Int`, type variables such as `a`, and type constructors such as `->` (which associates to the right). Two more examples of type constructors are: `[a]` is the type for lists of objects of type `a`, and `[(a,b)]` is the type for lists of pairs of objects. Clearly, functions can take functions as arguments, in which case they are called *higher-order*. Lazy evaluation is used, so that for example, `scanl` will produce the first element, `q`, of the result without needing to know anything about its list argument, `s`. The definition of `zipWith` illustrates the so-called *list comprehension*. An alternative definition uses `map` and uncurry:

```
    uncurry        :: (a -> b -> c) -> (a,b) -> c
    uncurry op p   = op (fst p) (snd p)
    zipWith op s t = map (uncurry op) (zip s t)
```

The partner to `uncurry` is `curry f x y = f (x,y)`. An alternative definition illustrates use of a lambda expression: `curry f = \x y -> f (x,y)`. The reader may like to supply the type. These two functions are named after the Curry-Howard isomorphism. The Haskell Standard Prelude defines `zipWith` without using `zip`, and then defines `zip = zipWith (,)`.

The following specifies that a type `a` is classified as `Num` if it has the operations (or methods) listed here:

```
class Num a where
    (+), (-), (*)        :: a -> a -> a
    negate, abs, signum  :: a -> a
    fromInteger          :: Integer -> a
    x - y                = x + negate y
    negate x             = 0 - x
```

All of the foregoing definitions are in the Standard Prelude. From here on, the code needs to be supplied. The program file starts with a few specified lines: the first line hides the Prelude definition of `cycle` because we are going to re-define it for other purposes; the second line says we need rational numbers; the third line gives the order in which to resolve ambiguity in numerical data.

```
    import Prelude hiding (cycle)
    import Data.Ratio
    default (Integer, Rational, Double)
```

We start by declaring how sequences, `[a]`, become an instance of `Num`. A prerequisite is that `a` is an instance of `Eq` and `Num`, indicated by `(Eq a, Num a) =>`. The definition of `(-)` is derived from `negate`.

```
    instance (Eq a, Num a) => Num [a] where
      negate             = map negate
      f+[]               = f
      []+g               = g
      (f0:f')+(g0:g')    = f0+g0 : f' + g'
      []*_               = []
      (0:f')*g           = 0 : f'*g
      _*[]               = []
      (f0:f')*g@(g0:g') = f0*g0 : (f0*|g' + f'*g)
      fromInteger c      = [fromInteger c]
      abs _              = error "abs not defined on sequences"
      signum _           = error "signum not defined on sequences"
```

Observe that addition is not defined by `f+g = zipWith (+) f g` (why?). Convolution product is derived from (1), but there are some things to note. Firstly, if $f_0 = 0$ then the zeroth term of the result is 0 and is delivered immediately. This may be regarded as a controversial quirk, but it enables certain equations to be used directly, as in the following (Catalan) example – in examples, definitions (which are placed in a program file) are interspersed with interactive requests for expression evaluation, indicated by the prompt "> ".

```
x :: Num a => [a]
x = [0,1]
c = 1 + x*c^2
> take 8 c
[1,1,2,5,14,42,132,429]
```

Secondly, the notation `g@`, is read as "`g` as". Thirdly, there are clauses for finite sequences – the empty list behaves here like zero (but note that 0 is embedded as `[0]`). Fourthly, the term $f_0g' = [f_0]g'$ becomes an explicit scalar product using `*|`, defined as an infix operator with precedence 7 (the same as `*`, and higher than `+`). The definition contains `(a*)` which illustrates the creation of a function by partial application of an operator (called *sectioning*).

```
infix 7 *|
(*|) :: Num a => a -> [a] -> [a]
a *| f = map (a*) f
```

A function like `map` is said to be *polymorphic* because any type can be assigned to its type variables (subject to consistency). By contrast, scalar multiplication `(*|)` has a *qualified* (*constrained* or *parametric*) polymorphic type: its type variable can range over only instances of class `Num`. The type stated could be omitted because it can be inferred due to the presence of `*`. On the other hand, if the explicit type given to `x` above was omitted, then Haskell would infer `x::[Integer]` and this is a *monomorphic* type which would restrict the use of `x`. With the qualified polymorphic type, `x` can appear in an expression where a sequence of elements of type `N` is expected, as long as `N` is an instance of `Num`. Any instance `N` of `Num` must provide a `fromInteger` method that shows how to embed integers into `N`, so `x` would be interpreted as `[N.fromInteger 0, N.fromInteger 1]`.

The `Num` class invites comparison with the specification of the signature of a ring. Likewise, Haskell's `Fractional` class may be compared to a ring-with-division, because a (partial) division operator `(/)`, or a multiplicative inverse (`recip`), is required. Rational numbers form the archetypal instance of `Fractional`, and any instance `F` must show how to embed the rationals in `F` by defining `fromRational :: Rational -> F`. Division on sequences, f/g, requires calculating the quotient q satisfying $f = qg$:

$$\begin{aligned} f_0 + xf' &= (q_0 + xq')g = q_0g + xq'g = q_0g_0 + x(q_0g' + q'g) \\ f_0 = q_0g_0 \quad &; \quad f' = q_0g' + q'g \\ q_0 = f_0/g_0 \quad &; \quad q' = (f' - q_0g')/g \\ q &= f_0/g_0 + x(f' - q_0g')/g \end{aligned}$$

Now we can say, at least approximately, how sequences become a ring-with-division:

```
instance (Eq a, Fractional a) => Fractional [a] where
  recip f           = 1/f
  _/[]              = error "divide by zero."
  []/_              = []
  (0:f')/(0:g')     = f'/g'
  (_:f')/(0:g')     = error "divide by zero"
  (f0:f')/g@(g0:g') = let q0=f0/g0 in q0:((f' - q0*|g')/g)
  fromRational c    = [fromRational c]
```

These simple definitions confront us with some of the difficulties in coding a satisfactory division operation that works for both finite and infinite sequences. One should investigate questions like: are $f/g = f * (1/g)$ and $f/f = 1$ faithfully implemented? To keep things simple, compromises have to be made.

Arithmetic and the convolution product rule are used to calculate a head-tail definition for square root, $\sqrt{f}$. The starting point is $f = \sqrt{f}\sqrt{f}$, and we calculate $(\sqrt{f})_0$ and $\sqrt{f}'$:

$$\begin{aligned}
f_0 = (\sqrt{f}\sqrt{f})_0 &= (\sqrt{f})_0(\sqrt{f})_0 \\
(\sqrt{f})_0 &= \sqrt{f_0} \\
f' = (\sqrt{f}\sqrt{f})' &= \sqrt{f_0}\sqrt{f}' + \sqrt{f}'\sqrt{f} \\
\sqrt{f}' &= f'/(\sqrt{f_0} + \sqrt{f}) \quad (f_0 \neq 0)
\end{aligned}$$

We shall trivialise $\sqrt{f_0}$ and restrict square root to fractional sequences with constant term 1. In the following code, the first clause is suggested by the identity $\sqrt{x^2 f} = x\sqrt{f}$, and the more general $\sqrt{x^{2n} f} = x^n\sqrt{f}$ is handled by recursion. An alternative definition of square root is derived in [58] by differentiating $r^2 = f$, rearranging and then integrating (which the reader may like to try).

```
sqroot (0:0:f'') = 0:sqroot f''
sqroot f@(1:f')  = 1:(f'/(1+sqroot f))
```

Sequence composition, $f \circ g = \sum_n f_n g^n$, is expanded thus:

$$\begin{aligned}
f \circ g &= f_0 + f_1 g^1 + f_2 g^2 + f_3 g^3 + \cdots \\
&= f_0 + g(f' \circ g) = f_0 + (g_0 + xg')(f' \circ g) \\
&= f_0 + g_0(f' \circ g) + xg'(f' \circ g)
\end{aligned}$$

When f is infinite, $g_0(f' \circ g)$ is not computable unless $g_0 = 0$. However, $f \circ g$ is computable for $g_0 \neq 0$ when f is finite ($p \circ [a] = [p(a)]$, p a polynomial). So we admit a potentially non-terminating clause and it is up to us to use it with care:

```
[] `o` _                = []
(f0:f') `o` g@(0:g')    = f0: g'*(f' `o` g)
(f0:f') `o` g@(g0:g')   = [f0] + (g0*|(f' `o` g))+
                            (0:g'*(f' `o` g))
```

The definition $f \circ g = \sum_n f_n g^n$ reveals x to be a left and right identity of composition. Composition distributes leftwards through sum, product, and quotient.

To calculate the converse, $g = f^\circ$, expand the composition $f \circ g = x$:

$$f_0 + xg'(f' \circ g) = x = 0 + x1$$

Hence, $g'(f' \circ g) = 1$, and

$$g_0 = 0; \qquad g' = 1/(f' \circ g)$$

The program code is:

```
converse(0:f') = g where g = 0: 1/(f' `o` g)
```

For the reciprocal of $f' \circ g$ to be defined, it is necessary that f'_0 is invertible, which entails that f' is invertible. The set of such "conversible" sequences forms a group, $(\mathbb{S}_C, \circ, ()^\circ, x)$.

The transforms Λ and Λ^{-1} are given names `e2o` and `o2e`, respectively [18]. Here they are, together with some other useful sequences:

```
e2o f     = zipWith (*) f facs
o2e f     = zipWith (/) f facs
from      :: Num a=>a->[a]
from      = iterate (+1)
nats, pos, zeros, facs :: Num a=>[a]
nats      = from 0
pos       = from 1
zeros     = 0:zeros
facs      = scanl (*) 1 pos
```

Differentiation and integration enjoy the appropriately succinct definitions,

```
deriv f = zipWith (*) pos (tail f)
integ f = 0:zipWith (/) f pos
```

The definitions $D\ \exp = \exp;\ \exp_0 = 1$ and $x^* = 1 + xx^*$ have the following solutions in Haskell. An `x` is affixed to prevent name clashes with existing names (for example `exp` in Haskell implements the function e^x). One can test $x^* = \Lambda \exp$ by checking the first few terms of their difference.

```
expx  :: (Eq a,Fractional a) => [a]
expx  = 1 + integ expx
starx :: (Eq a, Num a) => [a]
starx = 1 : starx
> takeW 6 (starx - e2o expx)
[0,0,0,0,0,0]
```

A rational a/b is presented in Haskell as `a%b`. The elements of `expx` are rationals, and `e2o` removes the factorial divisors, yielding `[1%1,1%1, ...]`. The following defines `takeW n` which is `take n` preceded by the conversion of whole rationals into integers (the (.) is function composition, and `properFraction` is in the Prelude).

```
makeWhole r  = case properFraction r of
                (n,0)         -> n
                otherwise     -> error "not whole"
makeAllWhole = map makeWhole
takeW n      = take n . makeAllWhole
```

Table 7 contains further core sequences defined by differential equations. All should be given the type `(Eq a, Fractional a) => [a]`, like `expx` above. The core sequence x^*, which we defined earlier, could be defined by `starx = 1+integ (starx^2)`, since $D\,x^* = (x^*)^2$; $x^*_0 = 1$ (but its elements would then be fractional). Also, one can define `binx r  = 1+integ (r*(binx r)/(1+x))` since $D\,b_r = (r/(1+x))b_r$; $b_r(0) = 1$. For two more examples, let us calculate definitions for arctan and arcsin.

$$D\ \tan^\circ = ((1+\tan^2)\circ\tan^\circ)^{-1} = 1/(1+x^2)$$

$$D\ \sin^\circ = (\cos\circ\sin^\circ)^{-1} = \left(\left(\sqrt{1-\sin^2}\right)\circ\sin^\circ\right)^{-1} = 1/\sqrt{1-x^2}$$

The latter uses the Pythagorean identity, $\sin^2+\cos^2 = 1$, which follows from the defining differential equations. Taking initial values into consideration, the solutions rendered in Haskell are immediate:

```
atanx   = integ (1/(1+x^2))
asinx   = integ (1/(sqroot (1-x^2)))
```

Here are checks of $\exp = (\sec+\tan)\circ\mathrm{gd}$ (gd is the Guddermanian function) and $D\ \sin^\circ = 1/\sqrt{1-x^2}$.

```
> takeW 6 (expx - ((secx + tanx) `o` gdx))
[0,0,0,0,0,0]
> takeW 6 (deriv (converse sinx) - (1/(sqroot (1-x^2))))
[0,0,0,0,0,0]
```

A bivariate sequence, $b(z,u)$, may be regarded as a (potentially doubly-infinite) matrix, $t = (b_{i,j})$, of coefficients of $z^i u^j$. It is implemented as a univariate sequence, s, of (homogeneous) polynomials such that s_n is the diagonal, $[b_{0,n},\ b_{1,n-1},\ldots b_{n,0}]$ of t. Thus, $b_{0,n}z^0u^n + b_{1,n-1}z^1u^{n-1}+\cdots+b_{n,0}z^nu^0$ is represented by $s_n = b_{0,n}x^0+b_{1,n-1}x^1+\cdots+b_{n,0}x^n$ (z becomes x, and u is redundant). So, $[u^kz^{n-k}]b = [x^k][x^n]s$, equivalently $[u^kz^n]b = [x^n][x^{n+k}]s$. The following depicts the t to s map:

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & \cdots \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & \cdots \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & \cdots \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & \cdots \end{bmatrix} \mapsto \begin{bmatrix} b_{0,0} & & & \\ b_{0,1} & b_{1,0} & & \\ b_{0,2} & b_{1,1} & b_{2,0} & \\ b_{0,3} & b_{1,2} & b_{2,1} & b_{3,0} \end{bmatrix}$$

`lgnx    = integ (1/(1+x))`	$D\lg\text{n} = 1/(1+x);\ \lg\text{n}_0 = 0$
`sinx    = integ cosx`	$D\sin = \cos;\ \sin_0 = 0$
`cosx    = 1-integ sinx`	$D\cos = -\sin;\ \cos_0 = 1$
`tanx    = integ (1+tanx^2)`	$D\tan = 1+\tan^2;\ \tan_0 = 0$
`secx    = 1+integ (secx * tanx)`	$D\sec = \sec\tan;\ sec_0 = 1$
`coshx   = 1+integ sinhx`	$D\cosh = \sinh;\ \cosh_0 = 1$
`sinhx   = integ coshx`	$D\sinh = \cosh;\ \sinh_0 = 0$
`tanhx   = integ (1-tanhx^2)`	$D\tanh = 1-\tanh^2;\ \tanh_0 = 0$
`gdx     = integ (1/coshx)`	$D\,\text{gd} = 1/\cosh;\ \text{gd}_0 = 0$

Table 7: Some core Sequences

The ring-with-division and square root operations pertaining to $b(z,u)$ are isomorphically transferred to the diagonal representation s. The representations for u and z are, $u = 0u^0z^0 + (1u^1z^0 + 0u^0z^1) \cong [0x^0, 1x^0 + 0x^1] = [[0],[1,0]]$ and $z = 0 + (0u + 1z) \cong [0, 0+1x] = [[0],[0,1]]$. Here is $(u+z)^*$ represented by s=`pascal` in Haskell,

```
u,z, pascal :: (Eq a, Num a) => [[a]]
u      = [[0],[1,0]]
z      = [[0],[0,1]]
pascal = starx `o` (u+z)
> take 6 pascal
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
```

This displays $[u^kz^{n-k}](u+z)^* = [x^k][x^n]s = \binom{n}{k}$. Commonly, we want to show a portion of t, such that $[x^k][x^n]t = b_{n,k} = [x^n][x^{n+k}]s$, or perhaps more commonly, such that $[x^k][x^n]t = n![u^kz^n]b = n![x^n][x^{n+k}]s = [x^n]\Lambda[x^{n+k}]s$. For example, let

$$b = \exp\circ(z+uz) = (\exp\circ z)(\exp\circ uz) = \sum_n\sum_{k=0}^{n}\frac{z^{n-k}}{(n-k)!}\frac{u^kz^k}{k!} = \sum_n\sum_{k=0}^{n}\binom{n}{k}\frac{u^kz^n}{n!}$$

Then $n![u^kz^n] = \binom{n}{k}$. The functions `unDiag` and `unDiage2o` transpose the s-representation into the desired t-representation (the reverse of the $t \mapsto s$ map depicted above). The function `unDiage2o` first removes factorial divisors associated with z. The functions `select` and `selectW` take an argument $[n_0, n_1, \ldots, n_m]$ saying what length to take from rows 0 to m of t. The version `selectW` converts whole rationals to integers. Bivariate counting sequences, $b_{n,k}$, are typically zero for $k > n$, and from these we select a lower triangular section. The `schroeder` sequence from section 3, item **C**, is an example which is ordinary in u and z, whilst `ebinom` below is exponential in z. The sequence `powerSums` of polynomials for summing powers has a polynomial of order $n+1$ at position n, so a lower trapezium section [2..4] is selected.

```
list, pluralList :: (Eq a, Num a) => [a]
list       = starx
pluralList = list - x - 1
```

```
schroeder  =  z + u*(pluralList ‘o‘ schroeder)

ebinom =  expx ‘o‘ (z+u*z)
> selectW [1..6] (unDiage2o ebinom)
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
powerSums  = ((expx ‘o‘ (u*z))-1)/((expx ‘o‘ z)-1)
> select [2..4] (unDiage2o powerSums)
[[0 % 1,1 % 1],[0 % 1,(-1) % 2,1 % 2],[0 % 1,1 % 6,(-1) % 2,1 % 3]]
```

A number of supporting functions are needed. The `unDiag` function expects its argument to be perfectly triangular, so `padTri` is first applied to fill out the triangle with zeros if necessary. Then `transpose m` detaches the heads of the rows of `m`, which make up the first column, `c`, and this becomes the first row of the result. A recursive invocation transposes the remaining sub-matrix, `m’`.

```
select s t   = zipWith take s t
selectW s t  = map makeAllWhole (zipWith take s t)
unDiag       :: Num a=> [[a]]->[[a]]
unDiag       = transpose . padTri
unDiage2o    :: Fractional a=> [[a]]->[[a]]
unDiage2o    = unDiag . (map e2o)
padTri t     = zipWith padRight t [1..]
padRight r k = r++(take (k-(length r)) zeros)

transpose []   = []
transpose m    = c:transpose m’
  where (c,m’) = foldr detachHead ([],[]) m
        detachHead   ([r0]) b  = (r0:fst b,snd b)
        detachHead   (r0:r’) b = (r0:fst b,r’:snd b)
```

Differentiation with respect to z (or u) is performed by `dz` (or `du`). Below are the definitions, plus a test based on the set partitions recurrence from section 3, item **L**. Instead of using `allZeros [1..6]`, the reader may wish to simply use `selectW [1..6]` and view the zeros (incidentally, one would then observe that the diagonal representation is not always a perfect triangle).

```
dz s = map deriv (tail s)
du s = map (reverse . deriv . reverse) (tail (padTri s))
set, nonEmptySet, emptySet :: (Eq a, Fractional a) => [a]
set          = expx
emptySet     = 1
nonEmptySet  = set - emptySet
parts        = set ‘o‘ (u*(nonEmptySet ‘o‘ z))

allEq c r    = foldr (\a b-> (a==c) && b) True r
allZeros s t = allEq True (map (allEq 0) (select s t))
> allZeros [1..6] ((dz parts) - (u*parts +u*du parts))
True
```

5 Exercising the implementation

As it stands, the implementation facilitates a great range of experimentation. We will demonstrate a few concrete examples. It will be seen that the implementation is a valuable assistant in the study of otherwise theoretical material.

The definitions in tables 4 and 5 transliterate into Haskell. Here are some examples (see items **B** and **D**). One has to be vigilant about when to use `e2o, take, takeW, select, selectW, unDiag` and `unDiage2o`. It is not necessary to give type declarations to all definitions, but it will be necessary for some. We leave that as a trial-and-error exercise.

```
cycle, perm :: (Eq a, Fractional a) => [a]
perm  = starx
lg g  = lgnx ‘o‘ (g-1)
cycle = lg starx
> takeW 6 (perm - (set ‘o‘ cycle))
[0,0,0,0,0,0]
cayleyTree            = x*(set ‘o‘ cayleyTree)
connectedAcyclicGraph = cayleyTree - cayleyTree^2 / 2
> takeW 8 (e2o connectedAcyclicGraph)
[0,1,1,3,16,125,1296,16807]

infix 7 |^
(|^)      :: (Eq a, Fractional a) => [a] -> a -> [a]
f |^ r    = expx ‘o‘ (r *| lg f)
legendre  = (1-2*u*z+z^2) |^ [-1%2]
> select [1..4] (unDiag legendre)
[[1 % 1],[0 % 1,1 % 1],[(-1) % 2,0 % 1,3 % 2],
[0 % 1,(-3) % 2,0 % 1,5 % 2]]
hermite   = expx ‘o‘ (2*u*z-z^2)
> select [1..4] (unDiage2o hermite)
[[1 % 1],[0 % 1,2 % 1],[(-2) % 1,0 % 1,4 % 1],
[0 % 1,(-12) % 1,0 % 1,8 % 1]]
```

No attention has been paid to efficiency or prettiness of results – all the computations are expected to work on small examples, resulting in small results. Endless examples could be given related to items **A-Z**. We must choose only a few, and we aim for variety.

The factorials are defined in the previous section using `scanl`; below they are generated directly from their differential equation, by a continued fraction recurrence, and by shuffle inverse (see items **E** and **P**). Shuffle product can also be defined by $f \otimes g = \Lambda(\Lambda^{-1}f * \Lambda^{-1}g))$ (but that involves rationals, even when f and g are integer sequences).

```
fac = 1+x*fac + x^2*(deriv fac)
> take 6 fac
[1,1,2,6,24,120]
cf_fac  = 1/(cfdenom 1)
```

```
 where cfdenom n = 1 - x*(2*n-1)-x^2*n^2*(1/(cfdenom (n+1)))
> takeW 6 cf_fac
[1,1,2,6,24,120]
infix 7 |><|    -- shuffle product
f@(f0:f') |><| g@(g0:g') = (f0 * g0): ((f' |><| g)+(f |><| g'))
_ |><| []                = []
[] |><| _                = []
shInv f@(f0:f') = (1/f0): (-f' |><| ((shInv f) |><| (shInv f)))
> takeW 6 (shInv (1-x))
[1,1,2,6,24,120]
```

The Newton transform, $(\mathcal{N}, \mathcal{N}^{-1})$, is an isomorphism between the Hadamard and infiltration rings (see item **K**). It is implemented here by (`i2h, h2i`), with a recursive variant, `rh2i`. We also translate the definitions of Δ and Σ directly to `delta` and `sigma`. Later, we shall define another version of Σ, named `prefixSums`, which produces a finite result on a finite sequence. Let's throw into our test the ubiquitous fibonacci sequence, defined by $f_{n+2} - f_{n+1} - f_n = 0$; $f_0 = f_1 = 1$. The first part can be re-expressed $b(E)f = 0$, where $b = x^2 - x - 1$, with solution $f = \frac{(\grave{b} * [1,1])[0..1]}{\grave{b}} = 1/(1 - x - x^2)$ (see item **S**). For illustration, we let Haskell take the last step.

```
h2i s          = (1/(1+x)) |><| s
i2h s          = (1/(1-x)) |><| s
rh2i s@(s0:s') = s0: rh2i (delta s)
delta s        = (tail s) - s
sigma s        = x*starx*s
recur          :: (Eq a, Num a) => a -> [a]
recur a        = a:recur a
fib            = (take 2 (rb*[1,1]))/rb where rb = reverse (x^2-x-1)
> takeW 10 (recur 1 + sigma (delta fib))
[1,1,2,3,5,8,13,21,34,55]
> takeW 10  (i2h (h2i fib))
[1,1,2,3,5,8,13,21,34,55]
```

The Hadamard product, `|*|`, and the infiltration product, `|^|` (and `infProd`) are now introduced, and testing them is left as an exercise.

```
infix 7 |*|
f |*| g = zipWith (*) f g

infix 7 |^|
f@(f0:f') |^| g@(g0:g') = (f0*g0): ((f'|^|g)+(f|^|g'))+(f'|^|g')
_ |^| []               = []
[] |^| _               = []

infProd f g = h2i (i2h f |*| i2h g)
```

Translation to and from falling factorial polynomials can be exercised as follows. There are, of course, more efficient ways of generating the data used here (`cycles, parts`), but we stick to a simple translation of the mathematical definitions. In the first test, we use the fact (item **N**) that $[0, 1, 5, 14, 30, 55, \ldots]$ is representable by a polynomial of degree 3. The second test compares falling factorials and cycle numbers.

```
cycles        = set ‘o‘ (u* (cycle ‘o‘ z))
fall n m      = product [n-i | i<-take m nats]
alt           :: Num a => a->[a]
alt r         = r:alt (-r)
altMat m      = zipWith op (alt 1) m
                where op sign r = zipWith (*) (alt sign) r
monom2FacPoly = select [1..] (unDiage2o parts)
facMonom2Poly = select [1..] (altMat (unDiage2o cycles))

toFacPoly p   = sum (zipWith (*|) p monom2FacPoly)
fromFacPoly p = sum (zipWith (*|) p facMonom2Poly)
squaresFacPoly= o2e (take 4 (h2i [0,1,5,14,30,55]))
> fromFacPoly squaresFacPoly
[0 % 1,1 % 6,1 % 2,1 % 3]
> [fall x i | i<- [0..5]] == take 6 facMonom2Poly
True
```

The Maclaurin and Taylor expansions (item **Y**) can be coded and tested:

```
maclaurin f = o2e (map head (iterate deriv f))
taylor f    = map o2e (zp (map (‘o‘ u) (iterate deriv f)))
              where zp (g0:g’) = g0 + z* (zp g’)

bsinx, tsinx :: [[Rational]]
bsinx = sinx ‘o‘ (u+z)
tsinx = taylor sinx
> select [1..8] bsinx == select [1..8] tsinx
True
```

Let us move beyond the **A-Z** items and look at some other examples. The Logan polynomials [32, sect. 6.5], have the tangent numbers as constant terms. Here they are, defined by a closed expression, $(\sin \circ z + u \cos \circ z)/(\cos \circ z - u * \sin \circ z)$, and by an iteration.

```
logan = (((sinx ‘o‘ z)+u*(cosx ‘o‘ z))/
        ((cosx ‘o‘ z)-u*(sinx ‘o‘ z)))
> selectW [2..5] (unDiage2o logan)
[[0,1],[1,0,1],[0,2,0,2],[2,0,8,0,6]]
loganPolys = iterate (\p -> (1+x^2) * deriv p) x
> take 4 loganPolys
[[0,1],[1,0,1],[0,2,0,2],[2,0,8,0,6]]
```

The Entringer triangle, E [80, 77], has the tangent numbers on the first column (disregarding the first element), and the secant numbers on the diagonal. Below, the triangle is generated first by a backwards and forwards (boustrophedonic) computation of partial sums, then as the diagonal (homogeneous) presentation of coefficients of $A = (\sin \circ u + \cos \circ u)/\cos \circ (u + z))$. The bivariate A is exponential in both u and z, and $E_{n,k} = \dfrac{[u^{n-k}z^n]A}{(n-k)!n!}$ [32, ex. 6.75]. Forward partial sums are prefix sums. Earlier, in item **I**, we met $\Sigma = xx^*$ for computing them, and we used this above to define `sigma`. But that operator always results in an infinite sequence, even when applied to a finite one. So here we use a different definition.

```
prefixSums       = scanl (+) 0
suffixSums       = reverse.prefixSums.reverse
alternate f g a = a:alternate g f (f a)
entringer        = alternate suffixSums prefixSums [1]
> take 7 entringer
[[1],[1,0],[0,1,1],[2,2,1,0],[0,2,4,5,5],[16,16,14,10,5,0],
[0,16,32,46,56,61,61]]
gkpEnt =   (((sinx `o` u) + (cosx `o` u))/(cosx `o` (u+z)))
ue2o   :: Fractional a => [a]->[a]
ue2o   = reverse.e2o.reverse
> map makeAllWhole (take 7 (map e2o (map ue2o gkpEnt)))
[[1],[1,0],[0,1,1],[2,2,1,0],[0,2,4,5,5],[16,16,14,10,5,0],
[0,16,32,46,56,61,61]]
```

The Moessner sieve generates the sequence $M_r = [1^r, 2^r, 3^r, 4^r, \ldots]$, given a positive integer r. Kozen and Silva [52] cite a variety of proofs including sequence-calculational [38] and coinduction methods [63]. Let us see how easily we can implement some of the computations in [52]. There it is shown that the Moessner procedure can be described as the computation of a succession of bivariate sequences, $b_n(z, u)$, usefully represented in diagonal (homogeneous) form, s_n, and that $[u^0z^r]b_n = [x^r][x^r]s_n = n^r$. The sequence of triangles begins with Pascal's triangle, $b_0 = p = (u + z)^*$, represented in diagonal form by s_0. Then the triangle-to-triangle step, $s_n \mapsto s_{n+1}$, is: take the row $\rho = [x^r]s_n = [\rho_0, \rho_1, \ldots, \rho_r]$, representing $h_n(z, u) = \rho_0u^rz^0 + \rho_1u^{r-1}z^1 + \cdots + \rho_ru^0z^r$, and compute s_{n+1} representing $b_{n+1} = h_n(z, 1) * p = (\rho_0z^0 + \rho_1z^1 + \cdots + \rho_rz^r) * p$. Here is the computation of the first three triangles, followed by the selection of $n^r = [x^r][x^r]s_n$ =`sn!!r!!r` from the first 5 triangles. The function `x2z` converts $\rho = [x^r]s_n = \rho_0 + \rho_1x + \cdots + \rho_rx^r$ to $h_n(z, 1)$ in bivariate diagonal form (there are many ways of doing this).

```
x2z rho      = sum (zipWith (\c zn->[[c]]*zn) rho zPowers)
               where zPowers = (iterate (*z) 1)
moessnerT r = iterate (\s -> (x2z (s!!r))*pascal) pascal
> take 3 (map (take 5) (moessnerT 4))
[[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]],
[[1],[1,5],[1,6,11],[1,7,17,15],[1,8,24,32,16]],
[[1],[1,9],[1,10,33],[1,11,43,65],[1,12,54,108,81]]]
nats2Power r = [sn!!r!!r | sn <- moessnerT r]
> take 5 (nats2Power 4)
[1,16,81,256,625]
```

The function `moessnerT` is easily changed to one which produces the sequence $[x^r]s_n$

representing $h_n(z,1)$, and this makes way for a generalisation. The iteration step is $[x^r]s_n \mapsto [x^r]s_{n+1}$, and the iteration starts with $[x^r]s_0 = 1$, representing $h_0(z,1)$.

```
moessnerH r = iterate (\rho -> ((x2z rho)*pascal)!!r) 1
> take 4 (map (take 5) (moessnerH 4))
[[1],[1,4,6,4,1],[1,8,24,32,16],[1,12,54,108,81]]
```

Kozen and Silva generalise Moessner's theorem to encompass theorems by Long and Paasche [52]. In the generalised implementation, there are two new parameters, $h_0(z,1)$ (regarded as univariate, to be converted to bivariate by x2z), and d, a sequence $[d_0, d_1, \ldots]$ of non-negative integers. The iteration step implements the following recurrence, in which the final subscript indicates the selection of the homogeneous polynomial of degree (deg h_n) + d_n:

$$h_{n+1}(z,u) = [h_n(z,1) * p]_{(\deg\ h_n)+d_n}$$

Thus, rather than a simple iteration, we scan along d, because step n requires d_n. Let c_n, $n > 0$, be the leading coefficient of $h_n(z,1)$ (which we extract using last, since the highest-order coefficient is at the end). The generalised theorem entails: (a) when $h_0(z,1) = 1$ and $d = [r,0,0,\ldots]$ we get Moessner's result, $c_n = n^r$; (b) when $h_0(z,1) = b + (a-b)z$ and $d = [r,0,0,\ldots]$, we get Long's result, $c_n = (a + (n-1)b)n^r$; and (c) when $h_0(z,1) = 1$ and $d = [d_0, d_1, \ldots]$, we get Paasche's result, $c_n = \prod_{i=0}^{n-1}(n-i)^{d_i}$. Here is an implementation.

```
ksmlp h0 d = map last (scanl step h0 d)
 where step hn dn = ((x2z hn)*pascal)!!((length hn - 1)+dn)

moessner r = ksmlp 1 (r:zeros)
long a b r = ksmlp [b,a-b] (r:zeros)
paascheFac = ksmlp 1 [1,1..]
superFac   = ksmlp 1 [1,2..]
```

The above computations convey some Haskell by example, and demonstrate a wealth of experimentation assisted by sequence operations. The core set of definitions are kept to a minimum, so that they are manageable in one file, and should not daunt beginners. In keeping to this principle, we have not implemented an equivalent of formal Laurent series, so we cannot accommodate a sequence for cot (and csc, and so on). However, we can define $x \cot = x \cos / \sin$; but in the form $(x \cos)/\sin$. Then, with reference to item **X**, let $c(r) = rx(\coth \circ rx)$, and test $x \coth = c(1/2) \circ (2x)$, $c(1/2) \circ -x = c(1/2)$ and $c(i) = x \cot$ (Gaussian rationals, introducing i, are in the next section):

```
xcotx, xcothx :: (Eq a, Fractional a) => [a]
xcotx   = (x*cosx)/sinx
xcothx  = (x*coshx)/sinhx
xcth r  = [r]*(x*(coshx `o` ([r]*x)))/(sinhx `o` ([r]*x))
> take 10 xcothx == take 10 ((xcth (1%2)) `o` (2*x))
```

```
True
> take 10 (xcth (1%2)) == take 10 ((xcth (1%2)) `o` (-x))
True
> take 10 xcotx == take 10 (xcth i)
True
```

Sometimes there is simply extra work to be done to convert a sequence expression into a form acceptable by our definitions. For example, the following expression for counting permutations by number of valleys is derived in [22]:

$$K(z,u) = 1 - \frac{1}{u} + \frac{\sqrt{u-1}}{u} \tan\circ(z\sqrt{u-1} + \arctan\circ\frac{1}{\sqrt{u-1}})$$

This fails to compute in our implementation for three reasons (can you spot them?). But, by using the double-angle identity for tan, and $i\tanh = \tan\circ ix$, it can be manipulated into the following form [20], which does compute:

$$K(z,u) = \frac{\sqrt{1-u}}{\sqrt{1-u} - \tanh\circ(z\sqrt{1-u})}$$

The question of how to circumscribe a minimal core set of definitions that perhaps manifest a timeless quality, is a challenging one. It seems only too easy to keep adding stuff, as the next section testifies.

6 Building on the implementation

Implementing sequence algebra is an example of mathematics-programming synergy, as found for example, in [62, 55, 79, 73, 65, 86, 81, 18]. One should note the chronology of language use: [62] uses Fortran, [55] uses psuedo-Algol, [79] uses Pascal, [73] uses Standard ML, [86] uses Maple, [81] uses Scheme, and [65, 18] use Haskell. Haskell is one of the most recent and ambitious in the evolution of programming languages. The story of its development [40] is an informative account of collaborative design in a scientific context. It clearly reveals the tensions between the pursuit of elegant tried-and-tested universal concepts, and pragmatically-motivated more complex and speculative features.

One has to face the fact that Haskell presents the casual newcomer with subtleties, some of which cause bafflement. This slightly detracts from our goal, but also means that the implementation of sequence algebra is a fine benchmark test: Haskell ought to host it well for relative beginners. There are two prominent sources of subtleties: lazy evaluation and type classes. The former might be discovered in working with infinite matrices, for example try rewriting `transpose`. The latter is likely to cause the most frustration. One could write an elucidation of potential "surprises" centred around implementing sequence algebra. That is beyond our scope, but we draw attention to the fact that some type declarations can be omitted, and some not. To take just one example, the final test of the previous section, `take 10 xcotx == take 10 (xcth i)`, does not go through if the type declaration for `xcotx` is omitted (then the system doesn't know to translate the rationals in `xcotx` to Gaussian rationals for comparison). On the other hand, we may omit an explicit type for `xcot` and use the test

```
makeReal (r:&0)  = r
makeReal _       = error "not real"
makeAllReal g    = map makeReal g
> take 10 xcotx == makeAllReal (take 10 (xcth i))
True
```

However, if the g is omitted from the definition of makeAllReal, then makeAllReal is given a different type and the test fails to type-check. Of course, such things have interesting explanations, but they are potentially off-putting for beginners.

These remarks notwithstanding, one cannot resist adding to the implementation in a myriad of ways. Here are a few next-steps, which the author has already taken, and which are left as fruitful exercises.

- Translate [86], and elements of [79], to use Haskell.
- Introduce Gaussian rationals as an instance of Num and Fractional, and test De Moivre's theorem (item **A**). Here is part of a definition and a test of Euler's identity:

```
infix  6  :&
data Gaussian a  = a :& a deriving (Eq, Read, Show)
i    :: (Eq a, Num a) => Gaussian a
i    = 0:&1
ix   :: (Eq a, Num a) => [Gaussian a]
ix = [0,i]

instance  (Num a) => Num (Gaussian a)  where
  -- define negate, +, abs, signum, fromInteger
  (x:&y) * (x':&y') =(x*x'-y*y') :& (x*y'+y*x')

instance  (Fractional a) => Fractional (Gaussian a)  where
  (x:&y) / (x':&y') =  (x*x'+y*y') / d :& (y*x'-x*y') / d
                       where d  = x'*x' + y'*y'
  fromRational a    = fromRational a :& 0

> take 10 (cosx + [i]*sinx) == take 10 (expx `o` ix)
True
```

- Introduce an instance, Shuffle a, of class Num, so that one can write s |><| t as S s * S t, and shuffle power $s^{n\otimes}$ as (S s)^n. Extend the following code, making Shuffle a an instance of Fractional. The first test below illustrates shuffle power. The second test involves the secant numbers, $s = \Lambda\,\mathrm{sec}$. These can be defined (see items **E** and **F**) by applying Λ to the differential equation for sec to give $s = 1 + s \otimes (\Lambda\tan)$, which, since $\tan_0 = 0$, can be written in Haskell as secNums = 1:secNums |><| (e2o tanx). Contrast this to the use of S:

```
newtype Shuffle a = S [a] deriving (Eq, Read, Show)
unS (S s) = s
```

```
instance (Eq a, Num a) => Num (Shuffle a) where
  negate (S s)  = S (negate s)
  (S s) + (S t) = S (s+t)
  (S s) * (S t) = S (s |><| t)
  fromInteger n = S [fromInteger n]
  abs _         = error "abs undefined on Shuffle"
  signum _      = error "signum undefined on Shuffle"

> takeW 6 (unS ((S starx)^2))
[1,2,4,8,16,32]
secNums = 1:unS (S secNums * (S (e2o tanx)))
> takeW 10 secNums
[1,0,1,0,5,0,61,0,1385,0]
```

- Introduce matrix computations. To keep the definitions simple, use the type `[[a]]` for a matrix, and presume, controversially, that it is used responsibly, in the sense that a matrix is presented as a list of rows of agreed length. Transpose is already defined in our implementation (written to work also for infinite matrices). Operations to define include determinant, characteristic polynomial, adjugate, Gaussian elimination, and different methods of inversion. Then one can test computations in proofs of the Cayley-Hamilton theorem, and experiment with bivariate Lagrange inversion (using 2×2 Jacobians).

- Sequences of sequences can become confused with matrices, so it is instructive to define:

  ```
  data Matrix a    = M [[a]] | D a
  instance (Num a) => Num (Matrix a) where  ...
  ```

  The idea is that if `s` is a square matrix of type `[[a]]`, then we can have `M s`. Definitions of addition, `M s + M t`, and multiplication, `M s * M t`, can (with dereliction of duty) assume that `s` and `t` are square of the same dimension. The element `D r` stands for the square diagonal matrix (of any dimension) with `r` along the diagonal. The instance definitions of addition and multiplication each require four clauses (MM, DM, MD, DD), negate has two clauses (M, D):

- Define

  ```
  data Matrix a    = M [[a]] | D a deriving (Eq, Read, Show)

  instance (Eq a, Num a) => Num (Matrix a) where
    negate (M m)   = M (matMap negate m)
    negate (D r)   = D (negate r)
    (M a) + (M b)  = M ...
    ... clauses for + and *

    fromInteger n = D (fromInteger n)
  ```

- Rewrite part III of [55] to use Haskell, making good use of classes and instances to reflect the algebraic structure.

7 Concluding remarks

It is clear that sequence algebra serves calculus: many sequence identities foretell relationships between analytic functions; it serves combinatorics: many counting sequences for discrete structures can be derived by sequence algebra; and it serves computation: it expresses the behaviour of certain kinds of automata; it leads to interpolation methods and summation formulae, and supports program calculation. The theory could hardly be more foundational, and constructing an implementation from scratch emphasises its concreteness, and has the potential to reinforce understanding.

We have exercised the implementation on examples from [32, 25], demonstrating that it makes a valuable companion to those texts. It could be applied to other texts, for example [15, 31, 4, 78, 76]. It can also serve as a centre-piece in a course on functional programming in mathematics. And, indeed, the experience of typing up and experimenting with the code, confronts one with intriguing issues in programming language design. There is zero-testing on sequence elements, which could be used to open a discussion on computability.

The on-line encyclopaedia of integer sequences [74] has hundreds of thousands of sequences. The sequences we have mentioned can be found using the OEIS search facility. It will be noticed that many of the sequences are accompanied by generating code written in various languages, including Haskell. One may like to investigate how many OEIS entries can be expressed in the "language" of tables 4 and 5. A Haskell interface to the OEIS is reported in [87].

Needless to say, to elaborate the topic more fully, with proof details and examples, one needs a book-sized exposition. Beyond that, the obvious question is how to make a seamless progression. A few programming-oriented suggestions are in section 6. On the theory side, we must acknowledge that sequence algebra is so low in the mathematical hierarchy, that it doesn't determine a narrow range of follow-up topics. Nevertheless, we mention a few. One is the classification of sequences, taking a lead from [75, Ch. 6]. Related to this is the computer algebra work done under the heading "generating functions" or "holonomic functions" [35, 46]. It remains to construct a bridge from the elementary level of the present paper to the use of a computer algebra package.

Established results on differential equations, including computer-algebraic, may be revisited with an eye to drawing out those which become particularly accessible when specialised to sequences. Just one suggestion is to bring the method of characteristics as used, for example in [22], into common parlance for sequence work.

Various multivariate directions beckon, including formal languages [5, 3] and multivariate Lagrange inversion [30]. We have also arrived at the threshold of analysis but we have not crossed it, except for bringing π into item **X**. It is natural to ask whether fluency in infinite sequences, as promoted here, has any bearing on how students approach Cauchy sequences and analytic functions. Related to this is the progression from chapter 1 to chapter 2 in [36] (and chapter VII of [19]). On another tack, one may use sequence algebra to motivate abstract algebra. For example, Eilenberg [19, ch. XVI, sect. 10] gives a proof of the Cayley Hamilton theorem using module concepts, and module concepts are used in [26, 27, 28] – papers whose titles echo [48, 49], but which involve a quantum-leap in mathematical sophistication. As a final remark, we note that the eponymous Haskell B. Curry, also abstracted from concrete operations on formal power series [16].

Acknowledgements

This work originated (some years ago) when I was an occasional visitor at the University of York. I am greatly indebted to Colin Runciman for providing that opportunity. Thanks are also due to Daniel Siemssen, Jeremy Jacob, and Detlef Plump for comments and encouragement.

References

[1] M. Aigner and G.M. Ziegler. *Proofs from THE BOOK, 3rd edn.* Springer, 2004.

[2] W. Basler. *Formal Power Series and Linear Systems of Meromorphic Differential Equations.* Springer, 2000.

[3] H Basold, H Hansen, J-É Pin, and J Rutten. Newton series, coinductively: a comparative study of composition. *Mathematical Structures in Computer Science*, pages 1–29, 2017.

[4] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopaedia of Mathematics.* Cambridge University Press, 1998. Translated from 1994 original in French.

[5] J. Berstel and C. Reutenauer. *Rational Series and their Languages*, volume 12 of *EATCS Monographs on Theoretical Computer Science.* Springer Verlag, 1988.

[6] R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice Hall International, 1997.

[7] R.S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.

[8] L. Brand. A Division Algebra for Sequences and Its Associated Operational Calculus. *The American Mathematical Monthly*, 71(7):719–728, 1964.

[9] L. Brand. *Differential and Difference Equations.* J. Wiley, 1966.

[10] B. Buchberger and M. Rosenkranz. Transforming problems from analysis to algebra: A case study in linear boundary problems. *J. Symbolic Computation*, 47:589–609, 2012.

[11] P. Cameron. *Notes on Counting: an Introduction to Enumerative Combinatorics.* Australian Mathematical Society Lecture Series. Cambridge University Press, 2017.

[12] A. Cayley. On the anayltical form called Trees. Second Part. *Philosophical Magazine*, XVIII:374–378, 1859.

[13] A. Cayley. A theorem on trees. *Quart. J. Pure Appl. Math.*, 23:376–378, 1889.

[14] W.Y.C. Chen. Context-free grammars, differential operators, and formal power series. *Theoretical Computer Science*, 117:113–129, 1993.

[15] L. Comtet. *Advanced Combinatorics: The Art of Finite and Infinite Expansions.* Springer, 1974.

[16] H. B. Curry. Abstract Differential Operators and Interpolation Formulas. *Portugaliae Mathematica*, 10(4):135–162, 1951.

[17] N. Dershowitz and S Zaks. The Cycle Lemma and Some Applications. *European J. Combinatorics*, 11(1):35–40, 1990.

[18] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing.* College Publications, 2012.

[19] S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.

[20] S. Elizalde and M. Noy. Consecutive patterns in permutations. *Adv. Applied Mathematics*, 30:110–125, 2003.

[21] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs.* The American Mathematical Society, 2003.

[22] C.J. Fewster and D. Siemssen. Enumerating Permutations by their Run Structure. *Electronic Journal of Combinatorics*, 21(4), 2014.

[23] P. Flajolet. Combinatorial Aspects of Continued Fractions. *Discrete Mathematics*, 32:125–161, 1980.

[24] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79:37–109, 1991.

[25] P. Flajolet and R. Sedgewick. *Analytic Combinatorics.* Cambridge University Press, 2009.

[26] L. Gatto. *Linear ODEs: an Algebraic Perspective.* Instituto Nacional de Matheática e Aplicada, Rio de Janeiro, 2012.

[27] L. Gatto and D Laksov. From linear recurrence relations to linear ODEs with constant coefficients. *J. Algebra and its Applicatins*, 5(6):3–19, 2016.

[28] L. Gatto and I. Scherbak. Remarks on the Cayely-Hamilton theorem. *arXiv:1510.03022*, 2015.

[29] I. M. Gessel. Lagrange inversion. *J. Comb. Theory Ser. A*, 144(C):212–249, November 2016.

[30] I.M. Gessel. A Combinatorial Proof of the Multivariate Lagrange Inversion Formula. *J. Combinatorial Theory, Series A*, 45:178–195, 1987.

[31] I.P. Goulden and D.M. Jackson. *Combinatorial Enumeration.* John Wiley & Sons, 1983.

[32] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics, 2nd edn.* Addison-Wesley, 1994.

[33] H.H. Hansen, C. Kupke, and J. Rutten. Stream Differential Equations: Specification Formats and Solution Methods. *Logical Methods in Computer Sciience*, 13(1:3):1–51, 2017.

[34] W.K. Hayman. Review of Generatingfunctionoly by H.S. Wilf. *Bulletin of Americam Math. Soc.*, 21(1):104–106, 1991.

[35] W. Hebisch and M. Rubey. Extended rate, more gfun. *J. Symb. Comput.*, 46(8):889–903, August 2011.

[36] P. Henrici. *Applied and Computational Complex Analysis, Vol 1.* John Wiley and Sons, 1974.

[37] R Hinze. Functional pearl: Streams and Unique Fixed Points. *ACM SIGPLAN Notices*, 43(9):189–200, 2008.

[38] R Hinze. Scans and convolutions – a calculational proof of Moessner's theorem. In Sven-Bodo Scholtz, editor, *Proc. 20th Intl. Symposium on The Implementation and Application of Functional Languages (IFL 08), vol. 5836 Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[39] R. Hinze. Concrete Stream Calculus: An Extended Study. *J. Functional Programming*, 20(5–6):463–535, November 2010.

[40] P. Hudak, J. Hughes, S. Peyton-Jones, and P. Wadler. A History of Haskell: Being Lazy With Class. In *Proc. Third ACM SIGPLAN History of Programming Languages Conf.* ACM, 2007.

[41] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the EATCS*, 62:222–259, 1997.

[42] N. Jacobson. *Lectures in Abstract Algebra, Vol 1.* Van Nostrand, 1951.

[43] W.P. Johnson. The Curious History of Faà di Bruno's Formula. *The American Mathematical Monthly*, 109:217–234, 2002.

[44] A. Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

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

[46] M. Kauers. The Holonomic Toolkit. In Carsten Schneider and Johannes Blümlein, editors, *Computer Algebra in Quantum Field Theory: Integration, Summation and Special Functions*, pages 119–144. Springer Vienna, 2013.

[47] W.F. Keigher. On the ring of hurwitz series. *Communications in Algebra*, 25(6):1845–1859, 1997.

[48] D. A. Klarner. Algebraic theory for difference and differential equations. *The American Mathematical Monthly*, 76(4):366–373, 1969.

[49] D.A. Klarner. Some Remarks on the Cayley-Hamilton Theorem. *The American Mathematical Monthly*, 83(5):367–369, 1976.

[50] K. Knopp. *Theory and Application of Infinite Series, 4th edn.* Blackie & Son, 1951.

[51] D.E. Knuth. Bracket Notation for the 'Coefficient of' Operator. In A. W. Roscoe, editor, *A Classical Mind*, pages 247–258. Prentice Hall International (UK) Ltd., 1994.

[52] D. Kozen and A. Silva. On Moessner's Theorem. *The American Mathematical Monthly*, 120(2):131–139, 2013.

[53] M. Kwapisz. The Power of a Matrix. *SIAM Rev.*, 40(3):703–705, 1998.

[54] I.E. Leonard. The Matrix Exponential. *SIAM Rev.*, 38(3):507–512, 1996.

[55] J.D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley, 1981.

[56] E. Liz. A Note on the Matrix Exponential. *SIAM Rev.*, 40(3):700–702, 1998.

[57] M.D. McIlroy. Power series, power serious. *J. of Functional Programming*, 3(9):325–337, 1999.

[58] M.D. McIlroy. The Music of Streams. *Inf. Proc. Letts.*, 77:189–195, 2001.

[59] D. Merlini, R. Sprugnoli, and M.C. Verri. Lagrange Inversion: When and How. *Acta Applicandae Mathematica*, 94(3):233–249, 2006.

[60] D. Merlini, R. Sprugnoli, and M.C. Verri. The Method of Coefficients. *The American Mathematical Monthly*, 114:40–57, 2007.

[61] H. Niederhausen. Finite Operator Calculus With Applications to Linear Recursions. Florida Altlantic University, 2010.

[62] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms, 2nd edn.* Academic Press, 1978.

[63] Milad Niqui and M. Rutten. An exercise in coinduction : Moessner ' s theorem. Technical Report SEN-1103, CWI, Amsterdam, 2011.

[64] I. Niven. Formal Power Series. *The American Mathematical Monthly*, 76:871–889, 1969.

[65] J.T. O'Donnell, C.V. Hall, and R. Page. *Discrete Mathematics using a Computer, 2nd edn.* Springer, 2006.

[66] I. Pak. History of Catalan Numbers. In *R.P. Stanley, The Catalan Numbers.* Cambridge University Press, 2014.

[67] D. Pavlović and M. Escardó. Calculus in Coinductive Form. In *Proc. of the 13th Annual IEEE Symposium on Logic in Computer Science*, pages 408–417. IEEE Computer Society Press, 1998.

[68] S. Peyton-Jones, editor. *The Haskell 98 Language Report.* www.Haskel.org, 2002. See also The Haskell 2010 Language Report, e.d. S. Marlow.

[69] J. Pitman. Enumerations of trees and forests related to branching processes and random walks. In *Microsurveys in Discrete Probability, number 41 in DIMACS Ser. Discrete Math. Theoret. Comp. Sci*, pages 163–180, 1998.

[70] G.N. Raney. Functional composition patterns and power series reversion. *Trans. American Mathematical Soc.*, 94:441–451, 1960.

[71] J. J. M. M. Rutten. A Coinductive Calculus of Streams. *Math. Struct. in Comp. Sci.*, 15(1):93–147, February 2005.

[72] J.J.M.M. Rutten. Coinductive counting with weighted automata. *J. Automata, Languages and Combinatorics*, 8(2):319–352, 2003a.

[73] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice-Hall, 1988.

[74] N.J.A. Sloane, editor. *The On-Line Encyclopedia of Integer Sequences.* Published electronically at https://oeis.org.

[75] R.P. Stanley. Hipparchus, Plutarch, Schröder, and Hough. *The American Mathematical Monthly*, 104:344–350, 1997.

[76] R.P. Stanley. *Enumerative Combinatorics: vol 2.* Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999. With a contribution by S. Fomin.

[77] R.P. Stanley. A Survey of Alternating Permutations. *Contemp. Math.*, 531:165–196, 2010.

[78] R.P. Stanley. *Enumerative Combinatorics: vol 1 (2nd edn.).* Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2011. (First ed. 1986).

[79] D. Stanton and D. White. *Constructive Combinatorics.* Undergraduate Texts in Mathematics. Springer-Verlag, 1986.

[80] R. Street. Trees, Permutations and the Tangent Function. *Reflections, Math. Assoc. of NSW*, 27(2):19–23, 2002.

[81] G.J. Sussman and J. Wisdom. *Functional Differential Geometry.* M.I.T. Press, 2012. with W. Farr.

[82] J. F. Traub. Generalized Sequences with Applications to the Discrete Calculus. *Mathematics of Computation*, 19(90):177–200, 1965.

[83] W.T. Tutte. On Elementary Calculus and the Good Formula. *Journal of Combinatorial Theory (B)*, 18:79–137, 1975.

[84] M. Ward. A Calculus of Sequences. *American Journal of Mathematics*, 58:255–266, 1936.

[85] H.S. Wilf. *Generatingfunctionology.* Academic Press, 1990. 2nd Edition (1994) available at http://www.cis.upenn.edu/˜wilf.

[86] H.S. Wilf. East Side, West Side ... an introduction to combinatorial families – with Maple programming. Available at http://www.cis.upenn.edu/˜wilf, 2002.

[87] J. Winter. QStream: A Suite of Streams. In R. Heckel and S. Milius, editors, *Algebra and Coalgebra in Computer Science. CALCO 2013.* Springer, 2013.

[88] D. Zeilberger. Garsia and Milne's bijective proof of the Inclusion-Exclusion principle. *Discrete Mathematics*, 51:109–110, 1984.